Competitive Programmer's Handbook

(modified for Python)

Antti Laaksonen[1]

Draft November 25, 2024

[1]This contains parts of "Competitive Programmer's Handbook". The included chapters have been modified by Inge Li Gørtz. The C++ code has been changed to Python, and minor changes have been made in chapter 12. The license of the book is Creative Commons BY-NC-SA. To get the full version of the original book go here: https://cses.fi/book/index.php

Contents

Part I

Basic techniques

Chapter 2

Time complexity

The efficiency of algorithms is important in competitive programming. Usually, it is easy to design an algorithm that solves the problem slowly, but the real challenge is to invent a fast algorithm. If the algorithm is too slow, it will get only partial points or no points at all.

The **time complexity** of an algorithm estimates how much time the algorithm will use for some input. The idea is to represent the efficiency as a function whose parameter is the size of the input. By calculating the time complexity, we can find out whether the algorithm is fast enough without implementing it.

2.1 Calculation rules

The time complexity of an algorithm is denoted $O(\cdots)$ where the three dots represent some function. Usually, the variable n denotes the input size. For example, if the input is an array of numbers, n will be the size of the array, and if the input is a string, n will be the length of the string.

Loops

A common reason why an algorithm is slow is that it contains many loops that go through the input. The more nested loops the algorithm contains, the slower it is. If there are k nested loops, the time complexity is $O(n^k)$.

For example, the time complexity of the following code is $O(n)$:

```
for i in range(n):
    // code
```

And the time complexity of the following code is $O(n^2)$:

```
for i in range(n):
    for j in range(n):
        // code
```

Order of magnitude

A time complexity does not tell us the exact number of times the code inside a loop is executed, but it only shows the order of magnitude. In the following examples, the code inside the loop is executed $3n$, $n+5$ and $\lceil n/2 \rceil$ times, but the time complexity of each code is $O(n)$.

```
for i in range(3*n):
    // code
```

```
for i in range(n+5):
    // code
```

```
for i in range(0,n,2):
    // code
```

As another example, the time complexity of the following code is $O(n^2)$:

```
for i in range(n):
    for j in range(i+1,n):
        // code
```

Phases

If the algorithm consists of consecutive phases, the total time complexity is the largest time complexity of a single phase. The reason for this is that the slowest phase is usually the bottleneck of the code.

For example, the following code consists of three phases with time complexities $O(n)$, $O(n^2)$ and $O(n)$. Thus, the total time complexity is $O(n^2)$.

```
for i in range(n):
    // code

for i in range(n):
    for j in range(n):
        // code

for i in range(n):
    // code
```

Several variables

Sometimes the time complexity depends on several factors. In this case, the time complexity formula contains several variables.

For example, the time complexity of the following code is $O(nm)$:

```
for i in range(n):
    for j in range(m):
        // code
```

Recursion

The time complexity of a recursive function depends on the number of times the function is called and the time complexity of a single call. The total time complexity is the product of these values.

For example, consider the following function:

```
def f(n):
    if (n == 1):
        return
    f(n-1)
```

The call `f`(n) causes n function calls, and the time complexity of each call is $O(1)$. Thus, the total time complexity is $O(n)$.

As another example, consider the following function:

```
def g(n):
    if (n == 1):
        return
    g(n-1)
    g(n-1)
```

In this case each function call generates two other calls, except for $n = 1$. Let us see what happens when g is called with parameter n. The following table shows the function calls produced by this single call:

function call	number of calls
$g(n)$	1
$g(n-1)$	2
$g(n-2)$	4
$\cdots$	$\cdots$
$g(1)$	2^{n-1}

Based on this, the time complexity is

$$1+2+4+\cdots+2^{n-1} = 2^n - 1 = O(2^n).$$

2.2 Complexity classes

The following list contains common time complexities of algorithms:

$O(1)$ The running time of a **constant-time** algorithm does not depend on the input size. A typical constant-time algorithm is a direct formula that calculates the answer.

$O(\log n)$ A **logarithmic** algorithm often halves the input size at each step. The running time of such an algorithm is logarithmic, because $\log_2 n$ equals the number of times n must be divided by 2 to get 1.

$O(\sqrt{n})$ A **square root algorithm** is slower than $O(\log n)$ but faster than $O(n)$. A special property of square roots is that $\sqrt{n} = n/\sqrt{n}$, so the square root $\sqrt{n}$ lies, in some sense, in the middle of the input.

$O(n)$ A **linear** algorithm goes through the input a constant number of times. This is often the best possible time complexity, because it is usually necessary to access each input element at least once before reporting the answer.

$O(n \log n)$ This time complexity often indicates that the algorithm sorts the input, because the time complexity of efficient sorting algorithms is $O(n \log n)$. Another possibility is that the algorithm uses a data structure where each operation takes $O(\log n)$ time.

$O(n^2)$ A **quadratic** algorithm often contains two nested loops. It is possible to go through all pairs of the input elements in $O(n^2)$ time.

$O(n^3)$ A **cubic** algorithm often contains three nested loops. It is possible to go through all triplets of the input elements in $O(n^3)$ time.

$O(2^n)$ This time complexity often indicates that the algorithm iterates through all subsets of the input elements. For example, the subsets of $\{1,2,3\}$ are $\emptyset$, $\{1\}$, $\{2\}$, $\{3\}$, $\{1,2\}$, $\{1,3\}$, $\{2,3\}$ and $\{1,2,3\}$.

$O(n!)$ This time complexity often indicates that the algorithm iterates through all permutations of the input elements. For example, the permutations of $\{1,2,3\}$ are $(1,2,3)$, $(1,3,2)$, $(2,1,3)$, $(2,3,1)$, $(3,1,2)$ and $(3,2,1)$.

An algorithm is **polynomial** if its time complexity is at most $O(n^k)$ where k is a constant. All the above time complexities except $O(2^n)$ and $O(n!)$ are polynomial. In practice, the constant k is usually small, and therefore a polynomial time complexity roughly means that the algorithm is *efficient*.

Most algorithms in this book are polynomial. Still, there are many important problems for which no polynomial algorithm is known, i.e., nobody knows how to solve them efficiently. **NP-hard** problems are an important set of problems, for which no polynomial algorithm is known[1].

[1] A classic book on the topic is M. R. Garey's and D. S. Johnson's *Computers and Intractability: A Guide to the Theory of NP-Completeness* [16].

2.3 Estimating efficiency

By calculating the time complexity of an algorithm, it is possible to check, before implementing the algorithm, that it is efficient enough for the problem. The starting point for estimations is the fact that a modern computer can perform some hundreds of millions of operations in a second.

For example, assume that the time limit for a problem is one second and the input size is $n = 10^5$. If the time complexity is $O(n^2)$, the algorithm will perform about $(10^5)^2 = 10^{10}$ operations. This should take at least some tens of seconds, so the algorithm seems to be too slow for solving the problem.

On the other hand, given the input size, we can try to *guess* the required time complexity of the algorithm that solves the problem. The following table contains some useful estimates assuming a time limit of one second.

input size	required time complexity
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n)$
$n \leq 500$	$O(n^3)$
$n \leq 5000$	$O(n^2)$
$n \leq 10^6$	$O(n \log n)$ or $O(n)$
n is large	$O(1)$ or $O(\log n)$

For example, if the input size is $n = 10^5$, it is probably expected that the time complexity of the algorithm is $O(n)$ or $O(n \log n)$. This information makes it easier to design the algorithm, because it rules out approaches that would yield an algorithm with a worse time complexity.

Still, it is important to remember that a time complexity is only an estimate of efficiency, because it hides the *constant factors*. For example, an algorithm that runs in $O(n)$ time may perform $n/2$ or $5n$ operations. This has an important effect on the actual running time of the algorithm.

2.4 Maximum subarray sum

There are often several possible algorithms for solving a problem such that their time complexities are different. This section discusses a classic problem that has a straightforward $O(n^3)$ solution. However, by designing a better algorithm, it is possible to solve the problem in $O(n^2)$ time and even in $O(n)$ time.

Given an array of n numbers, our task is to calculate the **maximum subarray sum**, i.e., the largest possible sum of a sequence of consecutive values in the array[2]. The problem is interesting when there may be negative values in the array. For example, in the array

−1	2	4	−3	5	2	−5	2

[2] J. Bentley's book *Programming Pearls* [5] made the problem popular.

the following subarray produces the maximum sum 10:

−1	2	4	−3	5	2	−5	2

We assume that an empty subarray is allowed, so the maximum subarray sum is always at least 0.

Algorithm 1

A straightforward way to solve the problem is to go through all possible subarrays, calculate the sum of values in each subarray and maintain the maximum sum. The following code implements this algorithm:

```
best = 0
for a in range(n):
    for b in range(a,n):
        sum = 0
        for k in range(a, b+1):
            sum += array[k]
        best = max(best,sum)
print(best)
```

The variables `a` and `b` fix the first and last index of the subarray, and the sum of values is calculated to the variable `sum`. The variable `best` contains the maximum sum found during the search.

The time complexity of the algorithm is $O(n^3)$, because it consists of three nested loops that go through the input.

Algorithm 2

It is easy to make Algorithm 1 more efficient by removing one loop from it. This is possible by calculating the sum at the same time when the right end of the subarray moves. The result is the following code:

```
best = 0
for a in range(n):
    sum = 0
    for b in range(a,n):
        sum += array[b]
        best = max(best,sum)
print(best)
```

After this change, the time complexity is $O(n^2)$.

Algorithm 3

Surprisingly, it is possible to solve the problem in $O(n)$ time[3], which means that just one loop is enough. The idea is to calculate, for each array position, the maximum sum of a subarray that ends at that position. After this, the answer for the problem is the maximum of those sums.

Consider the subproblem of finding the maximum-sum subarray that ends at position k. There are two possibilities:

1. The subarray only contains the element at position k.
2. The subarray consists of a subarray that ends at position $k-1$, followed by the element at position k.

In the latter case, since we want to find a subarray with maximum sum, the subarray that ends at position $k-1$ should also have the maximum sum. Thus, we can solve the problem efficiently by calculating the maximum subarray sum for each ending position from left to right.

The following code implements the algorithm:

```
best = 0
sum = 0
for k in range(n):
    sum = max(array[k],sum+array[k])
    best = max(best,sum)
print(best)
```

The algorithm only contains one loop that goes through the input, so the time complexity is $O(n)$. This is also the best possible time complexity, because any algorithm for the problem has to examine all array elements at least once.

Efficiency comparison

It is interesting to study how efficient algorithms are in practice. The following table shows the running times of the above algorithms for different values of n on a modern computer.

In each test, the input was generated randomly. The time needed for reading the input was not measured.

array size n	Algorithm 1	Algorithm 2	Algorithm 3
10^2	0.0 s	0.0 s	0.0 s
10^3	0.1 s	0.0 s	0.0 s
10^4	> 10.0 s	0.1 s	0.0 s
10^5	> 10.0 s	5.3 s	0.0 s
10^6	> 10.0 s	> 10.0 s	0.0 s
10^7	> 10.0 s	> 10.0 s	0.0 s

[3] In [5], this linear-time algorithm is attributed to J. B. Kadane, and the algorithm is sometimes called **Kadane's algorithm**.

The comparison shows that all algorithms are efficient when the input size is small, but larger inputs bring out remarkable differences in the running times of the algorithms. Algorithm 1 becomes slow when $n = 10^4$, and Algorithm 2 becomes slow when $n = 10^5$. Only Algorithm 3 is able to process even the largest inputs instantly.

Chapter 3

Sorting

Sorting is a fundamental algorithm design problem. Many efficient algorithms use sorting as a subroutine, because it is often easier to process data if the elements are in a sorted order.

For example, the problem ”does an array (list) contain two equal elements?” is easy to solve using sorting. If the array contains two equal elements, they will be next to each other after sorting, so it is easy to find them. Also, the problem ”what is the most frequent element in a array?” can be solved similarly.

There are many algorithms for sorting, and they are also good examples of how to apply different algorithm design techniques. The efficient general sorting algorithms work in $O(n \log n)$ time, and many algorithms that use sorting as a subroutine also have this time complexity.

3.1 Sorting theory

The basic problem in sorting is as follows:

> Given a lilst that contains n elements, your task is to sort the elements in increasing order.

For example, the array

1	3	8	2	9	2	5	6

will be as follows after sorting:

1	2	2	3	5	6	8	9

$O(n^2)$ algorithms

Simple algorithms for sorting a array work in $O(n^2)$ time. Such algorithms are short and usually consist of two nested loops. A famous $O(n^2)$ time sorting

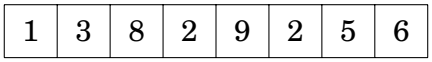
algorithm is **bubble sort** where the elements ”bubble” in the array according to their values.

Bubble sort consists of n rounds. On each round, the algorithm iterates through the elements of the array. Whenever two consecutive elements are found that are not in correct order, the algorithm swaps them. The algorithm can be implemented as follows:

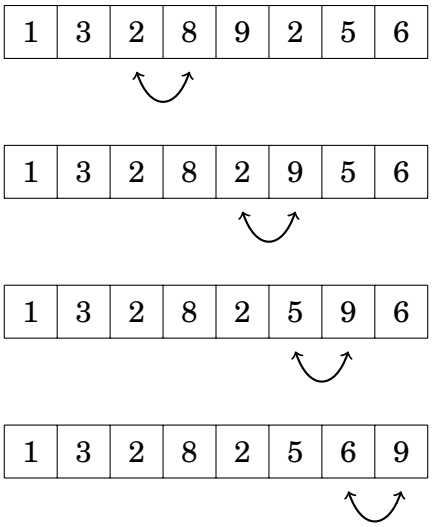
```
for i in range(n):
    for j in range(n-1):
        if (L[j] > L[j+1]):
            L[j],L[j+1] = L[j+1],L[j]
```

After the first round of the algorithm, the largest element will be in the correct position, and in general, after k rounds, the k largest elements will be in the correct positions. Thus, after n rounds, the whole array will be sorted.

For example, in the array



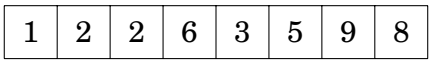
the first round of bubble sort swaps elements as follows:



Inversions

Bubble sort is an example of a sorting algorithm that always swaps *consecutive* elements in the array. It turns out that the time complexity of such an algorithm is *always* at least $O(n^2)$, because in the worst case, $O(n^2)$ swaps are required for sorting the array.

A useful concept when analyzing sorting algorithms is an **inversion**: a pair of array elements $(\texttt{array}[a], \texttt{array}[b])$ such that $a < b$ and $\texttt{array}[a] > \texttt{array}[b]$, i.e., the elements are in the wrong order. For example, the array



has three inversions: (6,3), (6,5) and (9,8). The number of inversions indicates how much work is needed to sort the array. An array is completely sorted when there are no inversions. On the other hand, if the array elements are in the reverse order, the number of inversions is the largest possible:

$$1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

Swapping a pair of consecutive elements that are in the wrong order removes exactly one inversion from the array. Hence, if a sorting algorithm can only swap consecutive elements, each swap removes at most one inversion, and the time complexity of the algorithm is at least $O(n^2)$.

$O(n \log n)$ algorithms

It is possible to sort an array efficiently in $O(n \log n)$ time using algorithms that are not limited to swapping consecutive elements. One such algorithm is **merge sort**[1], which is based on recursion.

Merge sort sorts a subarray $\texttt{array}[a \ldots b]$ as follows:

1. If $a = b$, do not do anything, because the subarray is already sorted.
2. Calculate the position of the middle element: $k = \lfloor (a+b)/2 \rfloor$.
3. Recursively sort the subarray $\texttt{array}[a \ldots k]$.
4. Recursively sort the subarray $\texttt{array}[k+1 \ldots b]$.
5. *Merge* the sorted subarrays $\texttt{array}[a \ldots k]$ and $\texttt{array}[k+1 \ldots b]$ into a sorted subarray $\texttt{array}[a \ldots b]$.

Merge sort is an efficient algorithm, because it halves the size of the subarray at each step. The recursion consists of $O(\log n)$ levels, and processing each level takes $O(n)$ time. Merging the subarrays $\texttt{array}[a \ldots k]$ and $\texttt{array}[k+1 \ldots b]$ is possible in linear time, because they are already sorted.

For example, consider sorting the following array:

1	3	6	2	8	2	5	9

The array will be divided into two subarrays as follows:

1	3	6	2

8	2	5	9

Then, the subarrays will be sorted recursively as follows:

1	2	3	6

2	5	8	9

Finally, the algorithm merges the sorted subarrays and creates the final sorted array:

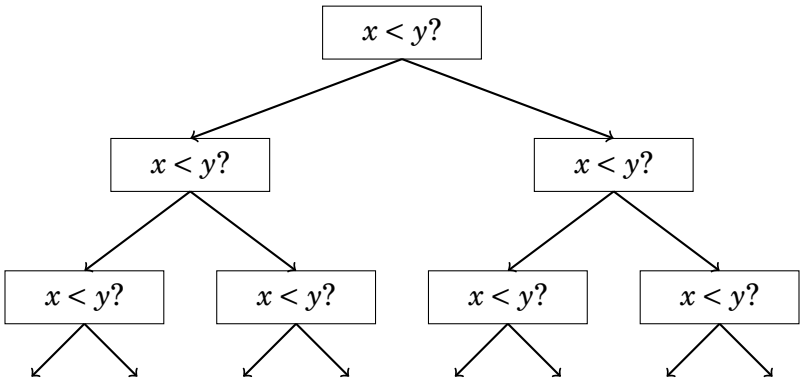
1	2	2	3	5	6	8	9

[1] According to [23], merge sort was invented by J. von Neumann in 1945.

Sorting lower bound

Is it possible to sort an array faster than in $O(n \log n)$ time? It turns out that this is *not* possible when we restrict ourselves to sorting algorithms that are based on comparing array elements.

The lower bound for the time complexity can be proved by considering sorting as a process where each comparison of two elements gives more information about the contents of the array. The process creates the following tree:



Here "$x < y$?" means that some elements x and y are compared. If $x < y$, the process continues to the left, and otherwise to the right. The results of the process are the possible ways to sort the array, a total of $n!$ ways. For this reason, the height of the tree must be at least

$$\log_2(n!) = \log_2(1) + \log_2(2) + \cdots + \log_2(n).$$

We get a lower bound for this sum by choosing the last $n/2$ elements and changing the value of each element to $\log_2(n/2)$. This yields an estimate

$$\log_2(n!) \geq (n/2) \cdot \log_2(n/2),$$

so the height of the tree and the minimum possible number of steps in a sorting algorithm in the worst case is at least $n \log n$.

Counting sort

The lower bound $n \log n$ does not apply to algorithms that do not compare array elements but use some other information. An example of such an algorithm is **counting sort** that sorts an array in $O(n)$ time assuming that every element in the array is an integer between $0 \ldots c$ and $c = O(n)$.

The algorithm creates a *bookkeeping* array, whose indices are elements of the original array. The algorithm iterates through the original array and calculates how many times each element appears in the array.

For example, the array

1	3	6	9	9	3	5	9

corresponds to the following bookkeeping array:

1	2	3	4	5	6	7	8	9
1	0	2	0	1	1	0	0	3

For example, the value at position 3 in the bookkeeping array is 2, because the element 3 appears 2 times in the original array.

Construction of the bookkeeping array takes $O(n)$ time. After this, the sorted array can be created in $O(n)$ time because the number of occurrences of each element can be retrieved from the bookkeeping array. Thus, the total time complexity of counting sort is $O(n)$.

Counting sort is a very efficient algorithm but it can only be used when the constant c is small enough, so that the array elements can be used as indices in the bookkeeping array.

3.2 Sorting in Python

It is almost never a good idea to use a home-made sorting algorithm in a contest, because there are good implementations available in programming languages. For example, the Python has a built-in function `sort` that can be easily used for sorting arrays and other data structures.

There are many benefits in using a built-in function. First, it saves time because there is no need to implement the function. Second, the built-in implementation is certainly correct and efficient: it is not probable that a home-made sorting function would be better.

In this section we will see how to use the Python `sort` function. The following code sorts a vector in increasing order:

```
L = [4,2,5,3,5,8,3]
L.sort()
```

After the sorting, the contents of the list will be $[2,3,3,4,5,5,8]$. The default sorting order is increasing, but a reverse order is possible as follows:

```
L.sort(reverse=True);
```

There is also a function `sorted` that builds and returns a new sorted list.

```
L = [4,2,5,3,5,8,3]
S = sorted(L)
```

The list `S` now contains the sorted list $[2,3,3,4,5,5,8]$

Comparison operators

The function `sort` requires that a **comparison operator** is defined for the data type of the elements to be sorted. When sorting, this operator will be used whenever it is necessary to find out the order of two elements.

Most Python data types have a built-in comparison operator, and elements of those types can be sorted automatically. For example, numbers are sorted according to their values and strings are sorted in alphabetical order.

Pairs are sorted primarily according to their first elements. However, if the first elements of two pairs are equal, they are sorted according to their second elements:

```
L = [(1,5),(2,3),(1,2)]
L.sort()
```

After this, the order of the pairs is (1,2), (1,5) and (2,3).

In a similar way, tuples (`tuple`) are sorted primarily by the first element, secondarily by the second element, etc.:

```
L = L[(2,1,4),(1,5,3),(2,1,3)]
L.sort()
```

After this, the order of the tuples is (1,5,3), (2,1,3) and (2,1,4).

You can also specify which key to sort tuples by. For example:

```
L = [('john', 'A', 15),('jane', 'B', 12),('dave', 'B', 10) ]
L.sort(key=lambda s: s[2])
```

After this, the order of the tuples is ('dave', 'B', 10), ('jane', 'B', 12) and ('john', 'A', 15).

3.3 Binary search

A general method for searching for an element in an array is to use a `for` loop that iterates through the elements of the array. For example, the following code searches for an element x in an array:

```
for i in range(n):
    if array[i] == x:
        # x found at index i
```

The time complexity of this approach is $O(n)$, because in the worst case, it is necessary to check all elements of the array. If the order of the elements is arbitrary, this is also the best possible approach, because there is no additional information available where in the array we should search for the element x.

However, if the array is *sorted*, the situation is different. In this case it is possible to perform the search much faster, because the order of the elements in

the array guides the search. The following **binary search** algorithm efficiently searches for an element in a sorted array in $O(\log n)$ time.

Method 1

The usual way to implement binary search resembles looking for a word in a dictionary. The search maintains an active region in the array, which initially contains all array elements. Then, a number of steps is performed, each of which halves the size of the region.

At each step, the search checks the middle element of the active region. If the middle element is the target element, the search terminates. Otherwise, the search recursively continues to the left or right half of the region, depending on the value of the middle element.

The above idea can be implemented as follows:

```
a = 0
b = n-1
while a <= b:
    k = (a+b)//2
    if array[k] == x:
        # x found at index k
    else:
        a = k+1
```

In this implementation, the active region is $a \ldots b$, and initially the region is $0 \ldots n-1$. The algorithm halves the size of the region at each step, so the time complexity is $O(\log n)$.

Method 2

An alternative method to implement binary search is based on an efficient way to iterate through the elements of the array. The idea is to make jumps and slow the speed when we get closer to the target element.

The search goes through the array from left to right, and the initial jump length is $n/2$. At each step, the jump length will be halved: first $n/4$, then $n/8$, $n/16$, etc., until finally the length is 1. After the jumps, either the target element has been found or we know that it does not appear in the array.

The following code implements the above idea:

```
k = 0
b = n//2
while b >= 1:
    while (k+b < n and array[k+b] <= x):
        k += b
    b //= 2
if array[k] == x:
    # x found at index k
```

During the search, the variable b contains the current jump length. The time complexity of the algorithm is $O(\log n)$, because the code in the `while` loop is performed at most twice for each jump length.

Python functions

The python module `bisect` contains the following functions that are based on binary search and work in logarithmic time:

- `bisect.bisect_left(array, x)` returns a pointer to the first array element whose value is at least x.
- `ubisect.bisect_right(array, x)` returns a pointer to the first array element whose value is larger than x.

The functions assume that the array is sorted. If there is no such element, the pointer points to the element after the last array element. For example, the following code finds out whether an array contains an element with value x:

```
k = bisect.bisect_left(array,x)
if k != len(array) and array[k] == x:
    # x found at index k
```

Then, the following code counts the number of elements whose value is x:

```
a = bisect.bisect_left(array,x)
b = bisect.bisect_right(array,x)
print(b-a)
```

You can use parameters `lo` and `hi` to specify a subset of the list which should be considered. By default the entire list is used. E.g., `bisect.bisect_left(array, x)= bisect.bisect_left(array, x, lo = 0, hi = len(array))`.

Finding the smallest solution

An important use for binary search is to find the position where the value of a *function* changes. Suppose that we wish to find the smallest value k that is a valid solution for a problem. We are given a function ok(x) that returns `true` if x is a valid solution and `false` otherwise. In addition, we know that ok(x) is `false` when $x < k$ and `true` when $x \geq k$. The situation looks as follows:

x	0	1	$\cdots$	$k-1$	k	$k+1$	$\cdots$
ok(x)	false	false	$\cdots$	false	true	true	$\cdots$

Now, the value of k can be found using binary search:

```
x = -1
b = z
while b >= 1:
    while (not ok(x+b)):
        x += b
    b //= 2
k = x+1
```

The search finds the largest value of x for which ok(x) is `false`. Thus, the next value $k = x+1$ is the smallest possible value for which ok(k) is `true`. The initial jump length z has to be large enough, for example some value for which we know beforehand that ok(z) is `true`.

The algorithm calls the function ok $O(\log z)$ times, so the total time complexity depends on the function ok. For example, if the function works in $O(n)$ time, the total time complexity is $O(n \log z)$.

Finding the maximum value

Binary search can also be used to find the maximum value for a function that is first increasing and then decreasing. Our task is to find a position k such that

- $f(x) < f(x+1)$ when $x < k$, and
- $f(x) > f(x+1)$ when $x \geq k$.

The idea is to use binary search for finding the largest value of x for which $f(x) < f(x+1)$. This implies that $k = x+1$ because $f(x+1) > f(x+2)$. The following code implements the search:

```
x = -1
b = z
while b >= 1:
    while (f(x+b) < f(x+b+1)):
        x += b
int k = x+1
```

Note that unlike in the ordinary binary search, here it is not allowed that consecutive values of the function are equal. In this case it would not be possible to know how to continue the search.

Chapter 4

Data structures

A **data structure** is a way to store data in the memory of a computer. It is important to choose an appropriate data structure for a problem, because each data structure has its own advantages and disadvantages. The crucial question is: which operations are efficient in the chosen data structure?

This chapter introduces the most important data structures in the Python standard library. It is a good idea to use the standard library whenever possible, because it will save a lot of time. Later in the book we will learn about more sophisticated data structures that are not available in the standard library.

4.1 Dynamic arrays

A **dynamic array** is an array whose size can be changed during the execution of the program. The most popular dynamic array in Python is the `list` structure. The following code creates an empty list and adds three elements to it:

```
L = []
L.append(3) # [3]
L.append(2) # [3,2]
L.append(5) # [3,2,5]
```

After this, the elements can be accessed like this:

```
print(L[0]) # 3
print(L[1]) # 2
print(L[2]) # 5
```

The function `len` returns the number of elements in the list. The following code iterates through the list and prints all elements in it:

```
for i in range(len(L)):
    print(L[i])
```

A shorter way to iterate through a list is as follows:

```
for x in L:
    print(x)
```

The function pop returns and removes the last element from the list:

```
L = []
L.append(5)
L.append(2)
print(L.pop()) # 2
print(L.pop()) # 5
```

The following code creates a list with five elements:

```
L = [2,4,2,5,1]
```

Another way to create a list is to give the number of elements and the initial value for each element:

```
// size 10, initial value 0
L = [0]*10
```

```
// size 10, initial value 5
L = [5]*10
```

The time of append and pop is $O(1)$.

Linear time operations The following operations all use $O(n)$ time, so use them with care!

You can insert at a given position using the function `insert(i,x)`, which inserts x at postition i. You can remove the first item in the list with value x using `remove(x)`. You can remove the item at index i using `del(i)`. All these operations use $O(n)$ time. Beware, that the command "`x in L`", where `L` is a list, uses $O(n)$ time to search the list for the element! (See next section for an efficient data structure for set membership).

4.2 Set structures

A **set** is a data structure that maintains a collection of elements with no duplicate elements. The basic operations of sets are element insertion, search and removal.

The Python standard library contains only an unordered set implementation: The structure `set` uses hashing, and its operations work in $O(1)$ time on average.

Curly braces or the set() function can be used to create sets. Note: to create an empty set you have to use set(), not {}; the latter creates an empty dictionary, a data structure that we discuss in the next section.

The following code creates a set that contains integers, and shows some of the operations. The function `add` adds an element to the set and the function `remove` removes an element from the set. To check in an element x is in the set s you can write "x in s". All three operations use $O(1)$ time on average.

```
s = set()
s.add(3)
s.add(2)
s.add(10);
print(s) # {10,2,3} (order might be different)
print(4 in s) # False
print(3 in s) # True
s.remove(3)
s.add(4)
print(3 in s) # False
print(4 in s) # True
```

A set can be used mostly like a list, but it is not possible to access the elements using the `[]` notation. But where a list uses $O(n)$ time for the `in` operation a set only uses $O(1)$ time. The following code creates a set, prints the number of elements in it, and then iterates through all the elements:

```
s = {2,5,6,8}
print(len(s)) # 4
for x in s:
    print(x)
```

An important property of sets is that all their elements are *distinct*. Thus, the function `insert` never adds an element to the set if it is already there. The following code illustrates this:

```
s = set()
s.add(5)
s.add(5)
s.add(5)
print(s) # {5}
```

4.3 Map structures/Dictionaries

A **map** or **dictionary** is a data structure that consists of key-value-pairs. Dictionaries are indexed by keys, which can be of many different types. Strings and numbers can be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples of strings and numbers.

You can think of a dictionary as a set of key: value pairs, with the requirement that the keys are unique (within one dictionary).

The Python standard library contains a map implementation that correspond

to the set implementation: the structure `dict` uses hashing and accessing elements takes $O(1)$ time on average.

A pair of braces creates an empty dictionary: {}. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.[1]

The following code creates a dictionary where the keys are strings and the values are integers:

```
d = {}
d["monkey"] = 4
d["banana"] = 3
d["harpsichord"] = 10
print(d["banana"]) # 3
```

You can check if a key k is in the dictionary d by writing "k in d":

```
if ("aybabtu" in d):
    # key exists
```

You can delete a key:value pair with del. If you store using a key that is already in use, the old value associated with that key is forgotten.

You can print all the keys and values in a dictionary d by writing `print(d)`. Writing `list(d)` returns a list of all the keys used in the dictionary d, in insertion order.

4.4 Other structures

Stack

A **stack** is a data structure that provides two $O(1)$ time operations: adding an element to the top, and removing an element from the top. It is only possible to access the top element of a stack.

In Python you can use a list to implement a stack. The following code shows how a stack can be used:

```
s = []
s.append(3)
s.append(2)
s.append(5)
print(s) # [3,2,5]
print(s.pop()) # 5
print(s) # [3,2]
```

[1] from python.org.

Deque

A **deque** is a dynamic array whose size can be efficiently changed at both ends of the array. To implement a deque, use the module `collections.deque` which was designed to have fast appends and pops from both ends. Like a list, a deque provides the functions `append` and `pop()`, but it also includes the functions `appendleft` and `popleft` which are not available in a list. All the operations takes $O(1)$ time.

A deque can be used as follows:

```
from collections import deque
d = deque()
d.append(5) # [5]
d.append(2) # [5,2]
d.appendleft(3) # [3,5,2]
d.pop() # [3,5]
d.popleft() # [5]
```

Queue

A **queue** provides two $O(1)$ time operations: adding an element to the end of the queue, and removing the first element in the queue. It is only possible to access the first and last element of a queue.

The following code shows how a queue can be used:

```
from collections import deque
q = deque()
q.append(3)
q.append(2)
q.append(5)
print(q.popleft()) #3
print(q.popleft()) # 2
```

Priority queue

A **priority queue** maintains a set of elements. The supported operations are insertion and, depending on the type of the queue, removal of either the minimum or maximum element. Insertion and removal take $O(\log n)$ time. A priority queue is usually implemented using a heap structure.

By default, the elements in a Python priority queue are sorted in increasing order, and it is possible to find and remove the smallest element in the queue. The following code illustrates this:

```
import heapq
q = []
heapq.heappush(q,3)
heapq.heappush(q,5)
heapq.heappush(q,7)
heapq.heappush(q,2)
print(heapq.heappop(q)) # 2
print(heapq.heappop(q)) # 3
heapq.heappush(q,1)
print(heapq.heappop()) # 1
```

You can turn an existing list L into a heap in linear time by using `heapq.heapify(L)`.

Chapter 5

Complete search

Complete search is a general method that can be used to solve almost any algorithm problem. The idea is to generate all possible solutions to the problem using brute force, and then select the best solution or count the number of solutions, depending on the problem.

Complete search is a good technique if there is enough time to go through all the solutions, because the search is usually easy to implement and it always gives the correct answer. If complete search is too slow, other techniques, such as greedy algorithms or dynamic programming, may be needed.

5.1 Generating subsets

We first consider the problem of generating all subsets of a set of n elements. For example, the subsets of $\{0,1,2\}$ are $\emptyset$, $\{0\}$, $\{1\}$, $\{2\}$, $\{0,1\}$, $\{0,2\}$, $\{1,2\}$ and $\{0,1,2\}$. There are two common methods to generate subsets: we can either perform a recursive search or exploit the bit representation of integers.

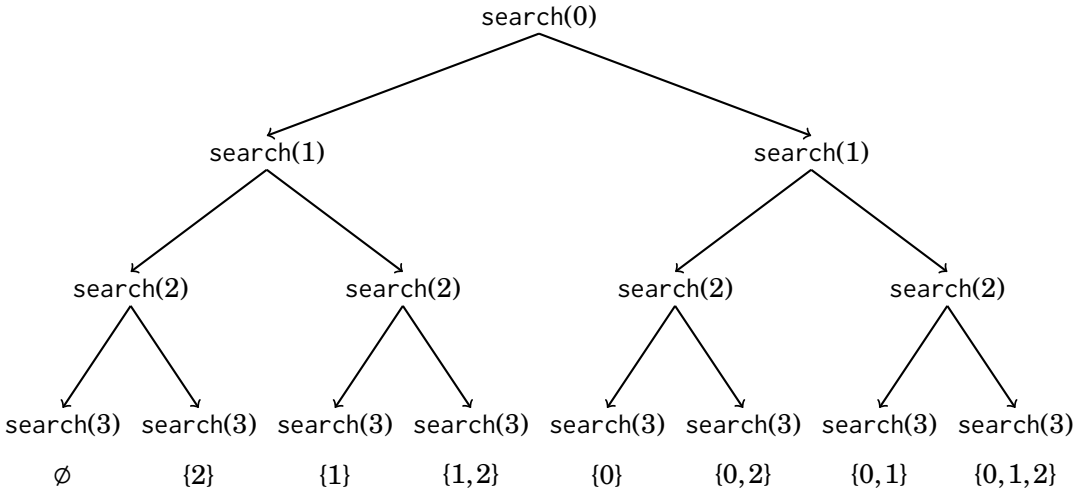
Method 1

An elegant way to go through all subsets of a set is to use recursion. The following function `search` generates the subsets of the set $\{0,1,\ldots,n-1\}$. The function maintains a list `subset` that will contain the elements of each subset. The search begins when the function is called with parameter 0.

```
subset = []
def search(k):
    if (k == n):
        # process subset
    else:
        search(k+1)
        subset.append(k)
        search(k+1)
        subset.pop()
```

When the function search is called with parameter k, it decides whether to include the element k in the subset or not, and in both cases, then calls itself with parameter $k+1$ However, if $k=n$, the function notices that all elements have been processed and a subset has been generated.

The following tree illustrates the function calls when $n=3$. We can always choose either the left branch (k is not included in the subset) or the right branch (k is included in the subset).



Method 2

Another way to generate subsets is based on the bit representation of integers. Each subset of a set of n elements can be represented as a sequence of n bits, which corresponds to an integer between $0 \ldots 2^n - 1$. The ones in the bit sequence indicate which elements are included in the subset.

The usual convention is that the last bit corresponds to element 0, the second last bit corresponds to element 1, and so on. For example, the bit representation of 25 is 11001, which corresponds to the subset $\{0,3,4\}$.

The following code goes through the subsets of a set of n elements

```
for b in range(1<<n):
    # process subset
```

The following code shows how we can find the elements of a subset that corresponds to a bit sequence. When processing each subset, the code builds a list that contains the elements in the subset.

```
for b in range(1<<n):
    subset = []
    for i in range(n):
        if (b&(1<<i)):
            subset.append(i)
    # process subset
```

5.2 Generating permutations

Next we consider the problem of generating all permutations of a set of n elements. For example, the permutations of $\{0,1,2\}$ are $(0,1,2)$, $(0,2,1)$, $(1,0,2)$, $(1,2,0)$, $(2,0,1)$ and $(2,1,0)$. Again, there are two approaches: we can either use recursion or go through the permutations iteratively.

Method 1

Like subsets, permutations can be generated using recursion. The following function `search` goes through the permutations of the set $\{0,1,\ldots,n-1\}$. The function builds a list `permutation` that contains the permutation, and the search begins when the function is called without parameters.

```
permutation = []
chosen = [False]*n
def search():
    if (len(permutation) == n):
        # process permutation
    else:
        for i in range(n):
            if (chosen[i]):
                continue
            chosen[i] = True
            permutation.append(i)
            search()
            chosen[i] = False
            permutation.pop()
```

Each function call adds a new element to `permutation`. The array `chosen` indicates which elements are already included in the permutation. If the size of `permutation` equals the size of the set, a permutation has been generated.

Method 2

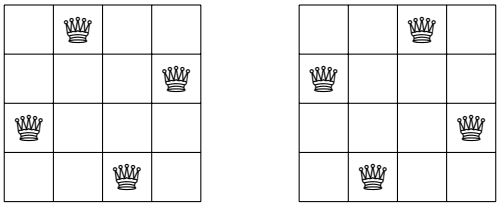
Another method for generating permutations is to begin with the permutation $\{0,1,\ldots,n-1\}$ and repeatedly use a function that constructs the next permutation in increasing order. The Python library `itertools` contains the function `permutation` that can be used for this:

```
from itertools import permutations
L = []
for i in range(n):
    L.append(i)
P = permutations(L)
for p in P:
    # process permutation p
```

5.3 Backtracking

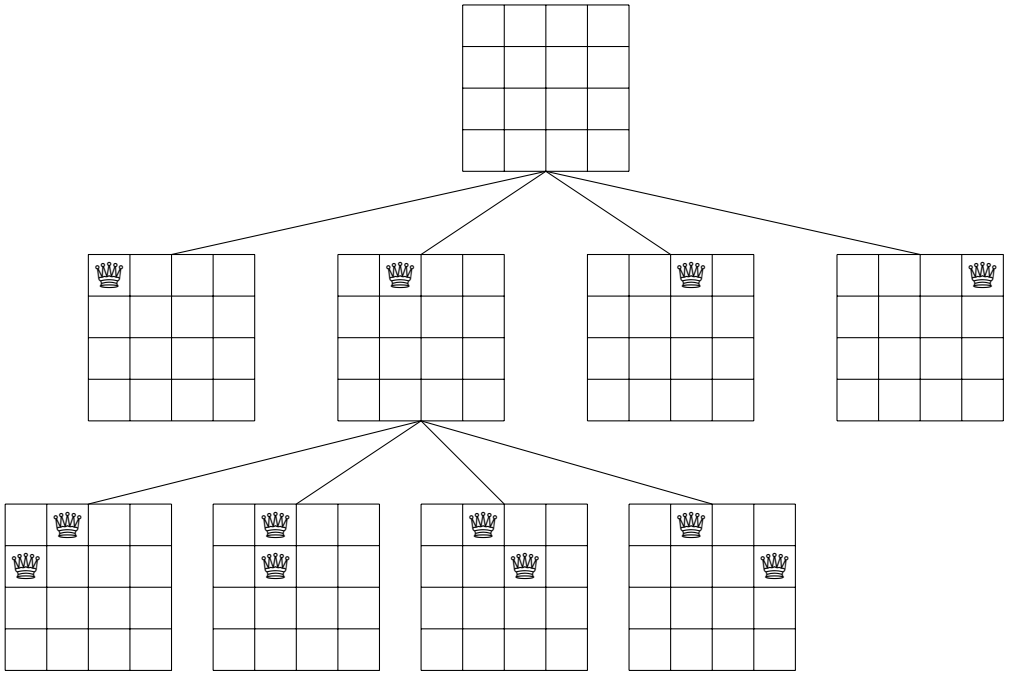
A **backtracking** algorithm begins with an empty solution and extends the solution step by step. The search recursively goes through all different ways how a solution can be constructed.

As an example, consider the problem of calculating the number of ways n queens can be placed on an $n \times n$ chessboard so that no two queens attack each other. For example, when $n = 4$, there are two possible solutions:



The problem can be solved using backtracking by placing queens to the board row by row. More precisely, exactly one queen will be placed on each row so that no queen attacks any of the queens placed before. A solution has been found when all n queens have been placed on the board.

For example, when $n = 4$, some partial solutions generated by the backtracking algorithm are as follows:



At the bottom level, the three first configurations are illegal, because the queens attack each other. However, the fourth configuration is valid and it can be extended to a complete solution by placing two more queens to the board. There is only one way to place the two remaining queens.

The algorithm can be implemented as follows:

```
count = 0
def search(y):
    global count
    if (y == n):
        count += 1
        return
    for x in range(n):
        if (column[x] or diag1[x+y] or diag2[x-y+n-1]):
            continue
        column[x] = diag1[x+y] = diag2[x-y+n-1] = True
        search(y+1)
        column[x] = diag1[x+y] = diag2[x-y+n-1] = False
```

The search begins by calling `search(0)`. The size of the board is $n \times n$, and the code calculates the number of solutions to `count`.

The code assumes that the rows and columns of the board are numbered from 0 to $n-1$. When the function `search` is called with parameter y, it places a queen on row y and then calls itself with parameter $y+1$. Then, if $y = n$, a solution has been found and the variable `count` is increased by one.

The array `column` keeps track of columns that contain a queen, and the arrays `diag1` and `diag2` keep track of diagonals. It is not allowed to add another queen to a column or diagonal that already contains a queen. For example, the columns and diagonals of the 4×4 board are numbered as follows:

0	1	2	3
0	1	2	3
0	1	2	3
0	1	2	3

`column`

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

`diag1`

3	4	5	6
2	3	4	5
1	2	3	4
0	1	2	3

`diag2`

Let $q(n)$ denote the number of ways to place n queens on an $n \times n$ chessboard. The above backtracking algorithm tells us that, for example, $q(8) = 92$. When n increases, the search quickly becomes slow, because the number of solutions increases exponentially. For example, calculating $q(16) = 14772512$ using the above algorithm already takes about a minute on a modern computer[1].

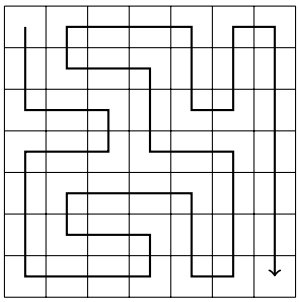
5.4 Pruning the search

We can often optimize backtracking by pruning the search tree. The idea is to add ”intelligence” to the algorithm so that it will notice as soon as possible if a

[1]There is no known way to efficiently calculate larger values of $q(n)$. The current record is $q(27) = 234907967154122528$, calculated in 2016 [28].

partial solution cannot be extended to a complete solution. Such optimizations can have a tremendous effect on the efficiency of the search.

Let us consider the problem of calculating the number of paths in an $n \times n$ grid from the upper-left corner to the lower-right corner such that the path visits each square exactly once. For example, in a 7×7 grid, there are 111712 such paths. One of the paths is as follows:



We focus on the 7×7 case, because its level of difficulty is appropriate to our needs. We begin with a straightforward backtracking algorithm, and then optimize it step by step using observations of how the search can be pruned. After each optimization, we measure the running time of the algorithm and the number of recursive calls, so that we clearly see the effect of each optimization on the efficiency of the search.

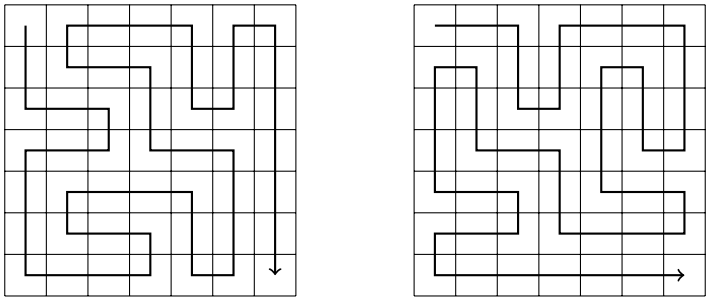
Basic algorithm

The first version of the algorithm does not contain any optimizations. We simply use backtracking to generate all possible paths from the upper-left corner to the lower-right corner and count the number of such paths.

- running time: 483 seconds
- number of recursive calls: 76 billion

Optimization 1

In any solution, we first move one step down or right. There are always two paths that are symmetric about the diagonal of the grid after the first step. For example, the following paths are symmetric:

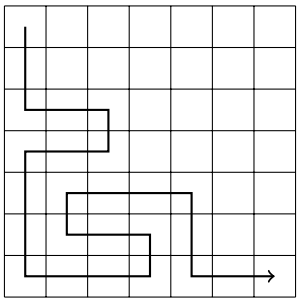


Hence, we can decide that we always first move one step down (or right), and finally multiply the number of solutions by two.

- running time: 244 seconds
- number of recursive calls: 38 billion

Optimization 2

If the path reaches the lower-right square before it has visited all other squares of the grid, it is clear that it will not be possible to complete the solution. An example of this is the following path:

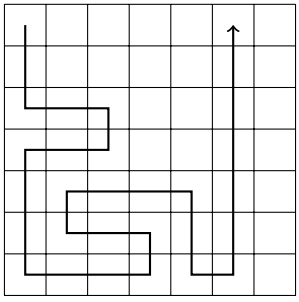


Using this observation, we can terminate the search immediately if we reach the lower-right square too early.

- running time: 119 seconds
- number of recursive calls: 20 billion

Optimization 3

If the path touches a wall and can turn either left or right, the grid splits into two parts that contain unvisited squares. For example, in the following situation, the path can turn either left or right:

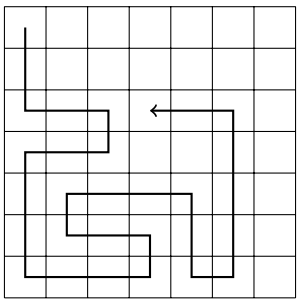


In this case, we cannot visit all squares anymore, so we can terminate the search. This optimization is very useful:

- running time: 1.8 seconds
- number of recursive calls: 221 million

Optimization 4

The idea of Optimization 3 can be generalized: if the path cannot continue forward but can turn either left or right, the grid splits into two parts that both contain unvisited squares. For example, consider the following path:



It is clear that we cannot visit all squares anymore, so we can terminate the search. After this optimization, the search is very efficient:

- running time: 0.6 seconds
- number of recursive calls: 69 million

Now is a good moment to stop optimizing the algorithm and see what we have achieved. The running time of the original algorithm was 483 seconds, and now after the optimizations, the running time is only 0.6 seconds. Thus, the algorithm became nearly 1000 times faster after the optimizations.

This is a usual phenomenon in backtracking, because the search tree is usually large and even simple observations can effectively prune the search. Especially useful are optimizations that occur during the first steps of the algorithm, i.e., at the top of the search tree.

5.5 Meet in the middle

Meet in the middle is a technique where the search space is divided into two parts of about equal size. A separate search is performed for both of the parts, and finally the results of the searches are combined.

The technique can be used if there is an efficient way to combine the results of the searches. In such a situation, the two searches may require less time than one large search. Typically, we can turn a factor of 2^n into a factor of $2^{n/2}$ using the meet in the middle technique.

As an example, consider a problem where we are given a list of n numbers and a number x, and we want to find out if it is possible to choose some numbers from the list so that their sum is x. For example, given the list $[2,4,5,9]$ and $x = 15$, we can choose the numbers $[2,4,9]$ to get $2+4+9 = 15$. However, if $x = 10$ for the same list, it is not possible to form the sum.

A simple algorithm to the problem is to go through all subsets of the elements and check if the sum of any of the subsets is x. The running time of such an

algorithm is $O(2^n)$, because there are 2^n subsets. However, using the meet in the middle technique, we can achieve a more efficient $O(2^{n/2})$ time algorithm[2]. Note that $O(2^n)$ and $O(2^{n/2})$ are different complexities because $2^{n/2}$ equals $\sqrt{2^n}$.

The idea is to divide the list into two lists A and B such that both lists contain about half of the numbers. The first search generates all subsets of A and stores their sums to a list S_A. Correspondingly, the second search creates a list S_B from B. After this, it suffices to check if it is possible to choose one element from S_A and another element from S_B such that their sum is x. This is possible exactly when there is a way to form the sum x using the numbers of the original list.

For example, suppose that the list is $[2,4,5,9]$ and $x = 15$. First, we divide the list into $A = [2,4]$ and $B = [5,9]$. After this, we create lists $S_A = [0,2,4,6]$ and $S_B = [0,5,9,14]$. In this case, the sum $x = 15$ is possible to form, because S_A contains the sum 6, S_B contains the sum 9, and $6+9 = 15$. This corresponds to the solution $[2,4,9]$.

We can implement the algorithm so that its time complexity is $O(2^{n/2})$. First, we generate *sorted* lists S_A and S_B, which can be done in $O(2^{n/2})$ time using a merge-like technique. After this, since the lists are sorted, we can check in $O(2^{n/2})$ time if the sum x can be created from S_A and S_B.

[2]This idea was introduced in 1974 by E. Horowitz and S. Sahni [19].

Chapter 6

Greedy algorithms

A **greedy algorithm** constructs a solution to the problem by always making a choice that looks the best at the moment. A greedy algorithm never takes back its choices, but directly constructs the final solution. For this reason, greedy algorithms are usually very efficient.

The difficulty in designing greedy algorithms is to find a greedy strategy that always produces an optimal solution to the problem. The locally optimal choices in a greedy algorithm should also be globally optimal. It is often difficult to argue that a greedy algorithm works.

6.1 Coin problem

As a first example, we consider a problem where we are given a set of coins and our task is to form a sum of money n using the coins. The values of the coins are $\texttt{coins} = \{c_1, c_2, \ldots, c_k\}$, and each coin can be used as many times we want. What is the minimum number of coins needed?

For example, if the coins are the euro coins (in cents)

$$\{1, 2, 5, 10, 20, 50, 100, 200\}$$

and $n = 520$, we need at least four coins. The optimal solution is to select coins $200 + 200 + 100 + 20$ whose sum is 520.

Greedy algorithm

A simple greedy algorithm to the problem always selects the largest possible coin, until the required sum of money has been constructed. This algorithm works in the example case, because we first select two 200 cent coins, then one 100 cent coin and finally one 20 cent coin. But does this algorithm always work?

It turns out that if the coins are the euro coins, the greedy algorithm *always* works, i.e., it always produces a solution with the fewest possible number of coins. The correctness of the algorithm can be shown as follows:

First, each coin 1, 5, 10, 50 and 100 appears at most once in an optimal solution, because if the solution would contain two such coins, we could replace

them by one coin and obtain a better solution. For example, if the solution would contain coins $5+5$, we could replace them by coin 10.

In the same way, coins 2 and 20 appear at most twice in an optimal solution, because we could replace coins $2+2+2$ by coins $5+1$ and coins $20+20+20$ by coins $50+10$. Moreover, an optimal solution cannot contain coins $2+2+1$ or $20+20+10$, because we could replace them by coins 5 and 50.

Using these observations, we can show for each coin x that it is not possible to optimally construct a sum x or any larger sum by only using coins that are smaller than x. For example, if $x = 100$, the largest optimal sum using the smaller coins is $50+20+20+5+2+2 = 99$. Thus, the greedy algorithm that always selects the largest coin produces the optimal solution.

This example shows that it can be difficult to argue that a greedy algorithm works, even if the algorithm itself is simple.

General case

In the general case, the coin set can contain any coins and the greedy algorithm *does not* necessarily produce an optimal solution.

We can prove that a greedy algorithm does not work by showing a counterexample where the algorithm gives a wrong answer. In this problem we can easily find a counterexample: if the coins are $\{1,3,4\}$ and the target sum is 6, the greedy algorithm produces the solution $4+1+1$ while the optimal solution is $3+3$.

It is not known if the general coin problem can be solved using any greedy algorithm[1]. However, as we will see in Chapter 7, in some cases, the general problem can be efficiently solved using a dynamic programming algorithm that always gives the correct answer.

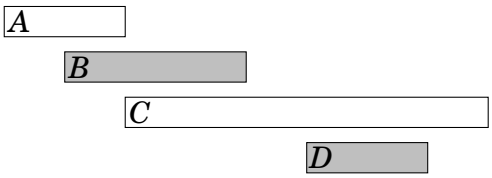
6.2 Scheduling

Many scheduling problems can be solved using greedy algorithms. A classic problem is as follows: Given n events with their starting and ending times, find a schedule that includes as many events as possible. It is not possible to select an event partially. For example, consider the following events:

event	starting time	ending time
A	1	3
B	2	5
C	3	9
D	6	8

In this case the maximum number of events is two. For example, we can select events B and D as follows:

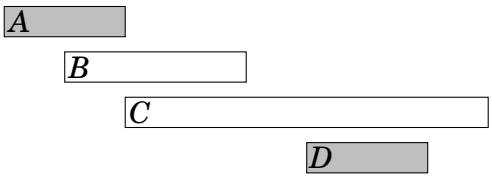
[1]However, it is possible to *check* in polynomial time if the greedy algorithm presented in this chapter works for a given set of coins [26].



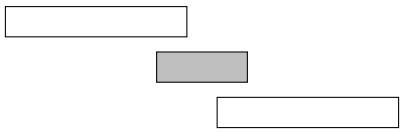
It is possible to invent several greedy algorithms for the problem, but which of them works in every case?

Algorithm 1

The first idea is to select as *short* events as possible. In the example case this algorithm selects the following events:



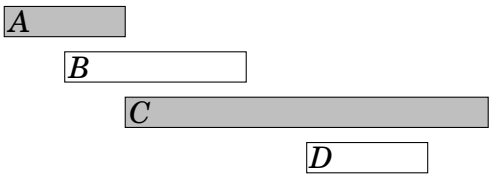
However, selecting short events is not always a correct strategy. For example, the algorithm fails in the following case:



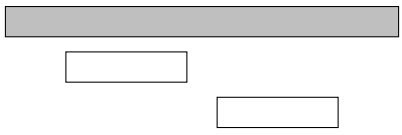
If we select the short event, we can only select one event. However, it would be possible to select both long events.

Algorithm 2

Another idea is to always select the next possible event that *begins* as *early* as possible. This algorithm selects the following events:



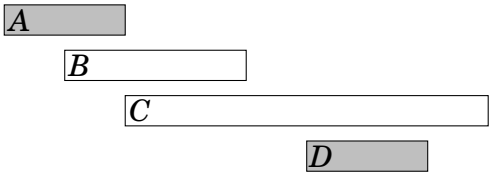
However, we can find a counterexample also for this algorithm. For example, in the following case, the algorithm only selects one event:



If we select the first event, it is not possible to select any other events. However, it would be possible to select the other two events.

Algorithm 3

The third idea is to always select the next possible event that *ends* as *early* as possible. This algorithm selects the following events:



It turns out that this algorithm *always* produces an optimal solution. The reason for this is that it is always an optimal choice to first select an event that ends as early as possible. After this, it is an optimal choice to select the next event using the same strategy, etc., until we cannot select any more events.

One way to argue that the algorithm works is to consider what happens if we first select an event that ends later than the event that ends as early as possible. Now, we will have at most an equal number of choices how we can select the next event. Hence, selecting an event that ends later can never yield a better solution, and the greedy algorithm is correct.

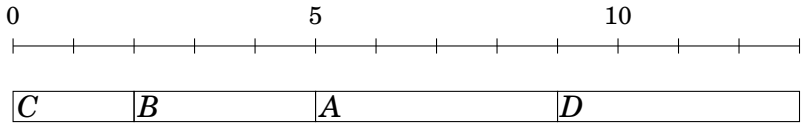
6.3 Tasks and deadlines

Let us now consider a problem where we are given n tasks with durations and deadlines and our task is to choose an order to perform the tasks. For each task, we earn $d - x$ points where d is the task's deadline and x is the moment when we finish the task. What is the largest possible total score we can obtain?

For example, suppose that the tasks are as follows:

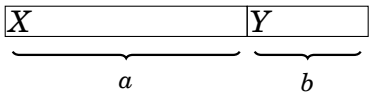
task	duration	deadline
A	4	2
B	3	5
C	2	7
D	4	5

In this case, an optimal schedule for the tasks is as follows:

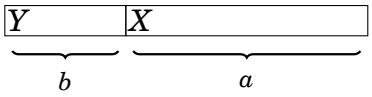


In this solution, C yields 5 points, B yields 0 points, A yields -7 points and D yields -8 points, so the total score is -10.

Surprisingly, the optimal solution to the problem does not depend on the deadlines at all, but a correct greedy strategy is to simply perform the tasks *sorted by their durations* in increasing order. The reason for this is that if we ever perform two tasks one after another such that the first task takes longer than the second task, we can obtain a better solution if we swap the tasks. For example, consider the following schedule:



Here $a > b$, so we should swap the tasks:



Now X gives b points less and Y gives a points more, so the total score increases by $a-b>0$. In an optimal solution, for any two consecutive tasks, it must hold that the shorter task comes before the longer task. Thus, the tasks must be performed sorted by their durations.

6.4 Minimizing sums

We next consider a problem where we are given n numbers $a_1, a_2, \ldots, a_n$ and our task is to find a value x that minimizes the sum

$$|a_1 - x|^c + |a_2 - x|^c + \cdots + |a_n - x|^c.$$

We focus on the cases $c = 1$ and $c = 2$.

Case $c = 1$

In this case, we should minimize the sum

$$|a_1 - x| + |a_2 - x| + \cdots + |a_n - x|.$$

For example, if the numbers are $[1,2,9,2,6]$, the best solution is to select $x = 2$ which produces the sum

$$|1-2| + |2-2| + |9-2| + |2-2| + |6-2| = 12.$$

In the general case, the best choice for x is the *median* of the numbers, i.e., the middle number after sorting. For example, the list $[1,2,9,2,6]$ becomes $[1,2,2,6,9]$ after sorting, so the median is 2.

The median is an optimal choice, because if x is smaller than the median, the sum becomes smaller by increasing x, and if x is larger then the median, the sum becomes smaller by decreasing x. Hence, the optimal solution is that x is the median. If n is even and there are two medians, both medians and all values between them are optimal choices.

Case $c = 2$

In this case, we should minimize the sum

$$(a_1 - x)^2 + (a_2 - x)^2 + \cdots + (a_n - x)^2.$$

For example, if the numbers are $[1,2,9,2,6]$, the best solution is to select $x = 4$ which produces the sum

$$(1-4)^2 + (2-4)^2 + (9-4)^2 + (2-4)^2 + (6-4)^2 = 46.$$

In the general case, the best choice for x is the *average* of the numbers. In the example the average is $(1+2+9+2+6)/5 = 4$. This result can be derived by presenting the sum as follows:

$$nx^2 - 2x(a_1 + a_2 + \cdots + a_n) + (a_1^2 + a_2^2 + \cdots + a_n^2)$$

The last part does not depend on x, so we can ignore it. The remaining parts form a function $nx^2 - 2xs$ where $s = a_1 + a_2 + \cdots + a_n$. This is a parabola opening upwards with roots $x = 0$ and $x = 2s/n$, and the minimum value is the average of the roots $x = s/n$, i.e., the average of the numbers $a_1, a_2, \ldots, a_n$.

6.5 Data compression

A **binary code** assigns for each character of a string a **codeword** that consists of bits. We can *compress* the string using the binary code by replacing each character by the corresponding codeword. For example, the following binary code assigns codewords for characters `A`–`D`:

character	codeword
A	00
B	01
C	10
D	11

This is a **constant-length** code which means that the length of each codeword is the same. For example, we can compress the string `AABACDACA` as follows:

00 00 01 00 10 11 00 10 00

Using this code, the length of the compressed string is 18 bits. However, we can compress the string better if we use a **variable-length** code where codewords may have different lengths. Then we can give short codewords for characters that appear often and long codewords for characters that appear rarely. It turns out that an **optimal** code for the above string is as follows:

character	codeword
A	0
B	110
C	10
D	111

An optimal code produces a compressed string that is as short as possible. In this case, the compressed string using the optimal code is

0 0 110 0 10 111 0 10 0,

so only 15 bits are needed instead of 18 bits. Thus, thanks to a better code it was possible to save 3 bits in the compressed string.

We require that no codeword is a prefix of another codeword. For example, it is not allowed that a code would contain both codewords 10 and 1011. The reason for this is that we want to be able to generate the original string from the compressed string. If a codeword could be a prefix of another codeword, this would not always be possible. For example, the following code is *not* valid:

character	codeword
A	10
B	11
C	1011
D	111

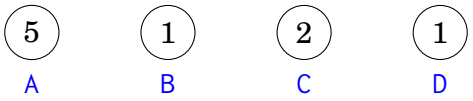
Using this code, it would not be possible to know if the compressed string 1011 corresponds to the string AB or the string C.

Huffman coding

Huffman coding[2] is a greedy algorithm that constructs an optimal code for compressing a given string. The algorithm builds a binary tree based on the frequencies of the characters in the string, and each character's codeword can be read by following a path from the root to the corresponding node. A move to the left corresponds to bit 0, and a move to the right corresponds to bit 1.

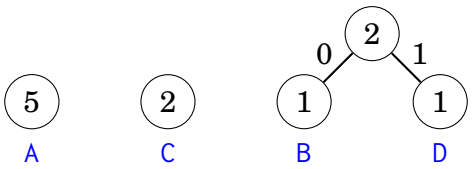
Initially, each character of the string is represented by a node whose weight is the number of times the character occurs in the string. Then at each step two nodes with minimum weights are combined by creating a new node whose weight is the sum of the weights of the original nodes. The process continues until all nodes have been combined.

Next we will see how Huffman coding creates the optimal code for the string AABACDACA. Initially, there are four nodes that correspond to the characters of the string:



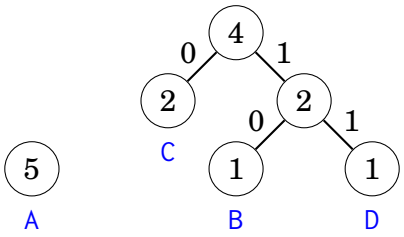
The node that represents character A has weight 5 because character A appears 5 times in the string. The other weights have been calculated in the same way.

The first step is to combine the nodes that correspond to characters B and D, both with weight 1. The result is:

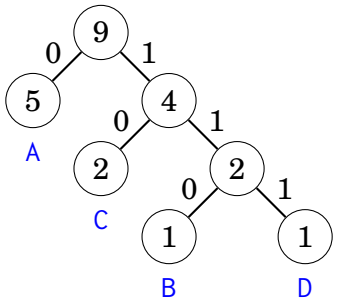


[2] D. A. Huffman discovered this method when solving a university course assignment and published the algorithm in 1952 [20].

After this, the nodes with weight 2 are combined:



Finally, the two remaining nodes are combined:



Now all nodes are in the tree, so the code is ready. The following codewords can be read from the tree:

character	codeword
A	0
B	110
C	10
D	111

Chapter 7

Dynamic programming

Dynamic programming is a technique that combines the correctness of complete search and the efficiency of greedy algorithms. Dynamic programming can be applied if the problem can be divided into overlapping subproblems that can be solved independently.

There are two uses for dynamic programming:

- **Finding an optimal solution**: We want to find a solution that is as large as possible or as small as possible.
- **Counting the number of solutions**: We want to calculate the total number of possible solutions.

We will first see how dynamic programming can be used to find an optimal solution, and then we will use the same idea for counting the solutions.

Understanding dynamic programming is a milestone in every competitive programmer's career. While the basic idea is simple, the challenge is how to apply dynamic programming to different problems. This chapter introduces a set of classic problems that are a good starting point.

7.1 Coin problem

We first focus on a problem that we have already seen in Chapter 6: Given a set of coin values $\texttt{coins} = \{c_1, c_2, \ldots, c_k\}$ and a target sum of money n, our task is to form the sum n using as few coins as possible.

In Chapter 6, we solved the problem using a greedy algorithm that always chooses the largest possible coin. The greedy algorithm works, for example, when the coins are the euro coins, but in the general case the greedy algorithm does not necessarily produce an optimal solution.

Now is time to solve the problem efficiently using dynamic programming, so that the algorithm works for any coin set. The dynamic programming algorithm is based on a recursive function that goes through all possibilities how to form the sum, like a brute force algorithm. However, the dynamic programming algorithm is efficient because it uses *memoization* and calculates the answer to each subproblem only once.

Recursive formulation

The idea in dynamic programming is to formulate the problem recursively so that the solution to the problem can be calculated from solutions to smaller subproblems. In the coin problem, a natural recursive problem is as follows: what is the smallest number of coins required to form a sum x?

Let $\texttt{solve}(x)$ denote the minimum number of coins required for a sum x. The values of the function depend on the values of the coins. For example, if $\texttt{coins} = \{1,3,4\}$, the first values of the function are as follows:

$$\begin{aligned}
\texttt{solve}(0) &= 0 \\
\texttt{solve}(1) &= 1 \\
\texttt{solve}(2) &= 2 \\
\texttt{solve}(3) &= 1 \\
\texttt{solve}(4) &= 1 \\
\texttt{solve}(5) &= 2 \\
\texttt{solve}(6) &= 2 \\
\texttt{solve}(7) &= 2 \\
\texttt{solve}(8) &= 2 \\
\texttt{solve}(9) &= 3 \\
\texttt{solve}(10) &= 3
\end{aligned}$$

For example, $\texttt{solve}(10) = 3$, because at least 3 coins are needed to form the sum 10. The optimal solution is $3+3+4=10$.

The essential property of solve is that its values can be recursively calculated from its smaller values. The idea is to focus on the *first* coin that we choose for the sum. For example, in the above scenario, the first coin can be either 1, 3 or 4. If we first choose coin 1, the remaining task is to form the sum 9 using the minimum number of coins, which is a subproblem of the original problem. Of course, the same applies to coins 3 and 4. Thus, we can use the following recursive formula to calculate the minimum number of coins:

$$\begin{aligned}
\texttt{solve}(x) = \min(&\texttt{solve}(x-1)+1, \\
&\texttt{solve}(x-3)+1, \\
&\texttt{solve}(x-4)+1).
\end{aligned}$$

The base case of the recursion is $\texttt{solve}(0) = 0$, because no coins are needed to form an empty sum. For example,

$$\texttt{solve}(10) = \texttt{solve}(7)+1 = \texttt{solve}(4)+2 = \texttt{solve}(0)+3 = 3.$$

Now we are ready to give a general recursive function that calculates the minimum number of coins needed to form a sum x:

$$\texttt{solve}(x) = \begin{cases}
\infty & x < 0 \\
0 & x = 0 \\
\min_{c \in \texttt{coins}} \texttt{solve}(x-c)+1 & x > 0
\end{cases}$$

First, if $x < 0$, the value is ∞, because it is impossible to form a negative sum of money. Then, if $x = 0$, the value is 0, because no coins are needed to form an

empty sum. Finally, if $x > 0$, the variable c goes through all possibilities how to choose the first coin of the sum.

Once a recursive function that solves the problem has been found, we can directly implement a solution in C++ (the constant `INF` denotes infinity):

```
def solve(x):
    if x < 0:
        return INF
    if x == 0:
        return 0
    best = INF
    for c in coins:
        best = min(best, solve(x-c)+1)
    return best
```

Still, this function is not efficient, because there may be an exponential number of ways to construct the sum. However, next we will see how to make the function efficient using a technique called memoization.

Using memoization

The idea of dynamic programming is to use **memoization** to efficiently calculate values of a recursive function. This means that the values of the function are stored in an array after calculating them. For each parameter, the value of the function is calculated recursively only once, and after this, the value can be directly retrieved from the array.

In this problem, we use lists

```
ready = [False]*N
value = [0]*N
```

where `ready`[x] indicates whether the value of `solve`(x) has been calculated, and if it is, `value`[x] contains this value. The constant N has been chosen so that all required values fit in the arrays.

Now the function can be efficiently implemented as follows:

```
def solve(x):
    if x < 0:
        return INF
    if x == 0:
        return 0
    if ready[x]:
        return value[x]
    best = INF
    for c in coins:
        best = min(best, solve(x-c)+1)
    value[x] = best
    ready[x] = true
    return best
```

The function handles the base cases $x < 0$ and $x = 0$ as previously. Then the function checks from `ready`[x] if `solve`(x) has already been stored in `value`[x], and if it is, the function directly returns it. Otherwise the function calculates the value of `solve`(x) recursively and stores it in `value`[x].

This function works efficiently, because the answer for each parameter x is calculated recursively only once. After a value of `solve`(x) has been stored in `value`[x], it can be efficiently retrieved whenever the function will be called again with the parameter x. The time complexity of the algorithm is $O(nk)$, where n is the target sum and k is the number of coins.

When you use recursion in Python you might have to increase the recursion limit by adding the following in the beginning of your code:

```
import sys
sys.setrecursionlimit(10**6)
```

This sets the recursion limit to 10^6. You might need another number than 10^6.

Note that we can also *iteratively* construct the array `value` of length $N+1$ using a loop that simply calculates all the values of `solve` for parameters $0 \ldots x$:

```
value[0] = 0
for x in range (1,n+1):
    value[x] = INF
    for c in coins:
        if (x >= c):
            value[x] = min(value[x], value[x-c]+1)
```

In fact, most competitive programmers prefer this implementation, because it is shorter and has lower constant factors. From now on, we also use iterative implementations in our examples. Still, it is often easier to think about dynamic programming solutions in terms of recursive functions.

Constructing a solution

Sometimes we are asked both to find the value of an optimal solution and to give an example how such a solution can be constructed. In the coin problem, for example, we can construct another list that indicates for each sum of money the first coin in an optimal solution:

```
first = [0]*(N+1)
```

Then, we can modify the algorithm as follows:

```
value[0] = 0
for x in range (1,n+1):
    value[x] = INF
    for c in coins:
        if (x >= c and value[x-c]+1 < value[x])
            value[x] = value[x-c]+1
            first[x] = c
```

After this, the following code can be used to print the coins that appear in an optimal solution for the sum n:

```
while (n > 0):
    print(first[n])
    n -= first[n]
```

Counting the number of solutions

Let us now consider another version of the coin problem where our task is to calculate the total number of ways to produce a sum x using the coins. For example, if $\texttt{coins} = \{1,3,4\}$ and $x = 5$, there are a total of 6 ways:

- $1+1+1+1+1$
- $1+1+3$
- $1+3+1$
- $3+1+1$
- $1+4$
- $4+1$

Again, we can solve the problem recursively. Let $\texttt{solve}(x)$ denote the number of ways we can form the sum x. For example, if $\texttt{coins} = \{1,3,4\}$, then $\texttt{solve}(5) = 6$ and the recursive formula is

$$\begin{aligned}\texttt{solve}(x) = &\texttt{solve}(x-1)+\\ &\texttt{solve}(x-3)+\\ &\texttt{solve}(x-4).\end{aligned}$$

Then, the general recursive function is as follows:

$$\texttt{solve}(x) = \begin{cases} 0 & x < 0 \\ 1 & x = 0 \\ \sum_{c \in \texttt{coins}} \texttt{solve}(x-c) & x > 0 \end{cases}$$

If $x < 0$, the value is 0, because there are no solutions. If $x = 0$, the value is 1, because there is only one way to form an empty sum. Otherwise we calculate the sum of all values of the form $\texttt{solve}(x-c)$ where c is in coins.

The following code constructs a list count such that count[x] equals the value of $\texttt{solve}(x)$ for $0 \le x \le n$:

```
count[0] = 1
for x in range(n):
    for c in coins:
        if (x-c >= 0):
            count[x] += count[x-c]
```

Often the number of solutions is so large that it is not required to calculate the exact number but it is enough to give the answer modulo m where, for example, $m = 10^9 + 7$. This can be done by changing the code so that all calculations are done modulo m. In the above code, it suffices to add the line

```
        count[x] %= m
```

after the line

```
        count[x] += count[x-c]
```

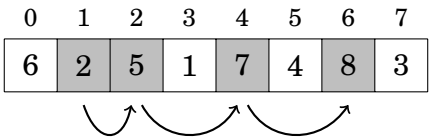
Now we have discussed all basic ideas of dynamic programming. Since dynamic programming can be used in many different situations, we will now go through a set of problems that show further examples about the possibilities of dynamic programming.

7.2 Longest increasing subsequence

Our first problem is to find the **longest increasing subsequence** in an array of n elements. This is a maximum-length sequence of array elements that goes from left to right, and each element in the sequence is larger than the previous element. For example, in the array

0	1	2	3	4	5	6	7
6	2	5	1	7	4	8	3

the longest increasing subsequence contains 4 elements:



Let length(k) denote the length of the longest increasing subsequence that ends at position k. Thus, if we calculate all values of length(k) where $0 \le k \le n-1$, we will find out the length of the longest increasing subsequence. For example, the values of the function for the above array are as follows:

$$\begin{aligned}
\texttt{length}(0) &= 1 \\
\texttt{length}(1) &= 1 \\
\texttt{length}(2) &= 2 \\
\texttt{length}(3) &= 1 \\
\texttt{length}(4) &= 3 \\
\texttt{length}(5) &= 2 \\
\texttt{length}(6) &= 4 \\
\texttt{length}(7) &= 2
\end{aligned}$$

For example, length(6) = 4, because the longest increasing subsequence that ends at position 6 consists of 4 elements.

To calculate a value of length(k), we should find a position $i < k$ for which array[i] < array[k] and length(i) is as large as possible. Then we know that length(k) = length(i) + 1, because this is an optimal way to add array[k] to a subsequence. However, if there is no such position i, then length(k) = 1, which means that the subsequence only contains array[k].

Since all values of the function can be calculated from its smaller values, we can use dynamic programming. In the following code, the values of the function will be stored in a list length.

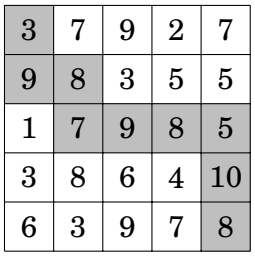
```
for k in range (n):
    length[k] = 1
    for i in range(k):
        if (array[i] < array[k]):
            length[k] = max(length[k],length[i]+1)
```

This code works in $O(n^2)$ time, because it consists of two nested loops. However, it is also possible to implement the dynamic programming calculation more efficiently in $O(n \log n)$ time. Can you find a way to do this?

7.3 Paths in a grid

Our next problem is to find a path from the upper-left corner to the lower-right corner of an $n \times n$ grid, such that we only move down and right. Each square contains a positive integer, and the path should be constructed so that the sum of the values along the path is as large as possible.

The following picture shows an optimal path in a grid:



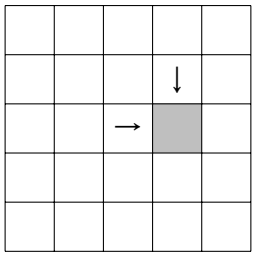
The sum of the values on the path is 67, and this is the largest possible sum on a path from the upper-left corner to the lower-right corner.

Assume that the rows and columns of the grid are numbered from 1 to n, and $\texttt{value}[y][x]$ equals the value of square (y,x). Let $\texttt{sum}(y,x)$ denote the maximum sum on a path from the upper-left corner to square (y,x). Now $\texttt{sum}(n,n)$ tells us the maximum sum from the upper-left corner to the lower-right corner. For example, in the above grid, $\texttt{sum}(5,5) = 67$.

We can recursively calculate the sums as follows:

$$\texttt{sum}(y,x) = \max(\texttt{sum}(y,x-1), \texttt{sum}(y-1,x)) + \texttt{value}[y][x]$$

The recursive formula is based on the observation that a path that ends at square (y,x) can come either from square $(y,x-1)$ or square $(y-1,x)$:



Thus, we select the direction that maximizes the sum. We assume that $\texttt{sum}(y,x) = 0$ if $y = 0$ or $x = 0$ (because no such paths exist), so the recursive formula also works when $y = 1$ or $x = 1$.

Since the function sum has two parameters, the dynamic programming list also has two dimensions. For example, we can use a list

```
sum = [[0 for i in range(N)] for j in range(N)]
```

and calculate the sums as follows:

```
for y in range(1,n+1):
    for x in range(1,n+1):
        sum[y][x] = max(sum[y][x-1],sum[y-1][x]) + value[y][x]
```

The time complexity of the algorithm is $O(n^2)$.

7.4 Knapsack problems

The term **knapsack** refers to problems where a set of objects is given, and subsets with some properties have to be found. Knapsack problems can often be solved

using dynamic programming.

In this section, we focus on the following problem: Given a list of weights $[w_1, w_2, \ldots, w_n]$, determine all sums that can be constructed using the weights. For example, if the weights are $[1,3,3,5]$, the following sums are possible:

0	1	2	3	4	5	6	7	8	9	10	11	12
X	X		X	X	X	X	X	X	X		X	X

In this case, all sums between $0 \ldots 12$ are possible, except 2 and 10. For example, the sum 7 is possible because we can select the weights $[1,3,3]$.

To solve the problem, we focus on subproblems where we only use the first k weights to construct sums. Let $\texttt{possible}(x,k) = \text{true}$ if we can construct a sum x using the first k weights, and otherwise $\texttt{possible}(x,k) = \text{false}$. The values of the function can be recursively calculated as follows:

$$\texttt{possible}(x,k) = \texttt{possible}(x-w_k, k-1) \vee \texttt{possible}(x, k-1)$$

The formula is based on the fact that we can either use or not use the weight w_k in the sum. If we use w_k, the remaining task is to form the sum $x - w_k$ using the first $k-1$ weights, and if we do not use w_k, the remaining task is to form the sum x using the first $k-1$ weights. As the base cases,

$$\texttt{possible}(x,0) = \begin{cases} \text{true} & x = 0 \\ \text{false} & x \neq 0 \end{cases}$$

because if no weights are used, we can only form the sum 0.

The following table shows all values of the function for the weights $[1,3,3,5]$ (the symbol "X" indicates the true values):

$k \backslash x$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	X												
1	X	X											
2	X	X		X	X								
3	X	X		X	X		X	X					
4	X	X		X	X	X	X	X	X	X		X	X

After calculating those values, $\texttt{possible}(x,n)$ tells us whether we can construct a sum x using *all* weights.

Let W denote the total sum of the weights. The following $O(nW)$ time dynamic programming solution corresponds to the recursive function:

```
possible[0][0] = True
for k in range(1,n):
    for x in range(W+1):
        if (x-w[k] >= 0):
            possible[x][k] |= possible[x-w[k]][k-1]
        possible[x][k] |= possible[x][k-1]
```

However, here is a better implementation that only uses a one-dimensional array possible[x] that indicates whether we can construct a subset with sum x. The trick is to update the list from right to left for each new weight:

```
possible[0] = True
for k in range(1,n+1):
    for x in range(W, -1, -1):
        if (possible[x]):
            possible[x+w[k]] = True
```

Note that the general idea presented here can be used in many knapsack problems. For example, if we are given objects with weights and values, we can determine for each weight sum the maximum value sum of a subset.

7.5 Edit distance

The **edit distance** or **Levenshtein distance**[1] is the minimum number of editing operations needed to transform a string into another string. The allowed editing operations are as follows:

- insert a character (e.g. ABC → ABCA)
- remove a character (e.g. ABC → AC)
- modify a character (e.g. ABC → ADC)

For example, the edit distance between LOVE and MOVIE is 2, because we can first perform the operation LOVE → MOVE (modify) and then the operation MOVE → MOVIE (insert). This is the smallest possible number of operations, because it is clear that only one operation is not enough.

Suppose that we are given a string x of length n and a string y of length m, and we want to calculate the edit distance between x and y. To solve the problem, we define a function distance(a,b) that gives the edit distance between prefixes x[$0 \ldots a$] and y[$0 \ldots b$]. Thus, using this function, the edit distance between x and y equals distance($n-1, m-1$).

We can calculate values of distance as follows:

$$\begin{aligned}\texttt{distance}(a,b) = \min(&\texttt{distance}(a,b-1)+1,\\ &\texttt{distance}(a-1,b)+1,\\ &\texttt{distance}(a-1,b-1)+\texttt{cost}(a,b)).\end{aligned}$$

Here cost(a,b) = 0 if x[a] = y[b], and otherwise cost(a,b) = 1. The formula considers the following ways to edit the string x:

- distance($a,b-1$): insert a character at the end of x

[1] The distance is named after V. I. Levenshtein who studied it in connection with binary codes [25].

- distance($a-1, b$): remove the last character from x
- distance($a-1, b-1$): match or modify the last character of x

In the two first cases, one editing operation is needed (insert or remove). In the last case, if x[a] = y[b], we can match the last characters without editing, and otherwise one editing operation is needed (modify).

The following table shows the values of distance in the example case:

		M	O	V	I	E
	0	1	2	3	4	5
L	1	1	2	3	4	5
O	2	2	1	2	3	4
V	3	3	2	1	2	3
E	4	4	3	2	2	2

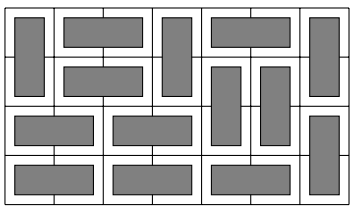
The lower-right corner of the table tells us that the edit distance between LOVE and MOVIE is 2. The table also shows how to construct the shortest sequence of editing operations. In this case the path is as follows:

		M	O	V	I	E
	0	1	2	3	4	5
L	1	1	2	3	4	5
O	2	2	1	2	3	4
V	3	3	2	1	2	3
E	4	4	3	2	2	2

The last characters of LOVE and MOVIE are equal, so the edit distance between them equals the edit distance between LOV and MOVI. We can use one editing operation to remove the character I from MOVI. Thus, the edit distance is one larger than the edit distance between LOV and MOV, etc.

7.6 Counting tilings

Sometimes the states of a dynamic programming solution are more complex than fixed combinations of numbers. As an example, consider the problem of calculating the number of distinct ways to fill an $n \times m$ grid using 1×2 and 2×1 size tiles. For example, one valid solution for the 4×7 grid is



and the total number of solutions is 781.

The problem can be solved using dynamic programming by going through the grid row by row. Each row in a solution can be represented as a string that contains m characters from the set $\{\sqcap, \sqcup, \sqsubset, \sqsupset\}$. For example, the above solution consists of four rows that correspond to the following strings:

- $\sqcap\sqsubset\sqsupset\sqcap\sqsubset\sqsupset\sqcap$
- $\sqcup\sqsubset\sqsupset\sqcup\sqcap\sqcap\sqcup$
- $\sqsubset\sqsupset\sqsubset\sqsupset\sqcup\sqcup\sqcap$
- $\sqsubset\sqsupset\sqsubset\sqsupset\sqsubset\sqsupset\sqcup$

Let $\mathrm{count}(k, x)$ denote the number of ways to construct a solution for rows $1 \ldots k$ of the grid such that string x corresponds to row k. It is possible to use dynamic programming here, because the state of a row is constrained only by the state of the previous row.

A solution is valid if row 1 does not contain the character $\sqcup$, row n does not contain the character $\sqcap$, and all consecutive rows are *compatible*. For example, the rows $\sqcup\sqsubset\sqsupset\sqcup\sqcap\sqcap\sqcup$ and $\sqsubset\sqsupset\sqsubset\sqsupset\sqcup\sqcup\sqcap$ are compatible, while the rows $\sqcap\sqsubset\sqsupset\sqcap\sqsubset\sqsupset\sqcap$ and $\sqsubset\sqsupset\sqsubset\sqsupset\sqsubset\sqsupset\sqcup$ are not compatible.

Since a row consists of m characters and there are four choices for each character, the number of distinct rows is at most 4^m. Thus, the time complexity of the solution is $O(n4^{2m})$ because we can go through the $O(4^m)$ possible states for each row, and for each state, there are $O(4^m)$ possible states for the previous row. In practice, it is a good idea to rotate the grid so that the shorter side has length m, because the factor 4^{2m} dominates the time complexity.

It is possible to make the solution more efficient by using a more compact representation for the rows. It turns out that it is sufficient to know which columns of the previous row contain the upper square of a vertical tile. Thus, we can represent a row using only characters $\sqcap$ and $\square$, where $\square$ is a combination of characters $\sqcup$, $\sqsubset$ and $\sqsupset$. Using this representation, there are only 2^m distinct rows and the time complexity is $O(n2^{2m})$.

As a final note, there is also a surprising direct formula for calculating the number of tilings[2]:

$$\prod_{a=1}^{\lceil n/2 \rceil} \prod_{b=1}^{\lceil m/2 \rceil} 4 \cdot (\cos^2 \frac{\pi a}{n+1} + \cos^2 \frac{\pi b}{m+1})$$

This formula is very efficient, because it calculates the number of tilings in $O(nm)$ time, but since the answer is a product of real numbers, a problem when using the formula is how to store the intermediate results accurately.

[2]Surprisingly, this formula was discovered in 1961 by two research teams [21, 34] that worked independently.

Chapter 8

Amortized analysis

The time complexity of an algorithm is often easy to analyze just by examining the structure of the algorithm: what loops does the algorithm contain and how many times the loops are performed. However, sometimes a straightforward analysis does not give a true picture of the efficiency of the algorithm.

Amortized analysis can be used to analyze algorithms that contain operations whose time complexity varies. The idea is to estimate the total time used to all such operations during the execution of the algorithm, instead of focusing on individual operations.

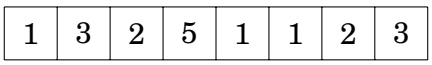
8.1 Two pointers method

In the **two pointers method**, two pointers are used to iterate through the array values. Both pointers can move to one direction only, which ensures that the algorithm works efficiently. Next we discuss two problems that can be solved using the two pointers method.

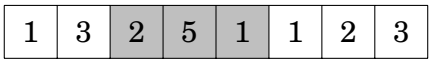
Subarray sum

As the first example, consider a problem where we are given an array of n positive integers and a target sum x, and we want to find a subarray whose sum is x or report that there is no such subarray.

For example, the array

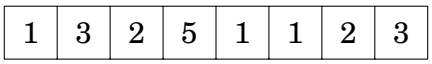


contains a subarray whose sum is 8:

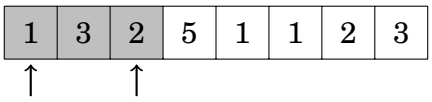


This problem can be solved in $O(n)$ time by using the two pointers method. The idea is to maintain pointers that point to the first and last value of a subarray. On each turn, the left pointer moves one step to the right, and the right pointer moves to the right as long as the resulting subarray sum is at most x. If the sum becomes exactly x, a solution has been found.

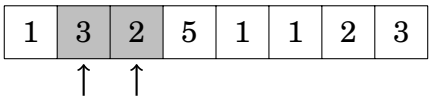
As an example, consider the following array and a target sum $x = 8$:



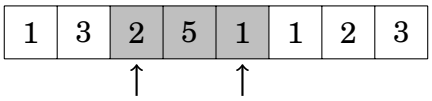
The initial subarray contains the values 1, 3 and 2 whose sum is 6:



Then, the left pointer moves one step to the right. The right pointer does not move, because otherwise the subarray sum would exceed x.



Again, the left pointer moves one step to the right, and this time the right pointer moves three steps to the right. The subarray sum is $2 + 5 + 1 = 8$, so a subarray whose sum is x has been found.



The running time of the algorithm depends on the number of steps the right pointer moves. While there is no useful upper bound on how many steps the pointer can move on a *single* turn. we know that the pointer moves *a total of* $O(n)$ steps during the algorithm, because it only moves to the right.

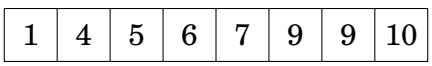
Since both the left and right pointer move $O(n)$ steps during the algorithm, the algorithm works in $O(n)$ time.

2SUM problem

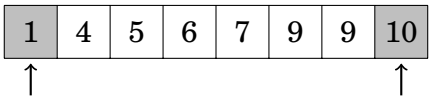
Another problem that can be solved using the two pointers method is the following problem, also known as the **2SUM problem**: given an array of n numbers and a target sum x, find two array values such that their sum is x, or report that no such values exist.

To solve the problem, we first sort the array values in increasing order. After that, we iterate through the array using two pointers. The left pointer starts at the first value and moves one step to the right on each turn. The right pointer begins at the last value and always moves to the left until the sum of the left and right value is at most x. If the sum is exactly x, a solution has been found.

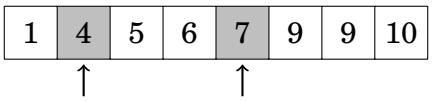
For example, consider the following array and a target sum $x = 12$:



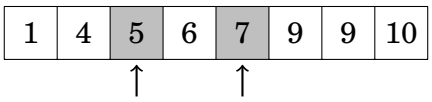
The initial positions of the pointers are as follows. The sum of the values is $1 + 10 = 11$ that is smaller than x.



Then the left pointer moves one step to the right. The right pointer moves three steps to the left, and the sum becomes $4 + 7 = 11$.



After this, the left pointer moves one step to the right again. The right pointer does not move, and a solution $5 + 7 = 12$ has been found.



The running time of the algorithm is $O(n \log n)$, because it first sorts the array in $O(n \log n)$ time, and then both pointers move $O(n)$ steps.

Note that it is possible to solve the problem in another way in $O(n \log n)$ time using binary search. In such a solution, we iterate through the array and for each array value, we try to find another value that yields the sum x. This can be done by performing n binary searches, each of which takes $O(\log n)$ time.

A more difficult problem is the **3SUM problem** that asks to find *three* array values whose sum is x. Using the idea of the above algorithm, this problem can be solved in $O(n^2)$ time[1]. Can you see how?

8.2 Nearest smaller elements

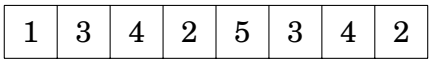
Amortized analysis is often used to estimate the number of operations performed on a data structure. The operations may be distributed unevenly so that most operations occur during a certain phase of the algorithm, but the total number of the operations is limited.

As an example, consider the problem of finding for each array element the **nearest smaller element**, i.e., the first smaller element that precedes the element in the array. It is possible that no such element exists, in which case the algorithm should report this. Next we will see how the problem can be efficiently solved using a stack structure.

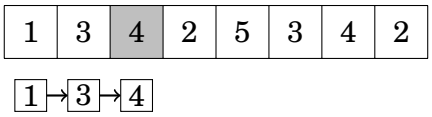
We go through the array from left to right and maintain a stack of array elements. At each array position, we remove elements from the stack until the top element is smaller than the current element, or the stack is empty. Then, we report that the top element is the nearest smaller element of the current element, or if the stack is empty, there is no such element. Finally, we add the current element to the stack.

As an example, consider the following array:

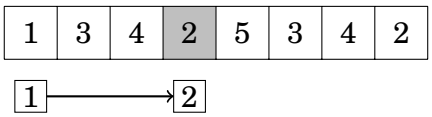
[1] For a long time, it was thought that solving the 3SUM problem more efficiently than in $O(n^2)$ time would not be possible. However, in 2014, it turned out [17] that this is not the case.



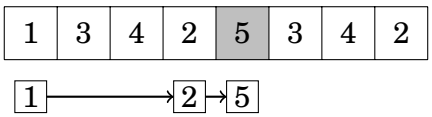
First, the elements 1, 3 and 4 are added to the stack, because each element is larger than the previous element. Thus, the nearest smaller element of 4 is 3, and the nearest smaller element of 3 is 1.



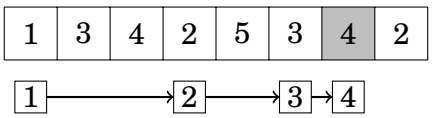
The next element 2 is smaller than the two top elements in the stack. Thus, the elements 3 and 4 are removed from the stack, and then the element 2 is added to the stack. Its nearest smaller element is 1:



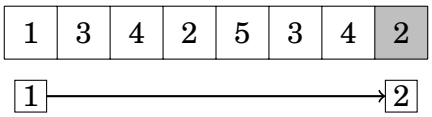
Then, the element 5 is larger than the element 2, so it will be added to the stack, and its nearest smaller element is 2:



After this, the element 5 is removed from the stack and the elements 3 and 4 are added to the stack:



Finally, all elements except 1 are removed from the stack and the last element 2 is added to the stack:



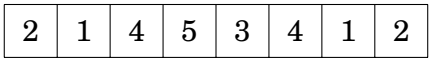
The efficiency of the algorithm depends on the total number of stack operations. If the current element is larger than the top element in the stack, it is directly added to the stack, which is efficient. However, sometimes the stack can contain several larger elements and it takes time to remove them. Still, each element is added *exactly once* to the stack and removed *at most once* from the stack. Thus, each element causes $O(1)$ stack operations, and the algorithm works in $O(n)$ time.

8.3 Sliding window minimum

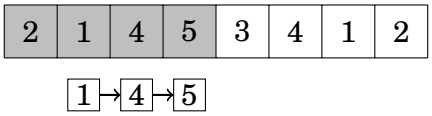
A **sliding window** is a constant-size subarray that moves from left to right through the array. At each window position, we want to calculate some information about the elements inside the window. In this section, we focus on the problem of maintaining the **sliding window minimum**, which means that we should report the smallest value inside each window.

The sliding window minimum can be calculated using a similar idea that we used to calculate the nearest smaller elements. We maintain a queue where each element is larger than the previous element, and the first element always corresponds to the minimum element inside the window. After each window move, we remove elements from the end of the queue until the last queue element is smaller than the new window element, or the queue becomes empty. We also remove the first queue element if it is not inside the window anymore. Finally, we add the new window element to the end of the queue.

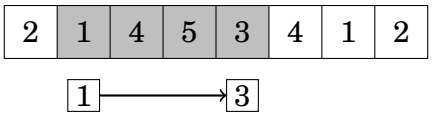
As an example, consider the following array:



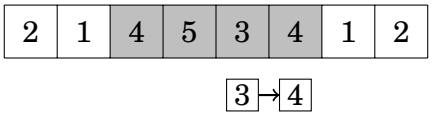
Suppose that the size of the sliding window is 4. At the first window position, the smallest value is 1:



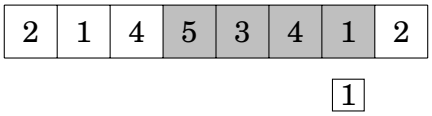
Then the window moves one step right. The new element 3 is smaller than the elements 4 and 5 in the queue, so the elements 4 and 5 are removed from the queue and the element 3 is added to the queue. The smallest value is still 1.



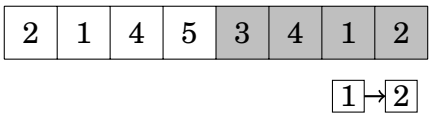
After this, the window moves again, and the smallest element 1 does not belong to the window anymore. Thus, it is removed from the queue and the smallest value is now 3. Also the new element 4 is added to the queue.



The next new element 1 is smaller than all elements in the queue. Thus, all elements are removed from the queue and it will only contain the element 1:



Finally the window reaches its last position. The element 2 is added to the queue, but the smallest value inside the window is still 1.



Since each array element is added to the queue exactly once and removed from the queue at most once, the algorithm works in $O(n)$ time.

Chapter 9

Range queries

In this chapter, we discuss data structures that allow us to efficiently process range queries. In a **range query**, our task is to calculate a value based on a subarray of an array. Typical range queries are:

- $\texttt{sum}_q(a,b)$: calculate the sum of values in range $[a,b]$
- $\texttt{min}_q(a,b)$: find the minimum value in range $[a,b]$
- $\texttt{max}_q(a,b)$: find the maximum value in range $[a,b]$

For example, consider the range $[3,6]$ in the following array:

0	1	2	3	4	5	6	7
1	3	8	4	6	1	3	4

In this case, $\texttt{sum}_q(3,6) = 14$, $\texttt{min}_q(3,6) = 1$ and $\texttt{max}_q(3,6) = 6$.

A simple way to process range queries is to use a loop that goes through all array values in the range. For example, the following function can be used to process sum queries on an array:

```
def sum(a, b) {
    s = 0
    for i in range(a, b+1):
        s += array[i]
    return s
```

This function works in $O(n)$ time, where n is the size of the array. Thus, we can process q queries in $O(nq)$ time using the function. However, if both n and q are large, this approach is slow. Fortunately, it turns out that there are ways to process range queries much more efficiently.

9.1 Static array queries

We first focus on a situation where the array is *static*, i.e., the array values are never updated between the queries. In this case, it suffices to construct a static data structure that tells us the answer for any possible query.

Sum queries

We can easily process sum queries on a static array by constructing a **prefix sum array**. Each value in the prefix sum array equals the sum of values in the original array up to that position, i.e., the value at position k is $\mathrm{sum}_q(0,k)$. The prefix sum array can be constructed in $O(n)$ time.

For example, consider the following array:

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2

The corresponding prefix sum array is as follows:

0	1	2	3	4	5	6	7
1	4	8	16	22	23	27	29

Since the prefix sum array contains all values of $\mathrm{sum}_q(0,k)$, we can calculate any value of $\mathrm{sum}_q(a,b)$ in $O(1)$ time as follows:

$$\mathrm{sum}_q(a,b) = \mathrm{sum}_q(0,b) - \mathrm{sum}_q(0,a-1)$$

By defining $\mathrm{sum}_q(0,-1) = 0$, the above formula also holds when $a = 0$.

For example, consider the range $[3,6]$:

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2

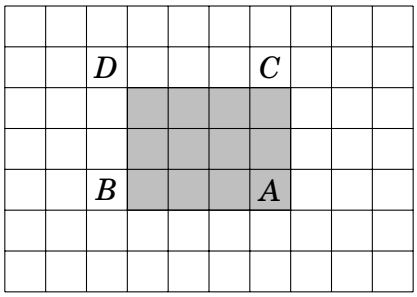
In this case $\mathrm{sum}_q(3,6) = 8+6+1+4 = 19$. This sum can be calculated from two values of the prefix sum array:

0	1	2	3	4	5	6	7
1	4	8	16	22	23	27	29

Thus, $\mathrm{sum}_q(3,6) = \mathrm{sum}_q(0,6) - \mathrm{sum}_q(0,2) = 27 - 8 = 19$.

It is also possible to generalize this idea to higher dimensions. For example, we can construct a two-dimensional prefix sum array that can be used to calculate the sum of any rectangular subarray in $O(1)$ time. Each sum in such an array corresponds to a subarray that begins at the upper-left corner of the array.

The following picture illustrates the idea:



The sum of the gray subarray can be calculated using the formula

$$S(A) - S(B) - S(C) + S(D),$$

where $S(X)$ denotes the sum of values in a rectangular subarray from the upper-left corner to the position of X.

Minimum queries

Minimum queries are more difficult to process than sum queries. Still, there is a quite simple $O(n \log n)$ time preprocessing method after which we can answer any minimum query in $O(1)$ time[1]. Note that since minimum and maximum queries can be processed similarly, we can focus on minimum queries.

The idea is to precalculate all values of $\min_q(a,b)$ where $b-a+1$ (the length of the range) is a power of two. For example, for the array

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2

the following values are calculated:

a	b	$\min_q(a,b)$
0	0	1
1	1	3
2	2	4
3	3	8
4	4	6
5	5	1
6	6	4
7	7	2

a	b	$\min_q(a,b)$
0	1	1
1	2	3
2	3	4
3	4	6
4	5	1
5	6	1
6	7	2

a	b	$\min_q(a,b)$
0	3	1
1	4	3
2	5	1
3	6	1
4	7	1
0	7	1

The number of precalculated values is $O(n \log n)$, because there are $O(\log n)$ range lengths that are powers of two. The values can be calculated efficiently using the recursive formula

$$\min_q(a,b) = \min(\min_q(a,a+w-1), \min_q(a+w,b)),$$

where $b-a+1$ is a power of two and $w = (b-a+1)/2$. Calculating all those values takes $O(n \log n)$ time.

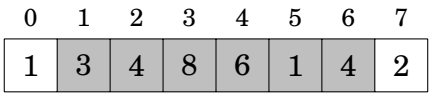
After this, any value of $\min_q(a,b)$ can be calculated in $O(1)$ time as a minimum of two precalculated values. Let k be the largest power of two that does not exceed $b-a+1$. We can calculate the value of $\min_q(a,b)$ using the formula

$$\min_q(a,b) = \min(\min_q(a,a+k-1), \min_q(b-k+1,b)).$$

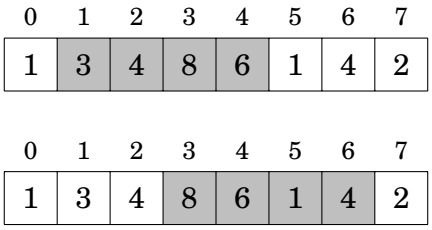
In the above formula, the range $[a,b]$ is represented as the union of the ranges $[a,a+k-1]$ and $[b-k+1,b]$, both of length k.

As an example, consider the range $[1,6]$:

[1] This technique was introduced in [4] and sometimes called the **sparse table** method. There are also more sophisticated techniques [13] where the preprocessing time is only $O(n)$, but such algorithms are not needed in competitive programming.



The length of the range is 6, and the largest power of two that does not exceed 6 is 4. Thus the range $[1,6]$ is the union of the ranges $[1,4]$ and $[3,6]$:



Since $\texttt{min}_q(1,4) = 3$ and $\texttt{min}_q(3,6) = 1$, we conclude that $\texttt{min}_q(1,6) = 1$.

9.2 Binary indexed tree

A **binary indexed tree** or a **Fenwick tree**[2] can be seen as a dynamic variant of a prefix sum array. It supports two $O(\log n)$ time operations on an array: processing a range sum query and updating a value.

The advantage of a binary indexed tree is that it allows us to efficiently update array values between sum queries. This would not be possible using a prefix sum array, because after each update, it would be necessary to build the whole prefix sum array again in $O(n)$ time.

Structure

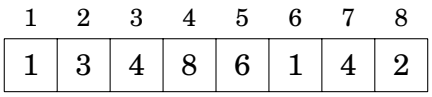
Even if the name of the structure is a binary indexed *tree*, it is usually represented as an array. In this section we assume that all arrays are one-indexed, because it makes the implementation easier.

Let $p(k)$ denote the largest power of two that divides k. We store a binary indexed tree as an array `tree` such that

$$\texttt{tree}[k] = \texttt{sum}_q(k - p(k) + 1, k),$$

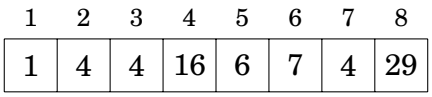
i.e., each position k contains the sum of values in a range of the original array whose length is $p(k)$ and that ends at position k. For example, since $p(6) = 2$, `tree`[6] contains the value of $\texttt{sum}_q(5,6)$.

For example, consider the following array:

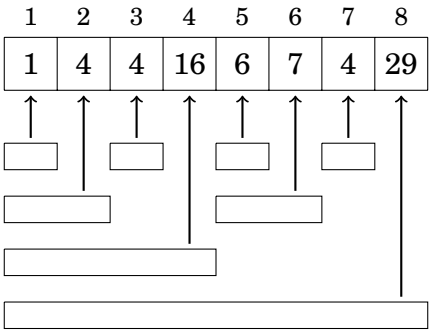


The corresponding binary indexed tree is as follows:

[2]The binary indexed tree structure was presented by P. M. Fenwick in 1994 [12].

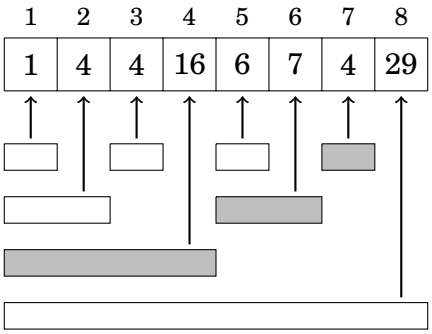


The following picture shows more clearly how each value in the binary indexed tree corresponds to a range in the original array:



Using a binary indexed tree, any value of $\mathrm{sum}_q(1,k)$ can be calculated in $O(\log n)$ time, because a range $[1,k]$ can always be divided into $O(\log n)$ ranges whose sums are stored in the tree.

For example, the range $[1,7]$ consists of the following ranges:



Thus, we can calculate the corresponding sum as follows:

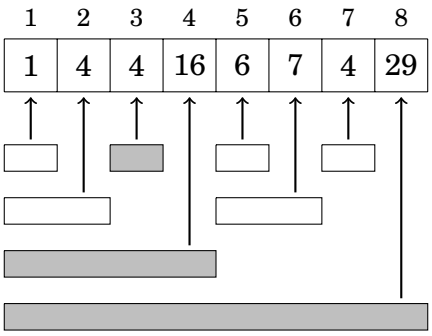
$$\mathrm{sum}_q(1,7) = \mathrm{sum}_q(1,4) + \mathrm{sum}_q(5,6) + \mathrm{sum}_q(7,7) = 16+7+4 = 27$$

To calculate the value of $\mathrm{sum}_q(a,b)$ where $a > 1$, we can use the same trick that we used with prefix sum arrays:

$$\mathrm{sum}_q(a,b) = \mathrm{sum}_q(1,b) - \mathrm{sum}_q(1,a-1).$$

Since we can calculate both $\mathrm{sum}_q(1,b)$ and $\mathrm{sum}_q(1,a-1)$ in $O(\log n)$ time, the total time complexity is $O(\log n)$.

Then, after updating a value in the original array, several values in the binary indexed tree should be updated. For example, if the value at position 3 changes, the sums of the following ranges change:



Since each array element belongs to $O(\log n)$ ranges in the binary indexed tree, it suffices to update $O(\log n)$ values in the tree.

Implementation

The operations of a binary indexed tree can be efficiently implemented using bit operations. The key fact needed is that we can calculate any value of $p(k)$ using the formula

$$p(k) = k \& -k.$$

The following function calculates the value of $\texttt{sum}_q(1,k)$:

```
def sum(k):
    s = 0
    while (k >= 1):
        s += tree[k]
        k -= k&-k
    return s
```

The following function increases the array value at position k by x (x can be positive or negative):

```
add(k, x):
    while (k <= n):
        tree[k] += x
        k += k&-k
```

The time complexity of both the functions is $O(\log n)$, because the functions access $O(\log n)$ values in the binary indexed tree, and each move to the next position takes $O(1)$ time.

9.3 Segment tree

A **segment tree**[3] is a data structure that supports two operations: processing a range query and updating an array value. Segment trees can support sum

[3]The bottom-up-implementation in this chapter corresponds to that in [30]. Similar structures were used in late 1970's to solve geometric problems [7].

queries, minimum and maximum queries and many other queries so that both operations work in $O(\log n)$ time.

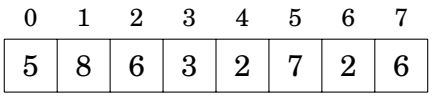
Compared to a binary indexed tree, the advantage of a segment tree is that it is a more general data structure. While binary indexed trees only support sum queries[4], segment trees also support other queries. On the other hand, a segment tree requires more memory and is a bit more difficult to implement.

Structure

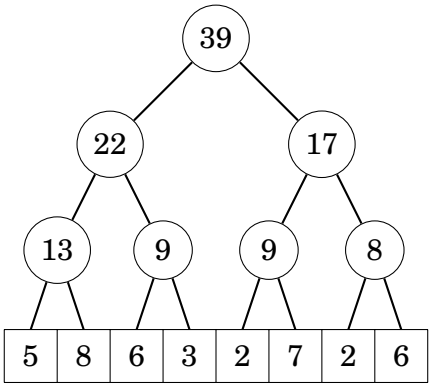
A segment tree is a binary tree such that the nodes on the bottom level of the tree correspond to the array elements, and the other nodes contain information needed for processing range queries.

In this section, we assume that the size of the array is a power of two and zero-based indexing is used, because it is convenient to build a segment tree for such an array. If the size of the array is not a power of two, we can always append extra elements to it.

We will first discuss segment trees that support sum queries. As an example, consider the following array:

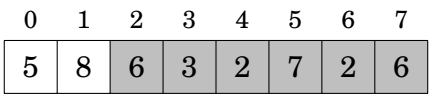


The corresponding segment tree is as follows:



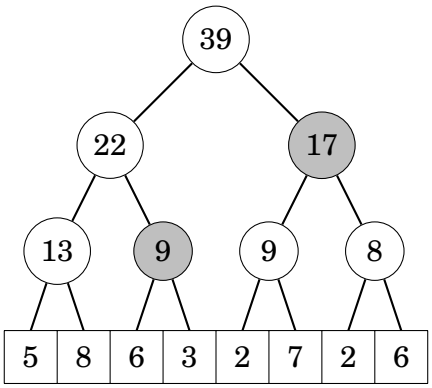
Each internal tree node corresponds to an array range whose size is a power of two. In the above tree, the value of each internal node is the sum of the corresponding array values, and it can be calculated as the sum of the values of its left and right child node.

It turns out that any range $[a,b]$ can be divided into $O(\log n)$ ranges whose values are stored in tree nodes. For example, consider the range [2,7]:



[4] In fact, using *two* binary indexed trees it is possible to support minimum queries [9], but this is more complicated than to use a segment tree.

Here $\texttt{sum}_q(2,7) = 6+3+2+7+2+6 = 26$. In this case, the following two tree nodes correspond to the range:

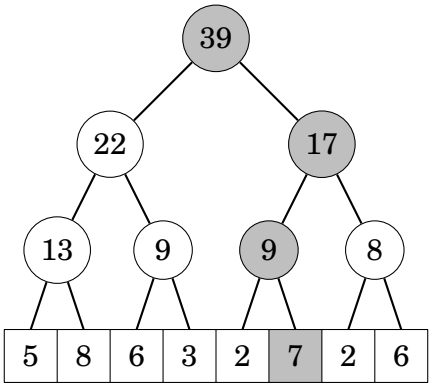


Thus, another way to calculate the sum is $9+17 = 26$.

When the sum is calculated using nodes located as high as possible in the tree, at most two nodes on each level of the tree are needed. Hence, the total number of nodes is $O(\log n)$.

After an array update, we should update all nodes whose value depends on the updated value. This can be done by traversing the path from the updated array element to the top node and updating the nodes along the path.

The following picture shows which tree nodes change if the array value 7 changes:

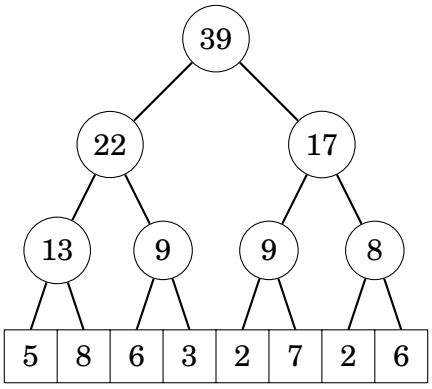


The path from bottom to top always consists of $O(\log n)$ nodes, so each update changes $O(\log n)$ nodes in the tree.

Implementation

We store a segment tree as a list of $2n$ elements where n is the size of the original array and a power of two. The tree nodes are stored from top to bottom: `tree`[1] is the top node, `tree`[2] and `tree`[3] are its children, and so on. Finally, the values from `tree`[n] to `tree`[$2n-1$] correspond to the values of the original array on the bottom level of the tree.

For example, the segment tree



is stored as follows:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
39	22	17	13	9	9	8	5	8	6	3	2	7	2	6

Using this representation, the parent of $\texttt{tree}[k]$ is $\texttt{tree}[\lfloor k/2 \rfloor]$, and its children are $\texttt{tree}[2k]$ and $\texttt{tree}[2k+1]$. Note that this implies that the position of a node is even if it is a left child and odd if it is a right child.

The following function calculates the value of $\texttt{sum}_q(a,b)$:

```
def sum(a, b):
    a += n
    b += n
    s = 0
    while (a <= b) :
        if (a%2 == 1):
            s += tree[a++]
        if (b%2 == 0):
            s += tree[b--]
        a /= 2
        b /= 2
    return s
```

The function maintains a range that is initially $[a+n, b+n]$. Then, at each step, the range is moved one level higher in the tree, and before that, the values of the nodes that do not belong to the higher range are added to the sum.

The following function increases the array value at position k by x:

```
add(k, x) {
    k += n
    tree[k] += x
    k /= 2
    while (k >= 1):
        tree[k] = tree[2*k]+tree[2*k+1]
```

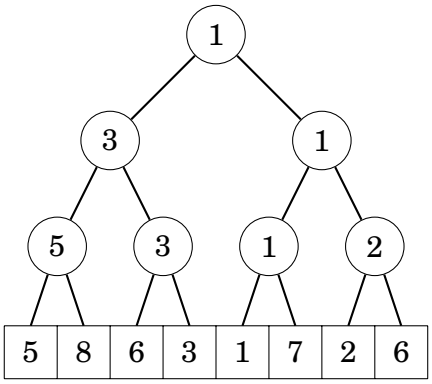
First the function updates the value at the bottom level of the tree. After this, the function updates the values of all internal tree nodes, until it reaches the top node of the tree.

Both the above functions work in $O(\log n)$ time, because a segment tree of n elements consists of $O(\log n)$ levels, and the functions move one level higher in the tree at each step.

Other queries

Segment trees can support all range queries where it is possible to divide a range into two parts, calculate the answer separately for both parts and then efficiently combine the answers. Examples of such queries are minimum and maximum, greatest common divisor, and bit operations and, or and xor.

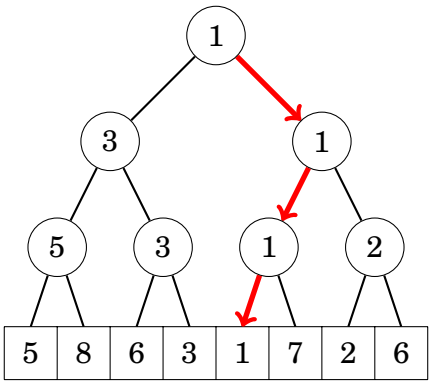
For example, the following segment tree supports minimum queries:



In this case, every tree node contains the smallest value in the corresponding array range. The top node of the tree contains the smallest value in the whole array. The operations can be implemented like previously, but instead of sums, minima are calculated.

The structure of a segment tree also allows us to use binary search for locating array elements. For example, if the tree supports minimum queries, we can find the position of an element with the smallest value in $O(\log n)$ time.

For example, in the above tree, an element with the smallest value 1 can be found by traversing a path downwards from the top node:



9.4 Additional techniques

Index compression

A limitation in data structures that are built upon an array is that the elements are indexed using consecutive integers. Difficulties arise when large indices are needed. For example, if we wish to use the index 10^9, the array should contain 10^9 elements which would require too much memory.

However, we can often bypass this limitation by using **index compression**, where the original indices are replaced with indices $1, 2, 3$, etc. This can be done if we know all the indices needed during the algorithm beforehand.

The idea is to replace each original index x with $c(x)$ where c is a function that compresses the indices. We require that the order of the indices does not change, so if $a < b$, then $c(a) < c(b)$. This allows us to conveniently perform queries even if the indices are compressed.

For example, if the original indices are 555, 10^9 and 8, the new indices are:

$$\begin{aligned} c(8) &= 1 \\ c(555) &= 2 \\ c(10^9) &= 3 \end{aligned}$$

Range updates

So far, we have implemented data structures that support range queries and updates of single values. Let us now consider an opposite situation, where we should update ranges and retrieve single values. We focus on an operation that increases all elements in a range $[a, b]$ by x.

Surprisingly, we can use the data structures presented in this chapter also in this situation. To do this, we build a **difference array** whose values indicate the differences between consecutive values in the original array. Thus, the original array is the prefix sum array of the difference array. For example, consider the following array:

0	1	2	3	4	5	6	7
3	3	1	1	1	5	2	2

The difference array for the above array is as follows:

0	1	2	3	4	5	6	7
3	0	−2	0	0	4	−3	0

For example, the value 2 at position 6 in the original array corresponds to the sum $3 - 2 + 4 - 3 = 2$ in the difference array.

The advantage of the difference array is that we can update a range in the original array by changing just two elements in the difference array. For example, if we want to increase the original array values between positions 1 and 4 by 5, it suffices to increase the difference array value at position 1 by 5 and decrease the value at position 5 by 5. The result is as follows:

0	1	2	3	4	5	6	7
3	5	−2	0	0	−1	−3	0

More generally, to increase the values in range $[a,b]$ by x, we increase the value at position a by x and decrease the value at position $b+1$ by x. Thus, it is only needed to update single values and process sum queries, so we can use a binary indexed tree or a segment tree.

A more difficult problem is to support both range queries and range updates. In Chapter 28 we will see that even this is possible.

Chapter 11

Basics of graphs

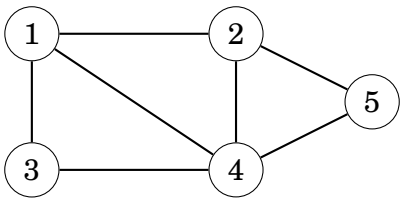
Many programming problems can be solved by modeling the problem as a graph problem and using an appropriate graph algorithm. A typical example of a graph is a network of roads and cities in a country. Sometimes, though, the graph is hidden in the problem and it may be difficult to detect it.

This part of the book discusses graph algorithms, especially focusing on topics that are important in competitive programming. In this chapter, we go through concepts related to graphs, and study different ways to represent graphs in algorithms.

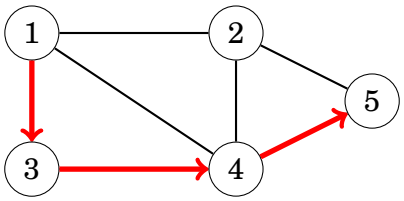
11.1 Graph terminology

A **graph** consists of **nodes** and **edges**. In this book, the variable n denotes the number of nodes in a graph, and the variable m denotes the number of edges. The nodes are numbered using integers $1, 2, \ldots, n$.

For example, the following graph consists of 5 nodes and 7 edges:



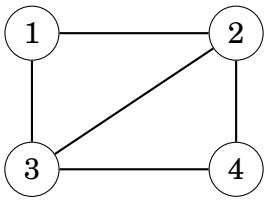
A **path** leads from node a to node b through edges of the graph. The **length** of a path is the number of edges in it. For example, the above graph contains a path $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ of length 3 from node 1 to node 5:



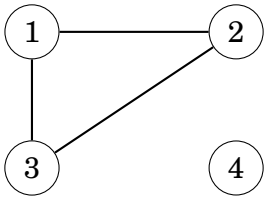
A path is a **cycle** if the first and last node is the same. For example, the above graph contains a cycle $1 \rightarrow 3 \rightarrow 4 \rightarrow 1$. A path is **simple** if each node appears at most once in the path.

Connectivity

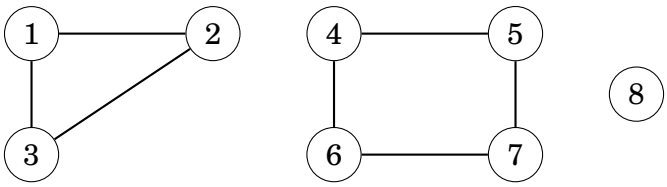
A graph is **connected** if there is a path between any two nodes. For example, the following graph is connected:



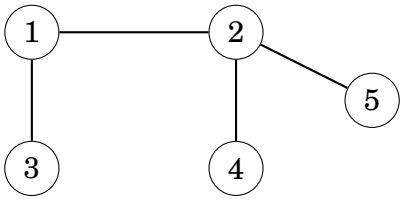
The following graph is not connected, because it is not possible to get from node 4 to any other node:



The connected parts of a graph are called its **components**. For example, the following graph contains three components: $\{1, 2, 3\}$, $\{4, 5, 6, 7\}$ and $\{8\}$.

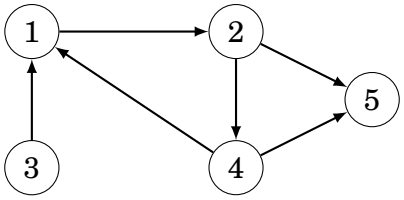


A **tree** is a connected graph that consists of n nodes and $n-1$ edges. There is a unique path between any two nodes of a tree. For example, the following graph is a tree:



Edge directions

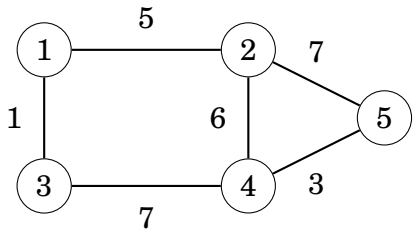
A graph is **directed** if the edges can be traversed in one direction only. For example, the following graph is directed:



The above graph contains a path $3 \to 1 \to 2 \to 5$ from node 3 to node 5, but there is no path from node 5 to node 3.

Edge weights

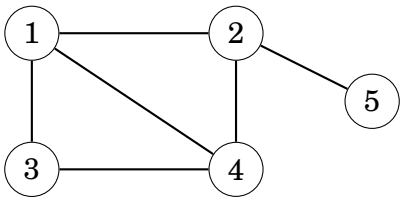
In a **weighted** graph, each edge is assigned a **weight**. The weights are often interpreted as edge lengths. For example, the following graph is weighted:



The length of a path in a weighted graph is the sum of the edge weights on the path. For example, in the above graph, the length of the path $1 \to 2 \to 5$ is 12, and the length of the path $1 \to 3 \to 4 \to 5$ is 11. The latter path is the **shortest** path from node 1 to node 5.

Neighbors and degrees

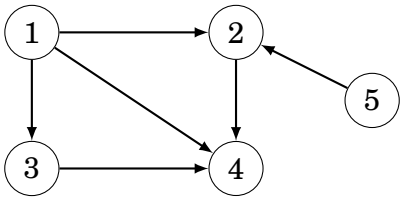
Two nodes are **neighbors** or **adjacent** if there is an edge between them. The **degree** of a node is the number of its neighbors. For example, in the following graph, the neighbors of node 2 are 1, 4 and 5, so its degree is 3.



The sum of degrees in a graph is always $2m$, where m is the number of edges, because each edge increases the degree of exactly two nodes by one. For this reason, the sum of degrees is always even.

A graph is **regular** if the degree of every node is a constant d. A graph is **complete** if the degree of every node is $n-1$, i.e., the graph contains all possible edges between the nodes.

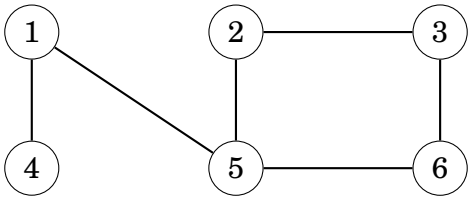
In a directed graph, the **indegree** of a node is the number of edges that end at the node, and the **outdegree** of a node is the number of edges that start at the node. For example, in the following graph, the indegree of node 2 is 2, and the outdegree of node 2 is 1.



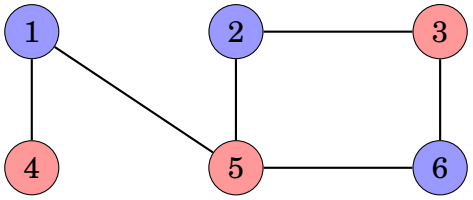
Colorings

In a **coloring** of a graph, each node is assigned a color so that no adjacent nodes have the same color.

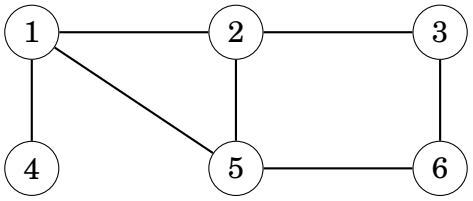
A graph is **bipartite** if it is possible to color it using two colors. It turns out that a graph is bipartite exactly when it does not contain a cycle with an odd number of edges. For example, the graph



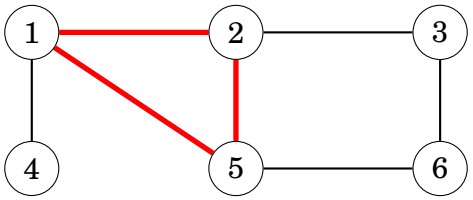
is bipartite, because it can be colored as follows:



However, the graph

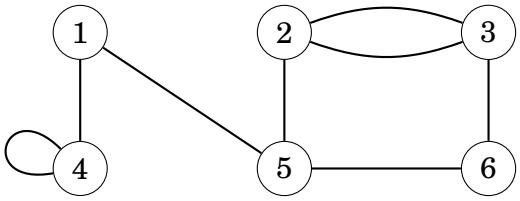


is not bipartite, because it is not possible to color the following cycle of three nodes using two colors:



Simplicity

A graph is **simple** if no edge starts and ends at the same node, and there are no multiple edges between two nodes. Often we assume that graphs are simple. For example, the following graph is *not* simple:



11.2 Graph representation

There are several ways to represent graphs in algorithms. The choice of a data structure depends on the size of the graph and the way the algorithm processes it. Next we will go through three common representations.

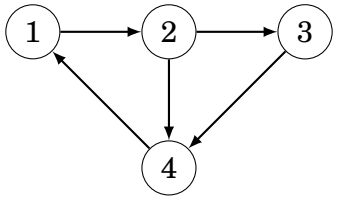
Adjacency list representation

In the adjacency list representation, each node x in the graph is assigned an **adjacency list** that consists of nodes to which there is an edge from x. Adjacency lists are the most popular way to represent graphs, and most algorithms can be efficiently implemented using them.

A convenient way to store the adjacency lists is to declare a list of lists as follows:

```
adj = [[] for i in range(N)]
```

The constant n is chosen so that all adjacency lists can be stored. Typically, it is enough to have $N = n + 1$ if the vertices[1]. For example, the graph

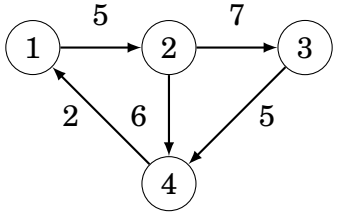


can be stored as follows:

```
adj[1].append(2)
adj[2].append(3)
adj[2].append(4)
adj[3].append(4)
adj[4].append(1)
```

If the graph is undirected, it can be stored in a similar way, but each edge is added in both directions.

For a weighted graph, the structure can be extended to store pairs instead of integers. In this case, the adjacency list of node a contains the pair (b, w) always when there is an edge from node a to node b with weight w. For example, the graph



[1] If the vertices in your graph is enumerated from 0 to $n-1$, then you can just use $N = n$.

can be stored as follows:

```
adj[1].append((2,5))
adj[2].append((3,7))
adj[2].append((4,6))
adj[3].append((4,5))
adj[4].append((1,2))
```

The benefit of using adjacency lists is that we can efficiently find the nodes to which we can move from a given node through an edge. For example, the following loop goes through all nodes to which we can move from node s:

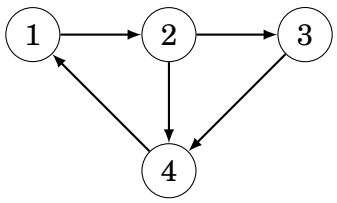
```
for u in adj[s]:
    # process node u
```

Adjacency matrix representation

An **adjacency matrix** is a two-dimensional array that indicates which edges the graph contains. We can efficiently check from an adjacency matrix if there is an edge between two nodes. The matrix can be stored as a list of lists

```
// Initialize adjacency matrix with 0's
adj = [[0]*N for i in range(N)]
```

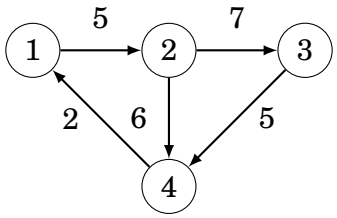
where each value adj[a][b] indicates whether the graph contains an edge from node a to node b. If the edge is included in the graph, then adj[a][b] = 1, and otherwise adj[a][b] = 0. For example, the graph



can be represented as follows:

	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	0	0	0	1
4	1	0	0	0

If the graph is weighted, the adjacency matrix representation can be extended so that the matrix contains the weight of the edge if the edge exists. Using this representation, the graph



corresponds to the following matrix:

	1	2	3	4
1	0	5	0	0
2	0	0	7	6
3	0	0	0	5
4	2	0	0	0

The drawback of the adjacency matrix representation is that the matrix contains n^2 elements, and usually most of them are zero. For this reason, the representation cannot be used if the graph is large.

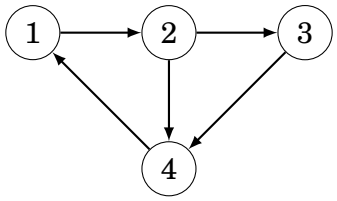
Edge list representation

An **edge list** contains all edges of a graph in some order. This is a convenient way to represent a graph if the algorithm processes all edges of the graph and it is not needed to find edges that start at a given node.

The edge list can be stored in a list

```
edges = []
```

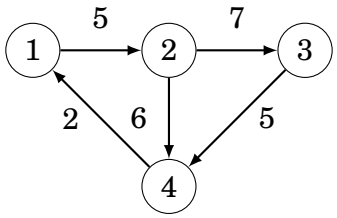
where each pair (a,b) denotes that there is an edge from node a to node b. Thus, the graph



can be represented as follows:

```
edges.append((1,2))
edges.append((2,3))
edges.append((2,4))
edges.append((3,4))
edges.append((4,1))
```

If the graph is weighted, the structure can be extended to a list of triples. Each element in this list is of the form (a,b,w), which means that there is an edge from node a to node b with weight w. For example, the graph



can be represented as follows:

```
edges.append((1,2,5))
edges.append((2,3,7))
edges.append((2,4,6))
edges.append((3,4,5))
edges.append((4,1,2))
```

Chapter 12

Graph traversal

This chapter discusses two fundamental graph algorithms: depth-first search and breadth-first search. Both algorithms are given a starting node in the graph, and they visit all nodes that can be reached from the starting node. The difference in the algorithms is the order in which they visit the nodes.

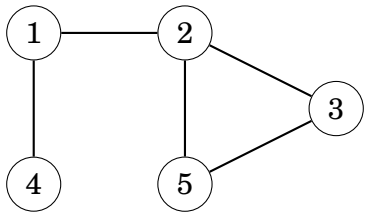
12.1 Depth-first search

Depth-first search (DFS) is a straightforward graph traversal technique. The algorithm begins at a starting node, and proceeds to all other nodes that are reachable from the starting node using the edges of the graph.

Depth-first search always follows a single path in the graph as long as it finds new nodes. After this, it returns to previous nodes and begins to explore other parts of the graph. The algorithm keeps track of visited nodes, so that it processes each node only once.

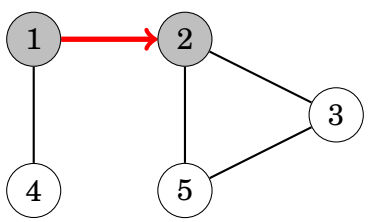
Example

Let us consider how depth-first search processes the following graph:

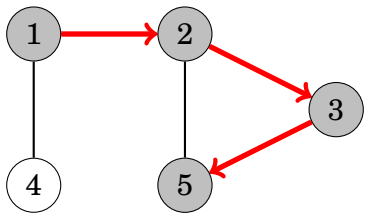


We may begin the search at any node of the graph; now we will begin the search at node 1.

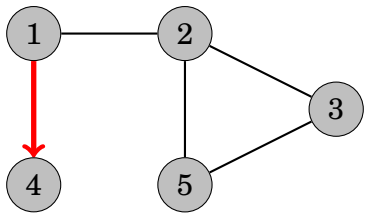
The search first proceeds to node 2:



After this, nodes 3 and 5 will be visited:



The neighbors of node 5 are 2 and 3, but the search has already visited both of them, so it is time to return to the previous nodes. Also the neighbors of nodes 3 and 2 have been visited, so we next move from node 1 to node 4:



After this, the search terminates because it has visited all nodes.

The time complexity of depth-first search is $O(n+m)$ where n is the number of nodes and m is the number of edges, because the algorithm processes each node and edge a constant number of times.

Implementation

Depth-first search can be conveniently implemented using recursion. The following function `dfs` begins a depth-first search at a given node. The function assumes that the graph is stored as adjacency lists in a list.

```
adj = [[] for i in range(N)]
```

and also maintains a list

```
visited = [False for i in range(N)]
```

that keeps track of the visited nodes. Initially, each list value is `false`, and when the search arrives at node s, the value of `visited`[s] becomes `true`. The function can be implemented as follows:

```
def dfs(s):
    if (visited[s]):
        return
    visited[s] = True
    # process node s
    for u in adj[s]:
        dfs(u)
```

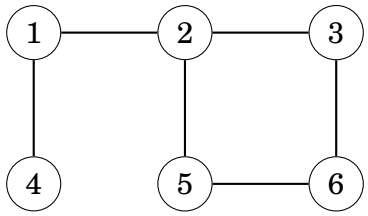
12.2 Breadth-first search

Breadth-first search (BFS) visits the nodes in increasing order of their distance from the starting node. Thus, we can calculate the distance from the starting node to all other nodes using breadth-first search. However, breadth-first search is more difficult to implement than depth-first search.

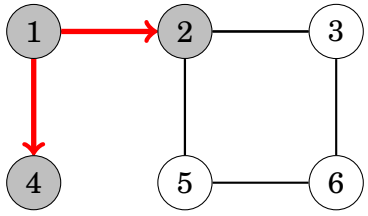
Breadth-first search goes through the nodes one level after another. First the search explores the nodes whose distance from the starting node is 1, then the nodes whose distance is 2, and so on. This process continues until all nodes have been visited.

Example

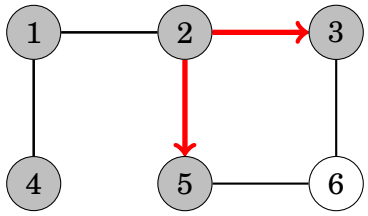
Let us consider how breadth-first search processes the following graph:



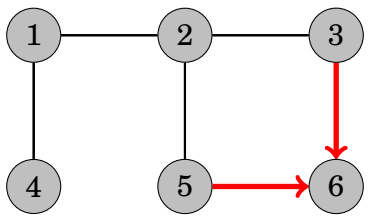
Suppose that the search begins at node 1. First, we process all nodes that can be reached from node 1 using a single edge:



After this, we proceed to nodes 3 and 5:



Finally, we visit node 6:



Now we have calculated the distances from the starting node to all nodes of the graph. The distances are as follows:

node	distance
1	0
2	1
3	2
4	1
5	2
6	3

Like in depth-first search, the time complexity of breadth-first search is $O(n+m)$, where n is the number of nodes and m is the number of edges.

Implementation

Breadth-first search is more difficult to implement than depth-first search, because the algorithm visits nodes in different parts of the graph. A typical implementation is based on a queue that contains nodes. At each step, the next node in the queue will be processed.

The following code assumes that the graph is stored as adjacency lists and maintains the following data structures:

```
from collections import deque
q = deque()
visited = [False for i in range(N)]
distance = [-1 for i in range(N)]
```

The queue `q` contains nodes to be processed in increasing order of their distance. New nodes are always added to the end of the queue, and the node at the beginning of the queue is the next node to be processed. The list `visited` indicates which nodes the search has already visited, and the list `distance` will contain the distances from the starting node to all nodes of the graph.

The search can be implemented as follows, starting at node x:

```
visited[x] = True
distance[x] = 0
q.append(x)
while q:
    s = q.popleft()
    # process node s
    for u in adj[s]:
        if (visited[u]):
                continue
        visited[u] = True
        distance[u] = distance[s]+1
        q.append(u)
```

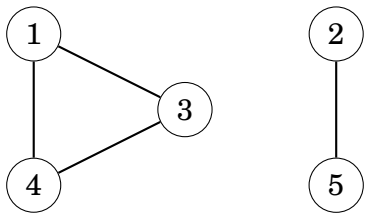
12.3 Applications

Using the graph traversal algorithms, we can check many properties of graphs. Usually, both depth-first search and breadth-first search may be used, but in practice, depth-first search is a better choice, because it is easier to implement. In the following applications we will assume that the graph is undirected.

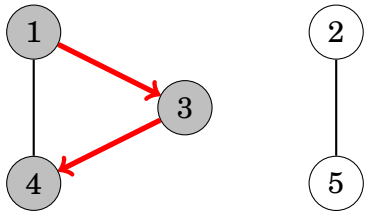
Connectivity check

A graph is connected if there is a path between any two nodes of the graph. Thus, we can check if a graph is connected by starting at an arbitrary node and finding out if we can reach all other nodes.

For example, in the graph



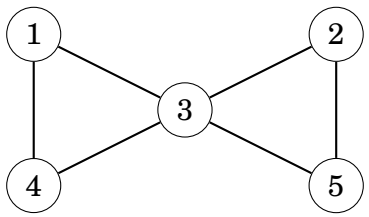
a depth-first search from node 1 visits the following nodes:



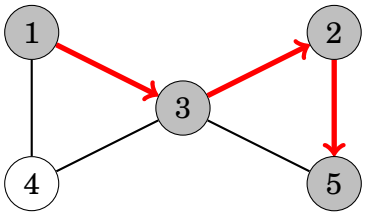
Since the search did not visit all the nodes, we can conclude that the graph is not connected. In a similar way, we can also find all connected components of a graph by iterating through the nodes and always starting a new depth-first search if the current node does not belong to any component yet.

Finding cycles

A graph contains a cycle if during a graph traversal, we find a node whose neighbor (other than the previous node in the current path) has already been visited. For example, the graph



contains two cycles and we can find one of them as follows:



After moving from node 2 to node 5 we notice that the neighbor 3 of node 5 has already been visited. Thus, the graph contains a cycle that goes through node 3, for example, $3 \to 2 \to 5 \to 3$.

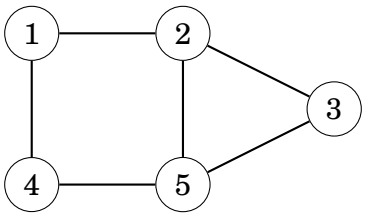
Another way to find out whether a graph contains a cycle is to simply calculate the number of nodes and edges in every component. If a component contains c nodes and no cycle, it must contain exactly $c - 1$ edges (so it has to be a tree). If there are c or more edges, the component surely contains a cycle.

Bipartiteness check

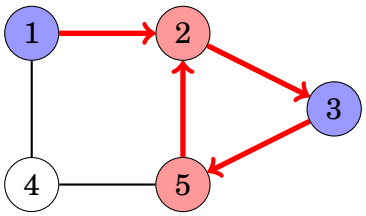
A graph is bipartite if its nodes can be colored using two colors so that there are no adjacent nodes with the same color. It is surprisingly easy to check if a graph is bipartite using graph traversal algorithms.

The idea is to color the starting node blue, all its neighbors red, all their neighbors blue, and so on. If at some point of the search we notice that two adjacent nodes have the same color, this means that the graph is not bipartite. Otherwise the graph is bipartite and one coloring has been found.

For example, the graph



is not bipartite, because a search from node 1 proceeds as follows:



We notice that the color of both nodes 2 and 5 is red, while they are adjacent nodes in the graph. Thus, the graph is not bipartite.

This algorithm always works, because when there are only two colors available, the color of the starting node in a component determines the colors of all other nodes in the component. It does not make any difference whether the starting node is red or blue.

Note that in the general case, it is difficult to find out if the nodes in a graph can be colored using k colors so that no adjacent nodes have the same color. Even when $k = 3$, no efficient algorithm is known but the problem is NP-hard.

Chapter 13

Shortest paths

Finding a shortest path between two nodes of a graph is an important problem that has many practical applications. For example, a natural problem related to a road network is to calculate the shortest possible length of a route between two cities, given the lengths of the roads.

In an unweighted graph, the length of a path equals the number of its edges, and we can simply use breadth-first search to find a shortest path. However, in this chapter we focus on weighted graphs where more sophisticated algorithms are needed for finding shortest paths.

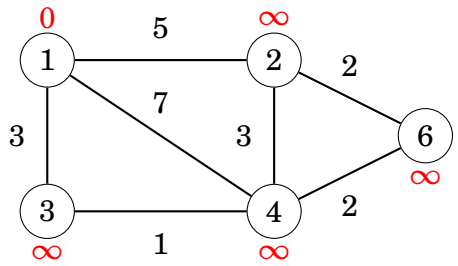
13.1 Bellman–Ford algorithm

The **Bellman–Ford algorithm**[1] finds shortest paths from a starting node to all nodes of the graph. The algorithm can process all kinds of graphs, provided that the graph does not contain a cycle with negative length. If the graph contains a negative cycle, the algorithm can detect this.

The algorithm keeps track of distances from the starting node to all nodes of the graph. Initially, the distance to the starting node is 0 and the distance to all other nodes in infinite. The algorithm reduces the distances by finding edges that shorten the paths until it is not possible to reduce any distance.

Example

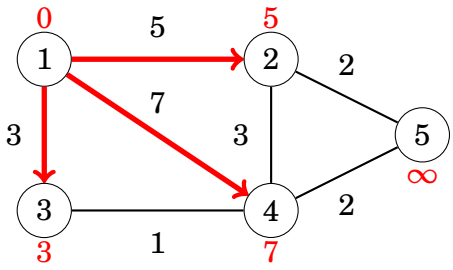
Let us consider how the Bellman–Ford algorithm works in the following graph:



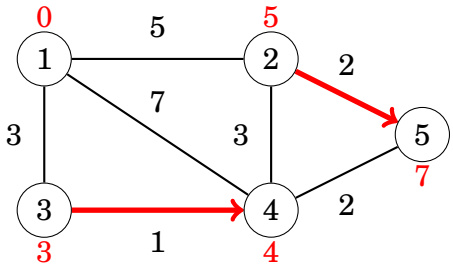
[1]The algorithm is named after R. E. Bellman and L. R. Ford who published it independently in 1958 and 1956, respectively [3, 15].

Each node of the graph is assigned a distance. Initially, the distance to the starting node is 0, and the distance to all other nodes is infinite.

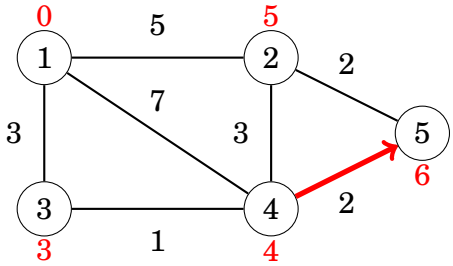
The algorithm searches for edges that reduce distances. First, all edges from node 1 reduce distances:



After this, edges $2 \to 5$ and $3 \to 4$ reduce distances:

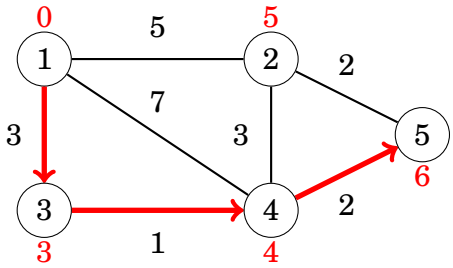


Finally, there is one more change:



After this, no edge can reduce any distance. This means that the distances are final, and we have successfully calculated the shortest distances from the starting node to all nodes of the graph.

For example, the shortest distance 3 from node 1 to node 5 corresponds to the following path:



Implementation

The following implementation of the Bellman–Ford algorithm determines the shortest distances from a node x to all nodes of the graph. The code assumes that the graph is stored as an edge list `edges` that consists of tuples of the form (a,b,w), meaning that there is an edge from node a to node b with weight w.

The algorithm consists of $n-1$ rounds, and on each round the algorithm goes through all edges of the graph and tries to reduce the distances. The algorithm constructs an list `distance` that will contain the distances from x to all nodes of the graph. The constant `INF` denotes an infinite distance.

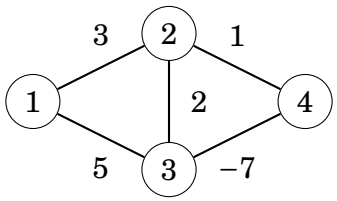
```
distance = [INF]*N
distance[x] = 0;
for i in range(n-1):
    for (a, b, w) in edges:
        distance[b] = min(distance[b], distance[a] + w)
```

The time complexity of the algorithm is $O(nm)$, because the algorithm consists of $n-1$ rounds and iterates through all m edges during a round. If there are no negative cycles in the graph, all distances are final after $n-1$ rounds, because each shortest path can contain at most $n-1$ edges.

In practice, the final distances can usually be found faster than in $n-1$ rounds. Thus, a possible way to make the algorithm more efficient is to stop the algorithm if no distance can be reduced during a round.

Negative cycles

The Bellman–Ford algorithm can also be used to check if the graph contains a cycle with negative length. For example, the graph



contains a negative cycle $2 \to 3 \to 4 \to 2$ with length -4.

If the graph contains a negative cycle, we can shorten infinitely many times any path that contains the cycle by repeating the cycle again and again. Thus, the concept of a shortest path is not meaningful in this situation.

A negative cycle can be detected using the Bellman–Ford algorithm by running the algorithm for n rounds. If the last round reduces any distance, the graph contains a negative cycle. Note that this algorithm can be used to search for a negative cycle in the whole graph regardless of the starting node.

SPFA algorithm

The **SPFA algorithm** ("Shortest Path Faster Algorithm") [11] is a variant of the Bellman–Ford algorithm, that is often more efficient than the original algorithm. The SPFA algorithm does not go through all the edges on each round, but instead, it chooses the edges to be examined in a more intelligent way.

The algorithm maintains a queue of nodes that might be used for reducing the distances. First, the algorithm adds the starting node x to the queue. Then, the algorithm always processes the first node in the queue, and when an edge $a \to b$ reduces a distance, node b is added to the queue.

The efficiency of the SPFA algorithm depends on the structure of the graph: the algorithm is often efficient, but its worst case time complexity is still $O(nm)$ and it is possible to create inputs that make the algorithm as slow as the original Bellman–Ford algorithm.

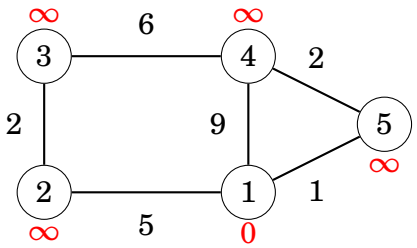
13.2 Dijkstra's algorithm

Dijkstra's algorithm[2] finds shortest paths from the starting node to all nodes of the graph, like the Bellman–Ford algorithm. The benefit of Dijsktra's algorithm is that it is more efficient and can be used for processing large graphs. However, the algorithm requires that there are no negative weight edges in the graph.

Like the Bellman–Ford algorithm, Dijkstra's algorithm maintains distances to the nodes and reduces them during the search. Dijkstra's algorithm is efficient, because it only processes each edge in the graph once, using the fact that there are no negative edges.

Example

Let us consider how Dijkstra's algorithm works in the following graph when the starting node is node 1:

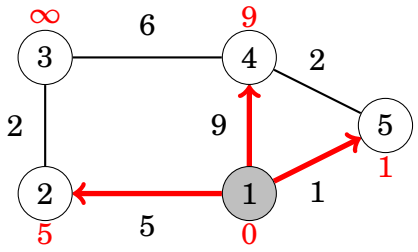


Like in the Bellman–Ford algorithm, initially the distance to the starting node is 0 and the distance to all other nodes is infinite.

At each step, Dijkstra's algorithm selects a node that has not been processed yet and whose distance is as small as possible. The first such node is node 1 with distance 0.

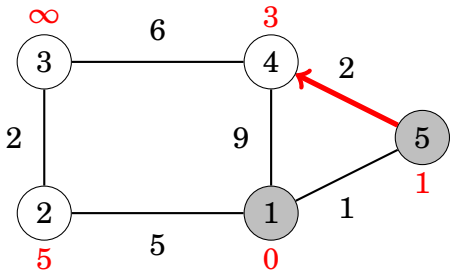
[2] E. W. Dijkstra published the algorithm in 1959 [8]; however, his original paper does not mention how to implement the algorithm efficiently.

When a node is selected, the algorithm goes through all edges that start at the node and reduces the distances using them:

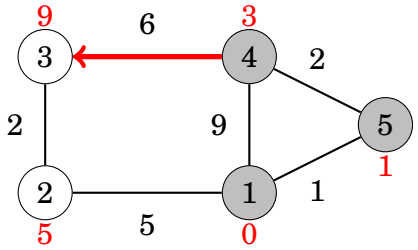


In this case, the edges from node 1 reduced the distances of nodes 2, 4 and 5, whose distances are now 5, 9 and 1.

The next node to be processed is node 5 with distance 1. This reduces the distance to node 4 from 9 to 3:

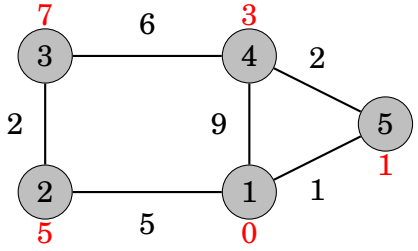


After this, the next node is node 4, which reduces the distance to node 3 to 9:



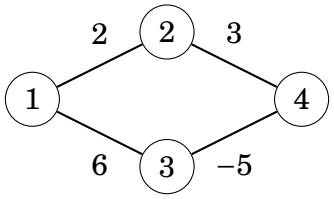
A remarkable property in Dijkstra's algorithm is that whenever a node is selected, its distance is final. For example, at this point of the algorithm, the distances 0, 1 and 3 are the final distances to nodes 1, 5 and 4.

After this, the algorithm processes the two remaining nodes, and the final distances are as follows:



Negative edges

The efficiency of Dijkstra's algorithm is based on the fact that the graph does not contain negative edges. If there is a negative edge, the algorithm may give incorrect results. As an example, consider the following graph:



The shortest path from node 1 to node 4 is $1 \to 3 \to 4$ and its length is 1. However, Dijkstra's algorithm finds the path $1 \to 2 \to 4$ by following the minimum weight edges. The algorithm does not take into account that on the other path, the weight -5 compensates the previous large weight 6.

Implementation

The following implementation of Dijkstra's algorithm calculates the minimum distances from a node x to other nodes of the graph. The graph is stored as adjacency lists so that `adj[a]` contains a pair (b,w) always when there is an edge from node a to node b with weight w.

An efficient implementation of Dijkstra's algorithm requires that it is possible to efficiently find the minimum distance node that has not been processed. An appropriate data structure for this is a priority queue that contains the nodes ordered by their distances. Using a priority queue, the next node to be processed can be retrieved in logarithmic time.

In the following code, the priority queue `q` contains pairs of the form (d,x), meaning that the current distance to node x is d. The list `distance` contains the distance to each node, and the list `processed` indicates whether a node has been processed. Initially the distance is 0 to x and ∞ to all other nodes.

```
import heapq
processed = [False]*N
distance = [INF]*N
distance[x] = 0
q = []
heapq.heappush(q, (0,x))
while (len(q) > 0):
    d, a = heapq.heappop(q)
    if (processed[a]):
        continue
    processed[a] = true
    for (b,w) in adj[a]:
        if (distance[a] + w < distance[b]):
                    distance[b] = distance[a] + w
                    heapq.heappush(q,(distance[b],b))
```

Note that there may be several instances of the same node in the priority queue; however, only the instance with the minimum distance will be processed.

The time complexity of the above implementation is $O(n + m \log m)$, because the algorithm goes through all nodes of the graph and adds for each edge at most one distance to the priority queue.

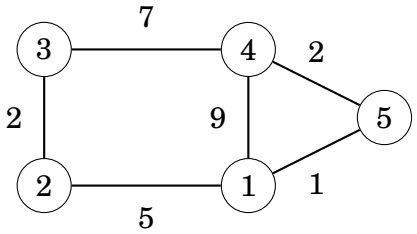
13.3 Floyd–Warshall algorithm

The **Floyd–Warshall algorithm**[3] provides an alternative way to approach the problem of finding shortest paths. Unlike the other algorithms of this chapter, it finds all shortest paths between the nodes in a single run.

The algorithm maintains a two-dimensional list that contains distances between the nodes. First, distances are calculated only using direct edges between the nodes, and after this, the algorithm reduces distances by using intermediate nodes in paths.

Example

Let us consider how the Floyd–Warshall algorithm works in the following graph:



Initially, the distance from each node to itself is 0, and the distance between nodes a and b is x if there is an edge between nodes a and b with weight x. All other distances are infinite.

In this graph, the initial list is as follows:

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	∞	∞
3	∞	2	0	7	∞
4	9	∞	7	0	2
5	1	∞	∞	2	0

The algorithm consists of consecutive rounds. On each round, the algorithm selects a new node that can act as an intermediate node in paths from now on, and distances are reduced using this node.

On the first round, node 1 is the new intermediate node. There is a new path between nodes 2 and 4 with length 14, because node 1 connects them. There is also a new path between nodes 2 and 5 with length 6.

[3]The algorithm is named after R. W. Floyd and S. Warshall who published it independently in 1962 [14, 35].

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	**14**	**6**
3	∞	2	0	7	∞
4	9	**14**	7	0	2
5	1	**6**	∞	2	0

On the second round, node 2 is the new intermediate node. This creates new paths between nodes 1 and 3 and between nodes 3 and 5:

	1	2	3	4	5
1	0	5	**7**	9	1
2	5	0	2	14	6
3	**7**	2	0	7	**8**
4	9	14	7	0	2
5	1	6	**8**	2	0

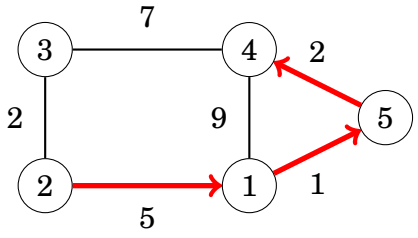
On the third round, node 3 is the new intermediate round. There is a new path between nodes 2 and 4:

	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	**9**	6
3	7	2	0	7	8
4	9	**9**	7	0	2
5	1	6	8	2	0

The algorithm continues like this, until all nodes have been appointed intermediate nodes. After the algorithm has finished, the list contains the minimum distances between any two nodes:

	1	2	3	4	5
1	0	5	7	3	1
2	5	0	2	8	6
3	7	2	0	7	8
4	3	8	7	0	2
5	1	6	8	2	0

For example, the list tells us that the shortest distance between nodes 2 and 4 is 8. This corresponds to the following path:



Implementation

The advantage of the Floyd–Warshall algorithm that it is easy to implement. The following code constructs a distance matrix where `distance`[a][b] is the shortest distance between nodes a and b. First, the algorithm initializes `distance` using the adjacency matrix `adj` of the graph:

```
for i in range(1,n+1):
    for j in range(1,n+1):
        if (i == j):
            distance[i][j] = 0
        elif (adj[i][j] > 0):
            distance[i][j] = adj[i][j]
        else:
            distance[i][j] = INF
```

After this, the shortest distances can be found as follows:

```
for k in range(1,n+1):
    for i in range(1,n+1):
        for j in range(1,n+1):
            distance[i][j] = min(distance[i][j], \
                                 distance[i][k]+distance[k][j])
```

The time complexity of the algorithm is $O(n^3)$, because it contains three nested loops that go through the nodes of the graph.

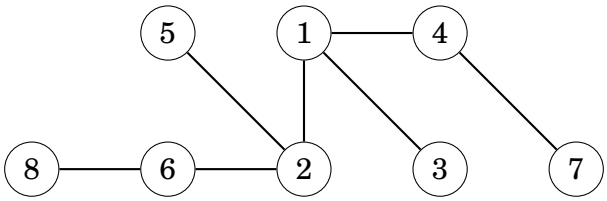
Since the implementation of the Floyd–Warshall algorithm is simple, the algorithm can be a good choice even if it is only needed to find a single shortest path in the graph. However, the algorithm can only be used when the graph is so small that a cubic time complexity is fast enough.

Chapter 14

Tree algorithms

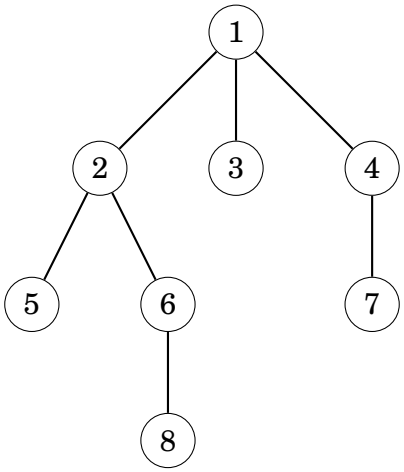
A **tree** is a connected, acyclic graph that consists of n nodes and $n-1$ edges. Removing any edge from a tree divides it into two components, and adding any edge to a tree creates a cycle. Moreover, there is always a unique path between any two nodes of a tree.

For example, the following tree consists of 8 nodes and 7 edges:



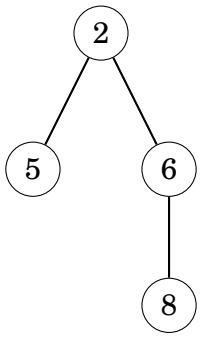
The **leaves** of a tree are the nodes with degree 1, i.e., with only one neighbor. For example, the leaves of the above tree are nodes 3, 5, 7 and 8.

In a **rooted** tree, one of the nodes is appointed the **root** of the tree, and all other nodes are placed underneath the root. For example, in the following tree, node 1 is the root node.



In a rooted tree, the **children** of a node are its lower neighbors, and the **parent** of a node is its upper neighbor. Each node has exactly one parent, except for the root that does not have a parent. For example, in the above tree, the children of node 2 are nodes 5 and 6, and its parent is node 1.

The structure of a rooted tree is *recursive*: each node of the tree acts as the root of a **subtree** that contains the node itself and all nodes that are in the subtrees of its children. For example, in the above tree, the subtree of node 2 consists of nodes 2, 5, 6 and 8:



14.1 Tree traversal

General graph traversal algorithms can be used to traverse the nodes of a tree. However, the traversal of a tree is easier to implement than that of a general graph, because there are no cycles in the tree and it is not possible to reach a node from multiple directions.

The typical way to traverse a tree is to start a depth-first search at an arbitrary node. The following recursive function can be used:

```
def dfs(s, e) {
    // process node s
    for u in adj[s]:
        if (u != e):
            dfs(u, s)
```

The function is given two parameters: the current node s and the previous node e. The purpose of the parameter e is to make sure that the search only moves to nodes that have not been visited yet.

The following function call starts the search at node x:

```
dfs(x, 0);
```

In the first call $e = 0$, because there is no previous node, and it is allowed to proceed to any direction in the tree.

Dynamic programming

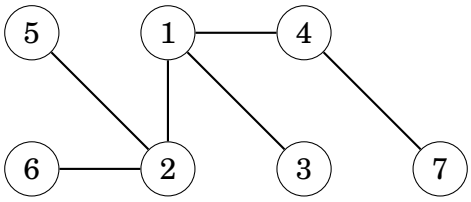
Dynamic programming can be used to calculate some information during a tree traversal. Using dynamic programming, we can, for example, calculate in $O(n)$ time for each node of a rooted tree the number of nodes in its subtree or the length of the longest path from the node to a leaf.

As an example, let us calculate for each node s a value count[s]: the number of nodes in its subtree. The subtree contains the node itself and all nodes in the subtrees of its children, so we can calculate the number of nodes recursively using the following code:

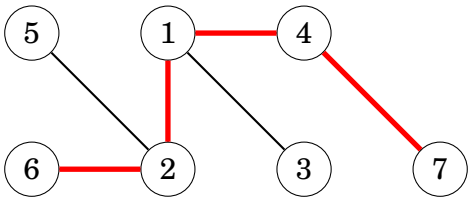
```
def dfs(s, e) {
    count[s] = 1
    for u in adj[s]:
        if (u == e):
            continue
        dfs(u, s)
        count[s] += count[u]
```

14.2 Diameter

The **diameter** of a tree is the maximum length of a path between two nodes. For example, consider the following tree:



The diameter of this tree is 4, which corresponds to the following path:



Note that there may be several maximum-length paths. In the above path, we could replace node 6 with node 5 to obtain another path with length 4.

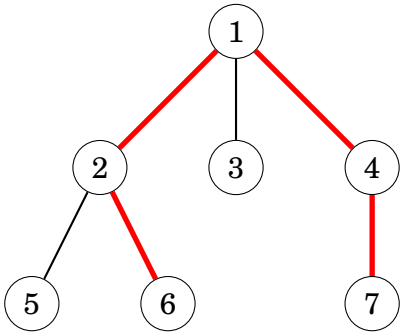
Next we will discuss two $O(n)$ time algorithms for calculating the diameter of a tree. The first algorithm is based on dynamic programming, and the second algorithm uses two depth-first searches.

Algorithm 1

A general way to approach many tree problems is to first root the tree arbitrarily. After this, we can try to solve the problem separately for each subtree. Our first algorithm for calculating the diameter is based on this idea.

An important observation is that every path in a rooted tree has a *highest point*: the highest node that belongs to the path. Thus, we can calculate for each node the length of the longest path whose highest point is the node. One of those paths corresponds to the diameter of the tree.

For example, in the following tree, node 1 is the highest point on the path that corresponds to the diameter:



We calculate for each node x two values:

- `toLeaf`(x): the maximum length of a path from x to any leaf
- `maxLength`(x): the maximum length of a path whose highest point is x

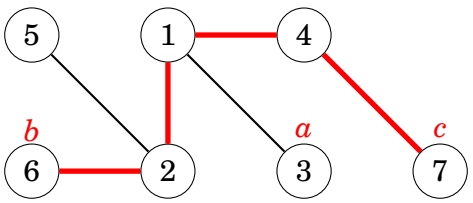
For example, in the above tree, `toLeaf`(1) = 2, because there is a path $1 \to 2 \to 6$, and `maxLength`(1) = 4, because there is a path $6 \to 2 \to 1 \to 4 \to 7$. In this case, `maxLength`(1) equals the diameter.

Dynamic programming can be used to calculate the above values for all nodes in $O(n)$ time. First, to calculate `toLeaf`(x), we go through the children of x, choose a child c with maximum `toLeaf`(c) and add one to this value. Then, to calculate `maxLength`(x), we choose two distinct children a and b such that the sum `toLeaf`(a) + `toLeaf`(b) is maximum and add two to this sum.

Algorithm 2

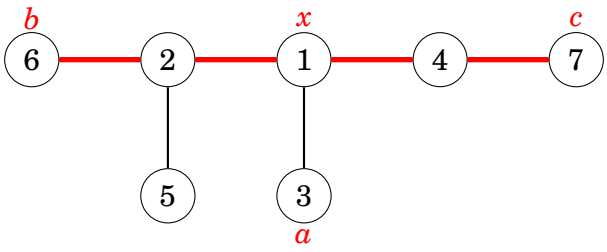
Another efficient way to calculate the diameter of a tree is based on two depth-first searches. First, we choose an arbitrary node a in the tree and find the farthest node b from a. Then, we find the farthest node c from b. The diameter of the tree is the distance between b and c.

In the following graph, a, b and c could be:



This is an elegant method, but why does it work?

It helps to draw the tree differently so that the path that corresponds to the diameter is horizontal, and all other nodes hang from it:

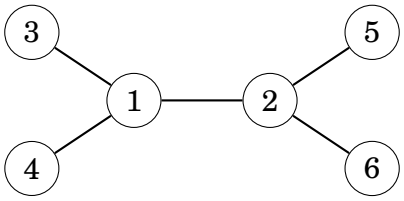


Node x indicates the place where the path from node a joins the path that corresponds to the diameter. The farthest node from a is node b, node c or some other node that is at least as far from node x. Thus, this node is always a valid choice for an endpoint of a path that corresponds to the diameter.

14.3 All longest paths

Our next problem is to calculate for every node in the tree the maximum length of a path that begins at the node. This can be seen as a generalization of the tree diameter problem, because the largest of those lengths equals the diameter of the tree. Also this problem can be solved in $O(n)$ time.

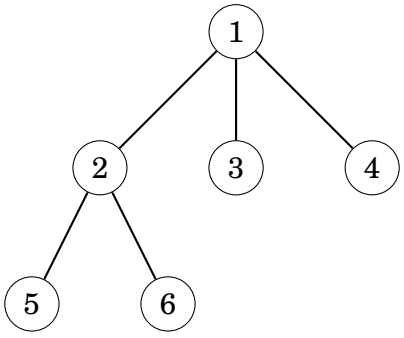
As an example, consider the following tree:



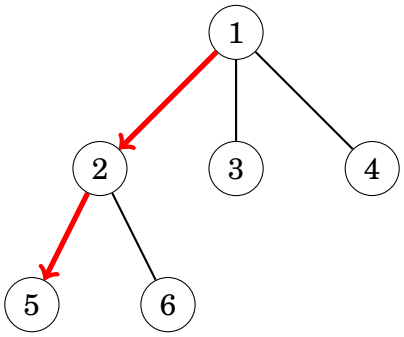
Let $\texttt{maxLength}(x)$ denote the maximum length of a path that begins at node x. For example, in the above tree, $\texttt{maxLength}(4) = 3$, because there is a path $4 \rightarrow 1 \rightarrow 2 \rightarrow 6$. Here is a complete table of the values:

node x	1	2	3	4	5	6
$\texttt{maxLength}(x)$	2	2	3	3	3	3

Also in this problem, a good starting point for solving the problem is to root the tree arbitrarily:

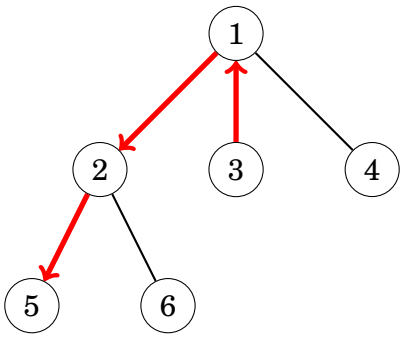


The first part of the problem is to calculate for every node x the maximum length of a path that goes through a child of x. For example, the longest path from node 1 goes through its child 2:

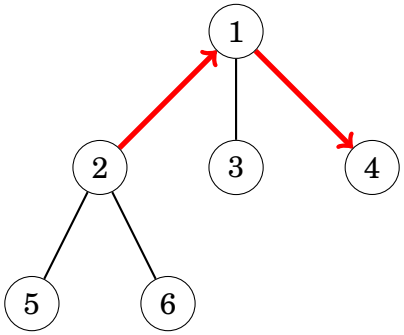


This part is easy to solve in $O(n)$ time, because we can use dynamic programming as we have done previously.

Then, the second part of the problem is to calculate for every node x the maximum length of a path through its parent p. For example, the longest path from node 3 goes through its parent 1:



At first glance, it seems that we should choose the longest path from p. However, this *does not* always work, because the longest path from p may go through x. Here is an example of this situation:



Still, we can solve the second part in $O(n)$ time by storing *two* maximum lengths for each node x:

- $\texttt{maxLength}_1(x)$: the maximum length of a path from x
- $\texttt{maxLength}_2(x)$ the maximum length of a path from x in another direction than the first path

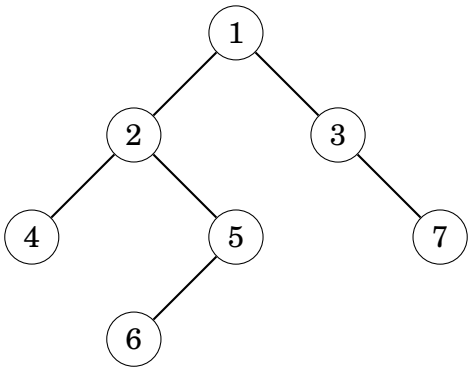
For example, in the above graph, $\texttt{maxLength}_1(1) = 2$ using the path $1 \to 2 \to 5$, and $\texttt{maxLength}_2(1) = 1$ using the path $1 \to 3$.

Finally, if the path that corresponds to $\texttt{maxLength}_1(p)$ goes through x, we conclude that the maximum length is $\texttt{maxLength}_2(p)+1$, and otherwise the maximum length is $\texttt{maxLength}_1(p)+1$.

14.4 Binary trees

A **binary tree** is a rooted tree where each node has a left and right subtree. It is possible that a subtree of a node is empty. Thus, every node in a binary tree has zero, one or two children.

For example, the following tree is a binary tree:



The nodes of a binary tree have three natural orderings that correspond to different ways to recursively traverse the tree:

- **pre-order**: first process the root, then traverse the left subtree, then traverse the right subtree
- **in-order**: first traverse the left subtree, then process the root, then traverse the right subtree
- **post-order**: first traverse the left subtree, then traverse the right subtree, then process the root

For the above tree, the nodes in pre-order are $[1,2,4,5,6,3,7]$, in in-order $[4,2,6,5,1,3,7]$ and in post-order $[4,6,5,2,7,3,1]$.

If we know the pre-order and in-order of a tree, we can reconstruct the exact structure of the tree. For example, the above tree is the only possible tree with pre-order $[1,2,4,5,6,3,7]$ and in-order $[4,2,6,5,1,3,7]$. In a similar way, the post-order and in-order also determine the structure of a tree.

However, the situation is different if we only know the pre-order and post-order of a tree. In this case, there may be more than one tree that match the orderings. For example, in both of the trees

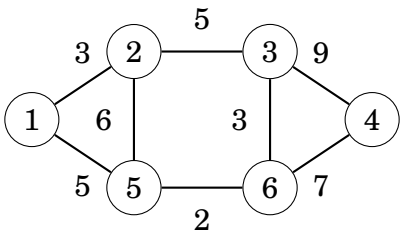
the pre-order is $[1,2]$ and the post-order is $[2,1]$, but the structures of the trees are different.

Chapter 15

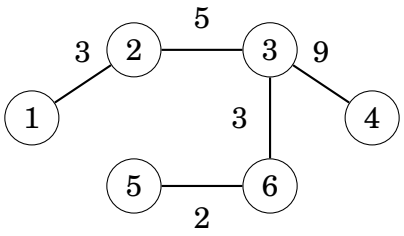
Spanning trees

A **spanning tree** of a graph consists of all nodes of the graph and some of the edges of the graph so that there is a path between any two nodes. Like trees in general, spanning trees are connected and acyclic. Usually there are several ways to construct a spanning tree.

For example, consider the following graph:

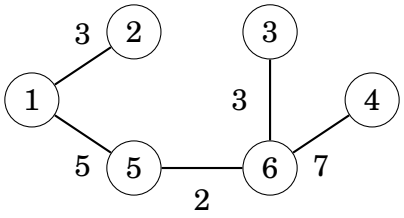


One spanning tree for the graph is as follows:

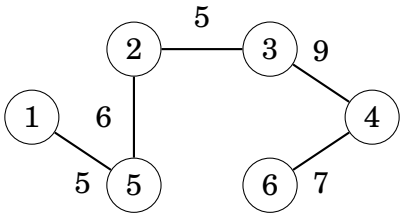


The weight of a spanning tree is the sum of its edge weights. For example, the weight of the above spanning tree is $3+5+9+3+2=22$.

A **minimum spanning tree** is a spanning tree whose weight is as small as possible. The weight of a minimum spanning tree for the example graph is 20, and such a tree can be constructed as follows:



In a similar way, a **maximum spanning tree** is a spanning tree whose weight is as large as possible. The weight of a maximum spanning tree for the example graph is 32:



Note that a graph may have several minimum and maximum spanning trees, so the trees are not unique.

It turns out that several greedy methods can be used to construct minimum and maximum spanning trees. In this chapter, we discuss two algorithms that process the edges of the graph ordered by their weights. We focus on finding minimum spanning trees, but the same algorithms can find maximum spanning trees by processing the edges in reverse order.

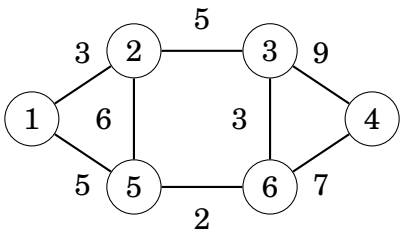
15.1 Kruskal's algorithm

In **Kruskal's algorithm**[1], the initial spanning tree only contains the nodes of the graph and does not contain any edges. Then the algorithm goes through the edges ordered by their weights, and always adds an edge to the tree if it does not create a cycle.

The algorithm maintains the components of the tree. Initially, each node of the graph belongs to a separate component. Always when an edge is added to the tree, two components are joined. Finally, all nodes belong to the same component, and a minimum spanning tree has been found.

Example

Let us consider how Kruskal's algorithm processes the following graph:



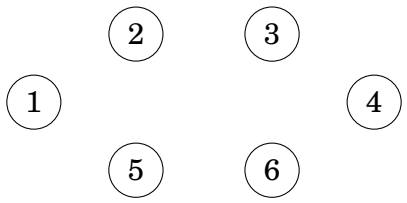
The first step of the algorithm is to sort the edges in increasing order of their weights. The result is the following list:

[1]The algorithm was published in 1956 by J. B. Kruskal [24].

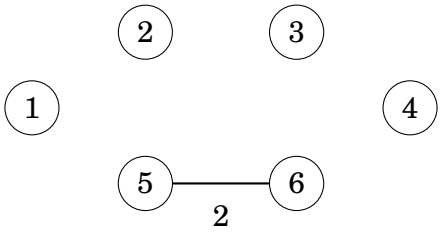
edge	weight
5–6	2
1–2	3
3–6	3
1–5	5
2–3	5
2–5	6
4–6	7
3–4	9

After this, the algorithm goes through the list and adds each edge to the tree if it joins two separate components.

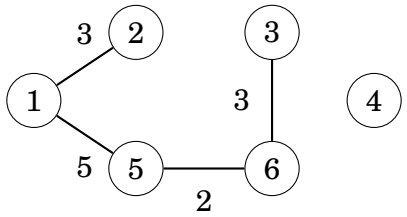
Initially, each node is in its own component:



The first edge to be added to the tree is the edge 5–6 that creates a component $\{5,6\}$ by joining the components $\{5\}$ and $\{6\}$:



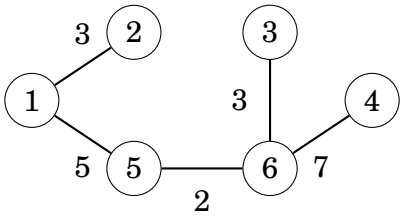
After this, the edges 1–2, 3–6 and 1–5 are added in a similar way:



After those steps, most components have been joined and there are two components in the tree: $\{1,2,3,5,6\}$ and $\{4\}$.

The next edge in the list is the edge 2–3, but it will not be included in the tree, because nodes 2 and 3 are already in the same component. For the same reason, the edge 2–5 will not be included in the tree.

Finally, the edge 4–6 will be included in the tree:

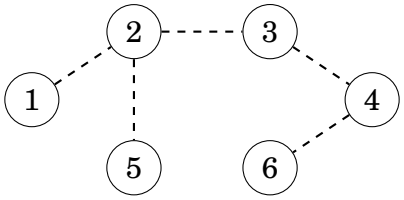


After this, the algorithm will not add any new edges, because the graph is connected and there is a path between any two nodes. The resulting graph is a minimum spanning tree with weight $2+3+3+5+7=20$.

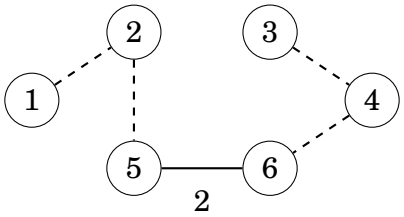
Why does this work?

It is a good question why Kruskal's algorithm works. Why does the greedy strategy guarantee that we will find a minimum spanning tree?

Let us see what happens if the minimum weight edge of the graph is *not* included in the spanning tree. For example, suppose that a spanning tree for the previous graph would not contain the minimum weight edge 5–6. We do not know the exact structure of such a spanning tree, but in any case it has to contain some edges. Assume that the tree would be as follows:



However, it is not possible that the above tree would be a minimum spanning tree for the graph. The reason for this is that we can remove an edge from the tree and replace it with the minimum weight edge 5–6. This produces a spanning tree whose weight is *smaller*:



For this reason, it is always optimal to include the minimum weight edge in the tree to produce a minimum spanning tree. Using a similar argument, we can show that it is also optimal to add the next edge in weight order to the tree, and so on. Hence, Kruskal's algorithm works correctly and always produces a minimum spanning tree.

Implementation

When implementing Kruskal's algorithm, it is convenient to use the edge list representation of the graph. The first phase of the algorithm sorts the edges in the list in $O(m \log m)$ time. After this, the second phase of the algorithm builds the minimum spanning tree as follows:

```
for (...):
  if (!same(a,b)) unite(a,b)
```

The loop goes through the edges in the list and always processes an edge a–b where a and b are two nodes. Two functions are needed: the function `same` determines if a and b are in the same component, and the function `unite` joins the components that contain a and b.

The problem is how to efficiently implement the functions `same` and `unite`. One possibility is to implement the function `same` as a graph traversal and check if we can get from node a to node b. However, the time complexity of such a function would be $O(n+m)$ and the resulting algorithm would be slow, because the function `same` will be called for each edge in the graph.

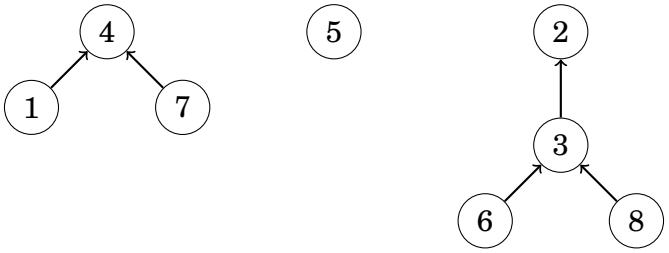
We will solve the problem using a union-find structure that implements both functions in $O(\log n)$ time. Thus, the time complexity of Kruskal's algorithm will be $O(m \log n)$ after sorting the edge list.

15.2 Union-find structure

A **union-find structure** maintains a collection of sets. The sets are disjoint, so no element belongs to more than one set. Two $O(\log n)$ time operations are supported: the `unite` operation joins two sets, and the `find` operation finds the representative of the set that contains a given element[2].

Structure

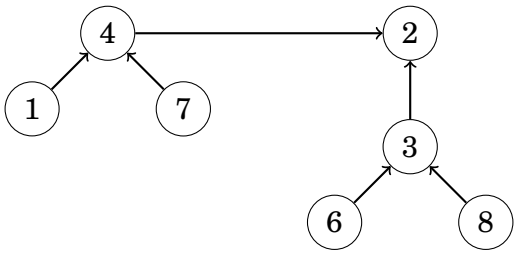
In a union-find structure, one element in each set is the representative of the set, and there is a chain from any other element of the set to the representative. For example, assume that the sets are $\{1,4,7\}$, $\{5\}$ and $\{2,3,6,8\}$:



[2]The structure presented here was introduced in 1971 by J. D. Hopcroft and J. D. Ullman [18]. Later, in 1975, R. E. Tarjan studied a more sophisticated variant of the structure [31] that is discussed in many algorithm textbooks nowadays.

In this case the representatives of the sets are 4, 5 and 2. We can find the representative of any element by following the chain that begins at the element. For example, the element 2 is the representative for the element 6, because we follow the chain $6 \rightarrow 3 \rightarrow 2$. Two elements belong to the same set exactly when their representatives are the same.

Two sets can be joined by connecting the representative of one set to the representative of the other set. For example, the sets $\{1,4,7\}$ and $\{2,3,6,8\}$ can be joined as follows:



The resulting set contains the elements $\{1,2,3,4,6,7,8\}$. From this on, the element 2 is the representative for the entire set and the old representative 4 points to the element 2.

The efficiency of the union-find structure depends on how the sets are joined. It turns out that we can follow a simple strategy: always connect the representative of the *smaller* set to the representative of the *larger* set (or if the sets are of equal size, we can make an arbitrary choice). Using this strategy, the length of any chain will be $O(\log n)$, so we can find the representative of any element efficiently by following the corresponding chain.

Implementation

The union-find structure can be implemented using lists. In the following implementation, the list `link` contains for each element the next element in the chain or the element itself if it is a representative, and the list `size` indicates for each representative the size of the corresponding set.

Initially, each element belongs to a separate set:

```
link = [i for i in range(n)]
size = [1 for i in range(n)]
```

The function `find` returns the representative for an element x. The representative can be found by following the chain that begins at x.

```
def find(x):
    while (x != link[x]):
        x = link[x]
    return x
```

The function `same` checks whether elements a and b belong to the same set. This can easily be done by using the function `find`:

```
def same(a, b):
    return (find(a) == find(b))
```

The function `unite` joins the sets that contain elements a and b (the elements have to be in different sets). The function first finds the representatives of the sets and then connects the smaller set to the larger set.

```
def unite(int a, int b):
    global size, link
    a = find(a)
    b = find(b)
    if (size[a] > size[b]):
        size[a] += size[b]
        link[b] = a
    else:
        size[b] += size[a]
        link[a] = b
```

The time complexity of the function `find` is $O(\log n)$ assuming that the length of each chain is $O(\log n)$. In this case, the functions `same` and `unite` also work in $O(\log n)$ time. The function `unite` makes sure that the length of each chain is $O(\log n)$ by connecting the smaller set to the larger set.

15.3 Prim's algorithm

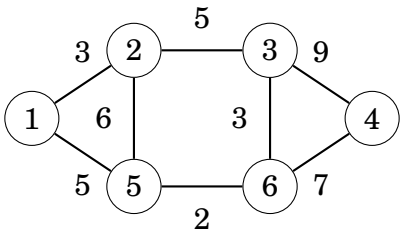
Prim's algorithm[3] is an alternative method for finding a minimum spanning tree. The algorithm first adds an arbitrary node to the tree. After this, the algorithm always chooses a minimum-weight edge that adds a new node to the tree. Finally, all nodes have been added to the tree and a minimum spanning tree has been found.

Prim's algorithm resembles Dijkstra's algorithm. The difference is that Dijkstra's algorithm always selects an edge whose distance from the starting node is minimum, but Prim's algorithm simply selects the minimum weight edge that adds a new node to the tree.

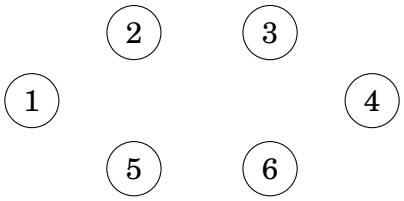
Example

Let us consider how Prim's algorithm works in the following graph:

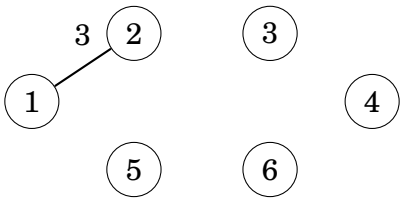
[3]The algorithm is named after R. C. Prim who published it in 1957 [27]. However, the same algorithm was discovered already in 1930 by V. Jarník.



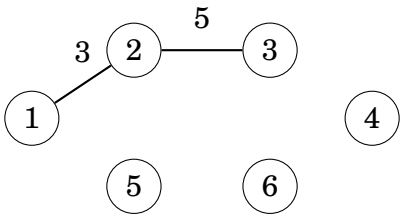
Initially, there are no edges between the nodes:



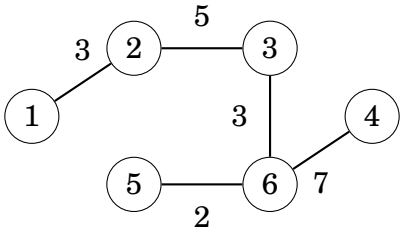
An arbitrary node can be the starting node, so let us choose node 1. First, we add node 2 that is connected by an edge of weight 3:



After this, there are two edges with weight 5, so we can add either node 3 or node 5 to the tree. Let us add node 3 first:



The process continues until all nodes have been included in the tree:



Implementation

Like Dijkstra's algorithm, Prim's algorithm can be efficiently implemented using a priority queue. The priority queue should contain all nodes that can be connected to the current component using a single edge, in increasing order of the weights of the corresponding edges.

The time complexity of Prim's algorithm is $O(n+m \log m)$ that equals the time complexity of Dijkstra's algorithm. In practice, Prim's and Kruskal's algorithms are both efficient, and the choice of the algorithm is a matter of taste. Still, most competitive programmers use Kruskal's algorithm.

Chapter 16

Directed graphs

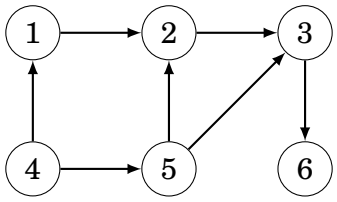
In this chapter, we focus on two classes of directed graphs:

- **Acyclic graphs**: There are no cycles in the graph, so there is no path from any node to itself[1].
- **Successor graphs**: The outdegree of each node is 1, so each node has a unique successor.

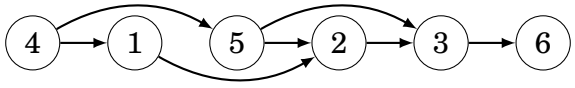
It turns out that in both cases, we can design efficient algorithms that are based on the special properties of the graphs.

16.1 Topological sorting

A **topological sort** is an ordering of the nodes of a directed graph such that if there is a path from node a to node b, then node a appears before node b in the ordering. For example, for the graph



one topological sort is $[4,1,5,2,3,6]$:



An acyclic graph always has a topological sort. However, if the graph contains a cycle, it is not possible to form a topological sort, because no node of the cycle can appear before the other nodes of the cycle in the ordering. It turns out that depth-first search can be used to both check if a directed graph contains a cycle and, if it does not contain a cycle, to construct a topological sort.

[1] Directed acyclic graphs are sometimes called DAGs.

Algorithm

The idea is to go through the nodes of the graph and always begin a depth-first search at the current node if it has not been processed yet. During the searches, the nodes have three possible states:

- state 0: the node has not been processed (white)
- state 1: the node is under processing (light gray)
- state 2: the node has been processed (dark gray)

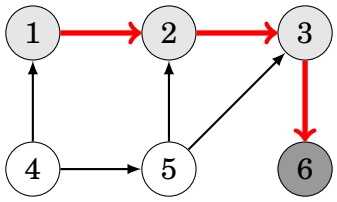
Initially, the state of each node is 0. When a search reaches a node for the first time, its state becomes 1. Finally, after all successors of the node have been processed, its state becomes 2.

If the graph contains a cycle, we will find this out during the search, because sooner or later we will arrive at a node whose state is 1. In this case, it is not possible to construct a topological sort.

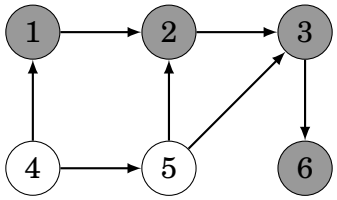
If the graph does not contain a cycle, we can construct a topological sort by adding each node to a list when the state of the node becomes 2. This list in reverse order is a topological sort.

Example 1

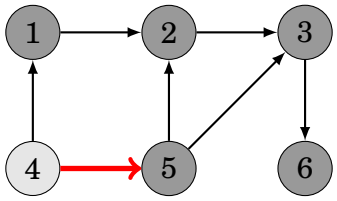
In the example graph, the search first proceeds from node 1 to node 6:



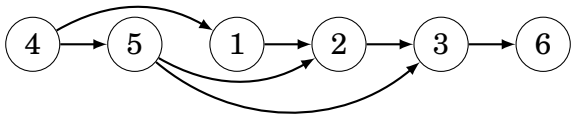
Now node 6 has been processed, so it is added to the list. After this, also nodes 3, 2 and 1 are added to the list:



At this point, the list is $[6,3,2,1]$. The next search begins at node 4:



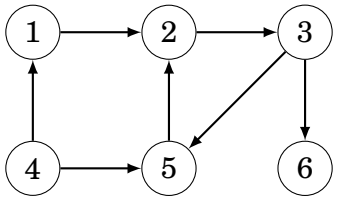
Thus, the final list is $[6,3,2,1,5,4]$. We have processed all nodes, so a topological sort has been found. The topological sort is the reverse list $[4,5,1,2,3,6]$:



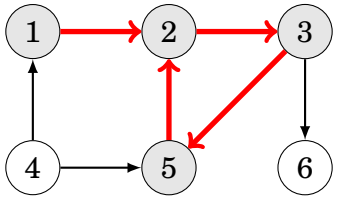
Note that a topological sort is not unique, and there can be several topological sorts for a graph.

Example 2

Let us now consider a graph for which we cannot construct a topological sort, because the graph contains a cycle:



The search proceeds as follows:



The search reaches node 2 whose state is 1, which means that the graph contains a cycle. In this example, there is a cycle $2 \rightarrow 3 \rightarrow 5 \rightarrow 2$.

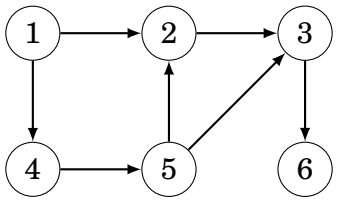
16.2 Dynamic programming

If a directed graph is acyclic, dynamic programming can be applied to it. For example, we can efficiently solve the following problems concerning paths from a starting node to an ending node:

- how many different paths are there?
- what is the shortest/longest path?
- what is the minimum/maximum number of edges in a path?
- which nodes certainly appear in any path?

Counting the number of paths

As an example, let us calculate the number of paths from node 1 to node 6 in the following graph:



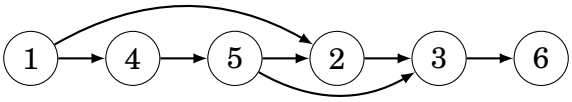
There are a total of three such paths:

- $1 \to 2 \to 3 \to 6$
- $1 \to 4 \to 5 \to 2 \to 3 \to 6$
- $1 \to 4 \to 5 \to 3 \to 6$

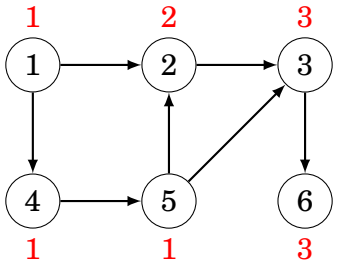
Let $\mathrm{paths}(x)$ denote the number of paths from node 1 to node x. As a base case, $\mathrm{paths}(1) = 1$. Then, to calculate other values of $\mathrm{paths}(x)$, we may use the recursion

$$\mathrm{paths}(x) = \mathrm{paths}(a_1) + \mathrm{paths}(a_2) + \cdots + \mathrm{paths}(a_k)$$

where $a_1, a_2, \ldots, a_k$ are the nodes from which there is an edge to x. Since the graph is acyclic, the values of $\mathrm{paths}(x)$ can be calculated in the order of a topological sort. A topological sort for the above graph is as follows:



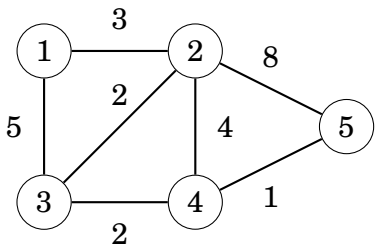
Hence, the numbers of paths are as follows:



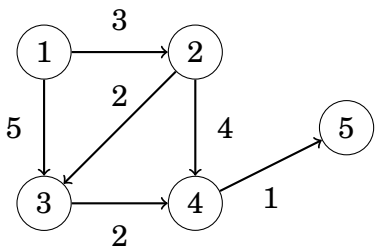
For example, to calculate the value of $\mathrm{paths}(3)$, we can use the formula $\mathrm{paths}(2) + \mathrm{paths}(5)$, because there are edges from nodes 2 and 5 to node 3. Since $\mathrm{paths}(2) = 2$ and $\mathrm{paths}(5) = 1$, we conclude that $\mathrm{paths}(3) = 3$.

Extending Dijkstra's algorithm

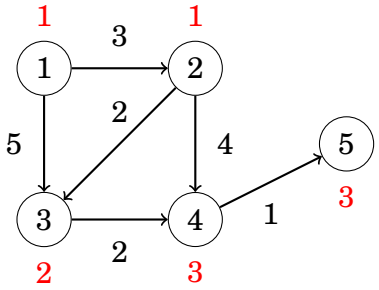
A by-product of Dijkstra's algorithm is a directed, acyclic graph that indicates for each node of the original graph the possible ways to reach the node using a shortest path from the starting node. Dynamic programming can be applied to that graph. For example, in the graph



the shortest paths from node 1 may use the following edges:



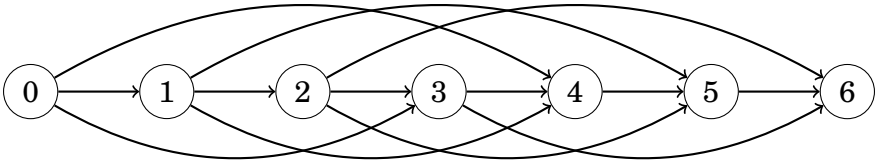
Now we can, for example, calculate the number of shortest paths from node 1 to node 5 using dynamic programming:



Representing problems as graphs

Actually, any dynamic programming problem can be represented as a directed, acyclic graph. In such a graph, each node corresponds to a dynamic programming state and the edges indicate how the states depend on each other.

As an example, consider the problem of forming a sum of money n using coins $\{c_1, c_2, \ldots, c_k\}$. In this problem, we can construct a graph where each node corresponds to a sum of money, and the edges show how the coins can be chosen. For example, for coins $\{1, 3, 4\}$ and $n = 6$, the graph is as follows:



Using this representation, the shortest path from node 0 to node n corresponds to a solution with the minimum number of coins, and the total number of paths from node 0 to node n equals the total number of solutions.

16.3 Successor paths

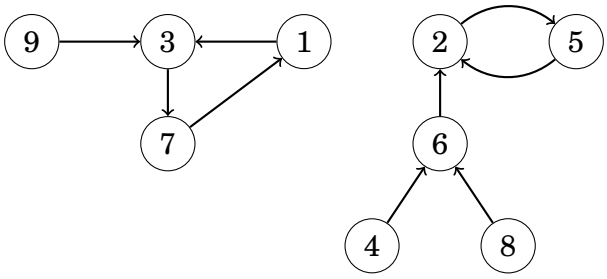
For the rest of the chapter, we will focus on **successor graphs**. In those graphs, the outdegree of each node is 1, i.e., exactly one edge starts at each node. A successor graph consists of one or more components, each of which contains one cycle and some paths that lead to it.

Successor graphs are sometimes called **functional graphs**. The reason for this is that any successor graph corresponds to a function that defines the edges of the graph. The parameter for the function is a node of the graph, and the function gives the successor of that node.

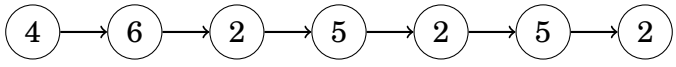
For example, the function

x	1	2	3	4	5	6	7	8	9
succ(x)	3	5	7	6	2	2	1	6	3

defines the following graph:



Since each node of a successor graph has a unique successor, we can also define a function succ(x,k) that gives the node that we will reach if we begin at node x and walk k steps forward. For example, in the above graph succ$(4,6) = 2$, because we will reach node 2 by walking 6 steps from node 4:



A straightforward way to calculate a value of succ(x,k) is to start at node x and walk k steps forward, which takes $O(k)$ time. However, using preprocessing, any value of succ(x,k) can be calculated in only $O(\log k)$ time.

The idea is to precalculate all values of succ(x,k) where k is a power of two and at most u, where u is the maximum number of steps we will ever walk. This can be efficiently done, because we can use the following recursion:

$$\text{succ}(x,k) = \begin{cases} \text{succ}(x) & k = 1 \\ \text{succ}(\text{succ}(x,k/2),k/2) & k > 1 \end{cases}$$

Precalculating the values takes $O(n \log u)$ time, because $O(\log u)$ values are calculated for each node. In the above graph, the first values are as follows:

x	1	2	3	4	5	6	7	8	9
succ$(x,1)$	3	5	7	6	2	2	1	6	3
succ$(x,2)$	7	2	1	2	5	5	3	2	7
succ$(x,4)$	3	2	7	2	5	5	1	2	3
succ$(x,8)$	7	2	1	2	5	5	3	2	7
$\cdots$									

After this, any value of $\text{succ}(x,k)$ can be calculated by presenting the number of steps k as a sum of powers of two. For example, if we want to calculate the value of $\text{succ}(x,11)$, we first form the representation $11 = 8+2+1$. Using that,

$$\text{succ}(x,11) = \text{succ}(\text{succ}(\text{succ}(x,8),2),1).$$

For example, in the previous graph

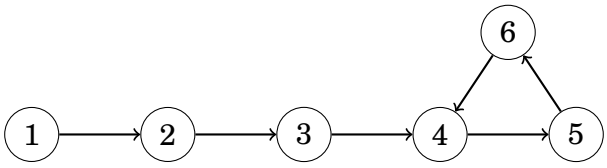
$$\text{succ}(4,11) = \text{succ}(\text{succ}(\text{succ}(4,8),2),1) = 5.$$

Such a representation always consists of $O(\log k)$ parts, so calculating a value of $\text{succ}(x,k)$ takes $O(\log k)$ time.

16.4 Cycle detection

Consider a successor graph that only contains a path that ends in a cycle. We may ask the following questions: if we begin our walk at the starting node, what is the first node in the cycle and how many nodes does the cycle contain?

For example, in the graph



we begin our walk at node 1, the first node that belongs to the cycle is node 4, and the cycle consists of three nodes (4, 5 and 6).

A simple way to detect the cycle is to walk in the graph and keep track of all nodes that have been visited. Once a node is visited for the second time, we can conclude that the node is the first node in the cycle. This method works in $O(n)$ time and also uses $O(n)$ memory.

However, there are better algorithms for cycle detection. The time complexity of such algorithms is still $O(n)$, but they only use $O(1)$ memory. This is an important improvement if n is large. Next we will discuss Floyd's algorithm that achieves these properties.

Floyd's algorithm

Floyd's algorithm[2] walks forward in the graph using two pointers a and b. Both pointers begin at a node x that is the starting node of the graph. Then, on each turn, the pointer a walks one step forward and the pointer b walks two steps forward. The process continues until the pointers meet each other:

```
a = succ(x)
b = succ(succ(x))
while (a != b):
    a = succ(a)
    b = succ(succ(b))
```

At this point, the pointer a has walked k steps and the pointer b has walked $2k$ steps, so the length of the cycle divides k. Thus, the first node that belongs to the cycle can be found by moving the pointer a to node x and advancing the pointers step by step until they meet again.

```
a = x
while (a != b):
    a = succ(a)
    b = succ(b)
first = a
```

After this, the length of the cycle can be calculated as follows:

```
b = succ(a)
length = 1
while (a != b):
    b = succ(b)
    length += 1
```

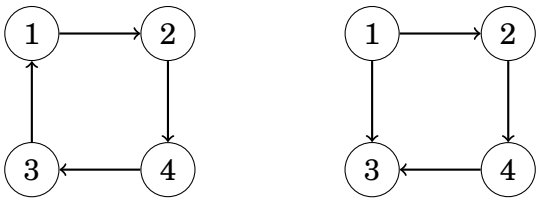
[2]The idea of the algorithm is mentioned in [22] and attributed to R. W. Floyd; however, it is not known if Floyd actually discovered the algorithm.

Chapter 17

Strong connectivity

In a directed graph, the edges can be traversed in one direction only, so even if the graph is connected, this does not guarantee that there would be a path from a node to another node. For this reason, it is meaningful to define a new concept that requires more than connectivity.

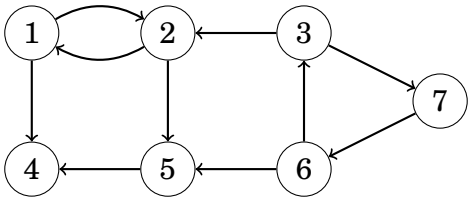
A graph is **strongly connected** if there is a path from any node to all other nodes in the graph. For example, in the following picture, the left graph is strongly connected while the right graph is not.



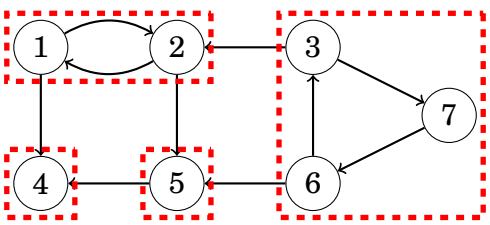
The right graph is not strongly connected because, for example, there is no path from node 2 to node 1.

The **strongly connected components** of a graph divide the graph into strongly connected parts that are as large as possible. The strongly connected components form an acyclic **component graph** that represents the deep structure of the original graph.

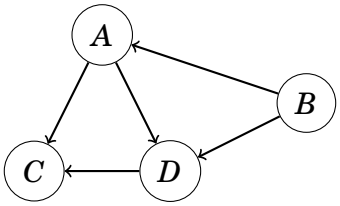
For example, for the graph



the strongly connected components are as follows:



The corresponding component graph is as follows:



The components are $A = \{1,2\}$, $B = \{3,6,7\}$, $C = \{4\}$ and $D = \{5\}$.

A component graph is an acyclic, directed graph, so it is easier to process than the original graph. Since the graph does not contain cycles, we can always construct a topological sort and use dynamic programming techniques like those presented in Chapter 16.

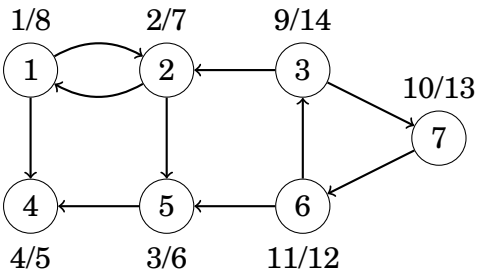
17.1 Kosaraju's algorithm

Kosaraju's algorithm[1] is an efficient method for finding the strongly connected components of a directed graph. The algorithm performs two depth-first searches: the first search constructs a list of nodes according to the structure of the graph, and the second search forms the strongly connected components.

Search 1

The first phase of Kosaraju's algorithm constructs a list of nodes in the order in which a depth-first search processes them. The algorithm goes through the nodes, and begins a depth-first search at each unprocessed node. Each node will be added to the list after it has been processed.

In the example graph, the nodes are processed in the following order:



The notation x/y means that processing the node started at time x and finished at time y. Thus, the corresponding list is as follows:

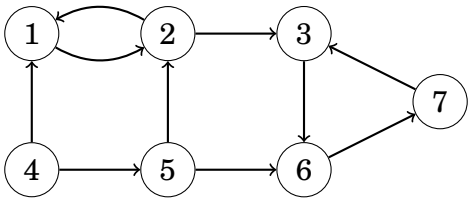
[1] According to [1], S. R. Kosaraju invented this algorithm in 1978 but did not publish it. In 1981, the same algorithm was rediscovered and published by M. Sharir [29].

node	processing time
4	5
5	6
2	7
1	8
6	12
7	13
3	14

Search 2

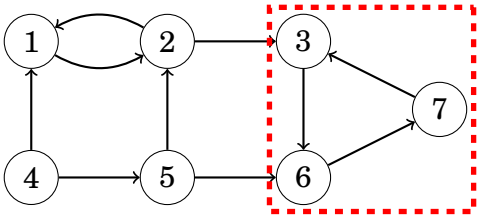
The second phase of the algorithm forms the strongly connected components of the graph. First, the algorithm reverses every edge in the graph. This guarantees that during the second search, we will always find strongly connected components that do not have extra nodes.

After reversing the edges, the example graph is as follows:



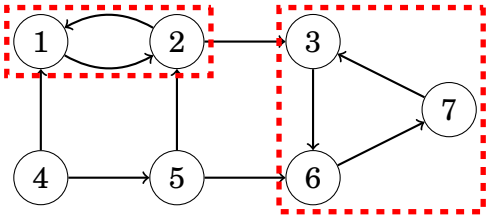
After this, the algorithm goes through the list of nodes created by the first search, in *reverse* order. If a node does not belong to a component, the algorithm creates a new component and starts a depth-first search that adds all new nodes found during the search to the new component.

In the example graph, the first component begins at node 3:

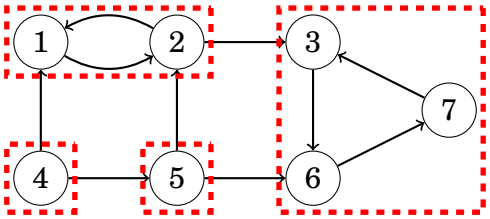


Note that since all edges are reversed, the component does not ”leak” to other parts in the graph.

The next nodes in the list are nodes 7 and 6, but they already belong to a component, so the next new component begins at node 1:



Finally, the algorithm processes nodes 5 and 4 that create the remaining strongly connected components:



The time complexity of the algorithm is $O(n+m)$, because the algorithm performs two depth-first searches.

17.2 2SAT problem

Strong connectivity is also linked with the **2SAT problem**[2]. In this problem, we are given a logical formula

$$(a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge \cdots \wedge (a_m \vee b_m),$$

where each a_i and b_i is either a logical variable ($x_1, x_2, \ldots, x_n$) or a negation of a logical variable ($\neg x_1, \neg x_2, \ldots, \neg x_n$). The symbols "$\wedge$" and "$\vee$" denote logical operators "and" and "or". Our task is to assign each variable a value so that the formula is true, or state that this is not possible.

For example, the formula

$$L_1 = (x_2 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_4)$$

is true when the variables are assigned as follows:

$$\begin{cases} x_1 = \text{false} \\ x_2 = \text{false} \\ x_3 = \text{true} \\ x_4 = \text{true} \end{cases}$$

[2]The algorithm presented here was introduced in [2]. There is also another well-known linear-time algorithm [10] that is based on backtracking.

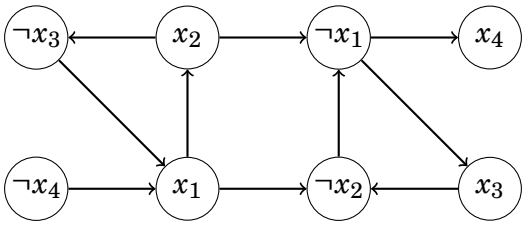
However, the formula

$$L_2 = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$$

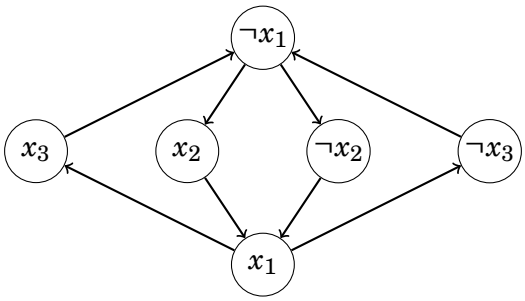
is always false, regardless of how we assign the values. The reason for this is that we cannot choose a value for x_1 without creating a contradiction. If x_1 is false, both x_2 and $\neg x_2$ should be true which is impossible, and if x_1 is true, both x_3 and $\neg x_3$ should be true which is also impossible.

The 2SAT problem can be represented as a graph whose nodes correspond to variables x_i and negations $\neg x_i$, and edges determine the connections between the variables. Each pair $(a_i \vee b_i)$ generates two edges: $\neg a_i \to b_i$ and $\neg b_i \to a_i$. This means that if a_i does not hold, b_i must hold, and vice versa.

The graph for the formula L_1 is:



And the graph for the formula L_2 is:



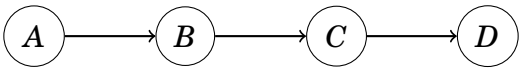
The structure of the graph tells us whether it is possible to assign the values of the variables so that the formula is true. It turns out that this can be done exactly when there are no nodes x_i and $\neg x_i$ such that both nodes belong to the same strongly connected component. If there are such nodes, the graph contains a path from x_i to $\neg x_i$ and also a path from $\neg x_i$ to x_i, so both x_i and $\neg x_i$ should be true which is not possible.

In the graph of the formula L_1 there are no nodes x_i and $\neg x_i$ such that both nodes belong to the same strongly connected component, so a solution exists. In the graph of the formula L_2 all nodes belong to the same strongly connected component, so a solution does not exist.

If a solution exists, the values for the variables can be found by going through the nodes of the component graph in a reverse topological sort order. At each step, we process a component that does not contain edges that lead to an unprocessed component. If the variables in the component have not been assigned values, their values will be determined according to the values in the component, and if

they already have values, they remain unchanged. The process continues until each variable has been assigned a value.

The component graph for the formula L_1 is as follows:



The components are $A = \{\neg x_4\}$, $B = \{x_1, x_2, \neg x_3\}$, $C = \{\neg x_1, \neg x_2, x_3\}$ and $D = \{x_4\}$. When constructing the solution, we first process the component D where x_4 becomes true. After this, we process the component C where x_1 and x_2 become false and x_3 becomes true. All variables have been assigned values, so the remaining components A and B do not change the variables.

Note that this method works, because the graph has a special structure: if there are paths from node x_i to node x_j and from node x_j to node $\neg x_j$, then node x_i never becomes true. The reason for this is that there is also a path from node $\neg x_j$ to node $\neg x_i$, and both x_i and x_j become false.

A more difficult problem is the **3SAT problem**, where each part of the formula is of the form $(a_i \vee b_i \vee c_i)$. This problem is NP-hard, so no efficient algorithm for solving the problem is known.

Chapter 18

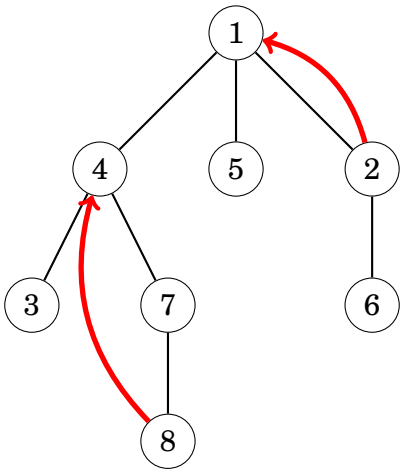
Tree queries

This chapter discusses techniques for processing queries on subtrees and paths of a rooted tree. For example, such queries are:

- what is the kth ancestor of a node?
- what is the sum of values in the subtree of a node?
- what is the sum of values on a path between two nodes?
- what is the lowest common ancestor of two nodes?

18.1 Finding ancestors

The kth **ancestor** of a node x in a rooted tree is the node that we will reach if we move k levels up from x. Let $\texttt{ancestor}(x,k)$ denote the kth ancestor of a node x (or 0 if there is no such an ancestor). For example, in the following tree, $\texttt{ancestor}(2,1) = 1$ and $\texttt{ancestor}(8,2) = 4$.



An easy way to calculate any value of $\texttt{ancestor}(x,k)$ is to perform a sequence of k moves in the tree. However, the time complexity of this method is $O(k)$, which may be slow, because a tree of n nodes may have a chain of n nodes.

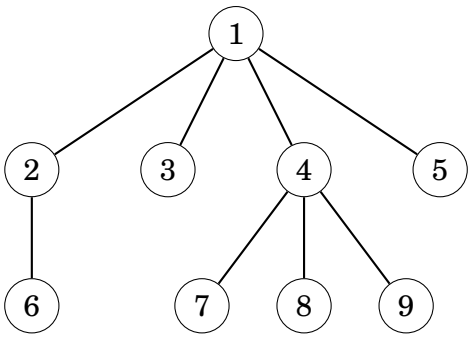
Fortunately, using a technique similar to that used in Chapter 16.3, any value of $\texttt{ancestor}(x,k)$ can be efficiently calculated in $O(\log k)$ time after preprocessing. The idea is to precalculate all values $\texttt{ancestor}(x,k)$ where $k \leq n$ is a power of two. For example, the values for the above tree are as follows:

x	1	2	3	4	5	6	7	8
$\texttt{ancestor}(x,1)$	0	1	4	1	1	2	4	7
$\texttt{ancestor}(x,2)$	0	0	1	0	0	1	1	4
$\texttt{ancestor}(x,4)$	0	0	0	0	0	0	0	0
$\cdots$								

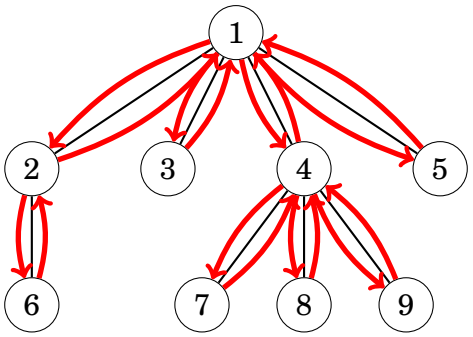
The preprocessing takes $O(n \log n)$ time, because $O(\log n)$ values are calculated for each node. After this, any value of $\texttt{ancestor}(x,k)$ can be calculated in $O(\log k)$ time by representing k as a sum where each term is a power of two.

18.2 Subtrees and paths

A **tree traversal array** contains the nodes of a rooted tree in the order in which a depth-first search from the root node visits them. For example, in the tree



a depth-first search proceeds as follows:



Hence, the corresponding tree traversal array is as follows:

1	2	6	3	4	7	8	9	5

Subtree queries

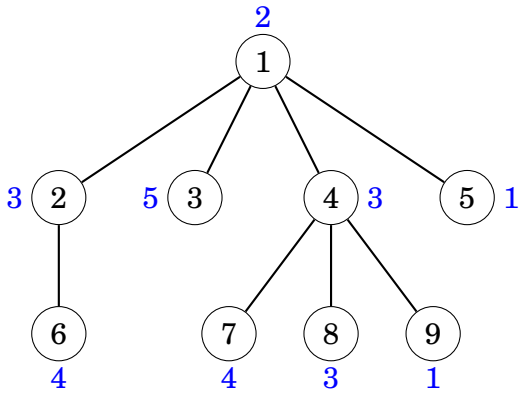
Each subtree of a tree corresponds to a subarray of the tree traversal array such that the first element of the subarray is the root node. For example, the following subarray contains the nodes of the subtree of node 4:

1	2	6	3	4	7	8	9	5

Using this fact, we can efficiently process queries that are related to subtrees of a tree. As an example, consider a problem where each node is assigned a value, and our task is to support the following queries:

- update the value of a node
- calculate the sum of values in the subtree of a node

Consider the following tree where the blue numbers are the values of the nodes. For example, the sum of the subtree of node 4 is $3+4+3+1=11$.



The idea is to construct a tree traversal array that contains three values for each node: the identifier of the node, the size of the subtree, and the value of the node. For example, the array for the above tree is as follows:

node id	1	2	6	3	4	7	8	9	5
subtree size	9	2	1	1	4	1	1	1	1
node value	2	3	4	5	3	4	3	1	1

Using this array, we can calculate the sum of values in any subtree by first finding out the size of the subtree and then the values of the corresponding nodes. For example, the values in the subtree of node 4 can be found as follows:

node id	1	2	6	3	4	7	8	9	5
subtree size	9	2	1	1	4	1	1	1	1
node value	2	3	4	5	3	4	3	1	1

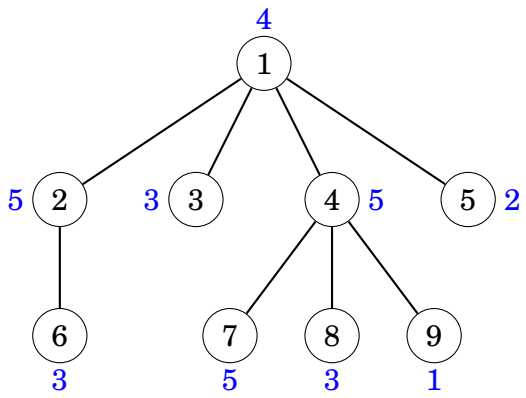
To answer the queries efficiently, it suffices to store the values of the nodes in a binary indexed or segment tree. After this, we can both update a value and calculate the sum of values in $O(\log n)$ time.

Path queries

Using a tree traversal array, we can also efficiently calculate sums of values on paths from the root node to any node of the tree. Consider a problem where our task is to support the following queries:

- change the value of a node
- calculate the sum of values on a path from the root to a node

For example, in the following tree, the sum of values from the root node to node 7 is $4+5+5=14$:



We can solve this problem like before, but now each value in the last row of the array is the sum of values on a path from the root to the node. For example, the following array corresponds to the above tree:

node id	1	2	6	3	4	7	8	9	5
subtree size	9	2	1	1	4	1	1	1	1
path sum	4	9	12	7	9	14	12	10	6

When the value of a node increases by x, the sums of all nodes in its subtree increase by x. For example, if the value of node 4 increases by 1, the array changes as follows:

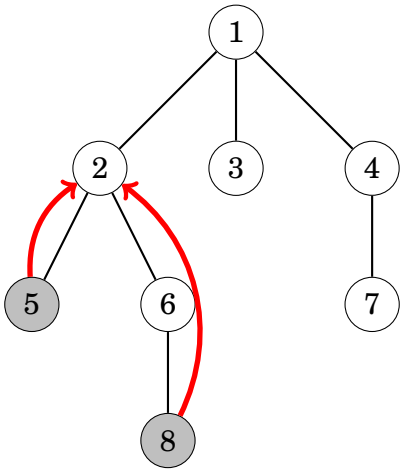
node id	1	2	6	3	4	7	8	9	5
subtree size	9	2	1	1	4	1	1	1	1
path sum	4	9	12	7	10	15	13	11	6

Thus, to support both the operations, we should be able to increase all values in a range and retrieve a single value. This can be done in $O(\log n)$ time using a binary indexed or segment tree (see Chapter 9.4).

18.3 Lowest common ancestor

The **lowest common ancestor** of two nodes of a rooted tree is the lowest node whose subtree contains both the nodes. A typical problem is to efficiently process queries that ask to find the lowest common ancestor of two nodes.

For example, in the following tree, the lowest common ancestor of nodes 5 and 8 is node 2:



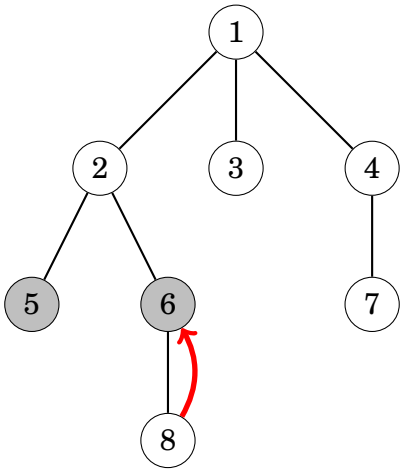
Next we will discuss two efficient techniques for finding the lowest common ancestor of two nodes.

Method 1

One way to solve the problem is to use the fact that we can efficiently find the kth ancestor of any node in the tree. Using this, we can divide the problem of finding the lowest common ancestor into two parts.

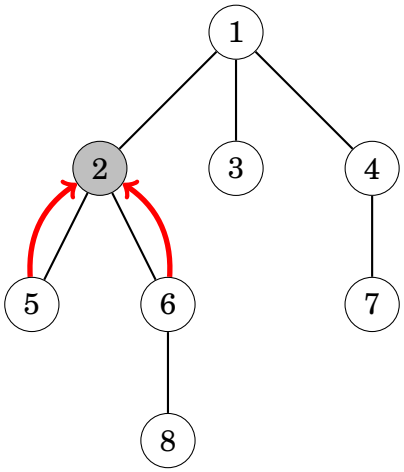
We use two pointers that initially point to the two nodes whose lowest common ancestor we should find. First, we move one of the pointers upwards so that both pointers point to nodes at the same level.

In the example scenario, we move the second pointer one level up so that it points to node 6 which is at the same level with node 5:



After this, we determine the minimum number of steps needed to move both pointers upwards so that they will point to the same node. The node to which the pointers point after this is the lowest common ancestor.

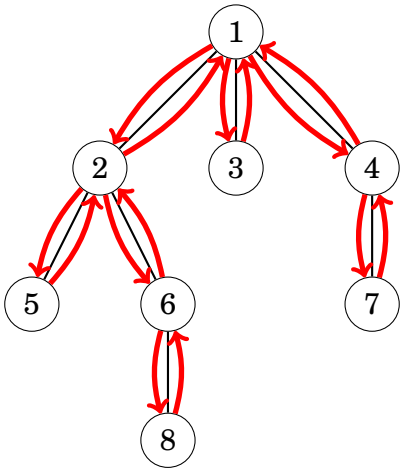
In the example scenario, it suffices to move both pointers one step upwards to node 2, which is the lowest common ancestor:



Since both parts of the algorithm can be performed in $O(\log n)$ time using precomputed information, we can find the lowest common ancestor of any two nodes in $O(\log n)$ time.

Method 2

Another way to solve the problem is based on a tree traversal array[1]. Once again, the idea is to traverse the nodes using a depth-first search:



However, we use a different tree traversal array than before: we add each node to the array *always* when the depth-first search walks through the node, and not only at the first visit. Hence, a node that has k children appears $k+1$ times in the array and there are a total of $2n-1$ nodes in the array.

[1]This lowest common ancestor algorithm was presented in [4]. This technique is sometimes called the **Euler tour technique** [33].

We store two values in the array: the identifier of the node and the depth of the node in the tree. The following array corresponds to the above tree:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
node id	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
depth	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

Now we can find the lowest common ancestor of nodes a and b by finding the node with the *minimum* depth between nodes a and b in the array. For example, the lowest common ancestor of nodes 5 and 8 can be found as follows:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
node id	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
depth	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1
				↑											

Node 5 is at position 2, node 8 is at position 5, and the node with minimum depth between positions 2...5 is node 2 at position 3 whose depth is 2. Thus, the lowest common ancestor of nodes 5 and 8 is node 2.

Thus, to find the lowest common ancestor of two nodes it suffices to process a range minimum query. Since the array is static, we can process such queries in $O(1)$ time after an $O(n \log n)$ time preprocessing.

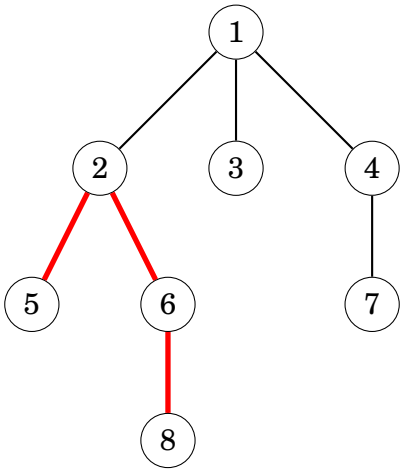
Distances of nodes

The distance between nodes a and b equals the length of the path from a to b. It turns out that the problem of calculating the distance between nodes reduces to finding their lowest common ancestor.

First, we root the tree arbitrarily. After this, the distance of nodes a and b can be calculated using the formula

$$\mathsf{depth}(a) + \mathsf{depth}(b) - 2 \cdot \mathsf{depth}(c),$$

where c is the lowest common ancestor of a and b and $\mathsf{depth}(s)$ denotes the depth of node s. For example, consider the distance of nodes 5 and 8:



The lowest common ancestor of nodes 5 and 8 is node 2. The depths of the nodes are $\texttt{depth}(5) = 3$, $\texttt{depth}(8) = 4$ and $\texttt{depth}(2) = 2$, so the distance between nodes 5 and 8 is $3 + 4 - 2 \cdot 2 = 3$.

18.4 Offline algorithms

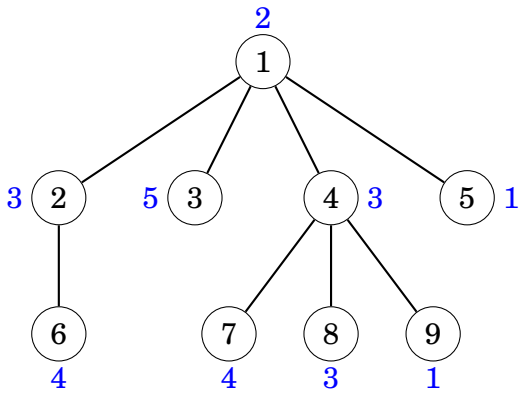
So far, we have discussed *online* algorithms for tree queries. Those algorithms are able to process queries one after another so that each query is answered before receiving the next query.

However, in many problems, the online property is not necessary. In this section, we focus on *offline* algorithms. Those algorithms are given a set of queries which can be answered in any order. It is often easier to design an offline algorithm compared to an online algorithm.

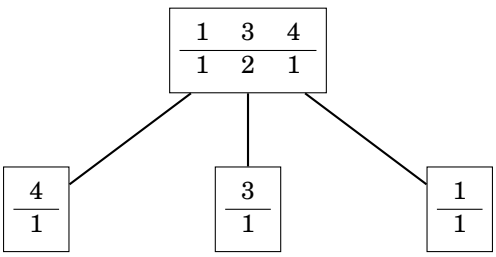
Merging data structures

One method to construct an offline algorithm is to perform a depth-first tree traversal and maintain data structures in nodes. At each node s, we create a data structure $\texttt{d}[s]$ that is based on the data structures of the children of s. Then, using this data structure, all queries related to s are processed.

As an example, consider the following problem: We are given a tree where each node has some value. Our task is to process queries of the form "calculate the number of nodes with value x in the subtree of node s". For example, in the following tree, the subtree of node 4 contains two nodes whose value is 3.



In this problem, we can use map structures to answer the queries. For example, the maps for node 4 and its children are as follows:



If we create such a data structure for each node, we can easily process all given queries, because we can handle all queries related to a node immediately after creating its data structure. For example, the above map structure for node 4 tells us that its subtree contains two nodes whose value is 3.

However, it would be too slow to create all data structures from scratch. Instead, at each node s, we create an initial data structure $\mathsf{d}[s]$ that only contains the value of s. After this, we go through the children of s and *merge* $\mathsf{d}[s]$ and all data structures $\mathsf{d}[u]$ where u is a child of s.

For example, in the above tree, the map for node 4 is created by merging the following maps:

Here the first map is the initial data structure for node 4, and the other three maps correspond to nodes 7, 8 and 9.

The merging at node s can be done as follows: We go through the children of s and at each child u merge $\mathsf{d}[s]$ and $\mathsf{d}[u]$. We always copy the contents from $\mathsf{d}[u]$ to $\mathsf{d}[s]$. However, before this, we *swap* the contents of $\mathsf{d}[s]$ and $\mathsf{d}[u]$ if $\mathsf{d}[s]$ is smaller than $\mathsf{d}[u]$. By doing this, each value is copied only $O(\log n)$ times during the tree traversal, which ensures that the algorithm is efficient.

To swap the contents of two data structures a and b efficiently, we can just use the following code:

```
a,b = b, a
```

Lowest common ancestors

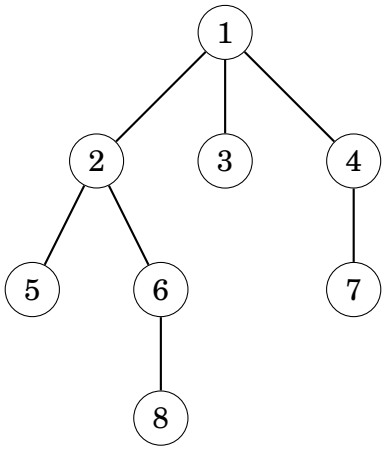
There is also an offline algorithm for processing a set of lowest common ancestor queries[2]. The algorithm is based on the union-find data structure (see Chapter 15.2), and the benefit of the algorithm is that it is easier to implement than the algorithms discussed earlier in this chapter.

The algorithm is given as input a set of pairs of nodes, and it determines for each such pair the lowest common ancestor of the nodes. The algorithm performs a depth-first tree traversal and maintains disjoint sets of nodes. Initially, each node belongs to a separate set. For each set, we also store the highest node in the tree that belongs to the set.

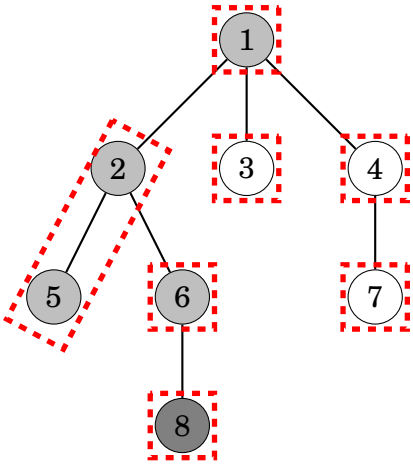
When the algorithm visits a node x, it goes through all nodes y such that the lowest common ancestor of x and y has to be found. If y has already been visited, the algorithm reports that the lowest common ancestor of x and y is the highest node in the set of y. Then, after processing node x, the algorithm joins the sets of x and its parent.

For example, suppose that we want to find the lowest common ancestors of node pairs $(5,8)$ and $(2,7)$ in the following tree:

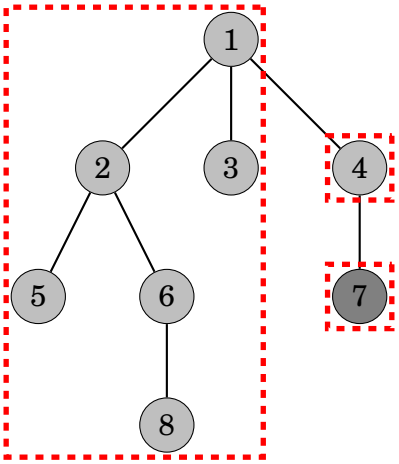
[2]This algorithm was published by R. E. Tarjan in 1979 [32].



In the following trees, gray nodes denote visited nodes and dashed groups of nodes belong to the same set. When the algorithm visits node 8, it notices that node 5 has been visited and the highest node in its set is 2. Thus, the lowest common ancestor of nodes 5 and 8 is 2:



Later, when visiting node 7, the algorithm determines that the lowest common ancestor of nodes 2 and 7 is 1:



Bibliography

[1] A. V. Aho, J. E. Hopcroft and J. Ullman. *Data Structures and Algorithms*, Addison-Wesley, 1983.

[2] B. Aspvall, M. F. Plass and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.

[3] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[4] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Latin American Symposium on Theoretical Informatics*, 88–94, 2000.

[5] J. Bentley. *Programming Pearls*. Addison-Wesley, 1999 (2nd edition).

[6] M.T. Goodrich and J.G. Kloss. Tiered Vectors: Efficient Dynamic Arrays for Rank-Based Sequences. In *6th International Workshop on Algorithms and Data Structures (WADS), 205–216, 1999.*

[7] J. Bentley and D. Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, C-29(7):571–577, 1980.

[8] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[9] M. Dima and R. Ceterchi. Efficient range minimum queries using binary indexed trees. *Olympiad in Informatics*, 9(1):39–44, 2015.

[10] S. Even, A. Itai and A. Shamir. On the complexity of time table and multi-commodity flow problems. *16th Annual Symposium on Foundations of Computer Science*, 184–193, 1975.

[11] D. Fanding. A faster algorithm for shortest-path – SPFA. *Journal of Southwest Jiaotong University*, 2, 1994.

[12] P. M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3):327–336, 1994.

[13] J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Annual Symposium on Combinatorial Pattern Matching*, 36–48, 2006.

[14] R. W. Floyd Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[15] L. R. Ford. Network flow theory. RAND Corporation, Santa Monica, California, 1956.

[16] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

[17] A. Grønlund and S. Pettie. Threesomes, degenerates, and love triangles. In *Proceedings of the 55th Annual Symposium on Foundations of Computer Science*, 621–630, 2014.

[18] J. E. Hopcroft and J. D. Ullman. A linear list merging algorithm. Technical report, Cornell University, 1971.

[19] E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM*, 21(2):277–292, 1974.

[20] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[21] P. W. Kasteleyn. The statistics of dimers on a lattice: I. The number of dimer arrangements on a quadratic lattice. *Physica*, 27(12):1209–1225, 1961.

[22] D. E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*, Addison–Wesley, 1998 (3rd edition).

[23] D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*, Addison–Wesley, 1998 (2nd edition).

[24] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[25] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8):707–710, 1966.

[26] D. Pearson. A polynomial-time algorithm for the change-making problem. *Operations Research Letters*, 33(3):231–234, 2005.

[27] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.

[28] 27-Queens Puzzle: Massively Parallel Enumeration and Solution Counting. `https://github.com/preusser/q27`

[29] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.

[30] P. Stańczyk. *Algorytmika praktyczna w konkursach Informatycznych*, MSc thesis, University of Warsaw, 2006.

[31] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[32] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.

[33] R. E. Tarjan and U. Vishkin. Finding biconnected componemts and computing tree functions in logarithmic parallel time. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, 12–20, 1984.

[34] H. N. V. Temperley and M. E. Fisher. Dimer problem in statistical mechanics – an exact result. *Philosophical Magazine*, 6(68):1061–1063, 1961.

[35] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.