Solutions for Introduction to algorithms

Philip Bille

Spring 2001

The author of this document takes absolutely no responsibility for the contents. This is merely a vague suggestion to a solution to some of the exercises posed in the book Introduction to algorithms by Cormen, Leiserson and Rivest. It is very likely that there are *many* errors and that the solutions are wrong. If you have found an error, have a better solution or wish to contribute in some constructive way please send a message to `beetle@diku.dk`.

It is important that you try *hard* to solve the exercises on your own. Use this document only as a last resort or to check if your instructor got it all wrong.

Since a second edition for the book recently has been done this document is longer being updated. Have fun with your algorithms.

Best regards,
Philip Bille

Last update: may 17, 2001

$1.1-2$

In line 5 of INSERTION-SORT alter $A[i] > key$ to $A[i] < key$ in order to sort the elements in nonincreasing order.

$1.1-3$

Algorithm 1 LINEAR-SEARCH(A, v)

Input: $A = \langle a_1, a_2, \ldots a_n \rangle$ and a value v.
Output: An index i such that $v = A[i]$ or **nil** if $v \notin A$
for $i \leftarrow 1$ **to** n **do**
  if $A[i] = v$ **then**
    return i
  end if
end for
return nil

$1.2-1$

This is a slight improvement of the requested algorithm: sorting is done *in-place* instead of using a second array. Assume that FIND-MIN(A, r, s) returns the index of the smallest element in A between indices r and s. Clearly, this can be implemented in $O(s-r)$ time if $r \geqslant s$.

Algorithm 2 SELECTION-SORT(A)

Input: $A = \langle a_1, a_2, \ldots a_n \rangle$
Output: sorted A.
for $i \leftarrow 1$ **to** n **do**
  $j \leftarrow$ FIND-MIN(A, i, n)
  $A[j] \leftrightarrow A[i]$
end for

The n calls of FIND-MIN gives the following bound on the time complexity:

$$\Theta\left(\sum_{i=1}^{n} i\right) = \Theta(n^2)$$

This holds for both the best- and worst-case running time.

$1.2-2$

Given that each element is equally likely to be the one searched for, a linear search will on the average have to search through half the elements. This is because half the time the wanted element will be in the first half and half the time it will be in the second half. Both the worst-case and average-case of LINEAR-SEARCH is $\Theta(n)$.

$1.2-3$

Solve the *Element Uniqueness Problem* in $\Theta(n \lg n)$ time. Simply sort the numbers and perform a linear run through the array searching for duplicates.

$1.2-5$

$n^3/1000 - 100n^2 - 100n + 3 = \Theta(n^3)$.

1.2 – 6

One can modify an algorithm to have a best-case running time by specializing it to handle a best-case input efficiently.

1.3 – 2

Algorithm 3 MERGE(A, p, q, r)

Input: Two sorted subarrays $A[p \ldots q]$, $A[q+1 \ldots r]$
Output: sorted $A[p \ldots r]$.
if $p \geqslant r$ **then**
  return
end if
$i \leftarrow p$
$j \leftarrow q+1$
while $i \leqslant q$ **and** $j \leqslant r$ **do**
  if $A[i] \leqslant A[j]$ **then**
    $B[i+j-1] \leftarrow A[i]$
    $i \leftarrow i+1$
  else
    $B[i+j-1] \leftarrow A[j]$
    $j \leftarrow j+1$
  end if
end while
$A \leftarrow B$

1.3 – 5

A recursive version of binary search on an array. Clearly, the worst-case running time is $\Theta(\lg n)$.

Algorithm 4 BINARY-SEARCH(A, v, p, r)

Input: A sorted array A and a value v.
Output: An index i such that $v = A[i]$ or **nil**.
if $p \geqslant r$ **and** $v \neq A[p]$ **then**
  return nil
end if
$j \leftarrow A[\lfloor (r-p)/2 \rfloor]$
if $v = A[j]$ **then**
  return j
else
  if $v < A[j]$ **then**
    return BINARY-SEARCH(A, v, p, j)
  else
    return BINARY-SEARCH(A, v, j, r)
  end if
end if

1.3 – 7

Give a $\Theta(n \lg n)$ time algorithm for determining if there exist two elements in an set S whose sum is exactly some value x.

Algorithm 5 CHECKSUMS(A, x)

Input: An array A and a value x.
Output: A boolean value indicating if there is two elements in A whose sum is x.
$A \leftarrow$ SORT(A)
$n \leftarrow$ *length*$[A]$
for $i \leftarrow$ **to** n **do**
  if $A[i] \geqslant 0$ **and** BINARY-SEARCH$(A, A[i] - x, 1, n)$ **then**
    return true
  end if
end for
return false

Clearly, this algorithm does the job. (It is assumed that **nil** cannot be true in the **if**-statement.)

1.4 – 1

Insertion sort beats merge sort when $8n^2 < 64n \lg n$, $\Rightarrow n < 8 \lg n$, $\Rightarrow 2^{n/8} < n$. This is true for $2 \leqslant n \leqslant 43$ (found by using a calculator).

Rewrite merge sort to use insertion sort for input of size 43 or less in order to improve the running time.

1 – 1

We assume that all months are 30 days and all years are 365.

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\lg n$	2^{10^6}	$2^{6 \cdot 10^7}$	$2^{36 \cdot 10^8}$	$2^{864 \cdot 10^8}$	$2^{2592 \cdot 10^9}$	$2^{94608 \cdot 10^{10}}$	$2^{94608 \cdot 10^{12}}$
$\sqrt{n}$	10^{12}	$36 \cdot 10^{14}$	$1296 \cdot 10^{16}$	$746496 \cdot 10^{16}$	$6718464 \cdot 10^{18}$	$8950673664 \cdot 10^{20}$	$8950673664 \cdot 10^{24}$
n	10^6	$6 \cdot 10^7$	$36 \cdot 10^8$	$864 \cdot 10^8$	$2592 \cdot 10^9$	$94608 \cdot 10^{10}$	$94608 \cdot 10^{12}$
$n \lg n$	62746	2801417	??	??	??	??	??
n^2	10^3	24494897	$6 \cdot 10^4$	293938	1609968	30758413	307584134
n^3	10^2	391	1532	4420	13736	98169	455661
2^n	19	25	31	36	41	49	56
$n!$	9	11	12	13	15	17	18

$2.1-1$

Let $f(n)$, $g(n)$ be asymptotically nonnegative. Show that $\max(f(n), g(n)) = \Theta(f(n) + g(n))$. This means that there exists positive constants c_1, c_2 and n_0 such that,

$$0 \leqslant c_1(f(n) + g(n)) \leqslant \max(f(n), g(n)) \leqslant c_2(f(n) + g(n))$$

for all $n \geqslant n_0$.

Selecting $c_2 = 1$ clearly shows the third inequality since the maximum must be smaller than the sum. c_1 should be selected as $1/2$ since the maximum is always greater than the weighted average of $f(n)$ and $g(n)$. Note the significance of the "asymptotically nonnegative" assumption. The first inequality could not be satisfied otherwise.

$2.1-4$

$2^{n+1} = O(2^n)$ since $2^{n+1} = 2 \cdot 2^n \leqslant 2 \cdot 2^n$! However 2^{2n} is not $O(2^n)$: by definition we have $2^{2n} = (2^n)^2$ which for no constant c asymptotically may be less than or equal to $c \cdot 2^n$.

$4.1-1$

Show that $T(n) = T(\lceil n/2 \rceil) + 1$ is $O(\lg n)$. Using substitution we want to prove that $T(n) \leqslant c \lg(n-b)$. Assume this holds for $\lceil n/2 \rceil$. We have:

$$\begin{aligned} T(n) &\leqslant c \lg(\lceil n/2 - b \rceil) + 1 \\ &< c \lg(n/2 - b + 1) + 1 \\ &= c \lg(\frac{n-2b+2}{2}) + 1 \\ &= c \lg(n-2b+2) - c \lg 2 + 1 \\ &\leqslant c \lg(n-b) \end{aligned}$$

The last inequality requires that $b \geqslant 2$ and $c \geqslant 1$.

$4.2-1$

Show an upper bound on $T(n) = 3T(\lfloor n/2 \rfloor) + n$ using iteration. We have:

$$\begin{aligned} T(n) &= n + 3T(\lfloor n/2 \rfloor) \\ &= n + 3(\lfloor n/2 \rfloor + 3T(\lfloor n/4 \rfloor)) \\ &= n + 3(\lfloor n/2 \rfloor + 3(\lfloor n/4 \rfloor + 3T(\lfloor n/8 \rfloor))) \\ &= n + 3\lfloor n/2 \rfloor + 9\lfloor n/4 \rfloor + 27T(\lfloor n/8 \rfloor) \end{aligned}$$

The ith term of the sum is $3^i \lfloor n/2^i \rfloor$, so the iteration must stop when $\lfloor n/2^i \rfloor = 1$ or ,equivalently, when $i \geqslant \lg n$. Using this fact and $\lfloor n/2^i \rfloor \leqslant n/2^i$ we get:

$$\begin{aligned} T(n) &\leqslant n + 3n/2 + 9n/4 + 27n/8 + \cdots + 3^{\lg n}\Theta(1) \\ &\leqslant n \sum_{i=0}^{\lg n} (3/2)^i + \Theta(n^{\lg 3}) \\ &= n \cdot \frac{(3/2)^{\lg n + 1} - 1}{3/2 - 1} + \Theta(n^{\lg 3}) \\ &= 2n(3/2)^{\lg n + 1} - 1 + \Theta(n^{\lg 3}) \\ &= 2n(3/2)^{\lg n} + 1/2 + \Theta(n^{\lg 3}) \\ &= 2nn^{\lg 3 - 1} + 1/2 + \Theta(n^{\lg 3}) \\ &= 2n^{\lg 3} + 1/2 + \Theta(n^{\lg 3}) \\ &= \Theta(n^{\lg 3}) \end{aligned}$$

$4.2-3$

Draw the recursion tree of $T(n) = 4T(\lfloor n/2 \rfloor) + n$. The height of the tree is $\lg n$, the out degree of each node will be 4 and the contribution of the ith level will be $4^i \lfloor n/2^i \rfloor$. Hence we have a bound on the sum given by:

$$
\begin{aligned}
T(n) &= 4T(\lfloor n/2 \rfloor) + n \\
&= \sum_{i=0}^{\lg n} 4^i \cdot \lfloor n/2^i \rfloor \\
&\leqslant \sum_{i=0}^{\lg n} 4^i \cdot n/2^i \\
&= n \sum_{i=0}^{\lg n} 2^i \\
&= n \cdot \frac{2^{\lg n+1} - 1}{2-1} \\
&= \Theta(n^2)
\end{aligned}
$$

$4.3-1$

Use the master method to find bounds for the following recursions. Note that $a = 4, b = 4$ and $n^{\log_2 4} = n^2$

- $T(n) = 4T(n/2) + n$. Since $n = O(n^{2-\epsilon})$ case 1 applies and we get $T(n) = \Theta(n^2)$.
- $T(n) = 4T(n/2) + n^2$. Since $n^2 = \Theta(n^2)$ we have $T(n) = \Theta(n^2 \lg n)$.
- $T(n) = 4T(n/2) + n^3$. Since $n^3 = \Omega(n^{2+\epsilon})$ and $4(n/2)^3 = 1/2n^3 \leqslant cn^3$ for some $c < 1$ we have that $T(n) = \Theta(n^3)$.

$7.1-1$

There is a most $2^{h+1}-1$ vertices in a complete binary tree of height h. Since the lower level need not be filled we may only have 2^h vertices.

$7.1-2$

Since the height of an n-element must satisfy that $2^h \leqslant n \leqslant 2^{h+1}-1 < 2^{h+1}$. we have $h \leqslant \lg n < h+1$. h is an integer so $h = \lfloor \lg n \rfloor$.

$7.1-3$

The heap property insures that the largest element in a subtree of a heap is at the root of the subtree.

$7.1-4$

The smallest element in a heap is always at a leaf of the tree.

$7.1-6$

No, the sequence $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12\rangle$ is not a heap.

$7.2-5$

Setting the root to 0 and all other nodes to 1, will cause the 0 to propagate to bottom of the tree using at least $\lg n$ operations each costing $O(1)$. Hence we have a $\Omega(\lg n)$ lower bound for HEAPIFY.

$7.4-3$

Show that the running time of heapsort is $\Omega(n \lg n)$. Simply make sure that the call to HEAPIFY always takes $\Omega(\lg n)$ time. This can be accomplised by using a heap where the corresponding array is sorted in increasing order.

$7.5-3$

To implement a queue or a stack simply enumerate the priority of the elements in a given order when they are inserted. If we wish to implement a stack we increment a counter before each insertion and then use this as the priority. This will give EXTRACT-MAX the desired behavior. Likewise, a queue can be implemented by decrementing a counter instead.

7.5 – 4

Algorithm 6 HEAP-INCREASE-KEY(A, i, k)

Input: A heap A and to integers i and k.
Output: A new heap where $A[i] \leftarrow \max(A[i], k)$.
if $A[i] \geqslant k$ **then**
  return
end if
while $i > 1$ **and** $A[\text{PARENT}(i)] < k$ **do**
  $A[i] \leftarrow A[\text{PARENT}(i)]$
  $i \leftarrow \text{PARENT}(i)$
end while
$A[i] \leftarrow k$

7.5 – 5

Algorithm 7 HEAP-DELETE(A, i)

Input: A heap A and integers i.
Output: The heap A with the element a position i deleted.
$A[i] \leftrightarrow A[\textit{heap-size}[A]]$
$\textit{heap-size}[A] \leftarrow \textit{heap-size}[A] - 1$
$\textit{key} \leftarrow A[i]$
if $\textit{key} \leqslant A[\text{PARENT}(i)]$ **then**
  HEAPIFY(A, i)
else
  while $i > 1$ **and** $A[\text{PARENT}(i)] < \textit{key}$ **do**
    $A[i] \leftarrow A[\text{PARENT}(i)]$
    $i \leftarrow \text{PARENT}(i)$
  end while
end if

7.5 – 6

Given k sorted lists with a total of n elements show how to merge them in $O(n \lg k)$ time. Insert all k elements a position 1 from each list into a heap. Use EXTRACT-MAX to obtain the first element of the merged list. Insert element at position 2 from the list where the largest element originally came from into the heap. Continuing in this fashion yields the desired algorithm. Clearly the running time is $O(n \lg k)$.

7 – 2

a. A d-ary heap can be implemented using a dimensional array as follows. The root is kept in $A[1]$, its d children are kept in order in $A[2]$ through $A[d+1]$ and so on. The procedures to map a node with index i to its parent and its jth child are given by:

D-ARY-PARENT(i)
return $\lfloor (i-2)/d + 1 \rfloor$

D-ARY-CHILD(i, j)
return $d(i-1) + j + 1$

b. Since each node has d children the height of the tree is $\Theta(\log_d n)$.

c. The HEAP-EXTRACT-MAX algorithm given in the text works fine for d-ary heaps; the problem is HEAPIFY. Here we need to compare the argument node to all its children. This takes $\Theta(d \log_d n)$ and dominates the overall time spent by HEAP-EXTRACT-MAX.

d. The HEAP-INSERT given in the text works fine as well. The worst-case running time is the height of the heap, that is $\Theta(\log_d n)$.

e. The HEAP-INCREASE-KEY algorithm described in exercise $7.5-4$ on page 9 works fine.

$8.1-2$

If all the elements of the array A is the same then PARTITION returns the $\lfloor n/2 \rfloor$. This can been seen by carefully analysing the code.

$8.1-3$

The running time of PARTITION is $\Theta(n)$ since each comparison involves and advance of a pointer. Pointers are advanced exactly n times.

$8.1-4$

To make QUICKSORT sort in nonincreasing order we must modify PARTITION. Simply alter the $\leqslant$'s on line 6 and 8 to $\geqslant$'s.

$8.2-1$

Show that QUICKSORT runs in $\Theta(n \lg n)$ time when all the elements of the array are the same. Since PARTITION in this case returns the middle element of the array (according to exercise $8.1-2$) this is obvious.

$8.2-2$

If the array is sorted in nonincreasing order then the *pivot* element that is selected to perform a partition around will be the largest element in the subarray. This will lead to a worst-case partitioning resulting in a worst-case time of $\Theta(n^2)$ for QUICKSORT.

$8-4$

a. Clearly, the QUICKSORT' does exactly the same as the original QUICKSORT and therefore works correctly.

b. Worst-case partitioning can cause the stack depth of QUICKSORT' to be $\Theta(n)$.

c. If we recursively call QUICKSORT' on the smallest subarray returned by PARTITION we will avoid the problem and retain a $O(\lg n)$ bound on the stack depth.

9.2 – 3

COUNTING-SORT will work correctly no matter what order A is processed in, however it is not stable. The modification to the **for** loop actually causes numbers with the same value to appear in reverse order in the output. This can seen by running a few examples.

9.2 – 5

Given n integer from 1 to k show how to count the number of elements from a to b in $O(1)$ time with $O(n+k)$ preprocessing time. As shown in COUNTING-SORT we can produce an array C such that $C[i]$ contains the number of elements less than or equal to i. Clearly, $C[b] - C[a]$ is the desired query.

9.3 – 4

Show how to sort n integers in the range 1 to n^2 in $O(n)$ time. First subtract each number by 1 yielding a range from 0 to $n^2 - 1$. Consider all number as 2-digit radix n. Each digit ranges from 0 to $n - 1$. Sort these using radix sort. This uses only two calls to counting sort. Finally, add 1 to all the numbers. This uses only $O(n)$ time.

$10.1-1$

Show how to find the the second smallest element of n elements using $n+\lceil \lg n \rceil$ comparisons. To find the smallest element simply construct a tournament as follows: Compare all the numbers in pairs. Only the smallest number of each pair is potentially the smallest of all so the problem is reduced to size $\lceil n/2 \rceil$. Continuing in this fashion until there is only one left clearly solves the problem.

Exactly $n-1$ comparisons are needed since the tournament can be drawn as an n-leaf binary tree which has $n-1$ internal nodes (show by induction on n). Each of these nodes correspond to a comparison.

We can use this binary tree to also locate the second smallest number. The path from the root to the smallest element (of height $\lceil \lg n \rceil$) must contain the second smallest element. Conducting a tournament among these uses $\lceil \lg n \rceil - 1$ comparisons.

The total number of comparisons are: $n-1+\lceil \lg n \rceil - 1 = n+\lceil \lg n \rceil - 2$.

$10.3-1$

Consider the analysis of the algorithm for groups of k. The number of elements less than (or greater than) the median of the medians x will be at least $\lceil \frac{k}{2} \rceil \left(\left\lceil \frac{1}{2} \lceil \frac{n}{k} \rceil \right\rceil - 2 \right) \geqslant \frac{n}{4} - k$. Hence, in the worst-case SELECT will be called recursively on at most $n - \left(\frac{n}{4} - k \right) = \frac{3n}{4} + k$ elements. The recurrence is

$$T(n) \leqslant T(\lceil n/k \rceil) + T(3n/4 + k) + O(n)$$

Solving by substitution we obtain a bound for which k the algorithm will be linear. Assume $T(n) \leqslant cn$ for all smaller n. We have:

$$\begin{aligned} T(n) &\leqslant c \left\lceil \frac{n}{k} \right\rceil + c \left(\frac{3n}{4} + k \right) + O(n) \\ &\leqslant \frac{cn}{k} + \frac{3cn}{k} + ck + O(n) \\ &= cn \left(\frac{1}{k} + \frac{3}{4} \right) + ck + O(n) \\ &\leqslant cn \end{aligned}$$

Where the last equation only holds for $k \geqslant 4$. Thus, we have shown that the algorithm will compute in linear time for any group size of 4 or more. In fact, the algorithm is $\Omega(n \lg n)$ for $k = 3$. This can be shown by example.

$10.3-3$

quicksort can be made to run in $O(n \lg n)$ time worst-case by noticing that we can perform "perfect partitioning": Simply use the linear time select to find the median and perform the partitioning around it. This clearly achieves the bound.

$10.3-5$

Assume that we have a routine MEDIAN that computes the median of an array in $O(n)$ time. Show how to find an arbitrary order statistic in $O(n)$.

Algorithm 8 SELECT(A, i)

Input: Array A and integer i.
Output: The ith largest element of A.
$x \leftarrow$ MEDIAN(A).
Partition A around x
if $i \leqslant \lfloor (n+1)/2 \rfloor$ **then**
  Recursively find the ith in the first half
else
  Recursively find $(i - \lfloor (n+1)/2 \rfloor)$th in the second half
end if

Clearly, this algorithm does the job.

$11.1-2$

Two stacks can be implemented in a single array without overflows occuring if they grow from each end and towards the middle.

$11.1-6$

Implement a queue using two stacks. Denote the two stacks S_1 and S_2. The ENQUEUE operation is simply implemented as a push on S_1. The dequeue operation is implemented as a pop on S_2. If S_2 is empty, successively pop S_1 and push S_2. The reverses the order of S_1 onto S_2.

The worst-case running time is $O(n)$ but the amortized complexity is $O(1)$ since each element is only moved a constant number of times.

$11.2-1$

No, INSERT and DELETE can not be implemented in $O(1)$ on a linked list. A scan through the list is required for both operations. INSERT must check that no duplicates exist.

$11.2-2$

A stack can be implemented using a linked list in the following way: PUSH is done by appending a new element to the front of the list and POP is done by deleting the first element of the list.

$11.2-6$

Algorithm 9 MERGE(L_1, L_2)

Input: Two sorted linked list's L_1, L_2.
Output: The merged list L of L_1 and L_2.
$x_1 \leftarrow head[L_1]$
$x_2 \leftarrow head[L_2]$
while $x_1 \neq$ **nil or** $x_2 \neq$ **nil do**
  if $key[x_1] < key[x_2]$ **then**
    LIST-INSERT(L, x_1)
    $x_1 \leftarrow next[x_1]$
  else
    LIST-INSERT(L, x_2)
    $x_2 \leftarrow next[x_2]$
  end if
end while
if $x_1 =$ **nil then**
  while $x_1 \neq$ **nil do**
    LIST-INSERT(L, x_1)
    $x_1 \leftarrow next[x_1]$
  end while
else
  while $x_2 \neq$ **nil do**
    LIST-INSERT(L, x_2)
    $x_2 \leftarrow next[x_2]$
  end while
end if

12.1 – 1

Find the maximum element in a direct-address table T of length m. In the worst-case searching the entire table is needed. Thus the procedure must take $O(m)$ time.

12.1 – 2

Describe how to implement a dynamic set with a bitvector. The elements have no satellite data. Simply set the ith bit to 1 if the i element is inserted and set the bit to 0 if it is deleted.

12.2 – 4

Consider keeping the chaining lists in sorted order. Searching will still take time proportional to the length of the list and therefore the running times are the same. The only difference is the insertions which now also take time proportional to the length of the list.

12.3 – 1

Searching a list of length n where each element contains a long key k and a small hash value $h(k)$ can be optimized in the following way: Comparing the keys should be done first comparing the hash values and if succesfull then comparing the keys.

13.1 – 2

The definitions clearly differ. If the heap property allowed the elements to be printed in sorted order in time $O(n)$ we would have an $O(n)$ time comparison sorting algorithm since BUILD-HEAP takes $O(n)$ time. This, however, is impossible since we know $\Omega(n \lg n)$ is a lower bound for sorting.

13.2 – 4

Show that the inorder walk of a n-node binary search tree implemented with a call to TREE-MINIMUN followed by $n - 1$ calls to TREE-SUCCESSOR takes $O(n)$ time.

Consider the algorithm at any given node during the algorithm. The algorithm will never go to the left subtree and going up will therefore cause it to never return to this same node. Hence the algorithm only traverses each edge at most twice and therefore the running time is $O(n)$.

13.2 – 6

If x is a leaf node, then if $p[x] = y$ and x is the left child then running TREE-SUCCESSOR yields y. Similarly if x is the right child then running TREE-PREDECESSOR yields y.

13.3 – 1

Algorithm 10 TREE-INSERT(z, k)

Input: A node z and value k.
Output: The binary tree with k inserted.
if $z =$ **nil then**
  key$[z] \leftarrow k$
  left$[z] \leftarrow$ **nil**
  right$[z] \leftarrow$ **nil**
else
  if $k <$ *key*$[z]$ **then**
    TREE-INSERT(*left*$[z], k$)
  else
    TREE-INSERT(*right*$[z], k$)
  end if
end if

13.3 – 4

Show that if a node in a binary search tree has two children then its successor has no left child and its predecessor has no right child. Let v be a node with two children. The nodes that immidiately precede v must be in the left subtree and the nodes that immidiately follow v must be in the right subtree. Thus the successor s must be in the right subtree and s will be the next node from v in an inorder walk. Therefore s cannot have a left child since this child since this would come before s in the inorder walk. Similarly, the predecessor has no right child.

$14.1-2$

Colouring the root black can obviously not violate any property.

$14.1-3$

By property 4 the longest and shortest path must contain the same number of black nodes. By property 3 every other nodes in the longest path must be black and therefore the length is at most twice that of the shortest path.

$14.1-4$

Consider a red-black tree with black-height k. If every nodes is black the total number of internal nodes is $2^k - 1$. If only every other nodes is black we can construct a tree with $2^{2k} - 1$ nodes.

$14.3-1$

If we choose to set the colour of a newly inserted node to black then property 3 is not violated but clearly property 4 is violated.

$14.3-3$

Inserting the keys $41, 38, 31, 12, 19, 8$ into an initially empty red-black tree yields the trees depicted in figure 1 on page 19.

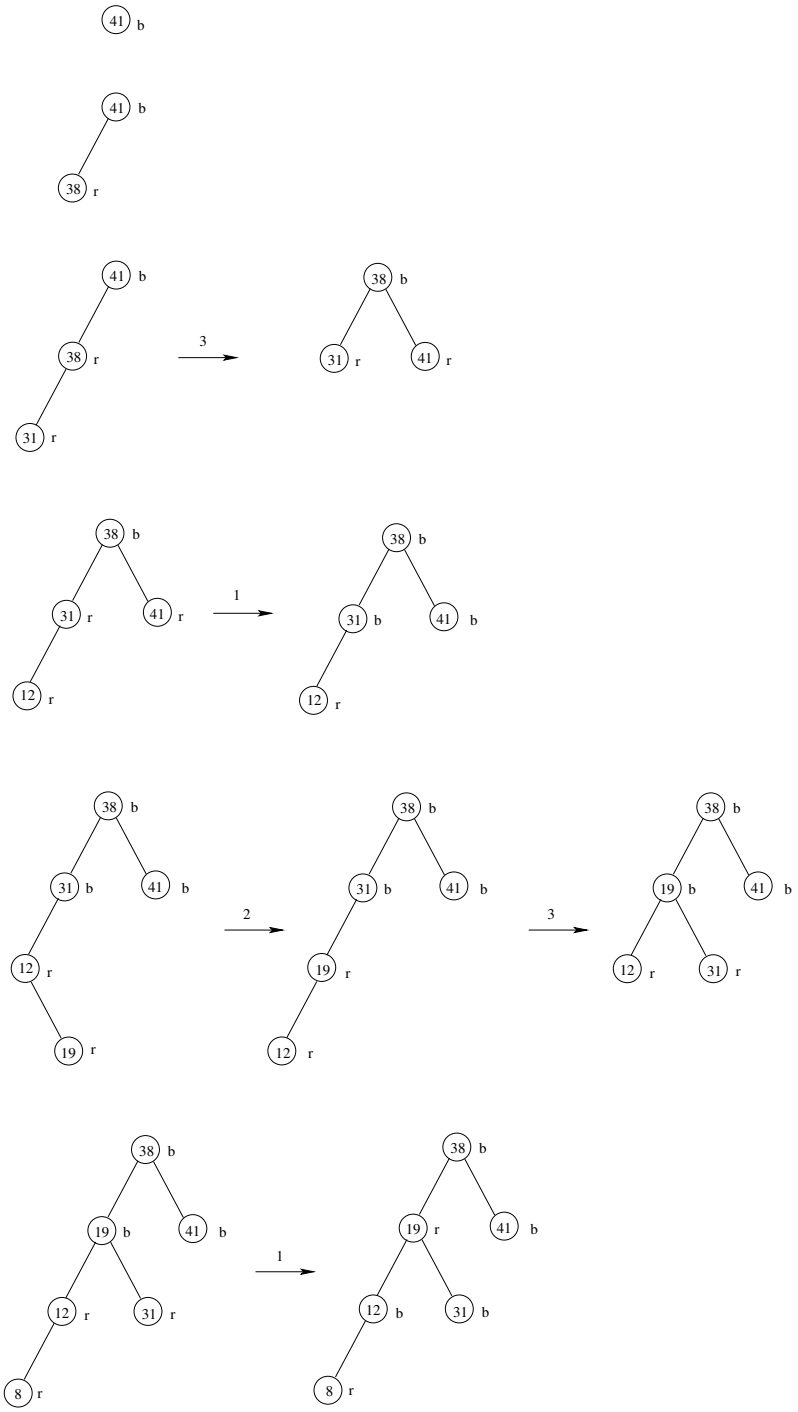

Figure 1: Inserting $41, 38, 31, 12, 19, 8$ into a red-black tree. The arrows indicate transformations. Notice that the root is always coloured black

$14.3 - 4$

Show that the *black height* property is preserved in figure 14.5 and 14.6. For 14.5 the black height for nodes A,B and D is $k + 1$ in both cases since all the subtrees have black height k. Node C has black height $k + 1$ on the left and $k + 2$ on the right since the black height of its black children is

$k+1$. For 14.6 it is clearly seen that both A, B and C have black height $k+1$. We see that the black height is well defined and the property is maintained through the transformations.

$14.4-6$

Inserting and immidiately deleting need not yield the same tree. Here is an example that alters the structure and one that changes the colour.

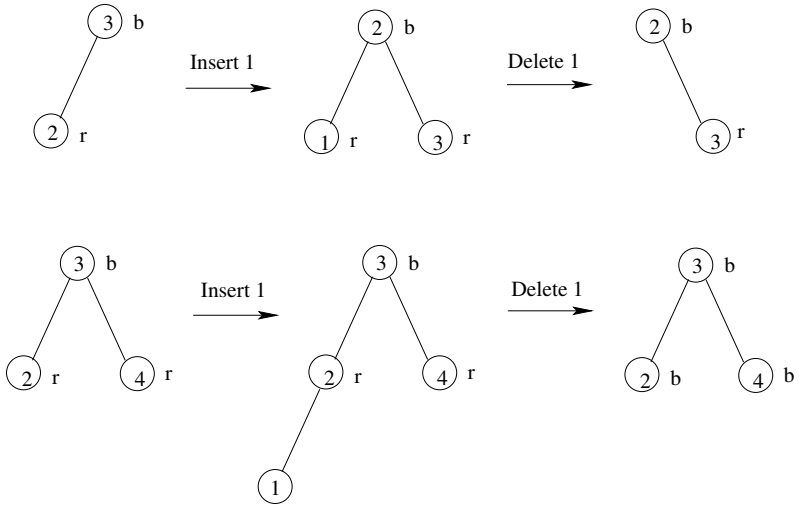

Figure 2: Inserting and deleting 1

16.1 – 1

Solve the matrix chain order for a specific problem. This can be done by computing MATRIX-CHAIN-ORDER(p) where $p = \langle 5, 10, 3, 12, 5, 50, 6 \rangle$ or simply using the equation:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leqslant k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j \end{cases}$$

The resulting table is the following:

i\j	1	2	3	4	5	6
1	0	150	330	405	1655	2010
2		0	360	330	2430	1950
3			0	180	930	1770
4				0	3000	1860
5					0	1500
6						0

The table is computed simply by the fact that $m[i,i] = 0$ for all i. This information is used to compute $m[i,i+1]$ for $i = 1, \ldots n-1$ an so on.

16.2 – 3

Draw a nice recursion tree. The MERGESORT algorithm performs at most a single call to any pair of indices of the array that is being sorted. In other words, the subproblems do not overlap and therefore memoization will not improve the running time.

16.3 – 3

Give an efficient memoized implementation of LCS-LENGTH. This can done directly by using:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1,j-1] + 1 & \text{if } i,j > 0 \text{ and } x_i = y_j \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Algorithm 11 LCS-LENGTH(X, Y)

Input: The two strings X and Y.
Output: The longest common substring of X and Y.
$m \leftarrow$ *length*$[X]$
$n \leftarrow$ *length*$[Y]$
for $i \leftarrow 1$ **to** m **do**
  for $j \leftarrow 1$ **to** n **do**
    $c[i,j] \leftarrow -1$
  end for
end for
return LOOKUP-LENGTH(X, Y, m, n)

Algorithm 12 LOOKUP-LENGTH(X, Y, i, j)

```
if c[i, j] > −1 then
  return c[i, j]
end if
if i = 0 or j = 0 then
  c[i, j] ← 0
else
  if x_i = y_i then
    c[i, j] ← LOOKUP-LENGTH(X, Y, i − 1, j − 1) + 1
  else
    c[i, j] ← max{LOOKUP-LENGTH(X, Y, i, j − 1), LOOKUP-LENGTH(X, Y, i − 1, j)}
  end if
end if
return c[i, j]
```

$16.3-5$

Given a sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$ we wish to find the longest monotonically increasing subsequence. Initially, "pack" the sequence by finding the smallest element and setting it to 1. Then find the second smallest element (not including 1) and insert it as 2. Continue until all elements in the table are in the range $1, \ldots, n$. Finding the longest common subsequence of X and $\langle 1, 2, \ldots, n \rangle$ yields the longest monotonically increasing subsequence of X. The running time is easily seen to be $O(n^2)$.

$16-1$

Compute the bitonic tour of n points in the plane. Sort the points and enumerate from left to right: $1, \ldots n$. For any i,$1 \leqslant i \leqslant n$, and for any k,$1 \leqslant k \leqslant n$, let $B[i, k]$ denote the minimun length of two disjoint bitonic paths, one from 1 to i, the other from 1 to k. When $i = k$ we have a minumin cost bitonic tour through the first i points. When $i = k = n$ we have a minimum cost bitonic tour through all n points. Note that we need only consider disjoint paths since any non-disjoint paths cannot be optimal due to the triangle inequality. We can now describe how to compute B using dynamic programming.

First define $B[0, 0] = 0$. We will determine the value of $B[i + 1, k]$ for some fixed i and for all k, $1 \leqslant k \leqslant i + 1$ by using B-values from the first i rows and i columns of B.

Case 1: $k < i$. The minimun cost disjoint paths from 1 to $i + 1$ and from 1 to k must contain the edge $(i, i + 1)$. Therefore

$$B[i + 1, k] = B[i, k] + w(i, i + 1)$$

Case 2: $k = i$. In other words, we a looking for $B[i + 1, i]$. The edge ending in $i + 1$ comes from u, $1 \leqslant u < i$. Hence

$$B[i + 1, i] = \min_{1 \leqslant u < 1} \{B[i, u] + w(u, i + 1)\}$$

Case 3: $k = i + 1$. The two edges entering $i + 1$ must come from i and from some u,$1 \leqslant u < i$. Therefore

$$B[i + 1, i + 1] = \min_{1 \leqslant u < 1} \{B[i, u] + w(i, i + 1) + w(u, i + 1)\}$$

$16-4$

The problem can be transformed into a tree colouring problem where we consider the supervisor tree and colour each node red if the employee is attending and white otherwise. The parent of a red node must be white. We wish to colour the tree so that the sum of the conviality of the nodes is maximised. The following recursions does the job. If x is a leaf with convivialty v and colour c then:

$$T(x,c) = \begin{cases} v & \text{if } x = \textsc{Red} \\ 0 & \text{if } x = \textsc{White} \end{cases}$$

If x is not a leaf then similarly:

$$T(x,c) = \begin{cases} v + \sum_i T(x.\mathit{child}_i, \textsc{White}) & \text{if } x = \textsc{Red} \\ \sum_i \max(T(x.\mathit{child}_i, \textsc{White}), T(x.\mathit{child}_i, \textsc{Red})) & \text{if } x = \textsc{White} \end{cases}$$

Implementing this recursion yields an $O(n)$ time algorithm using dynamic programming since there is exactly one subproblem for each node.

a. Simply $\max(T(\mathit{root}, \textsc{White}), T(\mathit{root}, \textsc{Red})$.

b. Simply $(T(\mathit{root}, \textsc{Red})$

Notes for the exercises

- Thanks to Jarl Friis for the tedious calculation of the table in exercise $16.1-1$.
- Thanks to Pawel Winter for providing a nice solution to $16-1$.

17.1 − 3

Show that selecting the activity with the least duration or with minimun overlap does not yield an optimal solution for the activity-selection problem. Consider figure 3:

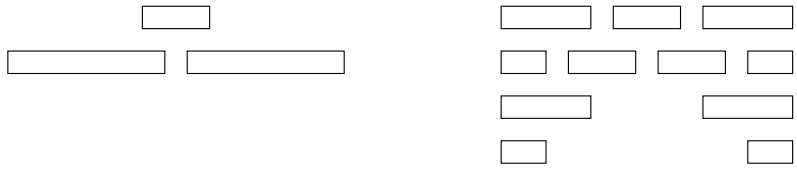

Figure 3: Two examples with other greedy strategies that go wrong

Selecting the activity with the least duration from the first example will result in selecting the topmost activity and none other. Clearly, this is worse than the optimal solution obtained by selecting the two activities in the second row.

The activity with the minimun overlap in the second example is the middle activity in the top row. However, selecting this activity eliminates the possibility of selecting the optimal solution depicted in the second row.

17.2 − 5

Describe an algorithm to find the smallest unit-length set, that contains all of the points $\{x_1, \ldots x_n\}$ on the real line. Consider the following very simple algorithm: Sort the points obtaining a new array $\{y_1, \ldots y_n\}$. The first interval is given by $[y_1, y_1 + 1]$. If y_i is the leftmost point not contained in any existing interval the next interval is $[y_i, y_i + 1]$ and so on.

This greedy algorithm does the job since the rightmost element of the set must be contained in an interval and we can do no better than the interval $[y_1, y_1 + 1]$. Additionally, any subproblem to the optimal solution must be optimal. This is easily seen by considering the problem for the points greater than $y_1 + 1$ and arguing inductively.

$18.1-2$

If a DECREMENT operation is added we can easily force the counter to change all bits per operation by calling DECREMENT and INCREMENT on 2^{k-1}. This gives a total running time of $\Theta(nk)$.

$18.1-3$

Consider a datastructure where n operations are performed. The ith operation costs i if i is an exact power of two and 1 otherwise. Determine the amortized cost of each operation using the aggregate method. Let c_i denote the cost of the ith operation. Summing the cost of the n operations yield:

$$\sum_{i=0}^{n} c_i \leqslant n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n + \sum_{j=0}^{\lg n - 1} 2^j = n + 2n - 1 < 3n$$

Thus the amortized time of each operation is less than $3n/n = O(1)$.

$18.2-1$

We wish to show an $O(1)$ amortized cost on stack operations where the stack is modified such that after k operations a backup is made. The size of the stack never exceeds k and if we assign an extra credit to each PUSH operation we can always pay for copying.

$18.2-2$

Redo exercise $18.1-3$ using the accounting method. Charging 3 credits per operation will do the job. This can be seen by example and by exercise $18.1-3$.

$18.2-3$

Keeping a pointer to the highest order bit and charging one extra credit for each bit we set allows us to do RESET in amortized $O(1)$ time.

$18.3-2$

Redo exercise $18.3-2$ using the potential method. Consider operation number $i = 2^j + k$, where j and $k \geqslant 0$ are integers and j is chosen as large as possible. Let the potential function be given by $\Phi(D_i) = 2k$. Clearly, this function satifies the requirements. There are two cases to consider for the ith operation:

If $k = 0$ then the actual cost is i and the amortized cost is given by:

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= i + 0 - 2 \cdot (2^j - 1 - 2^{j-1}) \\
&= i - (2 \cdot (2^j - 2^{j-1}) - 2) \\
&= i - (2^j \cdot (2-1) - 2) \\
&= i - i + 2 \\
&= 2
\end{aligned}$$

Otherwise the actual cost will be 1 and we find that

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= 1 + 2k - 2(k-1) \\
&= 3
\end{aligned}$$

$18.4-3$

Show that the amortized cost of TABLE-DELETE when $\alpha_{i-1} \geqslant 1/2$ is bounded above by a constant. Notice that the ith operation cannot cause the table to contract since contract only occurs when $\alpha_i < 1/4$. We need to consider cases for the load factor α_i. For both cases $num_i = num_{i-1} - 1$ and $size_i = size_{i-1}$.

Assume $\alpha_i \geqslant 1/2$.

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot (num_i + 1) - size_i) \\ &= -1\end{aligned}$$

Then consider $\alpha_i < 1/2$.

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot num_i - size_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 3 \cdot num_i - 3/2 \cdot size_i + 2 \\ &= 3\alpha_i \cdot size_i - 3/2 \cdot size_i + 2 \\ &< 3/2 \cdot size_i - 3/2 \cdot size_i + 2 \\ &= 2\end{aligned}$$

$18-2$

Construct a dynamic binary search datastructure. Let $k = \lceil \lg(n+1) \rceil$. Maintain k sorted arrays $A_0, A_1, \ldots A_{k-1}$ such that the length of the A_i is 2^i.

a. A SEARCH operation can be implemented by using a binary search on all the arrays. The worst-case complexity of this must be:

$$\sum_{i=0}^{k} \lg(2^i) = \sum_{i=0}^{k} i = \frac{k(k+1)}{2} = O(k^2) = O(\lg^2 n)$$

b. The worst-case of the INSERT operation occurs when all the sorted array have to be merged into a new array. That is, when n increases from $2^k - 1$ to 2^k for some k. Using k-way merging as in exercise $7.5-6$ with a total of n elements yields a running time of $O(n \lg k) = O(n \lg \lg n)$.

From the analysis of incrementing a binary counter we have that the ith bit is flipped a total of $\lfloor n/2^i \rfloor$ times in n INCREMENT operations. The correspondance to this problem is that every time the ith bit in n is flipped we need to merge the lists $A_i, A_{i-1}, \ldots A_0$ using time $O(2^i \lg i)$. The total running time is thus:

$$\sum_{i=1}^{\lfloor \lg n \rfloor} \lfloor n/2^i \rfloor 2^i \lg i < n \sum_{i=1}^{\lg n} \lg i = n \lg(\prod_{i=1}^{\lg n} i) = O(n \lg((\lg n)!)) = O(n \lg n \lg \lg n)$$

The amortized complexity is therefore $\lg n \lg \lg n$.

c. The DELETE operation should be implemented like the one used in dynamic tables (section 18.4). One should wait linear time before deallocating an array in order to avoid the worst-case complexity.

$20-2$

The algorithm MST-MERGEABLE-HEAP(G) is trivially implemented using the binomial heap operations. Consider the running of the algorithm step by step:

- The **for** loop of line $2-4$ requires $O(V)$ MAKE-HEAP operations and a total of $O(E)$ INSERT operations in line 4. Note that the E_i set can be at most $O(V)$ by definition. The total time is thus $O(V + E \lg V)$.
- Consider the **while** loop of line $5-12$.
  - We can at most extract $O(E)$ edges in line 7 taking a total of $(E \lg V)$ time.
  - The $i \neq j$ check can be done in time $O(1)$ by enumerating the sets.
  - The **then** branch is at most taken $O(V)$ times since it reduces the number of V_i's by one every time. Insertion into T can be done in $O(1)$ time using a linked list. Merging V_i's take $O(\lg V)$. Merging E_i's take $O(\lg E)$.
- The total time of the **while** loop is then $O(E \lg V + V \lg V + V \lg E)$.

The overall running time is seen to be $O(E \lg V)$.

$22.3-3$

Construct a sequence of m MAKE-SET, FIND-SET and UNION operations, n of which are MAKE-SET, that takes $\Omega(m \lg n)$ time when we use union by rank only.

First perform the n (assume n is a power of 2 for simplicity) MAKE-SET operations on each of the elements $\{x_1, x_2, \ldots x_3\}$. Then perform a union on each pair (x_1, x_2), (x_3, x_4) and so on yielding $n/2$ new sets. Continue the process until there is only a single set left. This must be done $\Omega(\lg n)$ each time using $\Omega(m)$.

$22.4-3$

The running time of the disjoint-set forest with only union by rank is $O(m \lg n)$. Any UNION or FIND-SET operation takes at most $O(\lg n)$ time since this is maximun rank of any representative by Lemma 22.2 and Corollary 22.5.

$22-1$

Consider an off-line minimun problem, where we are given n INSERT and m EXTRACT-MIN calls. The elements are all from the set $\{1, \ldots, n\}$. We are required to return an array *extracted*$[1 \ldots m]$ such that *extracted*$[i]$ is the return value of the ith EXTRACT-MIN call.

a. Consider the following sequence:

$4, 8, E, 3, E, 9, 2, 6, E, E, E, 1, 7, E, 5$

The corresponding *extracted* array is:

$4, 3, 8, 2, 6, 9, 1$

b. Running an example convinces one of the correctness of the algorithm.

c. Using UNION-FIND techniques we can construct an efficient implementation of the algorithm. Initially place the subsequences $I_1, \ldots I_{m+1}$ into disjoint-set and maintain an additional label and pointer for the root of each set. The label indicates the subquence number and the pointer points to the root of the next set.

We proceed as follows:

- Line 2 is implemented by a FIND-SET operation on i yielding the representative of subsequence I_j labeled j.
- The Finding the next set can be done by following the pointer from the root.
- A UNION operation is executed on the sets and a constant number of pointer updates is used to reestablish the structure.

The overall running time is seen to be $O(m\alpha(m, n))$.

$23.1-1$

Given the adjacency-list representation of a graph the out and in-degree of every node can easily be computed in $O(E+V)$ time.

$23.1-3$

The transpose of a directed graph G^T can be computed as follows: If the graph is represented by an adjacency-matrix A simply compute the transpose of the matrix A^T in time $O(V^2)$. If the graph is represented by an adjacency-list a single scan through this list is sufficient to construct the transpose. The time used is the $O(E+V)$.

$23.1-5$

The square of a graph can be computed as follows: If the graph is represented as an adjacency-matrix A simply compute the product A^2 where multiplication and addition is exchanged by and'ing and or'ing. Using the trivial algorithm yields a running time of $O(n^3)$. Using strassens algorithm improves the bound to $O(n^{\lg 7}) = O(n^{2,81})$.

If we are using the adjacency-list representation we can simply append lists eliminating duplicates. Assuming the lists are sorted we can proceed as follows: For each node v in some list replace the v with $Adj[v]$ and merge this into the list. Each list can be at most V long and therefore each merge operation takes at most $O(V)$ time. Thus the total running time is $O((E+V)V) = O(E+V^2)$.

$23.1-6$

Notice that if edge is present between edges v and u then v cannot be a sink and if the edges is not present then u cannot be a sink. Searching the adjancency-list in a linear fashion enables us to exclude one vertex at a time.

$23.2-3$

If breadth-first-search is run on a graph represented by an adjacency-matrix the time used scanning for neighbour can increase to $O(V^2)$ yielding a total running time of $O(V^2+V)$.

$23.2-6$

From graph theory we know that a graph is bipartite if and only if every cycle has even length. Using this fact we can simply modify breadth-first-search to compare the current distance with the distance of an encountered gray vertex. The running time will still be $O(E+V)$.

$23.2-7$

The diameter of a tree can computed in a bottom-up fashion using a recursive solution. If x is a node with a depth $d(x)$ in the tree then the diameter $D(x)$ must be:

$$D(x) = \begin{cases} \max\{\max_i\{D(x.child_i)\}, \max_{ij}\{d(x.child_i) + d(x.child_j)\} + 2\}, & \text{if } x \text{ is an internal node} \\ 0 & \text{if } x \text{ is a leaf} \end{cases}$$

Since the diameter must be in one of the subtrees or pass through the root and the longest path from the root must be the depth. The depth can easily be computed at the same time. Using dynamic programming we obtain a linear solution.

Actually, the problem can also be solved by computing the longest shortest path from an arbitrary node. The node farthest away will be the endpoint of a diameter and we can thus compute the longest shortest path from this node to obtain the diameter. See relevant litterature for a proof of this.

$23.3-1$

In the following table we indicate if there can be an edge (i,j) with the specified colours during a depth-first search. If the entry in the table is present we also indicate which type the edge might be: **T**ree, **F**orward or **B**ack edge.

(i,j)	WHITE	GRAY	BLACK
WHITE			
GRAY	T	B	F,C
BLACK			

$23.4-3$

Show how to determine if an undirected graph contains a cycle in $O(V)$ time. A depth-first search on an undirected graph yields only tree edges and back edges which can be seen by considering the cases in the table in exercise $23.3-1$. Then we have that an undirected graph is acyclic if and only if a depth-first search yields no back edges.

We now perform a depth-first search and if we discover a back edge then the graph must be acyclic. The time taken is $O(V)$ since if we discover more than V distinct edges the graph must be acyclic and we will have seen a back edge.

$23-3$

a. If G has an Euler tour any path going "into" a vertex must "leave" it. Conversely, if the in and out-degrees of any vertex is the same any we can construct a path that visits all edges.

b. Since the edges of any Euler graph can be split into disjoint cycles, we can simply find these cycles and "merge" them into an Euler tour.

Notes for the exercises

- Thanks to Stephen Alstrup for providing a solution to exercise $23.2-7$.

$24.1-1$

If (u,v) is a minimum weight edge then it will be safe for any cut with u and v on separate sides. Therefore (u,v) belongs to some minimun spanning tree.

$24.1-3$

Assume that (u,v) belongs to some minimum spanning tree. Consider a cut with u and v on separate sides. If (u,v) is not a light edge then clearly the graph would not be a minimum spanning tree.

$24.1-5$

Let e be the maximum-weight edge on some cycle in $G=(V,E)$ then e is not part of a minimum spanning tree of G. If e was in the minimum spanning tree then replacing it with any other edge on the cycle would yield a "better" minimum spanning tree.

$24.2-1$

Given a minimum spanning tree T we wish to sort the edges in Kruskals algorithm such that it produces T. For each edge e in T simply make sure that it preceeds any other edge not in T with weight $w(e)$.

$24.2-3$

Consider the running times of Prims algorithm implemented with either a binary heap or a Fibonnacci heap. Suppose $|E|=\Theta(V)$ then the running times are:

- Binary: $O(E \lg V) = O(V \lg V)$
- Fibonnacci: $O(E + V \lg V) = O(V \lg V)$

If $|E|=\Theta(V^2)$ then:

- Binary: $O(E \lg V) = O(V^2 \lg V)$
- Fibonnacci: $O(E + V \lg V) = O(V^2)$

The Fibonnacci heap beats the binary heap implementation of Prims algorithm when $|E| = \omega(V)$ since $O(E + V \lg V) = O(V \lg V)$ t for $|E| = O(V \lg V)$ but $O(E \lg V) = \omega(V \lg V)$ for $|E| = \omega(V)$. For $|E| = \omega(V \lg V)$ the Fibonnacci version clearly has a better running time than the ordinary version.

$24.2-4$

The running time of Kruskals algorithm can be analysed as follows:

- Sorting the edges: $O(E \lg E)$ time.
- $O(E)$ operations on a disjoint-set forest taking $O(E\alpha(E,V))$.

The sort dominates and hence the total time is $O(E \lg E)$. Sorting using counting sort when the edges fall in the range $1,\ldots,|V|$ yields $O(V+E) = O(E)$ time sorting. The total time is then $O(E\alpha(E,V))$. If the edges fall in the range $1,\ldots,W$ for any constant W we still need to use $\Omega(E)$ time for sorting and the total running time cannot be improved further.

$24.2-5$

The running time of Prims algorithm can be analysed as follows:

- $O(V)$ initialization.
- $O(V \cdot$ time for EXTRACT-MIN$)$.
- $O(E \cdot$ time for DECREASE-KEY$)$.

If the edges are in the range $1, \ldots, |V|$ the Van Emde Boas priority queue can speed up EXTRACT-MIN and DECREASE-KEY to $O(\lg \lg V)$ thus yielding a total running time of $O(V \lg \lg V + E \lg \lg V) = O(E \lg \lg V)$. If the edges are in the range from 1 to W we can implement the queue as an array $[1 \ldots W+1]$ where the ith slot holds a doubly linked list of the edges with weight i. The $W+1$st slot contains ∞. EXTRACT-MIN now runs in $O(W) = O(1)$ time since we can simply scan for the first nonempty slot and return the first element of that list. DECREASE-KEY runs in $O(1)$ time as well since it can be implemented by moving an element from one slot to another.

$24-1$

a. Show that a second-best minimun spanning tree can be obtained from the minimum spanning tree by replacing a single edge from the tree with another edge not in the tree.

Let T be a minimun spanning tree. We wish to find a tree that has the smallest possible weight that is larger than the weight of T. Clearly, any spanning tree can be obtained from another simply by replacing one or more edges from the tree. Consider replacing two or more edges from the tree. The second replacement edge must have a weight that is larger than the edge it replaced, since this edge could otherwise not be part of a minimum spanning tree. Therefore a second-best minimum spanning tree can be obtained by a single replacement.

The replacement edge should be the one yielding the smallest increase in weight. Hence it must the largest weight on some path from two vertices u and v.

b. Let $\max(u, v)$ be the weight of the edge of maximun weight on the unique path between u, v in a spanning tree. To compute this in $O(V^2)$ time for all nodes in the tree do the following:

For each node v perform a traversal of the tree. Inorder to compute $\max(v, k)$ for all $k \in V$ simply maintain the largest weight encounted so far for the path being investigated. Doing this yields a linear time algorithm for each node and we therefore obtain a total of $O(V^2)$ time.

c. Using the idea of the first question and the provided algorithm in the second question, we can now compute T_2 from T in the following way:

Compute $\max(u, v)$ for all vertices in T. Compute for any edge (u, v) not in T the difference $w(u, v) - \max(u, v)$. The two edges yielding the smallest positive difference should be replaced.

$25.2-3$

Consider stopping Dijkstra's algorithm just before extracting the last vertex v from the priority-queue. The shortest path estimate of this vertex must the shortest path since all edges going into v must have been relaxed. Additionally, v was to be extracted last so it will have the largest shortest path of all vertices and any relaxation from v will therefore not alter shortest path estimates. Therefore the modified algorithm is correct.

$25.2-5$

Consider running Dijkstra's algorithm on a graph, where the weight function is $w : E \to \{1, \ldots W-1\}$. To solve this efficiently, implement the priority queue by an array A of length $WV+1$. Any node with shortests path estimate d is kept in a linked list at $A[d]$. $A[WV+1]$ contains the nodes with ∞ as estimate.

EXTRACT-MIN is implemented by searching from the previous minimun shortest path estimate until a new is found. DECREASE-KEY simply moves vertices in the array. The EXTRACT-MIN operations takes a total of $O(VW)$ and the DECREASE-KEY operations take $O(E)$ time in total. Hence the running time of the modified algorithm will be $O(VW+E)$.

$25.2-6$

Consider the problem from the above exercise. Notice that every time a node v is extracted by EXTRACT-MIN the relaxations performed on the neighbour of v gives shortests path estimates in the range $\{d[v], \ldots d[v]+W-1\}$. Hence after every EXTRACT-MIN operation only W distinct shortest path estimates are in the priority queue at any time.

Converting the array implementation to a binary heap of the previous exercise must give a running time of $O(V+E)\lg W)$ since both the EXTRACT-MIN operation and the DECREASE-KEY operation take $O(\lg W)$ time. If we use a fibonnacci heap the running time can be further improved to $O(V \lg W + E)$.

$26.1-3$

The identity matrix for "multiplication" should look as the one given in the exercise since 0 is the identity for $+$ and ∞ is the identity for min.

$26.1-8$

The presence of a negative-weight cycle can be determined by looking at the diagonal of the matrix $D^{(n-1)}$ computed by an all-pairs shortest-path algorithm. If the diagonal contains any negative number there must be a negative-weight cycle.

$26.1-9$

As in the previous exercise when can determine the presence of a negative-weight cycle by looking for a negative number in the diagonal. If $D^{(m)}$ is the first time for which this occurs then clearly the negative-weight cycle has length m.

$26.2-5$

As in exercise $26.1-8$ a negative-weight cycle can be determined by looking at the diagonal of the output matrix.

$26.2-7$

We wish to compute the transitive closure of a directed graph $G = (V, E)$. Construct a new graph $G^* = (V, E^*)$ where E^* is initially empty. For each vertex v traverse the graph G adding edges for every node encountered in E^*. This takes $O(VE)$ time.

$27.1-6$

Let f_1 and f_2 be flows in a flow network $G = (V, E)$. The sum $f_1 + f_2$ is defined by $(f_1 + f_2)(u, v) = f_1(u, v) + f_2(u, v)$ for all $u, v \in V$. Of the tree flow properties the following are satified by $f_1 + f_2$:

Capacity constraint: May clearly be violated.

Skew symmetry: We have:

$$\begin{aligned}(f_1 + f_2)(u, v) &= f_1(u, v) + f_2(u, v) = -f_1(v, u) - f_2(v, u) \\ &= -(f_1(v, u) + f_2(v, u)) = -(f_1 + f_2)(v, u)\end{aligned}$$

Flow conservation: Let $u \in V - s, t$ be given. Then:

$$\begin{aligned}\sum_{v \in V} (f_1 + f_2)(u, v) &= \sum_{v \in V} (f_1(u, v) + f_2(u, v)) = \sum_{v \in V} f_1(u, v) + \sum_{v \in V} f_2(u, v) \\ &= 0 + 0 = 0\end{aligned}$$

$27.2-4$

Prove that for any vertices u and v and any flow and capacity functions f and c we have: $c_f(u, v) + c_f(v, u) = c(u, v) + c(u, v)$. Obvious since:

$$\begin{aligned}c_f(u, v) + c_f(v, u) &= c(u, v) - f(u, v) + c(v, u) - f(v, u) \\ &= c(u, v) + c(v, u) - f(u, v) + f(v, u) \\ &= c(u, v) + c(v, u)\end{aligned}$$

$27.2-7$

Show that the function f given by:

$$f_p(u, v) = \begin{cases} c_f(p) & \text{if } (u, v) \text{ is on } p \\ -c_f(p) & \text{if } (v, u) \text{ is on } p \\ 0 & \text{otherwise} \end{cases}$$

is a flow in G_f with value $|f_p| = c_f(p) > 0$. We simply check that the three properties are satified:

Capacity constraint: Since $c_f(p)$ is a minimum capacity on the path p and 0 otherwise no capacities can be exceeded.

Skew symmetry: If u and v are on p the definition clearly satifies this constraint. Otherwise the flow is 0 and the constraint is again satified.

Flow conservation: Since $c_f(p)$ is constant along a path flow conservation must clearly be preserved.

$27.2-9$

We wish to compute the edge connectivity of an undirected graph $G = (V, E)$ by running a maximum-flow algorithm on at most $|V|$ flow networks of the same size as G.

Let G_{uv} be the directed version of G. We will consider G_{uv} as a flow network where $s = u$ and $t = v$. We set the capacity of every edge to 1 so that the number of edges crossing any cut will equal the capacity of the cut. Let f_{uv} be a maximum flow of G_{uv}.

The edge connectivity can now be computed by finding $\min_{v \in V - \{u\}} |f_{uv}|$. This is can easily be seen by using the max-flow min-cut theorem.

$27.3-3$

We wish to give an upper bound on the length of any augmenting path found in the G' graph. The augmenting path is a simple path in the residual graph G'_f. The key observation is that edges in the residual graph may go from R to L. Hence a path must be of the form:

$$s \to L \to R \to \ldots \to L \to R \to t$$

Crossing between L and R as many times as it can without using a vertex twice. At most $2 + 2\min(|L|, |R|)$ vertices can be in the path and an upper bound on the length is therefore $2\min(|L|, |R|) + 1$.

$35.2-3$

Find two cases where the ANY-SEGMENT-INTERSECT go wrong if used to compute all intersections from left to right.

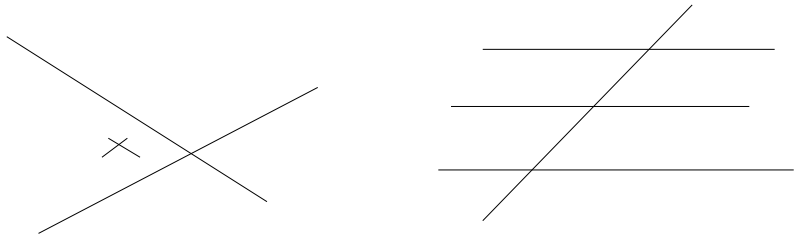

Figure 4: Two cases for the ANY-SEGMENT-INTERSECT

On the first illustration the rightmost intersection is found first and on the other the middle horisontal segment is not found.

$35.3-2$

Show that a lower bound of computing a convex hull of n points is $\Omega(n \lg n)$ if the computation model has the same lower bound for sorting.

The n points can be transformed into points in the plane by letting a point i map to the point (i, i^2). Computing the convex hull outputs the points in counterclockwise order and thus sorted.

$35-1$

a. Compute the convex layers of n points in the plane. Let h_i denote the number of points in the ith layer. Initially, compute the convex hull using Jarvis' march. Removing these points and repeating the procedure until no points exists does the job since $\sum h_i = n$ and the complete running time therefore is $O(n^2)$.

b. Clearly, if a lower bound for computing the convex hull is $\Omega(n \lg n)$ by exercise $35.3-2$ this must also be a lower bound for computing the convex layers.

$36.1-4$

The dynamic programming algorithm for the knapsack problem runs in time $O(nW)$ where n is the number of items and W is the maximun weight of the items. Since W do not depend on the size of the input the algorithm is not polynomial.

$36.1-5$

Suppose we have an algorithm A that accepts any string $x \in L$ in polynomial time but runs in superpolynomial time if $x \; not \in L$. There exists an algorithm that decides L since there must be an upper bound p on the time used to accept a string. If the algorithm does not give an answer after p time then we know that x will be rejected.

$36.1-6$

Consider an algorithm that calls $O(n)$ subroutines each taking linear time. The first call can produce $O(n)$ output which can be concatenated to the original input and used as input to the next giving it time $O(2n)$ and sofort. The total time used is then $\sum_{k=1}^{n} 2^k n$ which is clearly not polynomial. If however we only call a constant number of subroutines the algorithm will be polynomial.

$36.1-7$

Assume that L, L_1, $L_2 \in P$. The following statements hold:

- $L_1 \cup L_2 \in P$ since we can decide if $x \in L_1 \cup L_2$ by deciding if $x \in L_1$ and then if $x \in L_2$. If either holds then $x \in L_1 \cup L_2$ otherwise it is not.
- $L_1 \cap L_2 \in P$ since we can decide if $x \in L_1$ and then if $x \in L_2$. If both holds then $x \in L_1 \cap L_2$ otherwise it is not.
- $\overline{L} \in P$ since $x \in \overline{L} \iff x \notin L$.
- $L_1 L_2 \in P$. Given a string x of length n denote its substring from index i to j by x_{ij}. We can then decide x by deciding $x_{1k} \in L_1$ and $x_{k+1n} \in L2$ for all the n possible values of k.
- $L^* \in P$. We can prove this showing that the result holds for L^k for all k. We will use induction on k. If $k = 0$ we only consider the empty language and the result is trivial. Assume that $L^k \in P$ and consider $L^{k+1} = LL^k$. The above result on concatenation gives us that $LL^k \in P$.

$36.2-5$

Any NP complete language can be decided by an algorithm running in time $2^{O(n^k)}$ for some constant k simply by trying all the possible certificates.

$36.2-9$

We wish to show that $P \subseteq co-NP$. Assume that $L \in P$. Since P is closed under complement we have that $\overline{L} \in P$ and thus $\overline{L} \in NP$ giving us that $L \in co-NP$.

$36.2-10$

Show that $NP \neq co-NP \implies P \neq NP$. By contraposition the statement is the same as $P = NP \implies NP = co-NP$. Since P is closed under complement the statement is obvious.

$36.3-1$

Show that $\leqslant_p$ is a transitive relation. Let $L_1 \leqslant_p$ L2 and $L_2 \leqslant_p$ L3. Then there exists polynomial-time computable reduction functions f_1 and f_2 such that:

- $x \in L_1 \iff f_1(x) \in L_2$
- $x \in L_2 \iff f_2(x) \in L_3$

The function $f_2(f_1(x))$ is polynomial-time computable and satifies that $x \in L_1 \iff f_2(f_1(x)) \in L_3$ thus giving us that $L_1 \leqslant_p L_3$.

$36.3-2$

Show that if $L \leqslant_p \overline{L} \iff \overline{L} \leqslant_p L$. If $L \leqslant_p \overline{L}$ an f is the reduction function then we have by definition that $x \in L \iff f(x) \in \overline{L}$ for some x. This means that $x \notin L \iff f(x) \notin \overline{L}$ which is $x \in \overline{L} \iff f(x) \in L$ giving us that $\overline{L} \leqslant_p L$.

$36.3-5$

Assume that $L \in P$. We wish to show that L is P-complete unless it is the empty language or $\{0,1\}^*$. Given $L' \in P$ we can reduce it to L simply by using the polynomial-time algorithm for deciding L' to construct the reduction function f. Given x decide if $x \in L'$ and set $f(x)$ such that $f(x) \in L$ if $x \in L$‘ and $f(x) \notin L$ otherwise. Clearly, this is not possible for the two trivial languages mentioned above.

$36.3-6$

Show that $L \in NPC \iff \overline{L} \in co-NPC$. Assume $L \in NPC$. Then $L \in NP$ giving us that $\overline{L} \in co-NP$. Assume further that $L' \leqslant_p L$ for all $L \in NP$. This means that $x \in L' \iff f(x) \in L$ for some polynomial-time reduction function f. Then we have that $x \in \overline{L'} \iff f(x) \in \overline{L}$ which means that $\overline{L'} \leqslant_p \overline{L}$ for all $\overline{L'} \in co-NP$. The converse can be shown similarly.

$36.4-5$

If a formula is given in disjunctive normal form we can simply check if any of the AND'clauses can be satified to determine if the entire formula can be satified.

$36.5-4$

Show that the set-partitioning problem is NP-complete. Clearly, using the partion of the set as certificate shows that the problem is in NP. For the NP-hardness notice that there is a partition if and only if there is a subset $S' \subseteq S$ that sums to $(\sum_{x \in S} x)/2$.

$36.5-5$

By exercise $36.2-6$ the hamiltonian-path problem is in NP. To show that the problem is NP-hard construct a reduction from the hamilton-cycle problem. Given a graph G that has a hamilton-cycle pick any vertex v and make a "copy" of u that is connected to the same vertices as v. It can be shown that this graph has a hamiltonian path from v to u if and only if G has a hamilton-cycle.

$37.1-1$

A graph with two vertices and an edge between them has the optimal vertex cover of consisting of a single vertex. However, APPROX-VERTEX-COVER returns both vertices in this case.

$37.1-2$

The following graph will not yield an ratio bound of two using the proposed heuristic. The vertices are $V = \{a1, a2, a3, a4, a5, a6, a7, b1, b2, b3, b4, b5, b6, c1, c2, c3, c4, c5, c6\}$ and the adjancancy list representation is given by:

a1: b1, b2

a2: b3, b4

a3: b5, b6

a4: b1, b2, b3

a5: b4, b5, b6

a6: b1, b2, b3, b4

a7: b2, b3, b4, b5, b6

Additionally there should be an edge from b1 to c1 and from b2 to c2 and so forth.

$37.1-3$

To construct an optimal vertex cover for a tree simply pick a node k such that k has at least one leaf. Remove k and all vertices and edges incident to k and continue until no more vertices are left. The selected vertices will be an optimal vertex cover. To show that this algorithm excibits optimal substructure consider a k as above with vertices $l_1, l_2, \ldots, l_s$ as leafs in a graph $G = (V, E)$. Let G' be given by $V' = V \backslash \{k, l_1, l_2, \ldots, l_s\}$. Assuming we have an optimal vertex cover for G then all the leaves $l_1, l_2, \ldots, l_s$ must be covered and this can be obtained optimally by covering k.

$37.1-4$

The clique problem and vertex cover problem is related through a reduction. This, however, does not imply that there is a constant ratio bound approximation algorithm since the reductions may alter the ratio polynomially.

$37.2-1$

We transform an instance of the travelling salesman into another instance that satifies the triangle equality. Let k be the sum of weights of all edges. By adding k to all edges we obtain an instance that satifies the triangle inequality since k dominates the sums. The optimal tour remains unchanged since all tours contain the same number of edges and we therefore simply added nk to all tours. This does not contradict theorem 37.3 since the ratio bound of the transformed problem does not give a constant ratio bound on the original problem.

$37.4-3$

We can modify the approximation scheme to find a value greater than t that is the sum of some subset S' of a list S by running the approximation algorithm and then summing the elements of the complement list $S - S'$.

Notes for the exercises

- Thanks to David Pisinger for providing a solution to 37.1 − 2.