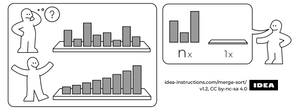
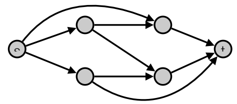
02110

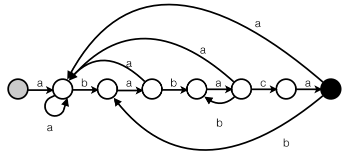
Inge Li Gørtz

Contents

- Divide-and-conquer
- Dynamic programming
- Maximum flow in networks
- Matchings and assignment problems
- Data structures:
  - Hash tables
  - Fenwick trees and dynamic arrays
  - Amortised data structures
- String matching
- Randomized algorithms
- NP-completeness

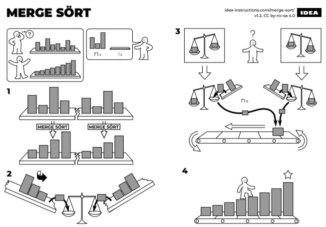


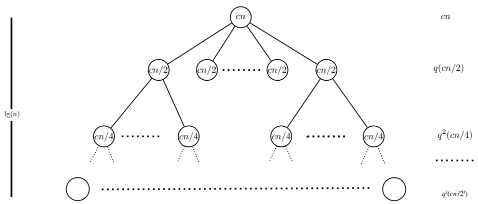




Divide-and-Conquer

- Algorithms: counting inversions
- Analysis:
  - Recursion trees.
  - Substitution method



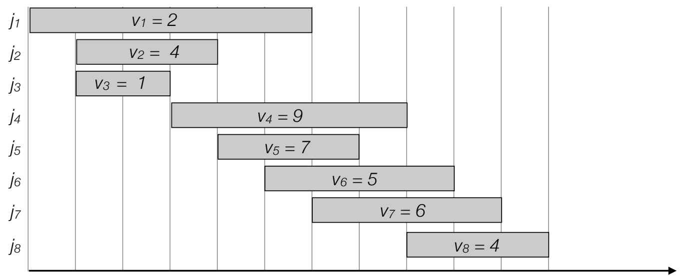


Dynamic Programming

- Greedy. Build solution incrementally, optimizing some local criterion.
- Divide-and-conquer. Break up problem into independent subproblems, solve each subproblem, and combine to get solution to original problem.
- Dynamic programming. Break up problem into overlapping subproblems, and build up solutions to larger and larger subproblems.
  - Can be used when the problem have "optimal substructure":
    - *Solution can be constructed from optimal solutions to subproblems*
    - *Use dynamic programming when subproblems overlap.*

Weighted interval scheduling

- Weighted interval scheduling problem
  - n jobs (intervals)
  - Job *i* starts at s_i, finishes at f_i and has weight/value v_i.
  - Goal: Find maximum weight subset of non-overlapping (compatible) jobs.

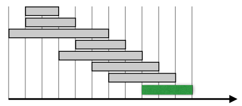


Weighted interval scheduling

- OPT(j) = value of optimal solution to the problem consisting job requests 1,2,..,*j*.
  - Case 1. OPT(*j*) selects job j
    OPT(j) = v_j + optimal solution to subproblem on 1,…,p(j)
  - Case 2. OPT(*j*) does not select job j
    OPT = optimal solution to subproblem 1,…j-1

- Recurrence:

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \text{otherwise} \end{cases}$$



Subset Sum

- Subset Sum
  - Given *n* items $\{1,\ldots,n\}$
  - Item i has weight w_i
  - Bound W
  - Goal: Select maximum weight subset S of items so that

$$\sum_{i \in S} w_i \leq W$$

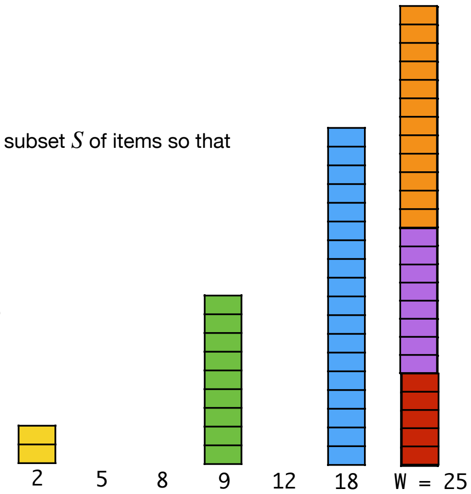
- Example
  - {2, 5, 8, 9, 12, 18} and W = 25.
  - Solution: 5 + 8 + 12 = 25.

Subset Sum

- Subset Sum
  - Given *n* items $\{1,\ldots,n\}$
  - Item i has weight w_i
  - Bound W
  - Goal: Select maximum weight subset S of items so that

$$\sum_{i \in S} w_i \leq W$$

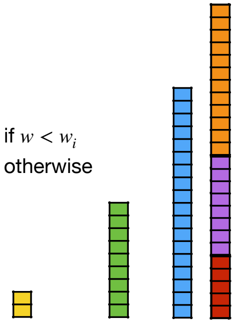
- Example
  - {2, 5, 8, 9, 12, 18} and W = 25.
  - Solution: 5 + 8 + 12 = 25.



Subset Sum

- $\mathcal{O}$ = optimal solution
- Consider element n.
  - Either in $\mathcal{O}$ or not.
    - $n \notin \mathcal{O}$: Optimal solution using items $\{1,\ldots,n-1\}$ is equal to $\mathcal{O}$.
    - $n \in \mathcal{O}$: Value of $\mathcal{O} = w_n$ + weight of optimal solution on $\{1,\ldots,n-1\}$ with capacity $W - w_n$.
- Recurrence

$$\text{OPT}(i,w) = \begin{cases} \text{OPT}(i-1,w) & \text{if } w < w_i \\ \max(\text{OPT}(i-1,w), w_i + \text{OPT}(i-1,w-w_i)) & \text{otherwise} \end{cases}$$



Subset Sum

- Recurrence:

$$\text{OPT}(i,w) = \begin{cases} \text{OPT}(i-1,w) & \text{if } w < w_i \\ \max(\text{OPT}(i-1,w), w_i + \text{OPT}(i-1,w-w_i)) & \text{otherwise} \end{cases}$$

- Example
  - {1, 2, 5, 8, 9} and W = 12

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

Knapsack

- $\mathcal{O}$ = optimal solution
- Consider element n.
  - Either in $\mathcal{O}$ or not.
    - $n \notin \mathcal{O}$: Optimal solution using items $\{1,\ldots,n-1\}$ is equal to $\mathcal{O}$.
    - $n \in \mathcal{O}$: Value of $\mathcal{O} = v_n$ + value on optimal solution on $\{1,\ldots,n-1\}$ with capacity $W - w_n$.
- Recurrence
  - $\text{OPT}(i,w)$ = optimal solution on $\{1,\ldots,i\}$ with capacity w.

$$\text{OPT}(i,w) = \begin{cases} \text{OPT}(i-1,w) & \text{if } w < w_i \\ \max(\text{OPT}(i-1,w), v_i + \text{OPT}(i-1,w-w_i)) & \text{otherwise} \end{cases}$$

- Running time $O(nW)$

Sequence alignment

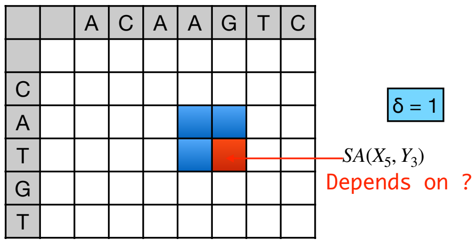
- How similar are ACAAGTC and CATGT.
- Align them such that
  - all items occurs in at most one pair.
  - no crossing pairs.
- Cost of alignment
  - gap penalty δ
  - mismatch cost for each pair of letters α(p,q).
- Goal: find minimum cost alignment.
- Input to problem: 2 strings A nd Y, gap penalty δ, and penalty matrix α(p,q).

```
A C A A G T C        A C A A - G T C
- C A T G T -        - C A - T G T -

1 mismatch, 2 gaps   0 mismatches, 4 gaps
```

Sequence alignment

$$SA(X_i, Y_j) = \begin{cases} j\delta & \text{if } i = 0 \\ i\delta & \text{if } j = 0 \\ \min \begin{cases} \alpha(x_i, y_j) + SA(X_{i-1}, Y_{j-1}), \\ \delta + SA(X_i, Y_{j-1}), \\ \delta + SA(X_{i-1}, Y_j)\} \end{cases} & \text{otherwise} \end{cases}$$



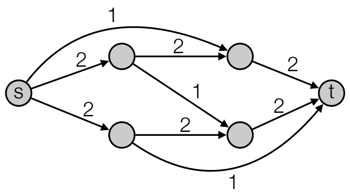
Penalty matrix

	A	C	G	T
A	0	1	2	2
C	1	0	2	3
G	2	2	0	1
T	2	3	1	0

Dynamic programming

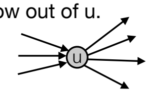
- **First formulate the problem recursively.**
  - Describe the *problem* recursively in a clear and precise way.
  - Give a recursive formula for the problem.
- **Bottom-up**
  - Identify all the subproblems.
  - Choose a memoization data structure.
  - Identify dependencies.
  - Find a good evaluation order.
- **Top-down**
  - Identify all the subproblems.
  - Choose a memoization data structure.
  - Identify base cases.
  - Remember to save results and check before computing.

Network Flow



- Network flow:
  - graph G=(V,E).
  - Special vertices s (source) and t (sink).
  - Every edge (u,v) has a capacity c(u,v) ≥ 0.
  - Flow:
    - **capacity constraint:** every edge e has a flow 0 ≤ f(u,v) ≤ c(u,v).
    - **flow conservation:** for all u ≠ s, t: flow into u equals flow out of u.

$$\sum_{v:(v,u)\in E} f(v,u) = \sum_{v:(u,v)\in E} f(u,v)$$



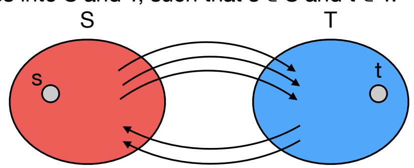
    - Value of flow f is the sum of flows out of s minus sum of flows into s:

$$|f| = \sum_{v:(s,v)\in E} f(s,v) - \sum_{v:(v,s)\in E} f(v,s)$$

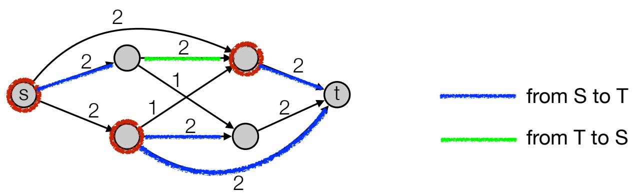
  - **Maximum flow problem:** find s-t flow of maximum value

s-t Cuts

- Cut: Partition of vertices into S and T, such that s ∊ S and t ∊ T.

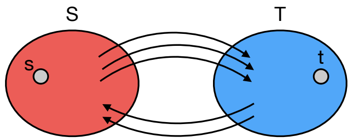


- Example



Network flow: s-t Cuts

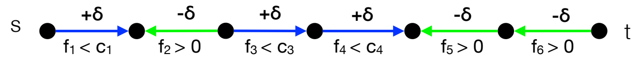
- Cut: Partition of vertices into S and T, such that s ∈ S and t ∈ T.



- Capacity of cut: total capacity of edges going ***from*** S ***to*** T.
- Flow across cut: flow from S to T minus flow from T to S.
- Value of flow any flow |f| ≤ c(S,T) for any s-t cut (S,T).
- Suppose we have found flow f and cut (S,T) such that |f| = c(S,T). Then f is a maximum flow and (S,T) is a minimum cut.

Augmenting paths

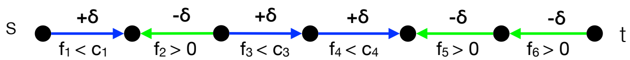
- Augmenting path: s-t path where
  - **forward** edges have leftover capacity
  - **backwards** edges have positive flow



- There is no augmenting path <=> f is a maximum flow.
- Ford-Fulkerson algorithm:
  - Repeatedly find augmenting path, use it, until no augmenting path exists
  - Running time: O(|f*| m).
- Edmonds-Karp algorithm:
  - Repeatedly find shortest augmenting path, use it, until no augmenting path exists
  - Use BFS to find a shortest augmenting path.
  - Running time: $O(nm^2)$
- Find minimum cut. All vertices to which there is an augmenting path from s goes into S, rest into T.

Augmenting paths

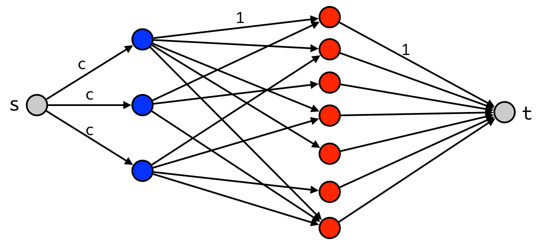
- Augmenting path: s-t path where
  - **forward** edges have leftover capacity
  - **backwards** edges have positive flow



- There is no augmenting path <=> f is a maximum flow.
- Scaling algorithm:
  - Set Δ = highest power of two that is no larger than the largest capacity out of s.
  - Until $\Delta < 1$
    - Repeatedly find augmenting path in G_Δ, use it, until no augmenting path exists.
    - Set $\Delta = \Delta/2$
  - Running time: $O(m^2 \log C)$.

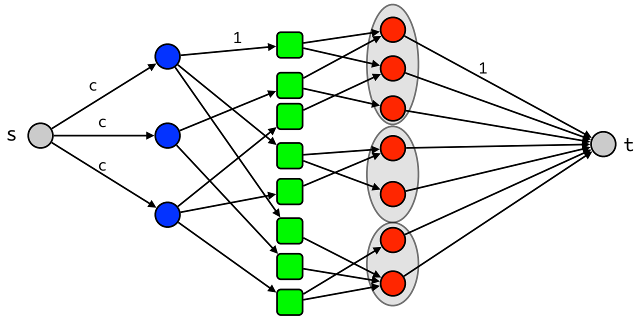
Network flow

- Can model and solve many problems via maximum flow.
  - Maximum bipartite matching
  - k edge-disjoint paths
  - capacities on vertices
  - Many sources/sinks
  - assignment problems: Example. X doctors, Y holidays, each doctor should work at at most c holidays, each doctor is available at some of the holidays.

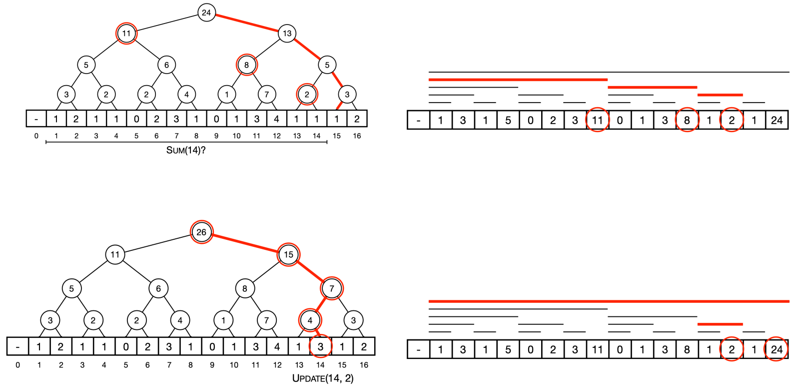


Scheduling of doctors

- X doctors, Y holidays, each doctor should work at at most c holidays, each doctor is available at some of the holidays.
- *Each doctor should work at most one day in each vacation period.*

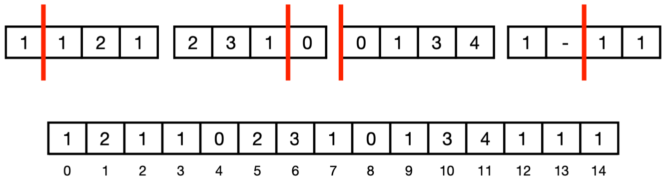


Partial sums



Dynamic array:

- 2-level rotated array

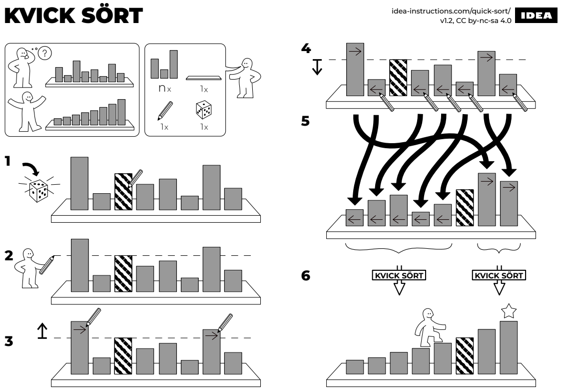


Amortized analysis

- Amortized analysis.
  - Time required to perform a sequence of data operations is averaged over all the operations performed.
- Example: dynamic tables with doubling and halving
  - If the table is full copy the elements to a new array of double size.
  - If the table is a quarter full copy the elements to a new array of half the size.
  - Worst case time for insertion or deletion: O(n)
  - Amortized time for insertion and deletion: O(1)
  - Any sequence of n insertions and deletions takes time O(n).
- Methods.
  - Aggregate method
  - Accounting method
  - Potential method

Randomized algorithms

- Contention resolution
- Minimum cut
- Coupon Collector.
- Selection
- Quicksort
- Hashing

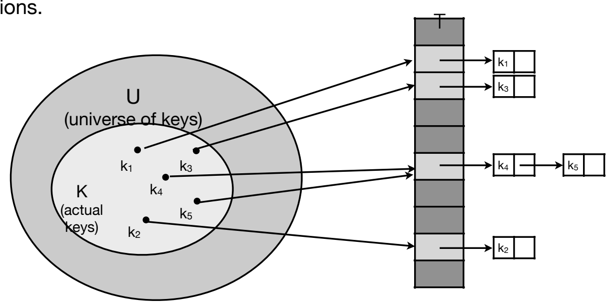


Hash tables and hash functions

- Theorem. We can solve the dictionary problem (without special assumptions) in:
  - O(n) space.
  - O(1) expected time per operation (lookup, insert, delete).
- Hash function. Given a prime p and a = $(a_1a_2\ldots a_r)_p$, define

$$h_a((x_1x_2\ldots x_r)_p) = a_1x_1 + a_2x_2 + \ldots + a_rx_r \mod p$$

- Then $H = \{h_a \mid (a_1a_2\ldots a_r)_p \in \{0,\ldots,p-1\}^r\}$ is a universal family of hash functions.

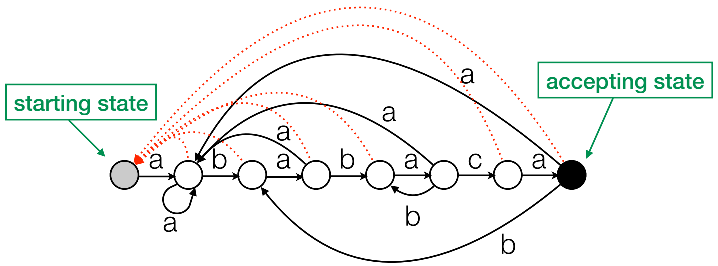


String Matching

- String matching problem:
  - string T (text) and string P (pattern) over an alphabet Σ. |T| = n, |P| = m.
  - Report all starting positions of occurrences of P in T.
- Knuth-Morris-Pratt (KMP). Running time: O(m + n)
- String matching automaton. Running time: O(n + m|Σ|)

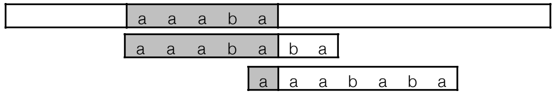
Finite Automaton

- Finite automaton: alphabet Σ = {a,b,c}. P= ababaca.



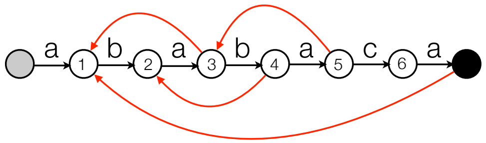
Knuth-Morris-Pratt (KMP)

- Matched P[1…q]: Find longest block P[1..k] that matches end of P[2..q].



- Find longest prefix P[1...k] of P that is a *proper* suffix of P[1...q]
- Array π[1…m]:
  - π[q] = max k < q such that P[1...k] is a suffix of P[1…q].
- Can be seen as finite automaton with *failure links*:

i	1	2	3	4	5	6	7
π[i]	0	0	1	2	3	0	1



P and NP

- P solvable in deterministic polynomial time.
- NP solvable in non-deterministic (with guessing) polynomial time. Only the time for the right guess is counted.
- P⊆NP (every problem T which is in P is also in NP).
- It is not known (but strongly believed) whether the inclusion is proper, that is whether there is a problem in NP which is not in P.
- There is subclass of NP which contains the hardest problems, NP-complete problems.
- Reductions.

Courses