Algorithms and Data Structures 2
Exam Notes
Week 2: Dynamic Programming 1

Mads Richardt

1. General Methodology and Theory

Dynamic Programming Principles

- Break problems into *overlapping subproblems* with *optimal substructure.*
- Use a recurrence relation to express solution of a large problem in terms of smaller ones.
- Implement either:
  1. **Top-down with memoization**: recursive + cache.
  2. **Bottom-up iteration**: fill table in order of subproblem dependencies.
- Running time = (number of subproblems) $\times$ (time per subproblem).
- Typical steps:
  1. Identify subproblems.
  2. Derive recurrence.
  3. Prove correctness (usually by induction).
  4. Analyze time/space.
  5. Reconstruct solution if needed.

Mathematical Tools

- Geometric sums: $\sum_{i=0}^{k} r^i = \frac{r^{k+1}-1}{r-1}$.
- Harmonic number: $H_n \approx \ln n + \gamma$.
- Logarithm rules: $\log_a b = \frac{\ln b}{\ln a}$.
- Recurrence solving (for DP analysis):

$$T(n) = T(n-1) + O(1) \implies O(n)$$
$$T(n) = T(n-1) + T(n-2) + O(1) \implies O(\varphi^n),\ \varphi = \tfrac{1+\sqrt{5}}{2}.$$

2. Notes from Slides and Textbook

Weighted Interval Scheduling (KT 6.1)

- Input: jobs $j = 1, \dots, n$, each with (s_j, f_j, v_j).
- Sort by finish time: $f_1 \le f_2 \le \cdots \le f_n$.
- Define $p(j)$ = largest $i < j$ with $f_i \le s_j$ (compatible).

- Recurrence:
$$OPT(j) = \begin{cases} 0 & j = 0, \\ \max\{v_j + OPT(p(j)),\ OPT(j-1)\} & j \geq 1. \end{cases}$$
- Running time: $O(n \log n)$ (sort + binary search for $p(j)$).

Job Planning (KT 6.2)

- Weekly jobs: revenue ℓ_i (low-stress) or h_i (high-stress).
- Constraint: if week i is high-stress, then week $i-1$ must be none.
- Recurrence:
$$OPT(i) = \max\{\ell_i + OPT(i-1),\ h_i + OPT(i-2)\}.$$
- Base: $OPT(0) = 0$, $OPT(1) = \max(\ell_1, h_1)$.

Office Switching (KT 6.4)

- Costs: N_i (NY), S_i (SF), moving cost M.
- State: $OPT(i, NY)$ = min cost up to month i, ending in NY.
- Recurrence:
$$\begin{aligned} OPT(i, NY) &= N_i + \min(OPT(i-1, NY),\ OPT(i-1, SF) + M), \\ OPT(i, SF) &= S_i + \min(OPT(i-1, SF),\ OPT(i-1, NY) + M). \end{aligned}$$
- Answer: $\min(OPT(n, NY), OPT(n, SF))$.

Grid Paths with Traps

- Grid $n \times n$, traps forbidden, moves: right or down.
- Let $P(i, j)$ = number of paths to (i, j).
- Recurrence:
$$P(i,j) = \begin{cases} 0 & \text{if trap at } (i,j), \\ 1 & \text{if } (i,j) = (1,1), \\ P(i-1,j) + P(i,j-1) & \text{otherwise.} \end{cases}$$
- Answer: $P(n, n)$.

Discrete Fréchet Distance

- Paths: $p_1, \ldots, p_n$, $q_1, \ldots, q_m$.
- Distance: $d(p, q)$ Euclidean.
- Define $L(i, j)$ = leash length needed to match $p_1 \ldots p_i$ with $q_1 \ldots q_j$.
- Recurrence:
$$L(i,j) = \max\big(d(p_i, q_j), \min\{L(i-1,j), L(i-1,j-1), L(i,j-1)\}\big).$$
- Base: $L(1, 1) = d(p_1, q_1)$.
- Answer: $L(n, m)$.

3. Solutions to Problem Set

Exercise 1: Weighted Interval Scheduling

Recursive with Memoization:

```
M[0..n] = empty
Compute-Opt(j):
  if j == 0: return 0
  if M[j] != empty: return M[j]
  M[j] = max(v[j] + Compute-Opt(p[j]), Compute-Opt(j-1))
  return M[j]
```

Iterative:

```
M[0] = 0
for j = 1..n:
  M[j] = max(v[j] + M[p[j]], M[j-1])
return M[n]
```

Exercise 2: Grid Paths

```
Paths[1..n][1..n]
for i = 1..n:
  for j = 1..n:
    if trap(i,j): Paths[i][j] = 0
    else if (i,j) == (1,1): Paths[i][j] = 1
    else: Paths[i][j] = Paths[i-1][j] + Paths[i][j-1]
return Paths[n][n]
```

Running time $O(n^2)$.

Exercise 3: Job Planning

```
OPT[0] = 0
OPT[1] = max(l1, h1)
for i = 2..n:
  OPT[i] = max(l[i] + OPT[i-1], h[i] + OPT[i-2])
return OPT[n]
```

Exercise 4: Office Switching

```
OPT_NY[1] = N1
OPT_SF[1] = S1
for i = 2..n:
  OPT_NY[i] = N[i] + min(OPT_NY[i-1], OPT_SF[i-1] + M)
  OPT_SF[i] = S[i] + min(OPT_SF[i-1], OPT_NY[i-1] + M)
return min(OPT_NY[n], OPT_SF[n])
```

Exercise 5: Discrete Fréchet Distance

```
for i = 1..n:
  for j = 1..m:
    if i == 1 and j == 1:
      L[i][j] = d(p1,q1)
    else if i == 1:
      L[i][j] = max(d(pi,qj), L[i][j-1])
    else if j == 1:
      L[i][j] = max(d(pi,qj), L[i-1][j])
    else:
      L[i][j] = max(d(pi,qj), min(L[i-1][j], L[i-1][j-1], L[i][j-1]))
return L[n][m]
```

Running time $O(nm)$, space $O(nm)$.

4. Summary

- Weighted Interval Scheduling: $OPT(j) = \max(v_j + OPT(p(j)), OPT(j-1))$.
- Job Planning: $OPT(i) = \max(\ell_i + OPT(i-1),\ h_i + OPT(i-2))$.
- Office Switching: $OPT(i, city) = \text{cost} + \min(\dots)$.
- Grid Paths: $P(i,j) = P(i-1,j) + P(i,j-1)$ (skip traps).
- Discrete Fréchet Distance: $L(i,j) = \max(d(p_i,q_j), \min\{\dots\})$.

Dynamic Programming

Algorithm Design 6.1, 6.2, 6.4

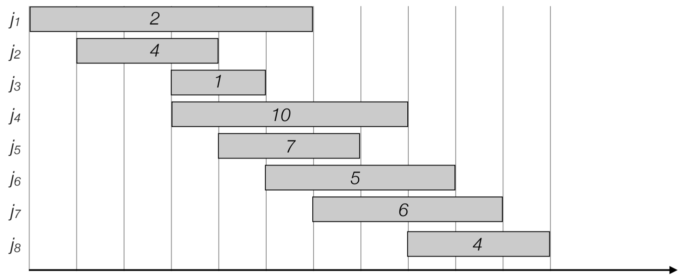
Thank you to Kevin Wayne for inspiration to slides

Applications

- In class (today and next time)

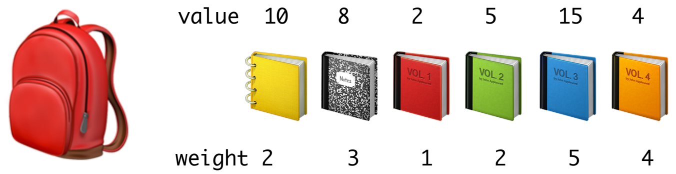
Applications

- In class (today and next time)
  - Weighted interval scheduling
    - Set of weighted intervals with start and finishing times
    - Goal: find maximum weight subset of non-overlapping intervals



Applications

- Today and next time
  - Weighted interval scheduling
  - Subset Sum and Knapsack
    - Set of items each having a weight and a value
    - Knapsack with a bounded capacity
    - Goal: fill knapsack so as to maximise the total value.



value	10	8	2	5	15	4
weight	2	3	1	2	5	4

Capacity 8

Applications

- Today and next time
  - Weighted interval scheduling
  - Subset Sum and Knapsack
  - Sequence alignment
    - Given two strings A and B how many edits (insertions, deletions, relabelings) is needed to turn A into B?

```
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
```

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.*

Computing Fibonacci numbers

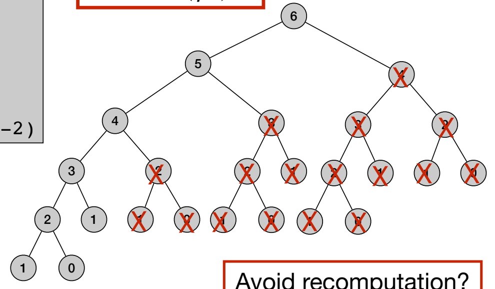
- Fibonacci numbers:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

- First try:

```
Fib(n)
if n = 0
  return 0
else if n = 1
  return 1
else
  return Fib(n-1) + Fib(n-2)
```

time $\Theta(\phi^n)$



Avoid recomputation?

Memoized Fibonacci numbers

- Fibonacci numbers:

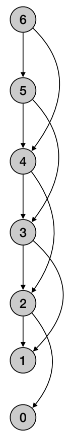
$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

- Remember already computed values:

```
for j=1 to n
  F[j] = null
Mem-Fib(n)

Mem-Fib(n)
if n = 0
  return 0
else if n = 1
  return 1
else
  if F[n] is empty
    F[n] = Mem-Fib(n-1) + Mem-Fib(n-2)
  return F[n]
```

time $\Theta(n)$



Bottom-up Fibonacci numbers

- Fibonacci numbers:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

- Remember already computed values:

```
Iter-Fib(n)
F[0] = 0
F[1] = 1
for i = 2 to n
  F[i] = F[i-1] + F[i-2]
return F[n]
```

time $\Theta(n)$
space $\Theta(n)$

Bottom-up Fibonacci numbers - save space

- Fibonacci numbers:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

- Remember last two computed values:

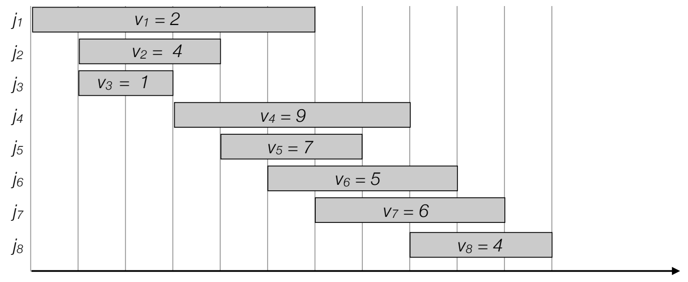
```
Iter-Fib(n)
previous = 0
current = 1
for i = 1 to n
  next = previous + current
  previous = current
  current = next
return current
```

time $\Theta(n)$
space $\Theta(1)$

Weighted Interval Scheduling

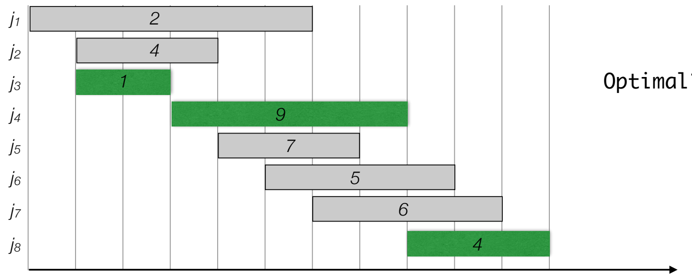
Weighted interval scheduling

- 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

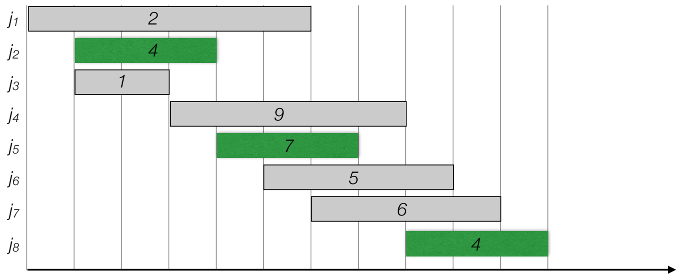
- Weighted interval scheduling problem
  - n jobs (intervals)
  - Job *i* starts at s_i, finishes at f_i and has weight/value v_i.
  - Goal: Find maximum weight subset of non-overlapping (compatible) jobs.



Optimal?

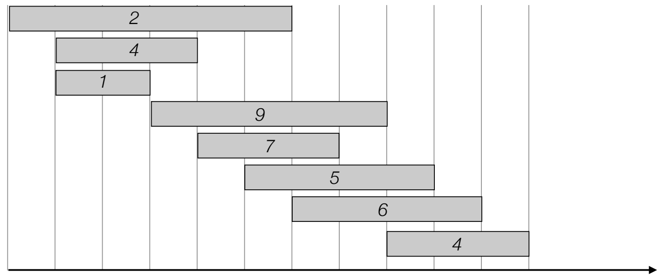
Weighted interval scheduling

- 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.



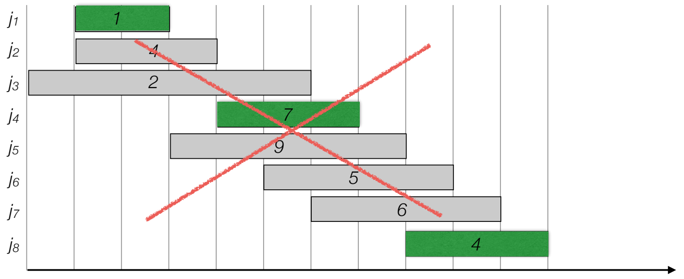
Weighted interval scheduling

- Label/sort jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$



Weighted interval scheduling

- Label/sort jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$
- ~~Greedy?~~

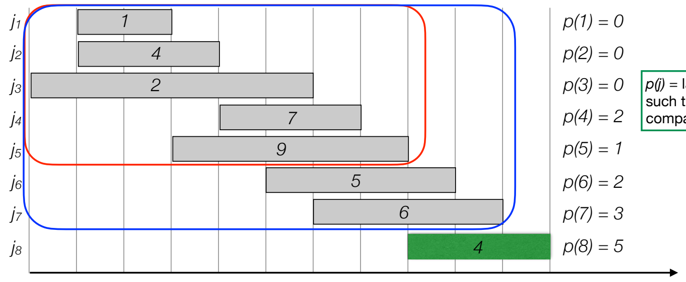


Weighted interval scheduling

- Label/sort jobs by finishing time: $f_1 \le f_2 \le \ldots \le f_n$
- Optimal solution OPT:
  - Case 1. OPT selects last job

    OPT = v_n + optimal solution to subproblem on the subset of jobs ending before job n starts
  - Case 2. OPT does not select last job

    OPT = optimal solution to subproblem on 1,…,n-1



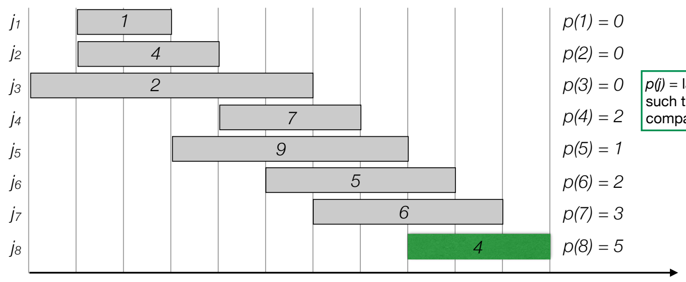
p(j) = largest index i < j such that job i is compatible with j.

Weighted interval scheduling

- Label/sort jobs by finishing time: $f_1 \le f_2 \le \ldots \le f_n$
- Optimal solution OPT:
  - Case 1. OPT selects last job

    OPT = v_n + optimal solution to subproblem on 1,…,p(n)
  - Case 2. OPT does not select last job

    OPT = optimal solution to subproblem on 1,…,n-1



p(j) = largest index i < j such that job i is compatible with j.

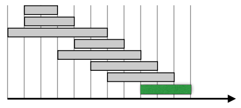
Weighted interval scheduling

- 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:

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



Weighted interval scheduling: brute force

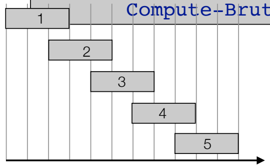
$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \texttt{otherwise} \end{cases}$$

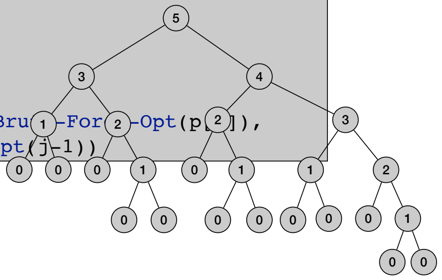
```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]
Compute-BruteForce—Opt(n)

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```

$\boxed{\text{time } \Theta(2^n)}$





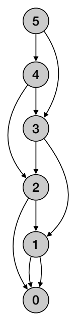
Weighted interval scheduling: memoization

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

for j=1 to n
  M[j] = null
M[0] = 0.
Compute-Memoized-Opt(n)

Compute-Memoized-Opt(j)
if M[j] is empty
  M[j] = max(v[j] + Compute-Memoized-Opt(p[j]),
         Compute-Memoized-Opt(j-1))
return M[j]
```



- Running time O(n log n):
  - Sorting takes O(n log n) time.
  - Computing p(n): O(n log n) - use log n time to find each p(i).
  - Each subproblem solved once.
  - Time to solve a subproblem constant.
- Space O(n)

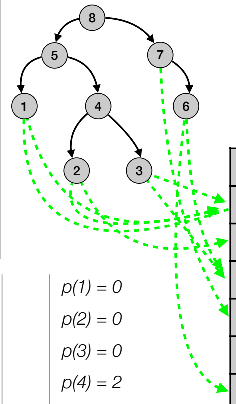
Weighted interval scheduling: memoization

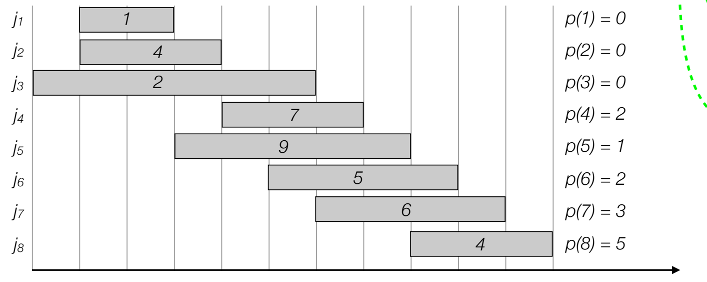
```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

for j=1 to n
   M[j] = empty
M[0] = 0.
Compute-Memoized-Opt(n)

Compute-Memoized-Opt(j)
if M[j] is empty
   M[j] = max(v[j] + Compute-Memoized-Opt(p[j]),
            Compute-Memoized-Opt(j-1))
return M[j]
```





p(1) = 0
p(2) = 0
p(3) = 0
p(4) = 2
p(5) = 1
p(6) = 2
p(7) = 3
p(8) = 5

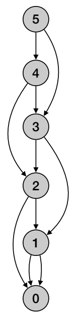
i	M[i]
0	0
1	1
2	4
3	4
4	11
5	11
6	11
7	11
8	15

Weighted interval scheduling: bottom-up

```
Compute-Bottom-Up-Opt(n, s[1..n], f[1..n], v[1..n])

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

M[0] = 0.
for j=1 to n
  M[j] = max(v[j] + M(p[j]), M(j-1))
return M[n]
```

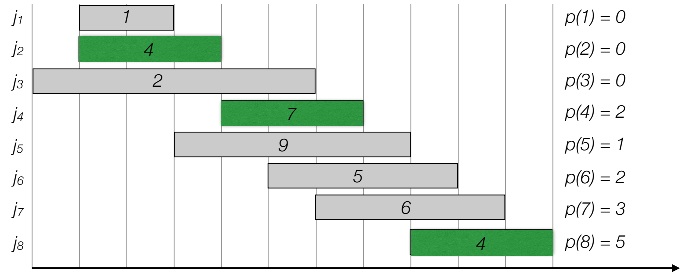


- Running time O(n log n):
  - Sorting takes O(n log n) time.
  - Computing p(n): O(n log n)
  - For loop: O(n) time
    - Each iteration takes constant time.
- Space O(n)

Weighted interval scheduling: find solution

```
Find-Solution(j)
if j=0
  Return emptyset
else if M[j] > M[j-1]
  return {j} ∪ Find-Solution(p[j])
else
  return Find-Solution(j-1)
```

Solution = 8 , 4 , 2



p(1) = 0
p(2) = 0
p(3) = 0
p(4) = 2
p(5) = 1
p(6) = 2
p(7) = 3
p(8) = 5

i	M[i]
0	0
1	1
2	4
3	4
4	11
5	11
6	11
7	11
8	15

Reading material

We will introduce the paradigm *dynamic programming*. You should read KT Section 6.1 and 6.2. $[w]$ on an exercise means it is a warmup exercise.

Exercises

1 $[w]$ **Weighted interval scheduling** Solve the following weighted interval scheduling problem using both the recursive method with memoization and the iterative method. The intervals are given as triples (s_i, f_i, v_i): $S = \{(1,7,4),(10,12,2),(2,5,3),(8,11,4),(12,13,3),(3,9,5),(3,4,3),(4,6,3),(5,8,2),(4,13,6)\}$.

2 Grid Paths Consider an $n \times n$ grid whose squares may have traps. It is not allowed to move to a square with a trap. Your task is to calculate the number of paths from the upper-left square to the lower-right square. You can only move right or down.

2.1 Give an algorithm that return the number of paths and analyse its running time.

2.2 Implement your algorithm on CSES: https://cses.fi/problemset/task/1638

3 Job planning Solve KT 6.2.

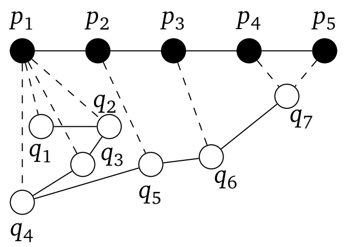
4 Office switching Solve KT 6.4.

5 Discrete Fréchet distance Consider Professor Bille going for a walk with his dog. The professor follows a path of points $p_1, \dots, p_n$ and the dog follows a path of points $q_1, \dots, q_m$. We assume that the walk is partitioned into a number of small steps, where the professor and the dog in each step either both move from p_i tp p_{i+1} and from q_j to q_{j+1}, respectively, or only one of them moves and the other one stays.

The goal is to find the smallest possible length L of the leash, such that the professor and the dog can move from p_1 and q_1, resp., to p_n and q_n. They cannot move backwards, and we only consider the distance between points. The distance L is also known as the discrete Fréchet distance.

We let $L(i,j)$ denote the smallest possible length of the leash, such that the professor and the dog can move from p_1 and q_1 to p_i and q_j, resp. For two points p and q, let $d(p,q)$ denote the distance between the them.

In the example below the dotted lines denote where Professor Bille (black nodes) and the dog (white nodes) are at time 1 to 8. The minimum leash length is $L = d(p_1, q_4)$.



5.1 Give a recursive formula for $L(i,j)$.

5.2 Give pseudo code for an algorithm that computes the length of the shortest possible leash. Analyze space and time usage of your solution.

5.3 Extend your algorithm to print out paths for the professor and the dog. The algorithm must return where the professor and the dog is at each time step. Analyze the time and space usage of your solution.

Puzzle of the week: 101 ants [1]There are 101 ants on a rod of length 1 metre. 100 of them are black and positioned randomly along the 1 metre rod. The 101st ant is red and is positioned in the middle of the rod. All ants are travelling at 1 metre/minute either right or left at the start. The ants are also perfectly elastic, so that if two ants collide they simply turn round and carry on at 1 metre/minute in the opposite direction. The rod has been capped at both ends so that an ant that reaches the end of the rod simply turns round and carries on travelling back towards the middle of the rod (still at 1 metre/minute). Each ant starts off heading in a random direction. What is the probability that after exactly 1 hour the red ant is back exactly in the middle of the rod?

The ants are arbitrarily positioned on the rod and are travelling at 1 metre/minute either right or left at the start.

[1]I got this puzzle from Raphäel Clifford