ADS2 — Dynamic Programming 1 (Week Plan)

Metadata

Field	Value
Title	ADS2 — DP1 Notes & Solutions
Date	2025-10-18 (Europe/Copenhagen)
Author	Mads Richardt
Sources used	

weekplan.pdf (Exercises block p.1); *DP1-4x1.pdf* (slides pp. 11–24, memoization & bottom-up); *Algorithm Design.pdf* (KT §6.1–6.2, WIS details); *6_2.pdf* (KT 6.2 Job Planning, Exercises p.313–314); *6_4.pdf* (KT 6.4 Office Switching, Exercises p.315–316).

Week plan filename	weekplan.pdf

General Methodology and Theory

- Dynamic programming (DP): overlapping subproblems + optimal substructure.
- Recipe: 1) choose subproblem index; 2) write correct recurrence; 3) pick order (top-down memo or bottom-up); 4) initialize bases; 5) prove/argue correctness; 6) analyze time/space; 7) recover solution (parent/backpointers).
- Complexity hygiene: sort once when needed; precompute helpers (e.g., predecessor index $p(j)$); space trimming when only previous rows/cols are needed.

Notes

- **Weighted Interval Scheduling (WIS)** — sort by finish times; let $p(j)$ be the rightmost non-overlapping job before j . Recurrence $M[j] = \max\{v_j + M[p(j)],\ M[j-1]\}$ with $M[0] = 0$. Solution via traceback using $v_j + M[p(j)] \ge M[j-1]$.
- **Grid Paths with traps** — count paths on an $n \times n$ grid moving only right/down, avoiding blocks. DP $dp[i][j] = 0$ if trap; else $dp[i][j] = dp[i-1][j] + dp[i][j-1]$ with borders guarded. (CSES 1638 uses modulo $10^9 + 7$.)
- **Job Planning (KT 6.2)** — choose each week: low ℓ_i , high h_i (requires week $i-1$ = none), or none. Let $OPT[i] = \max\{OPT[i-1] + \ell_i,\ OPT[i-2] + h_i,\ OPT[i-1]\}$; since $\ell_i \ge 0$, this simplifies to $OPT[i] = \max\{OPT[i-1] + \ell_i,\ OPT[i-2] + h_i\}$ with $OPT[0] = 0,\ OPT[1] = \max\{\ell_1, h_1\}$.
- **Office Switching (KT 6.4)** — two states per month: end in NY or SF. $NY[i] = \min\{NY[i-1],\ SF[i-1] + M\} + N_i$, $SF[i] = \min\{SF[i-1],\ NY[i-1] + M\} + S_i$; answer $\min\{NY[n], SF[n]\}$; backtrack for the offices sequence.

- **Discrete Fréchet distance** — sequences $p_1..p_n$ and $q_1..q_m$; leash length is minimax over coupled monotone walks. DP $L(i,j) = \max\Big(d(p_i,q_j),\ \min\{L(i-1,j),\ L(i-1,j-1),\ L(i,j-1)\}\Big)$ with edges-only moves and bases $L(1,1) = d(p_1,q_1)$, $L(i,1) = \max\{d(p_i,q_1), L(i-1,1)\}$, $L(1,j) = \max\{d(p_1,q_j), L(1,j-1)\}$.

Coverage Table

Weekplan ID	Canonical ID	Title/Label (verbatim)	Assignment Source	Text Source	Status
1	WIS	[w] Weighted interval scheduling — solve by memoization **and** iterative	weekplan.pdf §Exercises/1	DP1-4x1.pdf (slides pp. 11–24); Algorithm Design §6.1	Solved
2.1	—	Grid Paths — algorithm + analysis	weekplan.pdf §Exercises/2.1	CSES 1638 statement (external); weekplan description	Solved
2.2	CSES 1638	Grid Paths — implement on CSES	weekplan.pdf §Exercises/2.2	CSES 1638 I/O micro-card below	Solved
3	KT 6.2	Job planning — Solve KT 6.2	weekplan.pdf §Exercises/3	6_2.pdf Exercises p.313–314	Solved
4	KT 6.4	Office switching — Solve KT 6.4	weekplan.pdf §Exercises/4	6_4.pdf Exercises p.315–316	Solved
5.1	—	Discrete Fréchet distance — recursive formula for $L(i,j)$	weekplan.pdf §Exercises/5.1	weekplan.pdf figure p.1; Pasted image.png	Solved
5.2	—	Discrete Fréchet — pseudocode + time/space	weekplan.pdf §Exercises/5.2	weekplan.pdf figure p.1; Pasted image.png	Solved
5.3	—	Discrete Fréchet — output actual paths	weekplan.pdf §Exercises/5.3	weekplan.pdf figure p.1; Pasted image.png	Solved

MISMATCH: none. All week-plan items enumerated and solved.

Solutions

Exercise 1 — WIS

Source tags. Assignment: weekplan §1. Text: slides DP1-4x1 (WIS); KT §6.1–6.2.

Concept mapping. Intervals → weighted jobs; conflict if they overlap; build $p(j)$.

Method. Sort by finish; precompute $p[1..n]$ via binary search; fill $M[0..n]$ bottom-up; traceback.

```
Algorithm: weighted_interval_scheduling
Input: jobs 1..n with (s[i], f[i], v[i])
Output: optimal value and one optimal subset

sort by f ascending; compute p[1..n]
M[0] ← 0
for j = 1..n:
  M[j] ← max(v[j] + M[p[j]], M[j-1])
// recover
sol ← ∅; j ← n
while j > 0:
  if v[j] + M[p[j]] ≥ M[j-1]:
    sol ← sol ∪ {j}; j ← p[j]
  else:
    j ← j-1
return (M[n], sol)
// Time: O(n log n); Space: O(n)
```

Worked instance (from plan). Jobs 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)}.

- Sorted by finish: (3,4,3),(2,5,3),(4,6,3),(1,7,4),(5,8,2),(3,9,5),(8,11,4),(10,12,2),(12,13,3),(4,13,6).
- Optimal value $M[n] = 13$ with one certificate set (by original indices): {(3,4,3) id7, (4,6,3) id8, (8,11,4) id4, (12,13,3) id5}.

Verification. Feasible (non-overlapping), value 3+3+4+3=13; DP optimal by recurrence induction.

Pitfalls. Wrong $p(j)$; unstable tie-breaking on equal finishes; forgetting $M[0] = 0$.

Variant drill. If two intervals share finish time, break ties by smaller start to keep $p(j)$ monotone; correctness unchanged.

Transfer Pattern. Archetype: weighted independent set on interval graph. Recognition cues: intervals, weights, “non-overlap”. Mapping: vertices→jobs; edge when overlap; solve via finish-sorted DP with $p(j)$. Certificate: list of job indices. Anti-cues: arbitrary graphs (requires MWIS, not this DP).

Exercise 2.1 — Grid Paths (count & analyze)

Source tags. Assignment: weekplan §2.1; Text: plan statement.

```
Algorithm: grid_paths_count
Input: n, grid[1..n][1..n] with '.' free and '*' trap
Output: number of valid paths from (1,1) to (n,n)
```

```
dp[1..n][1..n] ← 0
if grid[1][1] ≠ '*': dp[1][1] ← 1
for i=1..n:
  for j=1..n:
    if grid[i][j] = '*': continue
    if i>1: dp[i][j] ← dp[i][j] + dp[i-1][j]
    if j>1: dp[i][j] ← dp[i][j] + dp[i][j-1]
return dp[n][n]
// Time: O(n^2); Space: O(n^2) (or O(n) with row rolling)
```

Verification. Each path's last step is from up or left; bases handle borders and traps.

Pitfalls. Missing modulo on platforms that require it; off-by-one at (1,1).

Variant drill. Add diagonal moves → add $+dp[i-1][j-1]$ term; still $O(n^2)$.

Transfer Pattern. Archetype: counting paths in a DAG. Cues: acyclic moves (right/down), obstacles. Mapping: nodes=grid cells, edges allowed moves, dp=node counts. Certificate: optional small grid hand-trace.

Exercise 2.2 — CSES 1638 I/O micro-card

- **Input.** n $(1 \le n \le 1000)$ then n lines of length n with characters in {'.','*'}. Use modulo 10^9+7 .
- **Output.** Single integer: number of paths from (1,1) to (n,n).
- **Edge cases.** Start or end is '*': answer 0. Prefer rolling row to cut space to $O(n)$.

Exercise 3 — KT 6.2 Job Planning

Source tags. Assignment: weekplan §3; Text: KT 6.2 (Exercises p.313–314).

Method. One-dim DP with skip for high-stress.

```
Algorithm: job_planning
Input: n; arrays l[1..n], h[1..n]
Output: OPT[n] (max value) and a plan

OPT[0] ← 0; prev[0] ← none
OPT[1] ← max(l[1], h[1]); prev[1] ← (l[1]≥h[1] ? L : H)
for i=2..n:
  a ← OPT[i-1] + l[i]   // take low this week
  b ← OPT[i-2] + h[i]   // take high, forcing i-1 = none
  if a ≥ b: OPT[i] ← a; prev[i] ← L
  else:     OPT[i] ← b; prev[i] ← H
// reconstruct by stepping i ← i-1 after L, or i ← i-2 after H
```

```
return (OPT[n], plan)
// Time: O(n); Space: O(n) (O(1) if only value needed)
```

Worked tiny example (plan table in weekplan). Weeks 1..4 with $\ell = [10, 1, 10, 10],\ h = [5, 50, 5, 1]$: value 70 via plan [none, H, L, L].

Pitfalls. Forgetting that “none” is dominated by taking ℓ_i when $\ell_i > 0$; wrong base for $i = 1$.

Transfer Pattern. Archetype: “house-robber with bonuses” (skip-one for high). Cues: “high requires a rest”, “weekly choice low/high”. Mapping: $OPT[i-2]$ for high; $OPT[i-1]$ for low. Certificate: week labels (L/H/∅) and sum.

Exercise 4 — KT 6.4 Office Switching

Source tags. Assignment: weekplan §4; Text: KT 6.4 (Exercises p.315–316).

```
Algorithm: office_switching
Input: n, move cost M; arrays N[1..n], S[1..n]
Output: min total cost and one optimal location sequence

NY[1] ← N[1]; SF[1] ← S[1]
for i=2..n:
  NY[i] ← min(NY[i-1], SF[i-1] + M) + N[i]
  SF[i] ← min(SF[i-1], NY[i-1] + M) + S[i]
// value and backtrack
cost ← min(NY[n], SF[n]) ; end ← (NY[n] ≤ SF[n] ? NY : SF)
// backtrack by comparing the chosen min at each i
return (cost, sequence)
// Time: O(n); Space: O(n) (O(1) for value)
```

Worked example (from text). $M = 10$, $N = [1, 3, 20, 30],\ S = [50, 20, 2, 4]$ → cost 20 with sequence [NY, NY, SF, SF].

Pitfalls. Starting bias (must allow either city in month 1); forgetting the +M on switches only.

Transfer Pattern. Archetype: two-state DP with switch penalty. Cues: “per-period cost + fixed switch cost”. Mapping: state per city; transition min of stay vs switch+M. Certificate: state sequence and accumulated cost table.

Exercise 5.1–5.3 — Discrete Fréchet Distance

Source tags. Assignment: weekplan §5.1–5.3; Text: figure in weekplan (also Pasted image.png).

5.1 Recurrence. $L(i,j) = \max\Big(d(p_i, q_j),\ \min\{L(i-1,j),\ L(i-1,j-1),\ L(i,j-1)\}\Big)$ with bases $L(1,1) = d(p_1, q_1)$, $L(i,1) = \max\{d(p_i, q_1), L(i-1,1)\}$, $L(1,j) = \max\{d(p_1, q_j), L(1, j-1)\}$.

5.2 Algorithm & bounds.

```
Algorithm: discrete_frechet
Input: sequences p[1..n], q[1..m]; distance d(·,·)
Output: L[n][m] (min leash length)

for i=1..n: for j=1..m:
  if i=1 and j=1: L[1][1] ← d(p1,q1)
  else if i=1:    L[1][j] ← max(d(p1,qj), L[1][j-1])
  else if j=1:    L[i][1] ← max(d(pi,q1), L[i-1][1])
  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]
// Time: O(nm); Space: O(nm) (O(min(n,m)) with two rows)
```

5.3 Paths (witness walks). Keep a parent pointer choosing the arg-min among {$L(i-1,j), L(i-1,j-1), L(i,j-1)$ } that achieved the minimum; break ties lexicographically (↖, ↑, ←). Backtrack from (n,m) to $(1,1)$ to output both timelines (professor & dog positions at each step). Space $O(nm)$ (or store only decisions and reconstruct online).

Verification. Monotone coupling constraint is enforced by grid neighbors; objective is the minimax (the outer max with the distance term). Standard exchange argument shows optimal substructure.

Pitfalls. Using sum instead of max; forgetting bases; not handling equal-distance ties (choose stable order for reproducibility).

Transfer Pattern. Archetype: alignment-style DP with minimax objective. Cues: two sequences, coupled monotone moves, distance metric, “shortest leash”. Mapping: DP grid; transitions from (i-1,j),(i-1,j-1),(i,j-1); value = max of local distance and best prefix. Certificate: value L(n,m) and a pair of monotone walks.

Puzzle — “101 ants” (week plan)

Claim. Probability the red ant (starting at the middle) is exactly at the middle after 1 hour is **1**.

- Reason: identical speed + elastic collisions are equivalent to ants passing through each other while keeping velocities; reflections at capped ends just flip direction. A single ant starting at the center with speed 1 crosses the center every 1 minute; after 60 minutes it is at the center deterministically, regardless of its initial direction.

Summary (1 page)

- **DP playbook**: define subproblems; prove recurrence; choose order; set bases; implement; recover witness. Prefer bottom-up for clarity; memoize when recursion mirrors the recurrence.
- **Patterns**: (i) interval DP (WIS) with predecessor $p(j)$; (ii) grid DAG counts; (iii) two-state switching with penalty; (iv) sequence alignment/minimax (Fréchet).

- **Complexities**: WIS $O(n \log n)$ time $O(n)$ space; Grid Paths $O(n^2)$ /$O(n)$; Job Planning $O(n)$ / $O(1)$; Office Switching $O(n)$ /$O(1)$; Fréchet $O(nm)$ /$O(\min\{n, m\})$.
- **Notation blurb**: $M[j]$ DP value; $p(j)$ predecessor index; $dp[i][j]$ 2-D DP; δ (delta) bottleneck; all indices 1-based.

Dynamic Programming

Algorithm Design 6.1, 6.2, 6.4

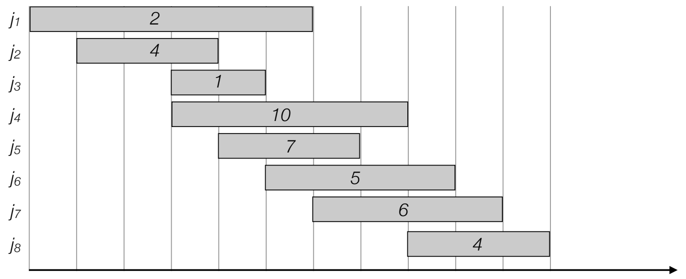
Thank you to Kevin Wayne for inspiration to slides

Applications

- In class (today and next time)

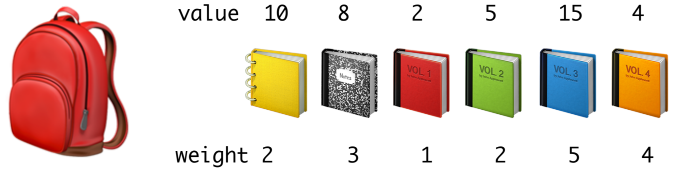
Applications

- 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

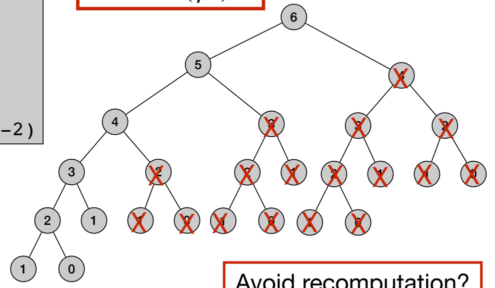
- Fibonacci numbers:

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

- 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

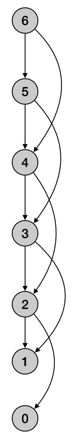
- Fibonacci numbers:

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

- 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
```

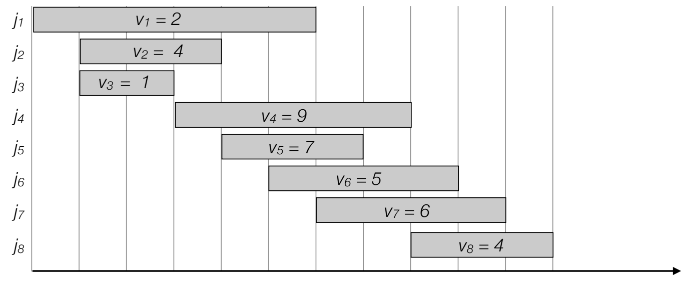
time $\Theta(n)$

space $\Theta(1)$

Weighted Interval Scheduling

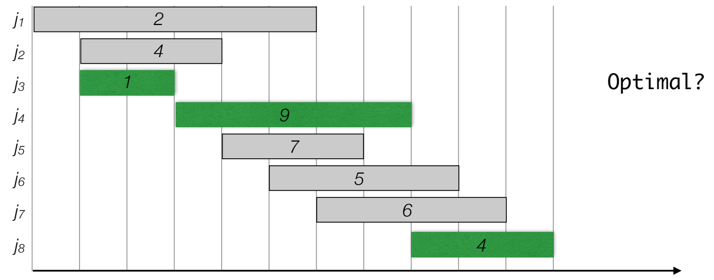
Weighted interval scheduling

- 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.



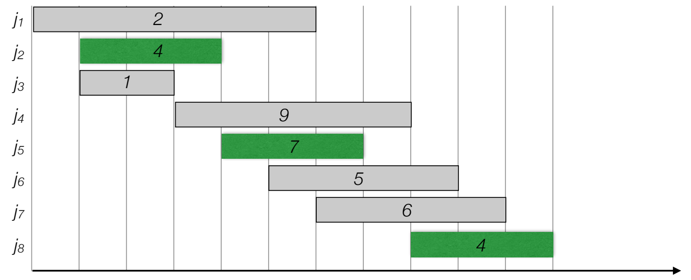
Weighted interval scheduling

- 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.



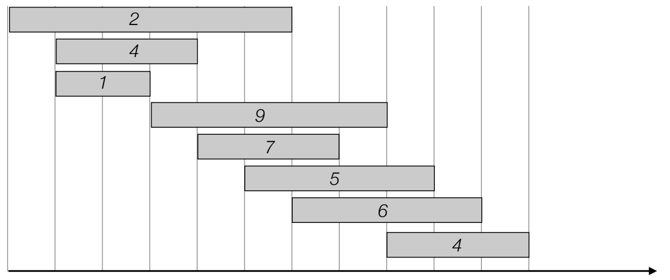
Weighted interval scheduling

- 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.



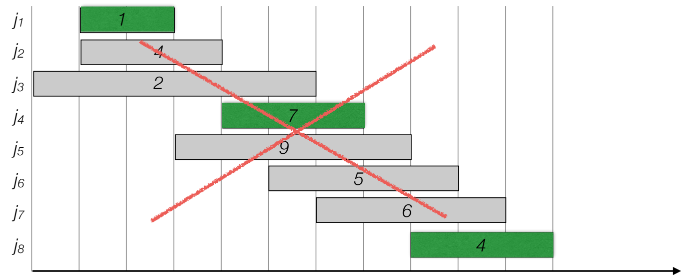
Weighted interval scheduling

- 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?~~

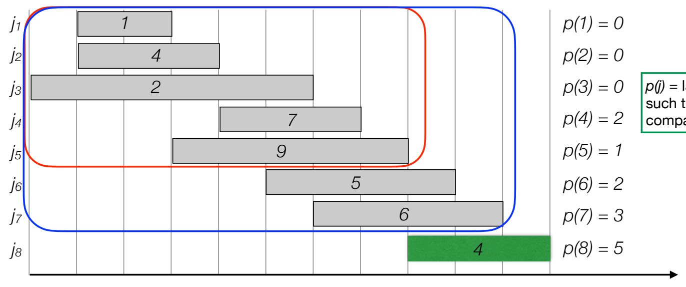


Weighted interval scheduling

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



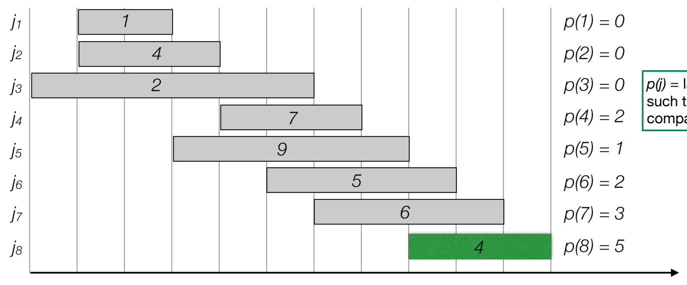
p(j) = largest index *i* < *j* such that job *i* is compatible with *j*.

Weighted interval scheduling

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

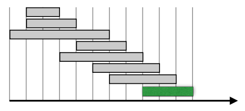
Weighted interval scheduling

- 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:

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

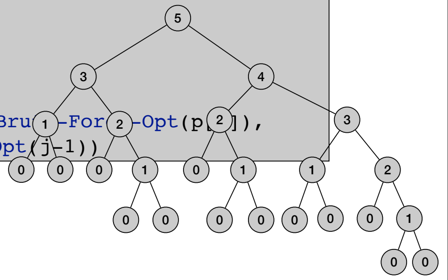


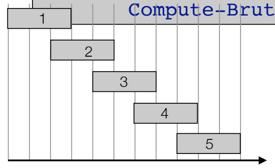
Weighted interval scheduling: brute force

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

```
~~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))
```

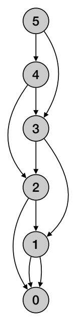
$\boxed{\text{time } \Theta(2^n)}$





Weighted interval scheduling: memoization

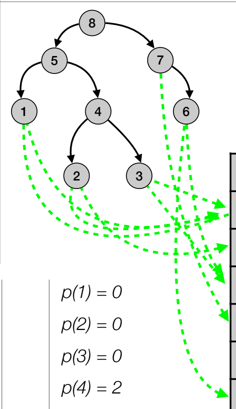
```
Input: n, s[1..n], f[1..n], v[1..n]

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]
```



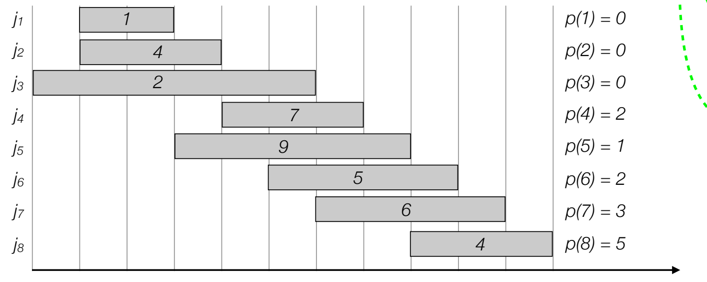
- Running time O(n log n):
  - Sorting takes O(n log n) time.
  - Computing p(n): O(n log n) - use log n time to find each p(i).
  - Each subproblem solved once.
  - Time to solve a subproblem constant.
- Space O(n)

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]
```



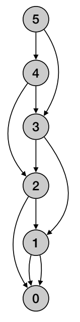
i	M[i]
0	0
1	1
2	4
3	4
4	11
5	11
6	11
7	11
8	15



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

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]
```



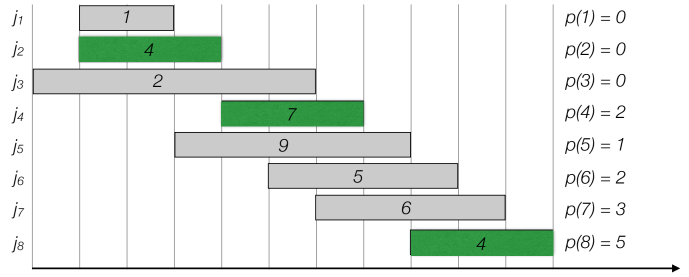
- Running time O(n log n):
  - Sorting takes O(n log n) time.
  - Computing p(n): O(n log n)
  - For loop: O(n) time
    - Each iteration takes constant time.
- Space O(n)

Weighted interval scheduling: find solution

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

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

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

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.

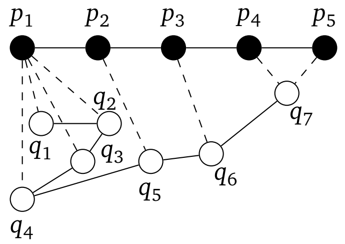
4 Office switching Solve KT 6.4.

5 Discrete Fréchet distance Consider Professor Bille going for a walk with his dog. The professor follows a path of points $p_1, \ldots, p_n$ and the dog follows a path of points $q_1, \ldots, q_m$. 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