COMP251: Dynamic programming (1)

Jérôme Waldispühl
School of Computer Science
McGill University

Based on (Cormen et al., 2002) & (Kleinberg & Tardos, 2005)
Algorithms paradigms

• **Greedy:**
  o Build up a solution incrementally.
  o Iteratively decompose and reduce the size of the problem.
  o Top-down approach.

• **Dynamic programming:**
  o Solve all possible sub-problems.
  o Assemble them to build up solutions to larger problems.
  o Bottom-up approach.
An example?

1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 = ?

20!

1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 = ?

21

Principle: Use answers previously computed for a smaller instance
INTRODUCTION
Activity-selection Problem

• **Input:** Set $S$ of $n$ activities, $a_1, a_2, ..., a_n$.
  - $s_i =$ start time of activity $i$.
  - $f_i =$ finish time of activity $i$.

• **Output:** Subset $A$ of maximum number of compatible activities.
  - 2 activities are compatible, if their intervals do not overlap.

Example:
Activity-selection Problem

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>$f_i$</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Activities sorted by finishing time.
Activity-selection Problem

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$f_i$</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Activities sorted by finishing time.
Activity-selection Problem

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>$f_i$</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Activities sorted by finishing time.
Activity-selection Problem

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>$f_i$</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Activities sorted by finishing time.
Optimal sub-structure

- Let $S_{ij} = \text{subset of activities in } S \text{ that start after } a_i \text{ finishes and finish before } a_j \text{ starts.}$

  \[ S_{ij} = \left\{ a_k \in S : \forall i, j \quad f_i \leq s_k < f_k \leq s_j \right\} \]

- $A_{ij} = \text{optimal solution to } S_{ij}$

- $A_{ij} = A_{ik} \cup \{ a_k \} \cup A_{kj}$
Greedy choice

Before theorem

<table>
<thead>
<tr>
<th># subproblems in optimal solution</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td># choices to consider</td>
<td>$j-i-1$</td>
</tr>
</tbody>
</table>

$A_{ij} = A_{ik} \cup \{ a_k \} \cup A_{kj}$

We can solve the problem $S_{ij}$ top-down:

- Consider all $a_k \in S_{ij}$
- Solve $S_{ik}$ and $S_{kj}$
- Pick the best $m$ such that $A_{ij} = A_{im} \cup \{ a_m \} \cup A_{im}$
Greedy choice

**Theorem:**
Let $S_{ij} \neq \emptyset$, and let $a_m$ be the activity in $S_{ij}$ with the earliest finish time: $f_m = \min\{f_k : a_k \in S_{ij}\}$. Then:

1. $a_m$ is used in some maximum-size subset of mutually compatible activities of $S_{ij}$.
2. $S_{im} = \emptyset$, so that choosing $a_m$ leaves $S_{mj}$ as the only nonempty subproblem.
Greedy choice

<table>
<thead>
<tr>
<th># subproblems in optimal solution</th>
<th>Before theorem</th>
<th>After theorem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

| # choices to consider             | j-i-1          | 1            |

\[
A_{ij} = A_{ik} \cup \{ a_k \} \cup A_{kj}
\]

\[
A_{ij} = \{ a_m \} \cup A_{mj}
\]

We can now solve the problem \( S_{ij} \) top-down:

- Choose \( a_m \in S_{ij} \) with the earliest finish time (greedy choice).
- Solve \( S_{mj} \).
Challenges

• Greedy choice is not always available.
• How to solve *efficiently* problems that exhibit an optimal substructures property?
WEIGHTED INTERVAL SCHEDULING
Weighted interval scheduling

- **Input**: Set $S$ of $n$ activities, $a_1, a_2, ..., a_n$.
  - $s_i =$ start time of activity $i$.
  - $f_i =$ finish time of activity $i$.
  - $w_i =$ weight of activity $i$

- **Output**: find maximum weight subset of mutually compatible activities.
  - 2 activities are compatible, if their intervals do not overlap.

Example:
Application of the greedy algorithm

W=9
✓

W=3
✗
Discussion

- **Optimal substructure:** ✓
  - $A_{ij} = \text{optimal solution to } S_{ij}$
  - $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$

- **Greedy Choice:** ❌
  - Select the activity with earliest finish time.
**Data structure**

**Notation:** All activities are sorted by finishing time $f_1 \leq f_2 \leq \ldots \leq f_n$

**Definition:** $p(j)$ = largest index $i < j$ such that activity/job $i$ is compatible with activity/job $j$.

**Examples:** $p(6)=4$, $p(5)=2$, $p(4)=2$, $p(2)=0$. 
Binary Choice

Notation: \( OPT(j) = \) value of the optimal solution to the problem including activities 1 to j
\[ = \max \text{ total weight of compatible activities 1 to j} \]

Case 1: OPT selects activity j
- Add weight \( w_j \)
- Cannot use incompatible activities
- Must include optimal solution on remaining compatible activities \( \{1, 2, \ldots, p(j)\} \).

Case 2: OPT does not select activity j
Must include optimal solution on other activities \( \{1, 2, \ldots, j-1\} \).

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max\{w_j + OPT(p(j)), OPT(j - 1)\} & \text{Otherwise}
\end{cases}
\]
Recursive call

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

Case 1

\[ w_6 + \text{OPT}(4) \]

\[ w_6 + w_4 + \text{OPT}(2) \]
\[ w_6 + w_4 + w_2 \]

\[ w_6 + w_3 + \text{OPT}(1) \]

\[ w_6 + \text{OPT}(3) \]

\[ w_6 + \text{OPT}(2) \]

Case 2

\[ \text{OPT}(5) \]

\[ w_5 + \text{OPT}(2) \]

\[ w_5 + w_2 \]

\[ w_5 + \text{OPT}(1) \]

\[ \text{OPT}(4) \]

\[ \ldots \]

Observation: \text{OPT}(j) is calculated multiple times...
Memoization

Memoization: Cache results of each subproblem; lookup as needed.

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.

M-Compute-Opt(j)
if M[j] is empty
    M[j] ← max(v[j]+M-Compute-Opt(p[j]),
                M-Compute-Opt(j-1))
return M[j].
Claim. Memoized version of algorithm takes $O(n \log n)$ time.

- Sort by finish time: $O(n \log n)$.
- Computing $p(\cdot)$: $O(n \log n)$ via sorting by start time.

- $\text{M-COMPUTE-OPT}(j)$: each invocation takes $O(1)$ time and either
  - (i) returns an existing value $M[j]$
  - (ii) fills in one new entry $M[j]$ and makes two recursive calls

- Progress measure $\Phi = \#$ nonempty entries of $M[\cdot]$.
  - initially $\Phi = 0$, throughout $\Phi \leq n$.
  - (ii) increases $\Phi$ by 1 $\Rightarrow$ at most $2n$ recursive calls.

- Overall running time of $\text{M-COMPUTE-OPT}(n)$ is $O(n)$.  

Remark. $O(n)$ if jobs are presorted by start and finish times.
DYNAMIC PROGRAMMING
Bottom-up

Observation: When we compute $M[j]$, we only need values $M[k]$ for $k<j$.

**BOTTOM-UP** ($n; s_1, \ldots, s_n; f_1, \ldots, f_n; v_1, \ldots, v_n$)

Sort jobs by finish time so that $f_1 \leq f_2 \leq \ldots \leq f_n$.
Compute $p(1), p(2), \ldots, p(n)$.

$M[0] \leftarrow 0$

for $j = 1$ TO $n$

\[ M[j] \leftarrow \max \{ v_j + M[p(j)], M[j-1] \} \]

**Main Idea of Dynamic Programming:** Solve the sub-problems in an order that makes sure when you need an answer, it's already been computed.
Finding a solution

Dyn. Prog. algorithm computes optimal value.

Q: How to find solution itself?
A: Bactrack!

Find-Solution(j)
if j = 0
    return ∅.
else if (v[j] + M[p[j]] > M[j–1])
    return { j } U Find-Solution(p[j])
else
    return Find-Solution(j–1).

Analysis. # of recursive calls ≤ n ⇒ O(n).
Example: Computing solution

<table>
<thead>
<tr>
<th>activity</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>predecessor</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Best weight M</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$V_j + M[p(j)]$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$M[j - 1]$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.
Example: Computing solution

<table>
<thead>
<tr>
<th>activity</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>predecessor</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Best weight $M$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_j+M[p(j)]$</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$M[j-1]$</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

M[0]=0

(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.

Diagram:
- $a_1$ finishes at time 2
- $a_2$ finishes at time 3
- $a_3$ starts at time 3 and finishes at time 5
- $a_4$ starts at time 5 and finishes at time 9
- $a_5$ starts at time 9 and finishes at time 10
Example: Computing solution

<table>
<thead>
<tr>
<th>activity</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>predecessor</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Best weight M</td>
<td>2</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$V_j + M[p(j)]$</td>
<td>2</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$M[j-1]$</td>
<td>0</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.
Example: Computing solution

<table>
<thead>
<tr>
<th>activity</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>predecessor</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Best weight M</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$V_j + M[p(j)]$</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$M[j-1]$</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.
Example: Computing solution

<table>
<thead>
<tr>
<th>activity</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>predecessor</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Best weight M</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>-</td>
</tr>
<tr>
<td>$V_j + M[p(j)]$</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>-</td>
</tr>
<tr>
<td>$M[j-1]$</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>-</td>
</tr>
</tbody>
</table>

(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.
Example: Computing solution

<table>
<thead>
<tr>
<th>activity</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>predecessor</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Best weight M</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>$V_j + M[p(j)]$</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>$M[j-1]$</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>

(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.
Example: Reconstruction

<table>
<thead>
<tr>
<th>activity</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>predecessor</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Best weight M</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>$V_j + M[p(j)]$</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>$M[j-1]$</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>

(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.
Example: Reconstruction

<table>
<thead>
<tr>
<th>activity</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>predecessor</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Best weight $M$</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>$V_j + M[p(j)]$</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>$M[j-1]$</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>

(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.
Example: Reconstruction

<table>
<thead>
<tr>
<th>activity</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>predecessor</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Best weight M</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>$V_j + M[p(j)]$</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>$M[j-1]$</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>

(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.