Dynamic Programming

T. M. Murali

February 28, March 5, 17, 19, 21, 2013
Algorithm Design Techniques

1. Goal: design efficient (polynomial-time) algorithms.
Algorithm Design Techniques

1. Goal: design efficient (polynomial-time) algorithms.

2. Greedy
   - Pro: natural approach to algorithm design.
   - Con: many greedy approaches to a problem. Only some may work.
   - Con: many problems for which no greedy approach is known.
Algorithm Design Techniques

1. Goal: design efficient (polynomial-time) algorithms.

2. Greedy
   ▶ Pro: natural approach to algorithm design.
   ▶ Con: many greedy approaches to a problem. Only some may work.
   ▶ Con: many problems for which no greedy approach is known.

3. Divide and conquer
   ▶ Pro: simple to develop algorithm skeleton.
   ▶ Con: conquer step can be very hard to implement efficiently.
   ▶ Con: usually reduces time for a problem known to be solvable in polynomial time.
**Algorithm Design Techniques**

1. **Goal:** design efficient (polynomial-time) algorithms.

2. **Greedy**
   - **Pro:** natural approach to algorithm design.
   - **Con:** many greedy approaches to a problem. Only some may work.
   - **Con:** many problems for which no greedy approach is known.

3. **Divide and conquer**
   - **Pro:** simple to develop algorithm skeleton.
   - **Con:** conquer step can be very hard to implement efficiently.
   - **Con:** usually reduces time for a problem known to be solvable in polynomial time.

4. **Dynamic programming**
   - More powerful than greedy and divide-and-conquer strategies.
   - *Implicitly* explore space of all possible solutions.
   - Solve multiple sub-problems and build up correct solutions to larger and larger sub-problems.
   - Careful analysis needed to ensure number of sub-problems solved is polynomial in the size of the input.
History of Dynamic Programming

- Bellman pioneered the systematic study of dynamic programming in the 1950s.
History of Dynamic Programming

- Bellman pioneered the systematic study of dynamic programming in the 1950s.
- The Secretary of Defense at that time was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.
  - “it’s impossible to use dynamic in a pejorative sense”
  - “something not even a Congressman could object to” (Bellman, R. E., Eye of the Hurricane, An Autobiography).
Applications of Dynamic Programming

- Computational biology: Smith-Waterman algorithm for sequence alignment.
- Operations research: Bellman-Ford algorithm for shortest path routing in networks.
- Control theory: Viterbi algorithm for hidden Markov models.
- Computer science (theory, graphics, AI, ...): Unix `diff` command for comparing two files.
Review: Interval Scheduling

Interval Scheduling

**INSTANCE:** Nonempty set \( \{(s_i, f_i), 1 \leq i \leq n\} \) of start and finish times of \( n \) jobs.

**SOLUTION:** The largest subset of mutually compatible jobs.

- Two jobs are *compatible* if they do not overlap.
Review: Interval Scheduling

Interval Scheduling

INSTANCE: Nonempty set \( \{(s_i, f_i), 1 \leq i \leq n\} \) of start and finish times of \( n \) jobs.

SOLUTION: The largest subset of mutually compatible jobs.

- Two jobs are \textit{compatible} if they do not overlap.
- Greedy algorithm: sort jobs in increasing order of finish times. Add next job to current subset only if it is compatible with previously-selected jobs.
**Weighted Interval Scheduling**

**INSTANCE:** Nonempty set \( \{(s_i, f_i), 1 \leq i \leq n\} \) of start and finish times of \( n \) jobs and a weight \( v_i \geq 0 \) associated with each job.

**SOLUTION:** A set \( S \) of mutually compatible jobs such that \( \sum_{i \in S} v_i \) is maximised.

![Diagram](image)

*Figure 6.1 A simple instance of weighted interval scheduling.*
Weighted Interval Scheduling

**INSTANCE:** Nonempty set \( \{(s_i, f_i), 1 \leq i \leq n\} \) of start and finish times of \( n \) jobs and a weight \( v_i \geq 0 \) associated with each job.

**SOLUTION:** A set \( S \) of mutually compatible jobs such that \( \sum_{i \in S} v_i \) is maximised.

![Diagram of job scheduling](image)

**Figure 6.1** A simple instance of weighted interval scheduling.

- Greedy algorithm can produce arbitrarily bad results for this problem.
Approach

- Sort jobs in increasing order of finish time and relabel: $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Job $i$ comes before job $j$ if $i < j$.
- $p(j)$ is the largest index $i < j$ such that job $i$ is compatible with job $j$. $p(j) = 0$ if there is no such job $i$.
- All jobs that come before job $p(j)$ are also compatible with job $j$.

![Diagram](image)

**Figure 6.2** An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval $j$.

- We will develop optimal algorithm from obvious statements about the problem.
Detour: a Binomial Identity

Pascal's triangle:
- Each element is a binomial coefficient.
- Each element is the sum of the two elements above it.

$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$

Proof: either we select the $n$th element or not...
Detour: a Binomial Identity

- Pascal’s triangle:
  - Each element is a binomial co-efficient.
  - Each element is the sum of the two elements above it.
Detour: a Binomial Identity

Pascal’s triangle:
- Each element is a binomial co-efficient.
- Each element is the sum of the two elements above it.

\[
\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}
\]
Detour: a Binomial Identity

- Pascal’s triangle:
  - Each element is a binomial co-efficient.
  - Each element is the sum of the two elements above it.

\[
\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}
\]

- Proof: either we select the \( n \)th element or not . . .
Sub-problems

Let $O$ be the optimal solution. Two cases to consider.

Case 1 job $n$ is not in $O$.

Case 2 job $n$ is in $O$. 
Sub-problems

Let $O$ be the optimal solution. Two cases to consider.

**Case 1** job $n$ is not in $O$. $O$ must be the optimal solution for jobs 
$\{1, 2, \ldots, n-1\}$.

**Case 2** job $n$ is in $O$. 

Suggests finding optimal solution for sub-problems consisting of jobs 
$\{1, 2, \ldots, j-1, j\}$, for all values of $j$. 

Sub-problems

Let $O$ be the optimal solution. Two cases to consider.

**Case 1** job $n$ is not in $O$. $O$ must be the optimal solution for jobs 
$\{1, 2, \ldots, n-1\}$.

**Case 2** job $n$ is in $O$.

- $O$ cannot use incompatible jobs 
  $\{p(n) + 1, p(n) + 2, \ldots, n-1\}$.
- Remaining jobs in $O$ must be the optimal solution for jobs 
  $\{1, 2, \ldots, p(n)\}$.
Sub-problems

Let \( O \) be the optimal solution. Two cases to consider.

Case 1: job \( n \) is not in \( O \). \( O \) must be the optimal solution for jobs \( \{1, 2, \ldots, n-1\} \).

Case 2: job \( n \) is in \( O \).

- \( O \) cannot use incompatible jobs \( \{p(n) + 1, p(n) + 2, \ldots, n-1\} \).
- Remaining jobs in \( O \) must be the optimal solution for jobs \( \{1, 2, \ldots, p(n)\} \).

\( O \) must be the best of these two choices!
Sub-problems

Let \( O \) be the optimal solution. Two cases to consider.

Case 1  job \( n \) is not in \( O \). \( O \) must be the optimal solution for jobs \( \{1, 2, \ldots, n - 1\} \).

Case 2  job \( n \) is in \( O \).

- \( O \) cannot use incompatible jobs \( \{p(n) + 1, p(n) + 2, \ldots, n - 1\} \).
- Remaining jobs in \( O \) must be the optimal solution for jobs \( \{1, 2, \ldots, p(n)\} \).

- \( O \) must be the best of these two choices!

- Suggests finding optimal solution for sub-problems consisting of jobs \( \{1, 2, \ldots, j - 1, j\} \), for all values of \( j \).
Recursion

Let $O_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($OPT(0) = 0$).
Recursion

- Let $O_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($OPT(0) = 0$).
- We are seeking $O_n$ with a value of $OPT(n)$.
Recursion

- Let $O_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($OPT(0) = 0$).
- We are seeking $O_n$ with a value of $OPT(n)$.
- To compute $OPT(j)$:
  
  Case 1 $j \notin O_j$:
Recursion

- Let $O_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($OPT(0) = 0$).
- We are seeking $O_n$ with a value of $OPT(n)$.
- To compute $OPT(j)$:
  
  **Case 1** $j \notin O_j$: $OPT(j) = OPT(j - 1)$.
Recursion

- Let $O_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($OPT(0) = 0$).
- We are seeking $O_n$ with a value of $OPT(n)$.
- To compute $OPT(j)$:
  - Case 1 $j \not\in O_j$: $OPT(j) = OPT(j - 1)$.
  - Case 2 $j \in O_j$: $v_j + OPT(p(j))$.
Recursion

- Let $O_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($OPT(0) = 0$).
- We are seeking $O_n$ with a value of $OPT(n)$.
- To compute $OPT(j)$:
  
  Case 1: $j \not\in O_j$: $OPT(j) = OPT(j - 1)$.
  
  Case 2: $j \in O_j$: $OPT(j) = v_j + OPT(p(j))$
Recursion

Let $O_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($OPT(0) = 0$).

We are seeking $O_n$ with a value of $OPT(n)$.

To compute $OPT(j)$:

Case 1 \( j \notin O_j \): \( OPT(j) = OPT(j - 1) \).

Case 2 \( j \in O_j \): \( OPT(j) = v_j + OPT(p(j)) \)

\[ OPT(j) = \max(v_j + OPT(p(j)), OPT(j - 1)) \]
Recursion

Let $O_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($OPT(0) = 0$).

We are seeking $O_n$ with a value of $OPT(n)$.

To compute $OPT(j)$:

Case 1 $j \notin O_j$: $OPT(j) = OPT(j - 1)$.

Case 2 $j \in O_j$: $OPT(j) = v_j + OPT(p(j))$

$OPT(j) = \max(v_j + OPT(p(j)), OPT(j - 1))$

When does request $j$ belong to $O_j$?
Recursion

Let \( O_j \) be the optimal solution for jobs \{1, 2, \ldots, j\} and \( OPT(j) \) be the value of this solution (\( OPT(0) = 0 \)).

We are seeking \( O_n \) with a value of \( OPT(n) \).

To compute \( OPT(j) \):

\begin{align*}
\text{Case 1 } j \notin O_j & : \quad OPT(j) = OPT(j - 1) \\
\text{Case 2 } j \in O_j & : \quad OPT(j) = v_j + OPT(p(j))
\end{align*}

\[ OPT(j) = \max(v_j + OPT(p(j)), OPT(j - 1)) \]

When does request \( j \) belong to \( O_j \)? If and only if \( v_j + OPT(p(j)) \geq OPT(j - 1) \).
Recursive Algorithm

\textbf{Compute-Opt}(j) \\
\text{If } j = 0 \text{ then} \\
\hspace{1em} \text{Return } 0 \\
\text{Else} \\
\hspace{1em} \text{Return } \max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j - 1)) \\
\text{Endif}
Recursive Algorithm

Compute-Opt(j)

If $j = 0$ then
    Return 0
Else
    Return $\max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j - 1))$
Endif

▶ Correctness of algorithm follows by induction.

Figure 6.4 An instance of weighted interval scheduling on which the simple Compute-Opt recursion will take exponential time. The values of all intervals in this instance are 1.
**Example of Recursive Algorithm**

**Figure 6.2** An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval $j$.

\[
\begin{align*}
\text{Index} & \quad v_1 = 2 & p(1) = 0 \\
1 & \quad v_2 = 4 & p(2) = 0 \\
2 & \quad v_3 = 4 & p(3) = 1 \\
3 & \quad v_4 = 7 & p(4) = 0 \\
4 & \quad v_5 = 2 & p(5) = 3 \\
5 & \quad v_6 = 1 & p(6) = 3 \\
6 & & \\
\end{align*}
\]

OPT(6) = 
OPT(5) = 
OPT(4) = 
OPT(3) = 
OPT(2) = 
OPT(1) = 
OPT(0) = 0
Example of Recursive Algorithm

**Figure 6.2** An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval $j$.

$$\text{OPT}(6) = \max(v_6 + \text{OPT}(p(6)), \text{OPT}(5)) = \max(1 + \text{OPT}(3), \text{OPT}(5))$$

$$\text{OPT}(5) =$$

$$\text{OPT}(4) =$$

$$\text{OPT}(3) =$$

$$\text{OPT}(2) =$$

$$\text{OPT}(1) =$$

$$\text{OPT}(0) = 0$$
Example of Recursive Algorithm

\[ \text{OPT}(6) = \max(v_6 + \text{OPT}(p(6)), \text{OPT}(5)) = \max(1 + \text{OPT}(3), \text{OPT}(5)) \]
\[ \text{OPT}(5) = \max(v_5 + \text{OPT}(p(j)), \text{OPT}(4)) = \max(2 + \text{OPT}(3), \text{OPT}(4)) \]
\[ \text{OPT}(4) = \]
\[ \text{OPT}(3) = \]
\[ \text{OPT}(2) = \]
\[ \text{OPT}(1) = \]
\[ \text{OPT}(0) = 0 \]

Figure 6.2: An instance of weighted interval scheduling with the functions \( p(j) \) defined for each interval \( j \).
Example of Recursive Algorithm

\[
\begin{align*}
\text{OPT}(6) &= \max(v_6 + \text{OPT}(p(6)), \text{OPT}(5)) = \max(1 + \text{OPT}(3), \text{OPT}(5)) \\
\text{OPT}(5) &= \max(v_5 + \text{OPT}(p(j)), \text{OPT}(4)) = \max(2 + \text{OPT}(3), \text{OPT}(4)) \\
\text{OPT}(4) &= \max(v_4 + \text{OPT}(p(4)), \text{OPT}(3)) = \max(7 + \text{OPT}(0), \text{OPT}(3)) \\
\text{OPT}(3) &= \\
\text{OPT}(2) &= \\
\text{OPT}(1) &= \\
\text{OPT}(0) &= 0
\end{align*}
\]

\textbf{Figure 6.2} An instance of weighted interval scheduling with the functions } p(j) \text{ defined for each interval } j.
Example of Recursive Algorithm

\[ \text{OPT}(6) = \max(v_6 + \text{OPT}(p(6)), \text{OPT}(5)) = \max(1 + \text{OPT}(3), \text{OPT}(5)) \]
\[ \text{OPT}(5) = \max(v_5 + \text{OPT}(p(j)), \text{OPT}(4)) = \max(2 + \text{OPT}(3), \text{OPT}(4)) \]
\[ \text{OPT}(4) = \max(v_4 + \text{OPT}(p(4)), \text{OPT}(3)) = \max(7 + \text{OPT}(0), \text{OPT}(3)) \]
\[ \text{OPT}(3) = \max(v_3 + \text{OPT}(p(3)), \text{OPT}(2)) = \max(4 + \text{OPT}(1), \text{OPT}(2)) \]
\[ \text{OPT}(2) = \]
\[ \text{OPT}(1) = \]
\[ \text{OPT}(0) = 0 \]

Figure 6.2  An instance of weighted interval scheduling with the functions \( p(j) \) defined for each interval \( j \).
Example of Recursive Algorithm

\[
\text{OPT}(6) = \max(v_6 + \text{OPT}(p(6)), \text{OPT}(5)) = \max(1 + \text{OPT}(3), \text{OPT}(5))
\]
\[
\text{OPT}(5) = \max(v_5 + \text{OPT}(p(5)), \text{OPT}(4)) = \max(2 + \text{OPT}(3), \text{OPT}(4))
\]
\[
\text{OPT}(4) = \max(v_4 + \text{OPT}(p(4)), \text{OPT}(3)) = \max(7 + \text{OPT}(0), \text{OPT}(3))
\]
\[
\text{OPT}(3) = \max(v_3 + \text{OPT}(p(3)), \text{OPT}(2)) = \max(4 + \text{OPT}(1), \text{OPT}(2))
\]
\[
\text{OPT}(2) = \max(v_2 + \text{OPT}(p(2)), \text{OPT}(1)) = \max(4 + \text{OPT}(0), \text{OPT}(1))
\]
\[
\text{OPT}(1) = v_1 = 2
\]
\[
\text{OPT}(0) = 0
\]

*Figure 6.2* An instance of weighted interval scheduling with the functions \(p(j)\) defined for each interval \(j\).
Example of Recursive Algorithm

\[
\begin{align*}
\text{OPT}(6) &= \max(v_6 + \text{OPT}(p(6)), \text{OPT}(5)) = \max(1 + \text{OPT}(3), \text{OPT}(5)) \\
\text{OPT}(5) &= \max(v_5 + \text{OPT}(p(5)), \text{OPT}(4)) = \max(2 + \text{OPT}(3), \text{OPT}(4)) \\
\text{OPT}(4) &= \max(v_4 + \text{OPT}(p(4)), \text{OPT}(3)) = \max(7 + \text{OPT}(0), \text{OPT}(3)) \\
\text{OPT}(3) &= \max(v_3 + \text{OPT}(p(3)), \text{OPT}(2)) = \max(4 + \text{OPT}(1), \text{OPT}(2)) \\
\text{OPT}(2) &= \max(v_2 + \text{OPT}(p(2)), \text{OPT}(1)) = \max(4 + \text{OPT}(0), \text{OPT}(1)) \\
\text{OPT}(1) &= v_1 = 2 \\
\text{OPT}(0) &= 0
\end{align*}
\]

*Figure 6.2* An instance of weighted interval scheduling with the functions \(p(j)\) defined for each interval \(j\).
Example of Recursive Algorithm

\[
\text{OPT}(6) = \max(v_6 + \text{OPT}(p(6)), \text{OPT}(5)) = \max(1 + \text{OPT}(3), \text{OPT}(5)) = 8
\]
\[
\text{OPT}(5) = \max(v_5 + \text{OPT}(p(5)), \text{OPT}(4)) = \max(2 + \text{OPT}(3), \text{OPT}(4)) = 8
\]
\[
\text{OPT}(4) = \max(v_4 + \text{OPT}(p(4)), \text{OPT}(3)) = \max(7 + \text{OPT}(0), \text{OPT}(3)) = 7
\]
\[
\text{OPT}(3) = \max(v_3 + \text{OPT}(p(3)), \text{OPT}(2)) = \max(4 + \text{OPT}(1), \text{OPT}(2)) = 6
\]
\[
\text{OPT}(2) = \max(v_2 + \text{OPT}(p(2)), \text{OPT}(1)) = \max(4 + \text{OPT}(0), \text{OPT}(1)) = 4
\]
\[
\text{OPT}(1) = v_1 = 2
\]
\[
\text{OPT}(0) = 0
\]

Figure 6.2: An instance of weighted interval scheduling with the functions \(p(j)\) defined for each interval \(j\).
Example of Recursive Algorithm

\[
\begin{align*}
\text{OPT}(6) &= \max(v_6 + \text{OPT}(p(6)), \text{OPT}(5)) = \max(1 + \text{OPT}(3), \text{OPT}(5)) \\
\text{OPT}(5) &= \max(v_5 + \text{OPT}(p(j)), \text{OPT}(4)) = \max(2 + \text{OPT}(3), \text{OPT}(4)) \\
\text{OPT}(4) &= \max(v_4 + \text{OPT}(p(4)), \text{OPT}(3)) = \max(7 + \text{OPT}(0), \text{OPT}(3)) \\
\text{OPT}(3) &= \max(v_3 + \text{OPT}(p(3)), \text{OPT}(2)) = \max(4 + \text{OPT}(1), \text{OPT}(2)) = 6 \\
\text{OPT}(2) &= \max(v_2 + \text{OPT}(p(2)), \text{OPT}(1)) = \max(4 + \text{OPT}(0), \text{OPT}(1)) = 4 \\
\text{OPT}(1) &= v_1 = 2 \\
\text{OPT}(0) &= 0
\end{align*}
\]

Figure 6.2 An instance of weighted interval scheduling with the functions \( p(j) \) defined for each interval \( j \).
Example of Recursive Algorithm

\[ \text{OPT}(6) = \max(v_6 + \text{OPT}(p(6)), \text{OPT}(5)) = \max(1 + \text{OPT}(3), \text{OPT}(5)) = 8 \]
\[ \text{OPT}(5) = \max(v_5 + \text{OPT}(p(j)), \text{OPT}(4)) = \max(2 + \text{OPT}(3), \text{OPT}(4)) = 8 \]
\[ \text{OPT}(4) = \max(v_4 + \text{OPT}(p(4)), \text{OPT}(3)) = \max(7 + \text{OPT}(0), \text{OPT}(3)) = 7 \]
\[ \text{OPT}(3) = \max(v_3 + \text{OPT}(p(3)), \text{OPT}(2)) = \max(4 + \text{OPT}(1), \text{OPT}(2)) = 6 \]
\[ \text{OPT}(2) = \max(v_2 + \text{OPT}(p(2)), \text{OPT}(1)) = \max(4 + \text{OPT}(0), \text{OPT}(1)) = 4 \]
\[ \text{OPT}(1) = v_1 = 2 \]
\[ \text{OPT}(0) = 0 \]

**Figure 6.2** An instance of weighted interval scheduling with the functions \( p(j) \) defined for each interval \( j \).
Example of Recursive Algorithm

OPT(6) = \max(v_6 + OPT(p(6)), OPT(5)) = \max(1 + OPT(3), OPT(5))
OPT(5) = \max(v_5 + OPT(p(j)), OPT(4)) = \max(2 + OPT(3), OPT(4)) = 8
OPT(4) = \max(v_4 + OPT(p(4)), OPT(3)) = \max(7 + OPT(0), OPT(3)) = 7
OPT(3) = \max(v_3 + OPT(p(3)), OPT(2)) = \max(4 + OPT(1), OPT(2)) = 6
OPT(2) = \max(v_2 + OPT(p(2)), OPT(1)) = \max(4 + OPT(0), OPT(1)) = 4
OPT(1) = v_1 = 2
OPT(0) = 0

Figure 6.2 An instance of weighted interval scheduling with the functions p(j) defined for each interval j.
Example of Recursive Algorithm

\[
\begin{align*}
\text{OPT}(6) &= \max(v_6 + \text{OPT}(p(6)), \text{OPT}(5)) = \max(1 + \text{OPT}(3), \text{OPT}(5)) = 8 \\
\text{OPT}(5) &= \max(v_5 + \text{OPT}(p(j)), \text{OPT}(4)) = \max(2 + \text{OPT}(3), \text{OPT}(4)) = 8 \\
\text{OPT}(4) &= \max(v_4 + \text{OPT}(p(4)), \text{OPT}(3)) = \max(7 + \text{OPT}(0), \text{OPT}(3)) = 7 \\
\text{OPT}(3) &= \max(v_3 + \text{OPT}(p(3)), \text{OPT}(2)) = \max(4 + \text{OPT}(1), \text{OPT}(2)) = 6 \\
\text{OPT}(2) &= \max(v_2 + \text{OPT}(p(2)), \text{OPT}(1)) = \max(4 + \text{OPT}(0), \text{OPT}(1)) = 4 \\
\text{OPT}(1) &= v_1 = 2 \\
\text{OPT}(0) &= 0
\end{align*}
\]

Figure 6.2  An instance of weighted interval scheduling with the functions \(p(j)\) defined for each interval \(j\).
Example of Recursive Algorithm

\[ OPT(6) = \max(v_6 + OPT(p(6)), OPT(5)) = \max(1 + OPT(3), OPT(5)) = 8 \]
\[ OPT(5) = \max(v_5 + OPT(p(j)), OPT(4)) = \max(2 + OPT(3), OPT(4)) = 8 \]
\[ OPT(4) = \max(v_4 + OPT(p(4)), OPT(3)) = \max(7 + OPT(0), OPT(3)) = 7 \]
\[ OPT(3) = \max(v_3 + OPT(p(3)), OPT(2)) = \max(4 + OPT(1), OPT(2)) = 6 \]
\[ OPT(2) = \max(v_2 + OPT(p(2)), OPT(1)) = \max(4 + OPT(0), OPT(1)) = 4 \]
\[ OPT(1) = v_1 = 2 \]
\[ OPT(0) = 0 \]

Optimal solution is job 5, job 3, and job 1.

\[ OPT(6) = \max(v_6 + OPT(p(6)), OPT(5)) = \max(1 + OPT(3), OPT(5)) = 8 \]
\[ OPT(5) = \max(v_5 + OPT(p(j)), OPT(4)) = \max(2 + OPT(3), OPT(4)) = 8 \]
\[ OPT(4) = \max(v_4 + OPT(p(4)), OPT(3)) = \max(7 + OPT(0), OPT(3)) = 7 \]
\[ OPT(3) = \max(v_3 + OPT(p(3)), OPT(2)) = \max(4 + OPT(1), OPT(2)) = 6 \]
\[ OPT(2) = \max(v_2 + OPT(p(2)), OPT(1)) = \max(4 + OPT(0), OPT(1)) = 4 \]
\[ OPT(1) = v_1 = 2 \]
\[ OPT(0) = 0 \]

Optimal solution is job 5, job 3, and job 1.

\[ OPT(6) = \max(v_6 + OPT(p(6)), OPT(5)) = \max(1 + OPT(3), OPT(5)) = 8 \]
\[ OPT(5) = \max(v_5 + OPT(p(j)), OPT(4)) = \max(2 + OPT(3), OPT(4)) = 8 \]
\[ OPT(4) = \max(v_4 + OPT(p(4)), OPT(3)) = \max(7 + OPT(0), OPT(3)) = 7 \]
\[ OPT(3) = \max(v_3 + OPT(p(3)), OPT(2)) = \max(4 + OPT(1), OPT(2)) = 6 \]
\[ OPT(2) = \max(v_2 + OPT(p(2)), OPT(1)) = \max(4 + OPT(0), OPT(1)) = 4 \]
\[ OPT(1) = v_1 = 2 \]
\[ OPT(0) = 0 \]

Optimal solution is job 5, job 3, and job 1.

\[ OPT(6) = \max(v_6 + OPT(p(6)), OPT(5)) = \max(1 + OPT(3), OPT(5)) = 8 \]
\[ OPT(5) = \max(v_5 + OPT(p(j)), OPT(4)) = \max(2 + OPT(3), OPT(4)) = 8 \]
\[ OPT(4) = \max(v_4 + OPT(p(4)), OPT(3)) = \max(7 + OPT(0), OPT(3)) = 7 \]
\[ OPT(3) = \max(v_3 + OPT(p(3)), OPT(2)) = \max(4 + OPT(1), OPT(2)) = 6 \]
\[ OPT(2) = \max(v_2 + OPT(p(2)), OPT(1)) = \max(4 + OPT(0), OPT(1)) = 4 \]
\[ OPT(1) = v_1 = 2 \]
\[ OPT(0) = 0 \]

Optimal solution is job 5, job 3, and job 1.
Example of Recursive Algorithm

\[ \text{OPT}(6) = \max(v_6 + \text{OPT}(p(6)), \text{OPT}(5)) = \max(1 + \text{OPT}(3), \text{OPT}(5)) = 8 \]
\[ \text{OPT}(5) = \max(v_5 + \text{OPT}(p(j)), \text{OPT}(4)) = \max(2 + \text{OPT}(3), \text{OPT}(4)) = 8 \]
\[ \text{OPT}(4) = \max(v_4 + \text{OPT}(p(4)), \text{OPT}(3)) = \max(7 + \text{OPT}(0), \text{OPT}(3)) = 7 \]
\[ \text{OPT}(3) = \max(v_3 + \text{OPT}(p(3)), \text{OPT}(2)) = \max(4 + \text{OPT}(1), \text{OPT}(2)) = 6 \]
\[ \text{OPT}(2) = \max(v_2 + \text{OPT}(p(2)), \text{OPT}(1)) = \max(4 + \text{OPT}(0), \text{OPT}(1)) = 4 \]
\[ \text{OPT}(1) = v_1 = 2 \]
\[ \text{OPT}(0) = 0 \]

Optimal solution is job 5, job 3, and job 1.
Running Time of Recursive Algorithm

Compute-Opt(j)
    If \( j = 0 \) then
        Return 0
    Else
        Return \( \max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j - 1)) \)
    Endif
Running Time of Recursive Algorithm

Compute-Opt(j)

If \( j = 0 \) then
   Return 0
Else
   Return \( \max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1)) \)
Endif

What is the running time of the algorithm?
Running Time of Recursive Algorithm

Compute-Opt\( (j)\)

If \( j = 0 \) then
    Return 0
Else
    Return \( \max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j - 1)) \)
Endif

▷ What is the running time of the algorithm? Can be exponential in \( n \).
Running Time of Recursive Algorithm

Compute-Opt(j)

If \( j = 0 \) then
    Return 0
Else
    Return \( \max (v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j - 1)) \)
Endif

- What is the running time of the algorithm? Can be exponential in \( n \).
- When \( p(j) = j - 2 \), for all \( j \geq 2 \): recursive calls are for \( j - 1 \) and \( j - 2 \).

![Figure 6.4](image-url) An instance of weighted interval scheduling on which the simple Compute-Opt recursion will take exponential time. The values of all intervals in this instance are 1.
Memoisation

- Store $\text{OPT}(j)$ values in a cache and reuse them rather than recompute them.
Memoisation

- Store $\text{OPT}(j)$ values in a cache and reuse them rather than recompute them.

---

\[
\text{M-Compute-Opt}(j) \\
\text{If } j = 0 \text{ then} \\
\quad \text{Return 0} \\
\text{Else if } M[j] \text{ is not empty then} \\
\quad \text{Return } M[j] \\
\text{Else} \\
\quad \text{Define } M[j] = \max(v_j + \text{M-Compute-Opt}(p(j)), \text{M-Compute-Opt}(j - 1)) \\
\quad \text{Return } M[j] \\
\text{Endif}
\]
Running Time of Memoisation

M-Compute-Opt(j)
If $j = 0$ then
    Return 0
Else if $M[j]$ is not empty then
    Return $M[j]$
Else
    Define $M[j] = \max(v_j + M\text{-Compute-Opt}(p(j)), M\text{-Compute-Opt}(j - 1))$
    Return $M[j]$
Endif

▶ Claim: running time of this algorithm is $O(n)$ (after sorting).
Running Time of Memoisation

M-Compute-Opt(j)
   If \( j = 0 \) then
      Return 0
   Else if \( M[j] \) is not empty then
      Return \( M[j] \)
   Else
      Define \( M[j] = \max(v_j + \text{M-Compute-Opt}(p(j)), \text{M-Compute-Opt}(j - 1)) \)
      Return \( M[j] \)
   Endif

Claim: running time of this algorithm is \( O(n) \) (after sorting).

Time spent in a single call to M-Compute-Opt is \( O(1) \) apart from time spent in recursive calls.

Total time spent is the order of the number of recursive calls to M-Compute-Opt.

How many such recursive calls are there in total?
Running Time of Memoisation

M-Compute-Opt(j)
If \( j = 0 \) then
  Return 0
Else if \( M[j] \) is not empty then
  Return \( M[j] \)
Else
  Define \( M[j] = \max(v_j + M-Compute-Opt(p(j)), M-Compute-Opt(j-1)) \)
  Return \( M[j] \)
Endif

- Claim: running time of this algorithm is \( O(n) \) (after sorting).
- Time spent in a single call to \( M-Compute-Opt \) is \( O(1) \) apart from time spent in recursive calls.
- Total time spent is the order of the number of recursive calls to \( M-Compute-Opt \).
- How many such recursive calls are there in total?
- Use number of filled entries in \( M \) as a measure of progress.
- Each time \( M-Compute-Opt \) issues two recursive calls, it fills in a new entry in \( M \).
- Therefore, total number of recursive calls is \( O(n) \).
Computing $O$ in Addition to $\text{OPT}(n)$
Computing $O$ in Addition to $\text{OPT}(n)$

- Explicitly store $O_j$ in addition to $\text{OPT}(j)$. 

- Recall: request $j$ belong to $O_j$ if and only if $v_j + \text{OPT}(p(j)) \geq \text{OPT}(j-1)$.

- Can recover $O_j$ from values of the optimal solutions in $O(j)$ time.
Computing $O$ in Addition to $\text{OPT}(n)$

- Explicitly store $O_j$ in addition to $\text{OPT}(j)$. Running time becomes $O(n^2)$.
Computing $O$ in Addition to $\text{OPT}(n)$

- Explicitly store $O_j$ in addition to $\text{OPT}(j)$. Running time becomes $O(n^2)$.
- Recall: request $j$ belong to $O_j$ if and only if $v_j + \text{OPT}(p(j)) \geq \text{OPT}(j - 1)$.
- Can recover $O_j$ from values of the optimal solutions in $O(j)$ time.
Computing $\mathcal{O}$ in Addition to $\text{OPT}(n)$

- Explicitly store $\mathcal{O}_j$ in addition to OPT($j$). Running time becomes $O(n^2)$.
- Recall: request $j$ belong to $\mathcal{O}_j$ if and only if $v_j + \text{OPT}(p(j)) \geq \text{OPT}(j - 1)$.
- Can recover $\mathcal{O}_j$ from values of the optimal solutions in $O(j)$ time.

---

Find-Solution($j$)

If $j = 0$ then

Output nothing

Else

If $v_j + M[p(j)] \geq M[j - 1]$ then

Output $j$ together with the result of Find-Solution($p(j)$)

Else

Output the result of Find-Solution($j - 1$)

Endif

Endif
From Recursion to Iteration

- Unwind the recursion and convert it into iteration.
- Can compute values in $M$ iteratively in $O(n)$ time.
- Find-Solution works as before.

Iterative-Compute-Opt

$M[0] = 0$

For $j = 1, 2, \ldots, n$

$M[j] = \max(v_j + M[p(j)], M[j - 1])$

Endfor
Basic Outline of Dynamic Programming

To solve a problem, we need a collection of sub-problems that satisfy a few properties:

1. There are a polynomial number of sub-problems.
2. The solution to the problem can be computed easily from the solutions to the sub-problems.
3. There is a natural ordering of the sub-problems from “smallest” to “largest”.
4. There is an easy-to-compute recurrence that allows us to compute the solution to a sub-problem from the solutions to some smaller sub-problems.
Basic Outline of Dynamic Programming

To solve a problem, we need a collection of sub-problems that satisfy a few properties:

1. There are a polynomial number of sub-problems.
2. The solution to the problem can be computed easily from the solutions to the sub-problems.
3. There is a natural ordering of the sub-problems from “smallest” to “largest”.
4. There is an easy-to-compute recurrence that allows us to compute the solution to a sub-problem from the solutions to some smaller sub-problems.

Difficulties in designing dynamic programming algorithms:

1. Which sub-problems to define?
2. How can we tie together sub-problems using a recurrence?
3. How do we order the sub-problems (to allow iterative computation of optimal solutions to sub-problems)?
Least Squares Problem

- Given scientific or statistical data plotted on two axes.
- Find the “best” line that “passes” through these points.

**Figure 6.6** A “line of best fit.”
Least Squares Problem

Given scientific or statistical data plotted on two axes.

Find the “best” line that “passes” through these points.

How do we formalise the problem?

Figure 6.6 A “line of best fit.”
Least Squares Problem

▶ Given scientific or statistical data plotted on two axes.
▶ Find the “best” line that “passes” through these points.
▶ How do we formalise the problem?

**LEAST SQUARES**

**INSTANCE:** Set \( P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\} \) of \( n \) points.

**SOLUTION:** Line \( L : y = ax + b \) that minimises

\[
\text{Error}(L, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2.
\]
Least Squares Problem

Figure 6.6 A “line of best fit.”

Given scientific or statistical data plotted on two axes.

Find the “best” line that “passes” through these points.

How do we formalise the problem?

LEAST SQUARES

INSTANCE: Set \( P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\} \) of \( n \) points.

SOLUTION: Line \( L : y = ax + b \) that minimises

\[
\text{Error}(L, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2.
\]

Solution is achieved by

\[
a = \frac{n \sum_{i} x_i y_i - (\sum_{i} x_i) (\sum_{i} y_i)}{n \sum_{i} x_i^2 - (\sum_{i} x_i)^2} \quad \text{and} \quad b = \frac{\sum_{i} y_i - a \sum_{i} x_i}{n}
\]
Segmented Least Squares

Figure 6.7 A set of points that lie approximately on two lines.
Segmented Least Squares

Figure 6.7 A set of points that lie approximately on two lines. Figure 6.8 A set of points that lie approximately on three lines.
Segmented Least Squares

Want to fit multiple lines through $P$.

Each line must fit contiguous set of $x$-coordinates.

Lines must minimise total error.
Segmented Least Squares

**Figure 6.7** A set of points that lie approximately on two lines.

**Figure 6.8** A set of points that lie approximately on three lines.
**Segmented Least Squares**

**INSTANCE:** Set $P = \{p_i = (x_i, y_i), 1 \leq i \leq n\}$ of $n$ points, $x_1 < x_2 < \cdots < x_n$.

**SOLUTION:** A integer $k$, a partition of $P$ into $k$ segments $\{P_1, P_2, \ldots, P_k\}$, $k$ lines $L_j : y = a_j x + b_j, 1 \leq j \leq k$ that minimise

$$\sum_{j=1}^{k} \text{Error}(L_j, P_j)$$

> A subset $P'$ of $P$ is a **segment** if $1 \leq i < j \leq n$ exist such that $P' = \{(x_i, y_i), (x_{i+1}, y_{i+1}), \ldots, (x_{j-1}, y_{j-1}), (x_j, y_j)\}$.

---

*Figure 6.7* A set of points that lie approximately on two lines.  *Figure 6.8* A set of points that lie approximately on three lines.
**Segmented Least Squares**

**INSTANCE:** Set $P = \{p_i = (x_i, y_i), 1 \leq i \leq n\}$ of $n$ points, $x_1 < x_2 < \cdots < x_n$ and a parameter $C > 0$.

**SOLUTION:** A integer $k$, a partition of $P$ into $k$ segments 
$\{P_1, P_2, \ldots, P_k\}$, $k$ lines $L_j : y = a_jx + b_j, 1 \leq j \leq k$ that minimise 
$$\sum_{j=1}^{k} \text{Error}(L_j, P_j) + Ck$$

- A subset $P'$ of $P$ is a *segment* if $1 \leq i < j \leq n$ exist such that $P' = \{(x_i, y_i), (x_{i+1}, y_{i+1}), \ldots, (x_{j-1}, y_{j-1}), (x_j, y_j)\}$.
Example of Segmented Least Squares

Input contains a set of two-dimensional points.
Consider the $x$-coordinates of the points in the input.
Example of Segmented Least Squares

Divide the points into segments; each segment contains consecutive points in the sorted order by $x$-coordinate.
Example of Segmented Least Squares

Fit the best line for each segment.
Example of Segmented Least Squares

Illegal solution: black point is not in any segment.
Example of Segmented Least Squares

Illegal solution: leftmost purple point has $x$-coordinate between last two points in green segment.
Formulating the Recursion I

- Observation: $p_n$ is part of some segment in the optimal solution. This segment starts at some point $p_i$.
- Let $OPT(i)$ be the optimal value for the points $\{p_1, p_2, \ldots, p_i\}$.
- Let $e_{i,j}$ denote the minimum error of any line that fits $\{p_i, p_2, \ldots, p_j\}$.
- We want to compute $OPT(n)$.

![Graphical representation of the recursion](image)

- If the last segment in the optimal partition is $\{p_i, p_{i+1}, \ldots, p_n\}$, then

$$OPT(n) = e_{i,n} + C + OPT(i - 1)$$
Formulating the Recursion II

- Consider the sub-problem on the points \( \{p_1, p_2, \ldots, p_j\} \)
- To obtain \( \text{OPT}(j) \), if the last segment in the optimal partition is \( \{p_i, p_{i+1}, \ldots, p_j\} \), then

\[
\text{OPT}(j) = e_{i,j} + C + \text{OPT}(i - 1)
\]

Since \( i \) can take only \( j \) distinct values,
\[
\text{OPT}(j) = \min_{1 \leq i \leq j} (e_{i,j} + C + \text{OPT}(i - 1))
\]

Segment \( \{p_i, p_{i+1}, \ldots, p_j\} \) is part of the optimal solution for this sub-problem if and only if the minimum value of \( \text{OPT}(j) \) is obtained using index \( i \).
Formulating the Recursion II

- Consider the sub-problem on the points \( \{p_1, p_2, \ldots, p_j\} \)
- To obtain \( \text{OPT}(j) \), if the last segment in the optimal partition is \( \{p_i, p_{i+1}, \ldots, p_j\} \), then
  \[
  \text{OPT}(j) = e_{i,j} + C + \text{OPT}(i - 1)
  \]
- Since \( i \) can take only \( j \) distinct values,
  \[
  \text{OPT}(j) = \min_{1 \leq i \leq j} (e_{i,j} + C + \text{OPT}(i - 1))
  \]
- Segment \( \{p_i, p_{i+1}, \ldots, p_j\} \) is part of the optimal solution for this sub-problem if and only if the minimum value of \( \text{OPT}(j) \) is obtained using index \( i \).
Dynamic Programming Algorithm

\[ \text{OPT}(j) = \min_{1 \leq i \leq j} \left( e_{i,j} + C + \text{OPT}(i - 1) \right) \]

Segmented-Least-Squares(n)

- Array \( M[0...n] \)
- Set \( M[0] = 0 \)
- For all pairs \( i \leq j \)
  - Compute the least squares error \( e_{i,j} \) for the segment \( p_i, \ldots, p_j \)
- Endfor
- For \( j = 1, 2, \ldots, n \)
  - Use the recurrence (6.7) to compute \( M[j] \)
- Endfor
- Return \( M[n] \)
Dynamic Programming Algorithm

$$\text{OPT}(j) = \min_{1 \leq i \leq j} (e_{i,j} + C + \text{OPT}(i - 1))$$

Segmented-Least-Squares(n)

Array $M[0 \ldots n]$
Set $M[0] = 0$
For all pairs $i \leq j$
    Compute the least squares error $e_{i,j}$ for the segment $p_i, \ldots, p_j$
Endfor
For $j = 1, 2, \ldots, n$
    Use the recurrence (6.7) to compute $M[j]$
Endfor
Return $M[n]$

- Running time is $O(n^3)$, can be improved to $O(n^2)$.
- We can find the segments in the optimal solution by backtracking.
RNA Molecules

RNA is a basic biological molecule. It is single stranded.
RNA molecules fold into complex “secondary structures.”
Secondary structure often governs the behaviour of an RNA molecule.
Various rules govern secondary structure formation:
RNA Molecules

- RNA is a basic biological molecule. It is single stranded.
- RNA molecules fold into complex “secondary structures.”
- Secondary structure often governs the behaviour of an RNA molecule.
- Various rules govern secondary structure formation:

1. Pairs of bases match up; each base matches with $\leq 1$ other base.
2. Adenine always matches with Uracil.
3. Cytosine always matches with Guanine.
4. There are no kinks in the folded molecule.
5. Structures are “knot-free”.

Figure 6.13 An RNA secondary structure. Thick lines connect adjacent elements of the sequence; thin lines indicate pairs of elements that are matched.
RNA Molecules

RNA is a basic biological molecule. It is single stranded.

RNA molecules fold into complex “secondary structures.”

Secondary structure often governs the behaviour of an RNA molecule.

Various rules govern secondary structure formation:

1. Pairs of bases match up; each base matches with \( \leq 1 \) other base.
2. Adenine always matches with Uracil.
3. Cytosine always matches with Guanine.
4. There are no kinks in the folded molecule.
5. Structures are “knot-free”.

Problem: given an RNA molecule, predict its secondary structure.
RNA Molecules

RNA is a basic biological molecule. It is single stranded.

RNA molecules fold into complex “secondary structures.”

Secondary structure often governs the behaviour of an RNA molecule.

Various rules govern secondary structure formation:

1. Pairs of bases match up; each base matches with $\leq 1$ other base.
2. Adenine always matches with Uracil.
3. Cytosine always matches with Guanine.
4. There are no kinks in the folded molecule.
5. Structures are “knot-free”.

Problem: given an RNA molecule, predict its secondary structure.

Hypothesis: In the cell, RNA molecules form the secondary structure with the lowest total free energy.
Formulating the Problem

- An **RNA molecule** is a string $B = b_1 b_2 \ldots b_n$; each $b_i \in \{A, C, G, U\}$.
- A **secondary structure on $B$** is a set of pairs $S = \{(i,j)\}$, where $1 \leq i, j \leq n$ and

\begin{itemize}
  \item 1. (No kinks.) If $(i, j) \in S$, then $i < j - 4$.
  \item 2. (Watson-Crick) The elements in each pair in $S$ consist of either \{A, U\} or \{C, G\} (in either order).
  \item 3. $S$ is a matching: no index appears in more than one pair.
  \item 4. (No knots) If $(i, j)$ and $(k, l)$ are two pairs in $S$, then we cannot have $i < k < j < l$.
\end{itemize}

\textbf{Figure 6.14} Two views of an RNA secondary structure. In the second view, (b), the string has been "stretched" lengthwise, and edges connecting matched pairs appear as noncrossing "bubbles" over the string.
Formulating the Problem

▶ An **RNA molecule** is a string $B = b_1 b_2 \ldots b_n$; each $b_i \in \{A, C, G, U\}$.
▶ A **secondary structure on** $B$ is a set of pairs $S = \{(i,j)\}$, where $1 \leq i, j \leq n$ and
  1. *(No kinks.*) If $(i,j) \in S$, then $i < j - 4$.
  2. *(Watson-Crick)* The elements in each pair in $S$ consist of either $\{A, U\}$ or $\{C, G\}$ (in either order).
  3. $S$ is a **matching**: no index appears in more than one pair.
  4. *(No knots)* If $(i,j)$ and $(k,l)$ are two pairs in $S$, then we cannot have $i < k < j < l$.

▶ The **energy** of a secondary structure $\propto$ the number of base pairs in it.
▶ Problem: Compute the largest secondary structure, i.e., with the largest number of base pairs.
Illegal Secondary Structures

A C A U G G C C A U G U

Watson-Crick

A C A U G G C C A U G U

Kink Matching

A C A U G G C C A U G U

Knot
Legal Secondary Structures

A C A U G G C C A U G U
Dynamic Programming Approach

- $\text{OPT}(j)$ is the maximum number of base pairs in a secondary structure for $b_1 b_2 \ldots b_j$. 

Insight: need sub-problems indexed both by start and by end.
Dynamic Programming Approach

- $OPT(j)$ is the maximum number of base pairs in a secondary structure for $b_1 b_2 \ldots b_j$. $OPT(j) = 0$, if $j \leq 5$. 
Dynamic Programming Approach

- \( OPT(j) \) is the maximum number of base pairs in a secondary structure for \( b_1b_2\ldots b_j \). \( OPT(j) = 0 \), if \( j \leq 5 \).
- In the optimal secondary structure on \( b_1b_2\ldots b_j \),
Dynamic Programming Approach

- $OPT(j)$ is the maximum number of base pairs in a secondary structure for $b_1 b_2 \ldots b_j$. $OPT(j) = 0$, if $j \leq 5$.
- In the optimal secondary structure on $b_1 b_2 \ldots b_j$
  1. if $j$ is not a member of any pair, use $OPT(j - 1)$. 

Insight: need sub-problems indexed both by start and by end.
Dynamic Programming Approach

- \( OPT(j) \) is the maximum number of base pairs in a secondary structure for \( b_1 b_2 \ldots b_j \). \( OPT(j) = 0 \), if \( j \leq 5 \).

- In the optimal secondary structure on \( b_1 b_2 \ldots b_j \):
  1. if \( j \) is not a member of any pair, use \( OPT(j - 1) \).
  2. if \( j \) pairs with some \( t < j - 4 \),

\[ \text{Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.} \]
Dynamic Programming Approach

- $OPT(j)$ is the maximum number of base pairs in a secondary structure for $b_1 b_2 \ldots b_j$. $OPT(j) = 0$, if $j \leq 5$.

- In the optimal secondary structure on $b_1 b_2 \ldots b_j$
  1. If $j$ is not a member of any pair, use $OPT(j - 1)$.
  2. If $j$ pairs with some $t < j - 4$, knot condition yields two independent sub-problems!

Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.
Dynamic Programming Approach

- \( OPT(j) \) is the maximum number of base pairs in a secondary structure for \( b_1 b_2 \ldots b_j \). \( OPT(j) = 0 \), if \( j \leq 5 \).

- In the optimal secondary structure on \( b_1 b_2 \ldots b_j \):
  1. if \( j \) is not a member of any pair, use \( OPT(j - 1) \).
  2. if \( j \) pairs with some \( t < j - 4 \), knot condition yields two independent sub-problems! \( OPT(t - 1) \) and ???

Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.
Dynamic Programming Approach

- $OPT(j)$ is the maximum number of base pairs in a secondary structure for $b_1 b_2 \ldots b_j$. $OPT(j) = 0$, if $j \leq 5$.

- In the optimal secondary structure on $b_1 b_2 \ldots b_j$
  1. if $j$ is not a member of any pair, use $OPT(j - 1)$.
  2. if $j$ pairs with some $t < j - 4$, knot condition yields two independent sub-problems! $OPT(t - 1)$ and ???

- Insight: need sub-problems indexed both by start and by end.

![Diagram](image)

Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.
Correct Dynamic Programming Approach

$OPT(i, j)$ is the maximum number of base pairs in a secondary structure for $b_i b_2 \ldots b_j$.

Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.
Correct Dynamic Programming Approach

\[ \text{OPT}(i, j) \text{ is the maximum number of base pairs in a secondary structure for } b_i b_2 \ldots b_j. \text{ OPT}(i, j) = 0, \text{ if } i \geq j - 4. \]
Correct Dynamic Programming Approach

OPT(i, j) is the maximum number of base pairs in a secondary structure for \( b_i b_2 \ldots b_j \). OPT(i, j) = 0, if \( i \geq j - 4 \).

In the optimal secondary structure on \( b_i b_2 \ldots b_j \)

\[
\text{OPT}(i, j) = \max \left( \right)
\]
Correct Dynamic Programming Approach

- \( OPT(i, j) \) is the maximum number of base pairs in a secondary structure for \( b_i b_2 \ldots b_j \). \( OPT(i, j) = 0 \), if \( i \geq j - 4 \).
- In the optimal secondary structure on \( b_i b_2 \ldots b_j \):
  1. if \( j \) is not a member of any pair, compute \( OPT(i, j - 1) \).

\[
OPT(i, j) = \max \left( OPT(i, j - 1), \right.
\]

\[
\left. \max_{t} \left( 1 + OPT(i, t - 1) + OPT(t + 1, j - 1) \right) \right)
\]

In the “inner” maximisation, \( t \) runs over all indices between \( i \) and \( j - 5 \) that are allowed to pair with \( j \).
Correct Dynamic Programming Approach

\[ \text{OPT}(i, j) \] is the maximum number of base pairs in a secondary structure for \( b_i b_2 \ldots b_j \). \( \text{OPT}(i, j) = 0, \) if \( i \geq j - 4 \).

In the optimal secondary structure on \( b_i b_2 \ldots b_j \)

1. if \( j \) is not a member of any pair, compute \( \text{OPT}(i, j - 1) \).
2. if \( j \) pairs with some \( t < j - 4 \), compute \( \text{OPT}(i, t - 1) \) and \( \text{OPT}(t + 1, j - 1) \).

\[
\text{OPT}(i, j) = \max \left( \text{OPT}(i, j - 1), \right.
\]

Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.
Correct Dynamic Programming Approach

- **OPT**(*i*, *j*) is the maximum number of base pairs in a secondary structure for \(b_ib_2\ldots b_j\). \(OPT(i, j) = 0\), if \(i \geq j - 4\).
- In the optimal secondary structure on \(b_ib_2\ldots b_j\)
  1. if \(j\) is not a member of any pair, compute \(OPT(i, j - 1)\).
  2. if \(j\) pairs with some \(t < j - 4\), compute \(OPT(i, t - 1)\) and \(OPT(t + 1, j - 1)\).
- Since \(t\) can range from \(i\) to \(j - 5\),

\[
OPT(i, j) = \max \left( OPT(i, j - 1), \right)
\]
Correct Dynamic Programming Approach

- \( \text{OPT}(i, j) \) is the maximum number of base pairs in a secondary structure for \( b_i b_2 \ldots b_j \). \( \text{OPT}(i, j) = 0 \), if \( i \geq j - 4 \).
- In the optimal secondary structure on \( b_i b_2 \ldots b_j \)
  1. if \( j \) is not a member of any pair, compute \( \text{OPT}(i, j - 1) \).
  2. if \( j \) pairs with some \( t < j - 4 \), compute \( \text{OPT}(i, t - 1) \) and \( \text{OPT}(t + 1, j - 1) \).
- Since \( t \) can range from \( i \) to \( j - 5 \),

\[
\text{OPT}(i, j) = \max \left( \text{OPT}(i, j - 1), \max_t \left( 1 + \text{OPT}(i, t - 1) + \text{OPT}(t + 1, j - 1) \right) \right)
\]
Correct Dynamic Programming Approach

- **OPT**(i, j) is the maximum number of base pairs in a secondary structure for $b_i b_2 \ldots b_j$. OPT(i, j) = 0, if $i \geq j - 4$.
- In the optimal secondary structure on $b_i b_2 \ldots b_j$
  1. if $j$ is not a member of any pair, compute OPT(i, j − 1).
  2. if $j$ pairs with some $t < j − 4$, compute OPT(i, t − 1) and OPT(t + 1, j − 1).
- Since $t$ can range from $i$ to $j − 5$,

$$OPT(i, j) = \max \left( OPT(i, j − 1), \max_{t} (1 + OPT(i, t − 1) + OPT(t + 1, j − 1)) \right)$$

- In the “inner” maximisation, $t$ runs over all indices between $i$ and $j − 5$ that are allowed to pair with $j$.

---

Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.
Example of Dynamic Programming Algorithm
Dynamic Programming Algorithm

\[
\text{OPT}(i, j) = \max \left( \text{OPT}(i, j - 1), \max_{t} (1 + \text{OPT}(i, t - 1) + \text{OPT}(t + 1, j - 1)) \right)
\]

- There are \( O(n^2) \) sub-problems.
- How do we order them from “smallest” to “largest”? 
Dynamic Programming Algorithm

\[ \text{OPT}(i, j) = \max \left( \text{OPT}(i, j - 1), \max_t (1 + \text{OPT}(i, t - 1) + \text{OPT}(t + 1, j - 1)) \right) \]

- There are \(O(n^2)\) sub-problems.
- How do we order them from "smallest" to "largest"?
- Note that computing \(\text{OPT}(i, j)\) involves sub-problems \(\text{OPT}(l, m)\) where \(m - l < j - i\).
Dynamic Programming Algorithm

\[
\text{OPT}(i,j) = \max \left( \text{OPT}(i,j-1), \max_t \left( 1 + \text{OPT}(i,t-1) + \text{OPT}(t+1,j-1) \right) \right)
\]

- There are \( O(n^2) \) sub-problems.
- How do we order them from “smallest” to “largest”? 
- Note that computing \( \text{OPT}(i,j) \) involves sub-problems \( \text{OPT}(l,m) \) where \( m-l < j-i \).

\[\text{Initialize } \text{OPT}(i,j) = 0 \text{ whenever } i \geq j-4\]
\[\text{For } k = 5, 6, \ldots, n-1 \]
\[\quad \text{For } i = 1, 2, \ldots n-k \]
\[\quad \quad \text{Set } j = i + k \]
\[\quad \quad \text{Compute } \text{OPT}(i,j) \text{ using the recurrence in (6.13)} \]
\[\quad \text{Endfor}\]
\[\text{Endfor}\]
\[\text{Return } \text{OPT}(1,n)\]
Dynamic Programming Algorithm

\[
\text{OPT}(i,j) = \max \left( \text{OPT}(i,j-1), \max_t (1 + \text{OPT}(i,t-1) + \text{OPT}(t+1,j-1)) \right)
\]

- There are \(O(n^2)\) sub-problems.
- How do we order them from “smallest” to “largest”?
- Note that computing \(\text{OPT}(i,j)\) involves sub-problems \(\text{OPT}(l,m)\) where \(m-l < j-i\).

\[
\begin{align*}
\text{Initialize } \text{OPT}(i,j) &= 0 \text{ whenever } i \geq j - 4 \\
\text{For } k &= 5, 6, \ldots, n-1 \\
\text{For } i &= 1, 2, \ldots n - k \\
\text{Set } j &= i + k \\
\text{Compute } \text{OPT}(i,j) \text{ using the recurrence in (6.13)} \\
\end{align*}
\]

Endfor
Endfor

Return \(\text{OPT}(1,n)\)

- Running time of the algorithm is \(O(n^3)\).
Example of Algorithm

RNA sequence $ACCGUAGU$

Initial values

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>i = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>j = 6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>j = 7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>j = 8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>j = 9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Filling in the values for $k = 5$

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>i = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>j = 6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>j = 7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>j = 8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>j = 9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Filling in the values for $k = 6$

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>i = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>j = 6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>j = 7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>j = 8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>j = 9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Filling in the values for $k = 7$

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>i = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>j = 6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>j = 7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>j = 8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>j = 9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Filling in the values for $k = 8$

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>i = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>j = 6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>j = 7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>j = 8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>j = 9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Google Search for “Dynamic Programming”

Did you mean: Dynamic Programming

dynamics - C
Programming forums for software developers and programmers.Languages with dynamic programming, anyone has an idea? ...www.thescientist.com/forum/topics/a01480.html - 26k -Cached - Similar pages - Note this

[PDF] ISS 16 (4) (1981) 261-274
File Format: PDF/Adobe Acrobat - View as HTML
Another important concept in dynamic programming involves the use of successive approximation in solving complicated functional equations of the type ...www.actuarios.org.uk/files/pdf/library/jss-16-4/0261-0274.pdf - Similar pages - Note this

List of Keywords Used - INFORMS: The Institute For Operations ...097 Dynamic programming-optimal control : Applications ...113 Dynamic programming, Bayesian: 114 Dynamic programming, Deterministic ...www.inform.org/article.php?id=797 - 55k -Cached - Similar pages - Note this

[PDF] # This program is a limited implementation of the ...File Format: PDF/Adobe Acrobat - View as HTML
on dynamic programming, # # Array Definitions, # Put initial manual calculations in the array. # This includes v1 - v4(x) calculations done manually ...www.startstrategygroup.com/DynamicProgramming.pdf - Similar pages - Note this

Method for image segmentation by minimizing the ratio between the ...Geiger et al., Dynamic Programming for Detecting, Tracking, and Matching Deformable Contours, Mar. 1995, pp. 294-302, Amini et al., Using Dynamic ...www.patentstorm.us/patents/6078698-claim1.html - 21k -Cached - Similar pages - Note this

Reduced State Sequence Detection Asynchronous Gaussian Multiple ...dynamic programming algorithm whose complexity is independent of the packet length and depends exponentially only on the number of ...ieeexplore.ieee.org/1045/14998/0074837.pdf - Similar pages - Note this

Asset Allocation Techniques in Static and Dynamic Settings
Possible techniques include but are not limited to discrete-time dynamic programming, continuous-time optimal control, Monte Carlo techniques, ...traweb.bea.gov/servlet/balduski/cf.html - 2k -Cached - Similar pages - Note this

Dynamic Calendar Programming | Chicago Web Calendars | Dynamic ...Dynamic Calendar Programming and Web Calendars are two specialties offered by CherryOne Website Design of Chicago. We are a dynamic website ...EasternUniversity/

▶ How do they know “Dynamic” and “Dymanic” are similar?
Sequence Similarity

- Given two strings, measure how similar they are.
- Given a database of strings and a query string, compute the string most similar to query in the database.

Applications:
- Online searches (Web, dictionary).
- Spell-checkers.
- Computational biology
- Speech recognition.
- Basis for Unix diff.
Defining Sequence Similarity

▶ “ocurrance” (wrong) vs “occurrence” (right).

- o-currance
  - occurrence

- o-curr-ance
  - occurrence-nce
Defining Sequence Similarity

▶ “ocurrance” (wrong) vs “occurrence” (right).

- o-currance
  - occurrence

- o-curr-ance
  - occurre-nce

- abbbaa--bbbaaabb
  - ababaaaaabbbba-b
Defining Sequence Similarity

- “occurrance” (wrong) vs “occurrence” (right).

  o-currance
  occurrence

  o-curr-ance
  occurre-nce

  abbbaa--bbaaab
  ababaaabbbaab

- *Edit distance* model: how many changes must you to make to one string to transform it into another?
- Changes allowed are deleting a letter, adding a letter, changing a letter.
Edit Distance

- Proposed by Needleman and Wunsch in the early 1970s.
- Input: two strings $x = x_1x_2x_3 \ldots x_m$ and $y = y_1y_2 \ldots y_n$.
- Sequences $\{1, 2, \ldots, m\}$ and $\{1, 2, \ldots, n\}$ represent positions in $x$ and $y$. 
Proposed by Needleman and Wunsch in the early 1970s.

Input: two strings $x = x_1x_2x_3 \ldots x_m$ and $y = y_1y_2 \ldots y_n$.

Sequences $\{1, 2, \ldots, m\}$ and $\{1, 2, \ldots, n\}$ represent positions in $x$ and $y$.

A matching of these sets is a set $M$ of ordered pairs such that

1. in each pair $(i, j)$, $1 \leq i \leq m$ and $1 \leq j \leq n$ and
2. no index from $x$ (respectively, from $y$) appears as the first (respectively, second) element in more than one ordered pair.

An index is not matched if it does not appear in the matching.
Edit Distance

Proposed by Needleman and Wunsch in the early 1970s.

Input: two strings \( x = x_1x_2x_3 \ldots x_m \) and \( y = y_1y_2 \ldots y_n \).

Sequences \( \{1, 2, \ldots, m\} \) and \( \{1, 2, \ldots, n\} \) represent positions in \( x \) and \( y \).

A matching of these sets is a set \( M \) of ordered pairs such that

1. in each pair \( (i, j) \), \( 1 \leq i \leq m \) and \( 1 \leq j \leq n \) and
2. no index from \( x \) (respectively, from \( y \)) appears as the first (respectively, second) element in more than one ordered pair.

An index is not matched if it does not appear in the matching.

A matching \( M \) is an alignment if there are no “crossing pairs” in \( M \): if \( (i, j) \in M \) and \( (i', j') \in M \) and \( i < i' \) then \( j < j' \).
Proposed by Needleman and Wunsch in the early 1970s.

Input: two strings $x = x_1 x_2 x_3 \ldots x_m$ and $y = y_1 y_2 \ldots y_n$.

Sequences $\{1, 2, \ldots, m\}$ and $\{1, 2, \ldots, n\}$ represent positions in $x$ and $y$.

A matching of these sets is a set $M$ of ordered pairs such that

1. in each pair $(i, j)$, $1 \leq i \leq m$ and $1 \leq j \leq n$ and
2. no index from $x$ (respectively, from $y$) appears as the first (respectively, second) element in more than one ordered pair.

An index is not matched if it does not appear in the matching.

A matching $M$ is an alignment if there are no “crossing pairs” in $M$: if $(i, j) \in M$ and $(i', j') \in M$ and $i < i'$ then $j < j'$.

Cost of an alignment is the sum of gap and mismatch penalties:

- Gap penalty  Penalty $\delta > 0$ for every unmatched index.
- Mismatch penalty  Penalty $\alpha_{x_i, y_j} > 0$ if $(i, j) \in M$ and $x_i \neq y_j$. 
Proposed by Needleman and Wunsch in the early 1970s.

Input: two strings $x = x_1x_2x_3 \ldots x_m$ and $y = y_1y_2 \ldots y_n$.

Sequences $\{1, 2, \ldots, m\}$ and $\{1, 2, \ldots, n\}$ represent positions in $x$ and $y$.

A matching of these sets is a set $M$ of ordered pairs such that

1. in each pair $(i, j)$, $1 \leq i \leq m$ and $1 \leq j \leq n$ and
2. no index from $x$ (respectively, from $y$) appears as the first (respectively, second) element in more than one ordered pair.

An index is not matched if it does not appear in the matching.

A matching $M$ is an alignment if there are no “crossing pairs” in $M$: if $(i, j) \in M$ and $(i', j') \in M$ and $i < i'$ then $j < j'$.

Cost of an alignment is the sum of gap and mismatch penalties:

- Gap penalty Penalty $\delta > 0$ for every unmatched index.
- Mismatch penalty Penalty $\alpha_{x_i,y_j} > 0$ if $(i, j) \in M$ and $x_i \neq y_j$.

Output: compute an alignment of minimal cost.
Dynamic Programming Approach

- Consider index $m \in x$ and index $n \in y$. Is $(m, n) \in M$?
Dynamic Programming Approach

- Consider index $m \in x$ and index $n \in y$. Is $(m, n) \in M$?
- Claim: $(m, n) \not\in M$
Dynamic Programming Approach

Consider index \( m \in x \) and index \( n \in y \). Is \( (m, n) \in M \)?

Claim: \( (m, n) \notin M \Rightarrow m \in x \text{ not matched or } n \in y \text{ not matched.} \)
Dynamic Programming Approach

Consider index $m \in x$ and index $n \in y$. Is $(m, n) \in M$?

Claim: $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.

How should we define sub-problems?
Dynamic Programming Approach

Consider index \( m \in x \) and index \( n \in y \). Is \( (m, n) \in M \)?

Claim: \( (m, n) \notin M \Rightarrow m \in x \) not matched or \( n \in y \) not matched.

How should we define sub-problems?

\( OPT(i, j) \): cost of optimal alignment between \( x = x_1x_2x_3 \ldots x_i \) and \( y = y_1y_2 \ldots y_j \).

\( (i, j) \in M \):
Dynamic Programming Approach

Consider index \( m \in x \) and index \( n \in y \). Is \( (m, n) \in M \)?

Claim: \( (m, n) \not\in M \Rightarrow m \in x \) not matched or \( n \in y \) not matched.

How should we define sub-problems?

\( OPT(i, j) \): cost of optimal alignment between \( x = x_1x_2x_3 \ldots x_i \) and \( y = y_1y_2 \ldots y_j \).

\( (i, j) \in M \): \( OPT(i, j) = \alpha_{x_iy_j} + OPT(i - 1, j - 1) \).
Dynamic Programming Approach

- Consider index $m \in x$ and index $n \in y$. Is $(m, n) \in M$?
- Claim: $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.
- How should we define sub-problems?
- $OPT(i, j)$: cost of optimal alignment between $x = x_1x_2x_3 \ldots x_i$ and $y = y_1y_2 \ldots y_j$.
  - $(i, j) \in M$: $OPT(i, j) = \alpha_{xiyj} + OPT(i - 1, j - 1)$.
  - $i$ not matched:
Dynamic Programming Approach

Consider index \( m \in x \) and index \( n \in y \). Is \((m, n) \in M\)?

- **Claim:** \((m, n) \notin M \Rightarrow m \in x \) not matched or \( n \in y \) not matched.

- How should we define sub-problems?

- **\( OPT(i, j) \):** cost of optimal alignment between \( x = x_1x_2x_3 \ldots x_i \) and \( y = y_1y_2 \ldots y_j \).
  - \((i, j) \in M: \) \( OPT(i, j) = \alpha_{x_iy_j} + OPT(i - 1, j - 1) \).
  - \( i \) not matched: \( OPT(i, j) = \delta + OPT(i - 1, j) \).
Dynamic Programming Approach

Consider index \( m \in x \) and index \( n \in y \). Is \( (m, n) \in M \)?

Claim: \( (m, n) \not\in M \Rightarrow m \in x \) not matched or \( n \in y \) not matched.

How should we define sub-problems?

\( OPT(i, j) \): cost of optimal alignment between \( x = x_1x_2x_3\ldots x_i \) and \( y = y_1y_2\ldots y_j \).

\( (i, j) \in M \): \( OPT(i, j) = \alpha_{x_iy_j} + OPT(i - 1, j - 1) \).

\( i \) not matched: \( OPT(i, j) = \delta + OPT(i - 1, j) \).

\( j \) not matched: \( OPT(i, j) = \delta + OPT(i, j - 1) \).
Dynamic Programming Approach

Consider index \( m \in x \) and index \( n \in y \). Is \((m, n) \in M\)?

Claim: \((m, n) \notin M \Rightarrow m \in x \) not matched or \( n \in y \) not matched.

How should we define sub-problems?

\( OPT(i, j) \): cost of optimal alignment between \( x = x_1x_2x_3 \ldots x_i \) and \( y = y_1y_2 \ldots y_j \).

- \((i, j) \in M\): \( OPT(i, j) = \alpha_{x_iy_j} + OPT(i-1, j-1) \).
- \( i \) not matched: \( OPT(i, j) = \delta + OPT(i-1, j) \).
- \( j \) not matched: \( OPT(i, j) = \delta + OPT(i, j-1) \).

\[ OPT(i, j) = \min \left( \alpha_{x_iy_j} + OPT(i-1, j-1), \delta + OPT(i-1, j), \delta + OPT(i, j-1) \right) \]

\((i, j) \in M\) if and only if minimum is achieved by the first term.

What are the base cases?
Dynamic Programming Approach

- Consider index $m \in x$ and index $n \in y$. Is $(m, n) \in M$?
- Claim: $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.
- How should we define sub-problems?
- $OPT(i, j)$: cost of optimal alignment between $x = x_1 x_2 x_3 \ldots x_i$ and $y = y_1 y_2 \ldots y_j$.
  - $(i, j) \in M$: $OPT(i, j) = \alpha_{x_i y_j} + OPT(i - 1, j - 1)$.
  - $i$ not matched: $OPT(i, j) = \delta + OPT(i - 1, j)$.
  - $j$ not matched: $OPT(i, j) = \delta + OPT(i, j - 1)$.

  $$OPT(i, j) = \min \left( \alpha_{x_i y_j} + OPT(i - 1, j - 1), \delta + OPT(i - 1, j), \delta + OPT(i, j - 1) \right)$$

  - $(i, j) \in M$ if and only if minimum is achieved by the first term.
- What are the base cases? $OPT(i, 0) = OPT(0, i) = i\delta$. 
Dynamic Programming Algorithm

\[ \text{OPT}(i,j) = \min \left( \alpha_{x_i,y_j} + \text{OPT}(i - 1, j - 1), \delta + \text{OPT}(i - 1, j), \delta + \text{OPT}(i, j - 1) \right) \]

Alignment \((X,Y)\)

Array \(A[0\ldots m, 0\ldots n]\)

Initialize \(A[i,0] = i\delta\) for each \(i\)

Initialize \(A[0,j] = j\delta\) for each \(j\)

For \(j = 1,\ldots, n\)

\[ \text{For } i = 1,\ldots, m \]

Use the recurrence (6.16) to compute \(A[i,j]\)

Endfor

Endfor

Return \(A[m,n]\)
Dynamic Programming Algorithm

\[ \text{OPT}(i,j) = \min \left( \alpha_{x_i y_j} + \text{OPT}(i-1,j-1), \delta + \text{OPT}(i-1,j), \delta + \text{OPT}(i,j-1) \right) \]

Alignment \((X,Y)\)

Array \(A[0...m, 0...n]\)

Initialize \(A[i,0] = \delta i\) for each \(i\)

Initialize \(A[0,j] = \delta j\) for each \(j\)

For \(j = 1, \ldots, n\)

\(\text{For } i = 1, \ldots, m\)

Use the recurrence (6.16) to compute \(A[i,j]\)

\(\text{Endfor}\)

\(\text{Endfor}\)

Return \(A[m,n]\)

- Running time is \(O(mn)\). Space used in \(O(mn)\).
Dynamic Programming Algorithm

\[
\text{OPT}(i,j) = \min \left( \alpha_{x_i, y_j} + \text{OPT}(i-1,j-1), \delta + \text{OPT}(i-1,j), \delta + \text{OPT}(i,j-1) \right)
\]

Alignment \((X,Y)\)

Array \(A[0\ldots m, 0\ldots n]\)

Initialize \(A[i,0] = i\delta \) for each \(i\)

Initialize \(A[0,j] = j\delta \) for each \(j\)

For \(j = 1, \ldots, n\)

For \(i = 1, \ldots, m\)

Use the recurrence (6.16) to compute \(A[i,j]\)

Endfor

Endfor

Return \(A[m,n]\)

- Running time is \(O(mn)\). Space used in \(O(mn)\).
- Can compute \(\text{OPT}(m, n)\) in \(O(mn)\) time and \(O(m + n)\) space (Hirschberg 1975, Chapter 6.7).
Dynamic Programming Algorithm

\[ \text{OPT}(i, j) = \min \left( \alpha_{x_i y_j} + \text{OPT}(i - 1, j - 1), \delta + \text{OPT}(i - 1, j), \delta + \text{OPT}(i, j - 1) \right) \]

Alignment \((X, Y)\)

Array \(A[0 \ldots m, 0 \ldots n]\)

Initialize \(A[i, 0] = i\delta\) for each \(i\)

Initialize \(A[0, j] = j\delta\) for each \(j\)

For \(j = 1, \ldots, n\)

For \(i = 1, \ldots, m\)

Use the recurrence (6.16) to compute \(A[i, j]\)

Endfor

Endfor

Return \(A[m, n]\)

- Running time is \(O(mn)\). Space used in \(O(mn)\).
- Can compute \(\text{OPT}(m, n)\) in \(O(mn)\) time and \(O(m + n)\) space (Hirschberg 1975, Chapter 6.7).
- Can compute alignment in the same bounds by combining dynamic programming with divide and conquer.
Graph-theoretic View of Sequence Alignment

Grid graph $G_{xy}$:
- Rows labelled by symbols in $x$ and columns labelled by symbols in $y$.
- Edges from node $(i,j)$ to $(i,j+1)$, to $(i+1,j)$, and to $(i+1,j+1)$.
- Edges directed upward and to the right have cost $\delta$.
- Edge directed from $(i,j)$ to $(i+1,j+1)$ has cost $\alpha^{x_{i+1}y_{j+1}}$.

Figure 6.17 A graph-based picture of sequence alignment.
Graph-theoretic View of Sequence Alignment

Grid graph $G_{xy}$:
- Rows labelled by symbols in $x$ and columns labelled by symbols in $y$.
- Edges from node $(i, j)$ to $(i, j + 1)$, to $(i + 1, j)$, and to $(i + 1, j + 1)$.
- Edges directed upward and to the right have cost $\delta$.
- Edge directed from $(i, j)$ to $(i + 1, j + 1)$ has cost $\alpha_{x_{i+1}y_{j+1}}$.

$f(i, j)$: minimum cost of a path in $G_{XY}$ from $(0, 0)$ to $(i, j)$.

Claim: $f(i, j) = \text{OPT}(i, j)$ and diagonal edges in the shortest path are the matched pairs in the alignment.

Figure 6.17 A graph-based picture of sequence alignment.
Motivation

- Computational finance:
  - Each node is a financial agent.
  - The cost $c_{uv}$ of an edge $(u, v)$ is the cost of a transaction in which we buy from agent $u$ and sell to agent $v$.
  - Negative cost corresponds to a profit.

- Internet routing protocols
  - Dijkstra’s algorithm needs knowledge of the entire network.
  - Routers only know which other routers they are connected to.
  - Algorithm for shortest paths with negative edges is decentralised.
  - We will not study this algorithm in the class. See Chapter 6.9.
Problem Statement

▶ Input: a directed graph $G = (V, E)$ with a cost function $c : E \rightarrow \mathbb{R}$, i.e., $c_{uv}$ is the cost of the edge $(u, v) \in E$.

▶ A negative cycle is a directed cycle whose edges have a total cost that is negative.

▶ Two related problems:

1. If $G$ has no negative cycles, find the shortest s-t path: a path of from source $s$ to destination $t$ with minimum total cost.
2. Does $G$ have a negative cycle?
Problem Statement

- **Input:** a directed graph $G = (V, E)$ with a cost function $c : E \rightarrow \mathbb{R}$, i.e., $c_{uv}$ is the cost of the edge $(u, v) \in E$.
- A *negative cycle* is a directed cycle whose edges have a total cost that is negative.
- Two related problems:
  1. If $G$ has no negative cycles, find the *shortest s-t path*: a path of from source $s$ to destination $t$ with minimum total cost.
  2. Does $G$ have a *negative cycle*?

![Graph](image)

**Figure 6.20** In this graph, one can find s-t paths of arbitrarily negative cost (by going around the cycle $C$ many times).
Approaches for Shortest Path Algorithm

1. Dijsktra’s algorithm.

2. Add some large constant to each edge.
Approaches for Shortest Path Algorithm

1. Dijkstra’s algorithm. Computes incorrect answers because it is greedy.

2. Add some large constant to each edge. Computes incorrect answers because the minimum cost path changes.

Figure 6.21 (a) With negative edge costs, Dijkstra’s Algorithm can give the wrong answer for the Shortest-Path Problem. (b) Adding 3 to the cost of each edge will make all edges nonnegative, but it will change the identity of the shortest s-t path.
Dynamic Programming Approach

- Assume $G$ has no negative cycles.
- Claim: There is a shortest path from $s$ to $t$ that is *simple* (does not repeat a node)
Dynamic Programming Approach

- Assume $G$ has no negative cycles.
- Claim: There is a shortest path from $s$ to $t$ that is simple (does not repeat a node) and hence has at most $n - 1$ edges.
Dynamic Programming Approach

- Assume $G$ has no negative cycles.
- Claim: There is a shortest path from $s$ to $t$ that is simple (does not repeat a node) and hence has at most $n - 1$ edges.
- How do we define sub-problems?
Dynamic Programming Approach

- Assume $G$ has no negative cycles.
- Claim: There is a shortest path from $s$ to $t$ that is *simple* (does not repeat a node) and hence has at most $n - 1$ edges.
- How do we define sub-problems?
  - Shortest $s$-$t$ path has $\leq n - 1$ edges: how we can reach $t$ using $i$ edges, for different values of $i$?
  - We do not know which nodes will be in shortest $s$-$t$ path: how we can reach $t$ from each node in $V$?
Dynamic Programming Approach

- Assume $G$ has no negative cycles.
- Claim: There is a shortest path from $s$ to $t$ that is *simple* (does not repeat a node) and hence has at most $n - 1$ edges.
- How do we define sub-problems?
  - Shortest $s$-$t$ path has $\leq n - 1$ edges: how we can reach $t$ using $i$ edges, for different values of $i$?
  - We do not know which nodes will be in shortest $s$-$t$ path: how we can reach $t$ from each node in $V$?
- Sub-problems defined by varying the number of edges in the shortest path and by varying the starting node in the shortest path.
Dynamic Programming Recursion

- \( OPT(i, v) \): minimum cost of a \( v-t \) path that uses at most \( i \) edges.
- \( t \) is not explicitly mentioned in the sub-problems.
- Goal is to compute \( OPT(n - 1, s) \).
Dynamic Programming Recursion

- \( OPT(i, v) \): minimum cost of a \( v-t \) path that uses at most \( i \) edges.
- \( t \) is not explicitly mentioned in the sub-problems.
- Goal is to compute \( OPT(n - 1, s) \).

\[
OPT(i, v) = \min\left( \begin{array}{c}
OPT(i - 1, v) \\
\min_{w \in V} (c_{vw} + OPT(i - 1, w))
\end{array} \right)
\]

**Figure 6.22** The minimum-cost path \( P \) from \( v \) to \( t \) using at most \( i \) edges.

- Let \( P \) be the optimal path whose cost is \( OPT(i, v) \).
**Dynamic Programming Recursion**

- $OPT(i, v)$: minimum cost of a $v$-$t$ path that uses at most $i$ edges.
- $t$ is not explicitly mentioned in the sub-problems.
- Goal is to compute $OPT(n - 1, s)$.

Let $P$ be the optimal path whose cost is $OPT(i, v)$.

1. If $P$ actually uses $i - 1$ edges, then $OPT(i, v) = OPT(i - 1, v)$.
2. If first node on $P$ is $w$, then $OPT(i, v) = c_{vw} + OPT(i - 1, w)$.

![Graph](image.png)

**Figure 6.22** The minimum-cost path $P$ from $v$ to $t$ using at most $i$ edges.
Dynamic Programming Recursion

- \( \textit{OPT}(i, v) \): minimum cost of a \( v \)-\( t \) path that uses at most \( i \) edges.
- \( t \) is not explicitly mentioned in the sub-problems.
- Goal is to compute \( \textit{OPT}(n - 1, s) \).

![Graph](image)

**Figure 6.22** The minimum-cost path \( P \) from \( v \) to \( t \) using at most \( i \) edges.

- Let \( P \) be the optimal path whose cost is \( \textit{OPT}(i, v) \).
  1. If \( P \) actually uses \( i - 1 \) edges, then \( \textit{OPT}(i, v) = \textit{OPT}(i - 1, v) \).
  2. If first node on \( P \) is \( w \), then \( \textit{OPT}(i, v) = c_{vw} + \textit{OPT}(i - 1, w) \).

\[
\textit{OPT}(i, v) = \min \left( \textit{OPT}(i - 1, v), \min_{w \in V} (c_{vw} + \textit{OPT}(i - 1, w)) \right)
\]
Example of Dynamic Programming Recursion

$$\text{OPT}(i, v) = \min \left( \text{OPT}(i - 1, v), \min_{w \in V} (c_{vw} + \text{OPT}(i - 1, w)) \right)$$
Example of Dynamic Programming Recursion

$$\text{OPT}(i, v) = \min \left( \text{OPT}(i - 1, v), \min_{w \in V} \left( c_{vw} + \text{OPT}(i - 1, w) \right) \right)$$
Example of Dynamic Programming Recursion

\[
OPT(i, v) = \min \left( OPT(i - 1, v), \min_{w \in V} \left( c_{vw} + OPT(i - 1, w) \right) \right)
\]
Example of Dynamic Programming Recursion

\[ \text{OPT}(i, v) = \min \left( \text{OPT}(i - 1, v), \min_{w \in V} (c_{vw} + \text{OPT}(i - 1, w)) \right) \]
Example of Dynamic Programming Recursion

\[
\text{OPT}(i, \nu) = \min \left( \text{OPT}(i - 1, \nu), \min_{w \in V} (c_{\nu w} + \text{OPT}(i - 1, w)) \right)
\]

\[
\begin{array}{c|cccccc}
\hline
& 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
\text{t} & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{a} & \infty & -3 & \infty & \infty & \infty & \infty \\
\text{b} & \infty & \infty & \infty & \infty & \infty & \infty \\
\text{c} & \infty & \infty & \infty & \infty & \infty & \infty \\
\text{d} & \infty & \infty & \infty & \infty & \infty & \infty \\
\text{e} & \infty & \infty & \infty & \infty & \infty & \infty \\
\hline
\end{array}
\]
Example of Dynamic Programming Recursion

$$\text{OPT}(i, v) = \min \left( \text{OPT}(i - 1, v), \min_{w \in V} (c_{vw} + \text{OPT}(i - 1, w)) \right)$$
Example of Dynamic Programming Recursion

$$OPT(i, v) = \min \left( OPT(i - 1, v), \min_{w \in V} (c_{vw} + OPT(i - 1, w)) \right)$$
Example of Dynamic Programming Recursion

$$\text{OPT}(i, v) = \min \left( \text{OPT}(i - 1, v), \min_{w \in V} (c_{vw} + \text{OPT}(i - 1, w)) \right)$$
Example of Dynamic Programming Recursion

$$\text{OPT}(i, v) = \min \left( \text{OPT}(i - 1, v), \min_{w \in V} (c_{vw} + \text{OPT}(i - 1, w)) \right)$$

**Table:**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>$\infty$</td>
<td>-3</td>
<td>-3</td>
<td>-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>0</td>
<td>-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>$\infty$</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>$\infty$</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>$\infty$</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Diagram:**

[Diagram showing a graph with vertices a, b, c, d, e, and t, with edges and weights labeled.]
**Example of Dynamic Programming Recursion**

\[
\text{OPT}(i, v) = \min \left( \text{OPT}(i - 1, v), \min_{w \in V} \left( c_{vw} + \text{OPT}(i - 1, w) \right) \right)
\]
Example of Dynamic Programming Recursion

$$OPT(i, v) = \min \left( OPT(i - 1, v), \min_{w \in V} (c_{vw} + OPT(i - 1, w)) \right)$$
Example of Dynamic Programming Recursion

\[
\text{OPT}(i, v) = \min \left( \text{OPT}(i - 1, v), \min_{w \in V} \left( c_{vw} + \text{OPT}(i - 1, w) \right) \right)
\]
Example of Dynamic Programming Recursion

\[ \text{OPT}(i, v) = \min \left( \text{OPT}(i - 1, v), \min_{w \in V} \left( c_{vw} + \text{OPT}(i - 1, w) \right) \right) \]
Example of Dynamic Programming Recursion

$$OPT(i, v) = \min \left( OPT(i - 1, v), \min_{w \in V} (c_{vw} + OPT(i - 1, w)) \right)$$
Alternate Dynamic Programming Formulation

- $OPT(i, v)$: minimum cost of a $v$-$t$ path that uses exactly $i$ edges. Goal is to compute
Alternate Dynamic Programming Formulation

- $OPT(i, v)$: minimum cost of a $v$-$t$ path that uses exactly $i$ edges. Goal is to compute

$$
\min_{i=1}^{n-1} OPT(i, s).
$$
Alternate Dynamic Programming Formulation

- $OPT(i, v)$: minimum cost of a $v$-$t$ path that uses exactly $i$ edges. Goal is to compute
  \[
  \min_{i=1}^{n-1} OPT(i, s).
  \]

- Let $P$ be the optimal path whose cost is $OPT(i, v)$. 

T. M. Murali February 28, March 5, 17, 19, 21, 2013 Dynamic Programming
**Alternate Dynamic Programming Formulation**

- **OPT**(i, v): minimum cost of a v-t path that uses exactly i edges. Goal is to compute

\[
\min_{i=1}^{n-1} \text{OPT}(i, s).
\]

- Let P be the optimal path whose cost is OPT(i, v).
  - If first node on P is w, then \( \text{OPT}(i, v) = c_{vw} + \text{OPT}(i - 1, w) \).
Alternate Dynamic Programming Formulation

- \( OPT(i, v) \): minimum cost of a \( v-t \) path that uses exactly \( i \) edges. Goal is to compute
  \[
  \min_{i=1}^{n-1} OPT(i, s).
  \]

- Let \( P \) be the optimal path whose cost is \( OPT(i, v) \).
  - If first node on \( P \) is \( w \), then \( OPT(i, v) = c_{vw} + OPT(i-1, w) \).
    \[
    OPT(i, v) = \min_{w \in V} \left( c_{vw} + OPT(i-1, w) \right)
    \]
Alternate Dynamic Programming Formulation

- $OPT(i, v)$: minimum cost of a $v$-$t$ path that uses exactly $i$ edges. Goal is to compute

$$\min_{i=1}^{n-1} OPT(i, s).$$

- Let $P$ be the optimal path whose cost is $OPT(i, v)$.
  - If first node on $P$ is $w$, then $OPT(i, v) = c_{vw} + OPT(i - 1, w)$.

$$OPT(i, v) = \min_{w \in V} (c_{vw} + OPT(i - 1, w))$$

- Compare the two desired solutions:

$$\min_{i=1}^{n-1} OPT(i, s) = \min_{i=1}^{n-1} \left( \min_{w \in V} (c_{sw} + OPT(i - 1, w)) \right)$$

$$OPT(n - 1, s) = \min \left( OPT(n - 2, s), \min_{w \in V} (c_{sw} + OPT(n - 2, w)) \right)$$
Bellman-Ford Algorithm

\[ \text{OPT}(i, v) = \min \left( \text{OPT}(i - 1, v), \min_{w \in V} (c_{vw} + \text{OPT}(i - 1, w)) \right) \]

---

Shortest-Path(G, s, t)

\[ n = \text{number of nodes in } G \]

Array \( M[0 \ldots n-1, V] \)

Define \( M[0, t] = 0 \) and \( M[0, v] = \infty \) for all other \( v \in V \)

For \( i = 1, \ldots, n - 1 \)

For \( v \in V \) in any order

Compute \( M[i, v] \) using the recurrence (6.23)

Endfor

Endfor

Return \( M[n - 1, s] \)

Bellman-Ford Algorithm

$$\text{OPT}(i, v) = \min \left( \text{OPT}(i - 1, v), \min_{w \in V} (c_{vw} + \text{OPT}(i - 1, w)) \right)$$

---

**Shortest-Path**(G,s,t)

\[ n = \text{number of nodes in } G \]
\[ \text{Array } M[0...n-1, V] \]

Define \( M[0, t] = 0 \) and \( M[0, v] = \infty \) for all other \( v \in V \)

For \( i = 1, \ldots, n - 1 \)

For \( v \in V \) in any order

Compute \( M[i, v] \) using the recurrence (6.23)

Endfor

Endfor

Return \( M[n - 1, s] \)

- Space used is \( O(n^2) \). Running time is \( O(n^3) \).
- If shortest path uses \( k \) edges, we can recover it in \( O(kn) \) time by tracing back through smaller sub-problems.
An Improved Bound on the Running Time

- Suppose $G$ has $n$ nodes and $m \ll \binom{n}{2}$ edges. Can we demonstrate a better upper bound on the running time?
An Improved Bound on the Running Time

Suppose \( G \) has \( n \) nodes and \( m \ll \binom{n}{2} \) edges. Can we demonstrate a better upper bound on the running time?

\[
M[i, v] = \min \left( M[i - 1, v], \min_{w \in V} (c_{vw} + M[i - 1, w]) \right)
\]
An Improved Bound on the Running Time

- Suppose $G$ has $n$ nodes and $m \ll \binom{n}{2}$ edges. Can we demonstrate a better upper bound on the running time?

$$M[i, v] = \min \left( M[i - 1, v], \min_{w \in V} (c_{vw} + M[i - 1, w]) \right)$$

- $w$ only needs to range over neighbours of $v$.

- If $n_v$ is the number of neighbours of $v$, then in each round, we spend time equal to

$$\sum_{v \in V} n_v =$$
An Improved Bound on the Running Time

- Suppose $G$ has $n$ nodes and $m \ll \binom{n}{2}$ edges. Can we demonstrate a better upper bound on the running time?

$$M[i, v] = \min \left( M[i - 1, v], \min_{w \in V} (c_{vw} + M[i - 1, w]) \right)$$

- $w$ only needs to range over neighbours of $v$.
- If $n_v$ is the number of neighbours of $v$, then in each round, we spend time equal to

$$\sum_{v \in V} n_v = m.$$ 

- The total running time is $O(mn)$. 

Improving the Memory Requirements

\[ M[i, v] = \min \left( M[i - 1, v], \min_{w \in V} (c_{vw} + M[i - 1, w]) \right) \]

- The algorithm uses \( O(n^2) \) space to store the array \( M \).
Improving the Memory Requirements

\[ M[i, v] = \min \left( M[i - 1, v], \min_{w \in V} (c_{vw} + M[i - 1, w]) \right) \]

- The algorithm uses \( O(n^2) \) space to store the array \( M \).
- Observe that \( M[i, v] \) depends only on \( M[i - 1, *] \) and no other indices.
Improving the Memory Requirements

\[ M[i, v] = \min \left( M[i - 1, v], \min_{w \in V} (c_{vw} + M[i - 1, w]) \right) \]

- The algorithm uses \( O(n^2) \) space to store the array \( M \).
- Observe that \( M[i, v] \) depends only on \( M[i - 1, *] \) and no other indices.
- Modified algorithm:
  1. Maintain two arrays \( M \) and \( M' \) indexed over \( V \).
  2. At the beginning of each iteration, copy \( M \) into \( M' \).
  3. To update \( M \), use

\[ M[v] = \min \left( M'[v], \min_{w \in V} (c_{vw} + M'[w]) \right) \]
Improving the Memory Requirements

\[ M[i, v] = \min \left( M[i - 1, v], \min_{w \in V} (c_{vw} + M[i - 1, w]) \right) \]

- The algorithm uses \( O(n^2) \) space to store the array \( M \).
- Observe that \( M[i, v] \) depends only on \( M[i - 1, \ast] \) and no other indices.
- Modified algorithm:
  1. Maintain two arrays \( M \) and \( M' \) indexed over \( V \).
  2. At the beginning of each iteration, copy \( M \) into \( M' \).
  3. To update \( M \), use

\[ M[v] = \min \left( M'[v], \min_{w \in V} (c_{vw} + M'[w]) \right) \]

- Claim: at the beginning of iteration \( i \), \( M \) stores values of \( \text{OPT}(i - 1, v) \) for all nodes \( v \in V \).
- Space used is \( O(n) \).
Computing the Shortest Path: Algorithm

\[ M[v] = \min \left( M'[v], \min_{w \in V} (c_{vw} + M'[w]) \right) \]

- How can we recover the shortest path that has cost \( M[v] \)?
Computing the Shortest Path: Algorithm

\[ M[v] = \min \left( M'[v], \min_{w \in V} (c_{vw} + M'[w]) \right) \]

- How can we recover the shortest path that has cost \( M[v] \)?
- For each node \( v \), maintain \( f(v) \), the first node after \( v \) in the current shortest path from \( v \) to \( t \).
- To update \( f(v) \), if we ever set \( M[v] \) to \( \min_{w \in V} (c_{vw} + M'[w]) \), set \( f(v) \) to be the node \( w \) that attains this minimum.
- At the end, follow \( f(v) \) pointers from \( s \) to \( t \).
Example of Maintaining Pointers

\[ M[v] = \min \left( M'[v], \min_{w \in N_v} (c_{vw} + M'[w]) \right) \]
Example of Maintaining Pointers

\[ M[v] = \min \left( M'[v], \min_{w \in N_v} (c_{vw} + M'[w]) \right) \]
Example of Maintaining Pointers

\[ M[v] = \min \left( M'[v], \min_{w \in N_v} (c_{vw} + M'[w]) \right) \]
Example of Maintaining Pointers

\[ M[v] = \min \left( M'[v], \min_{w \in N_v} (c_{vw} + M'[w]) \right) \]
Example of Maintaining Pointers

\[
M[v] = \min \left( M'[v], \min_{w \in N_v} (c_{vw} + M'[w]) \right)
\]

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
\hline
t & 0 & 0 & 0 & 0 & 0 & 0 \\
a & \infty & -3 & -3 & -4 & -6 & \\
b & \infty & \infty & 0 & -2 & -2 & \\
c & \infty & 3 & 3 & 3 & 3 & \\
d & \infty & 4 & 3 & 3 & 2 & \\
e & \infty & 2 & 0 & 0 & 0 & \\
\end{array}
\]
Example of Maintaining Pointers

\[ M[v] = \min \left( M'[v], \min_{w \in N_v} (c_{vw} + M'[w]) \right) \]
Example of Maintaining Pointers

\[ M[v] = \min \left( M'[v], \min_{w \in N_v} (c_{vw} + M'[w]) \right) \]
Computing the Shortest Path: Correctness

- Pointer graph $P(V, F)$: each edge in $F$ is $(v, f(v))$.
  - Can $P$ have cycles?
  - Is there a path from $s$ to $t$ in $P$?
  - Can there be multiple paths $s$ to $t$ in $P$?
  - Which of these is the shortest path?

```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$a$</td>
<td>8</td>
<td>-3</td>
<td>-3</td>
<td>-4</td>
<td>-6</td>
<td>-6</td>
</tr>
<tr>
<td>$b$</td>
<td>8</td>
<td>$\infty$</td>
<td>0</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
</tr>
<tr>
<td>$c$</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$d$</td>
<td>8</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>$e$</td>
<td>$\infty$</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Diagram:

- $a$ to $b$: 6
- $b$ to $e$: 8
- $c$ to $e$: 8
- $d$ to $e$: 8
- $d$ to $t$: 3
- $b$ to $d$: 4
- $b$ to $a$: -1
- $b$ to $c$: -2
- $c$ to $d$: -3
- $a$ to $d$: -3
- $a$ to $t$: -4
- $e$ to $t$: 2

- $t$ to $t$: 0
Computing the Shortest Path: Cycles in $P$

$$M[v] = \min \left( M'[v], \min_{w \in V} (c_{vw} + M'[w]) \right)$$

- Claim: If $P$ has a cycle $C$, then $C$ has negative cost.
Computing the Shortest Path: Cycles in $P$

\[ M[v] = \min \left( M'[v], \min_{w \in V} (c_{vw} + M'[w]) \right) \]

- **Claim:** If $P$ has a cycle $C$, then $C$ has negative cost.
  - Suppose we set $f(v) = w$. Between this assignment and the assignment of $f(v)$ to some other node, $M[v] \geq c_{vw} + M[w]$ (because $M[w]$ may itself decrease).

\[ \sum_{i=1}^{k-1} c_{vi} + c_{vk} > M[v_1] - M[v_k] \]

**Corollary:** if $G$ has no negative cycles that $P$ does not either.
Computing the Shortest Path: Cycles in $P$

\[ M[v] = \min \left( M'[v], \min_{w \in V} (c_{vw} + M'[w]) \right) \]

- **Claim:** If $P$ has a cycle $C$, then $C$ has negative cost.
  - Suppose we set $f(v) = w$. Between this assignment and the assignment of $f(v)$ to some other node, $M[v] \geq c_{vw} + M[w]$ (because $M[w]$ may itself decrease).
  - Let $v_1, v_2, \ldots, v_k$ be the nodes in $C$ and assume that $(v_k, v_1)$ is the last edge to have been added.
  - What is the situation just before this addition?
Computing the Shortest Path: Cycles in $P$

$$M[v] = \min \left( M'[v], \min_{w \in V} (c_{vw} + M'[w]) \right)$$

- **Claim:** If $P$ has a cycle $C$, then $C$ has negative cost.
  - Suppose we set $f(v) = w$. Between this assignment and the assignment of $f(v)$ to some other node, $M[v] \geq c_{vw} + M[w]$ (because $M[w]$ may itself decrease).
  - Let $v_1, v_2, \ldots, v_k$ be the nodes in $C$ and assume that $(v_k, v_1)$ is the last edge to have been added.
  - What is the situation just before this addition?
  - $M[v_i] - M[v_{i+1}] \geq c_{v_i,v_{i+1}}$, for all $1 \leq i < k - 1$.
  - $M[v_k] - M[v_1] > c_{v_k,v_1}$. 

Computing the Shortest Path: Cycles in $P$

$$M[v] = \min \left( M'[v], \min_{w \in V} (c_{vw} + M'[w]) \right)$$

- **Claim:** If $P$ has a cycle $C$, then $C$ has negative cost.
  - Suppose we set $f(v) = w$. Between this assignment and the assignment of $f(v)$ to some other node, $M[v] \geq c_{vw} + M[w]$ (because $M[w]$ may itself decrease).
  - Let $v_1, v_2, \ldots, v_k$ be the nodes in $C$ and assume that $(v_k, v_1)$ is the last edge to have been added.
  - What is the situation just before this addition?
  - $M[v_i] - M[v_{i+1}] \geq c_{v_i v_{i+1}}$, for all $1 \leq i < k - 1$.
  - $M[v_k] - M[v_1] > c_{v_k v_1}$.
  - Adding all these inequalities, $0 > \sum_{i=1}^{k-1} c_{v_i v_{i+1}} + c_{v_k v_1} = \text{cost of } C$.

$\text{T. M. Murali February 28, March 5, 17, 19, 21, 2013 Dynamic Programming}$
Computing the Shortest Path: Cycles in $P$

$$M[v] = \min \left( M'[v], \min_{w \in V} (c_{vw} + M'[w]) \right)$$

- **Claim:** If $P$ has a cycle $C$, then $C$ has negative cost.
  - Suppose we set $f(v) = w$. Between this assignment and the assignment of $f(v)$ to some other node, $M[v] \geq c_{vw} + M[w]$ (because $M[w]$ may itself decrease).
  - Let $v_1, v_2, \ldots v_k$ be the nodes in $C$ and assume that $(v_k, v_1)$ is the last edge to have been added.
  - What is the situation just before this addition?
    - $M[v_i] - M[v_{i+1}] \geq c_{v_i,v_{i+1}}$, for all $1 \leq i < k - 1$.
    - $M[v_k] - M[v_1] > c_{v_k,v_1}$.
    - Adding all these inequalities, $0 > \sum_{i=1}^{k-1} c_{v_i,v_{i+1}} + c_{v_k,v_1} = \text{cost of } C$.

- **Corollary:** if $G$ has no negative cycles that $P$ does not either.
Computing the Shortest Path: Paths in $P$

- Let $P$ be the pointer graph upon termination of the algorithm.
- Consider the path $P_v$ in $P$ obtained by following the pointers from $v$ to $f(v) = v_1$, to $f(v_1) = v_2$, and so on.
Computing the Shortest Path: Paths in $P$

- Let $P$ be the pointer graph upon termination of the algorithm.
- Consider the path $P_v$ in $P$ obtained by following the pointers from $v$ to $f(v) = v_1$, to $f(v_1) = v_2$, and so on.
- Claim: $P_v$ terminates at $t$. 


Computing the Shortest Path: Paths in $P$

- Let $P$ be the pointer graph upon termination of the algorithm.
- Consider the path $P_v$ in $P$ obtained by following the pointers from $v$ to $f(v) = v_1$, to $f(v_1) = v_2$, and so on.
- Claim: $P_v$ terminates at $t$.
- Claim: $P_v$ is the shortest path in $G$ from $v$ to $t$. 
Bellman-Ford Algorithm: One Array

\[ M[v] = \min\left( M[v], \min_{w \in V} (c_{vw} + M[w]) \right) \]

- We can prove algorithm’s correctness in this case as well.
Bellman-Ford Algorithm: Early Termination

\[ M[v] = \min \left( M[v], \min_{w \in V} (c_{vw} + M[w]) \right) \]

- In general, after \( i \) iterations, the path whose length is \( M[v] \) may have many more than \( i \) edges.
Bellman-Ford Algorithm: Early Termination

\[ M[v] = \min \left( M[v], \min_{w \in V} (c_{vw} + M[w]) \right) \]

- In general, after \( i \) iterations, the path whose length is \( M[v] \) may have many more than \( i \) edges.
- Early termination: If \( M \) does not change after processing all the nodes, we have computed all the shortest paths to \( t \).