Dynamic Programming

T. M. Murali

March 20, 25, 27, April 1, 2024

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

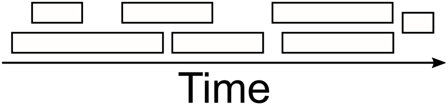
- Bellman pioneered the systematic study of dynamic programming in the 1950s.
- The Secretary of Defense at that time was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.
  - "it's impossible to use dynamic in a pejorative sense"
  - "something not even a 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.

- Input: Start and end time of each ride.
- Constraint: Cannot be in two places at one time.
- Goal: Compute the largest number of rides you can be on in one day.

Review: Interval Scheduling



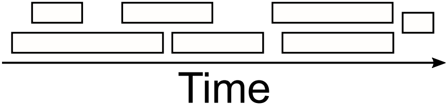
INTERVAL SCHEDULING

INSTANCE: Set $\{(s(i), f(i)), 1 \le i \le n\}$ 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.
- For any input set of jobs, algorithm must provably compute the largest set of compatible jobs.

Review: Interval Scheduling



INTERVAL SCHEDULING

INSTANCE: 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.
- For any input set of jobs, algorithm must provably compute the largest set of compatible jobs.
- 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

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 the value $\sum_{i \in S} v_i$ is maximised.

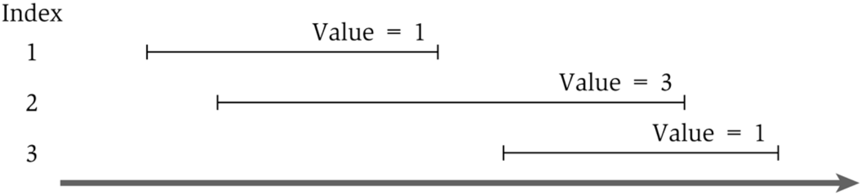


Figure 6.1 A simple instance of weighted interval scheduling.

- Dynamic Programming: Weighted Interval Scheduling: Greedy Algorithm

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 the value $\sum_{i \in S} v_i$ is maximised.

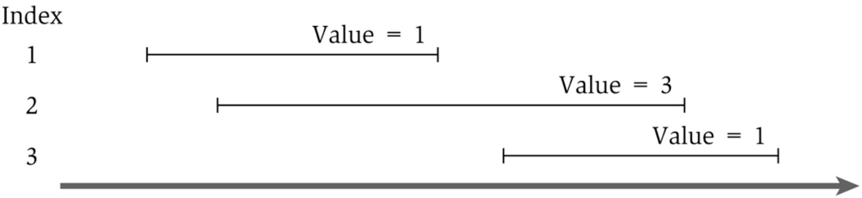
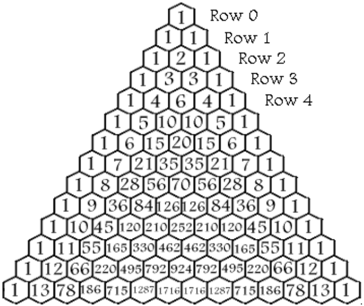


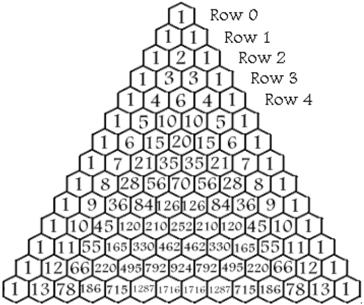
Figure 6.1 A simple instance of weighted interval scheduling.

- ▸ Dynamic Programming: Weighted Interval Scheduling: Greedy Algorithm Greedy algorithm can produce arbitrarily bad results for this problem.

Detour: a Binomial Identity

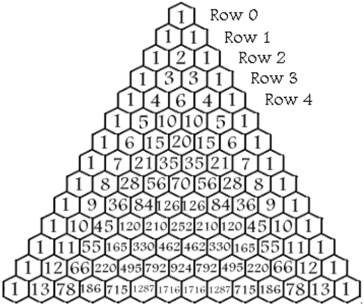


Detour: a Binomial Identity



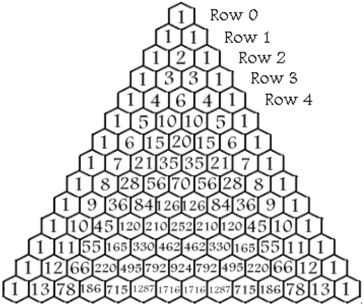
- Pascal's triangle: ▸ Dynamic Programming: Pascal's triangle

Detour: a Binomial Identity



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

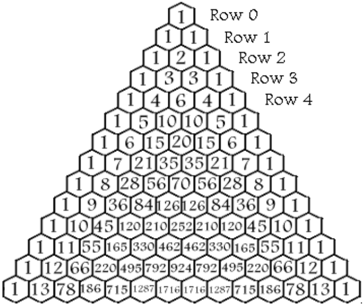
Detour: a Binomial Identity



- Pascal's triangle: Dynamic Programming: 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



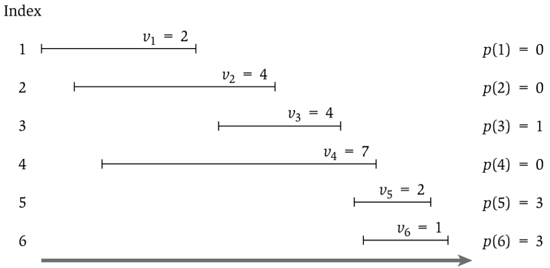
- Pascal's triangle: ▸ Dynamic Programming: Pascal's triangle
  - Each element is a binomial co-efficient.
  - Each element is the sum of the two elements above it.

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

- Proof: either we include the nth element in a subset or not ...

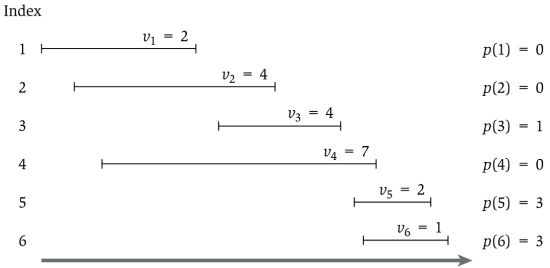
Approach

- 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. Dynamic Programming: Weighted Interval Scheduling: Compatible jobs



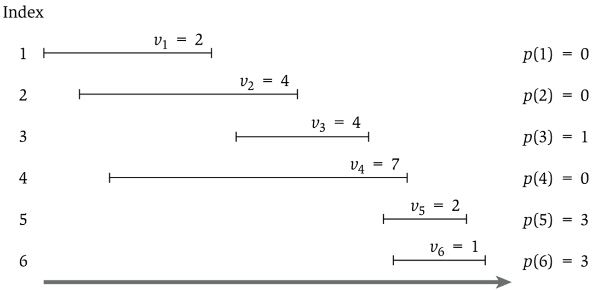
Approach

- 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. ▸ Dynamic Programming: Weighted Interval Scheduling: Compatible jobs
- All jobs that come before job $p(j)$ are also compatible with job j.



- We will develop optimal algorithm from obvious statements about the problem.

Sub-problems

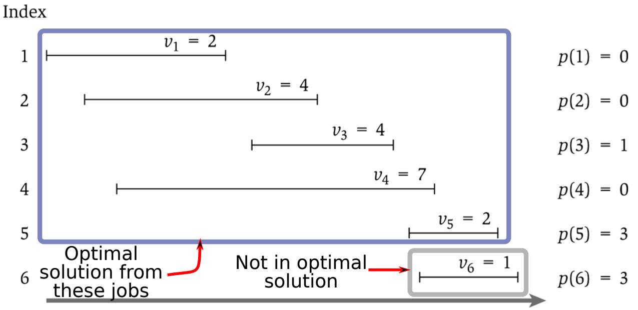


- Let $\mathcal{O}$ be the optimal solution: it contains a subset of the input jobs. Two cases to consider. One of these cases must be true.

  Case 1: job n is not in $\mathcal{O}$.

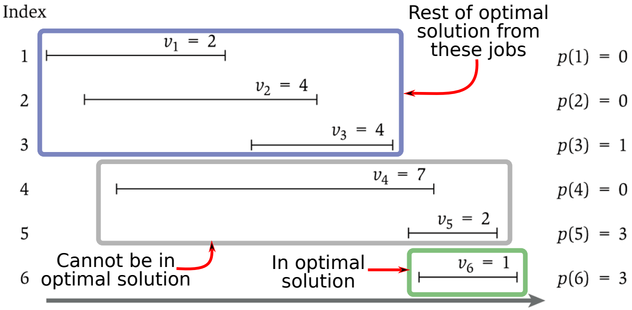
  Case 2: job n is in $\mathcal{O}$.

Sub-problems



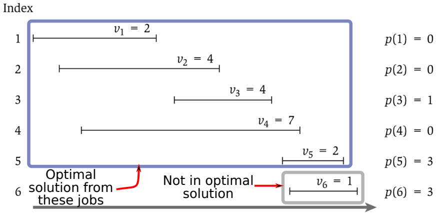
- Let $\mathcal{O}$ be the optimal solution: it contains a subset of the input jobs. Two cases to consider. One of these cases must be true.
  - Case 1: job n is not in $\mathcal{O}$. $\mathcal{O}$ must be the optimal solution for jobs $\{1, 2, \ldots, n-1\}$.
  - Case 2: job n is in $\mathcal{O}$.

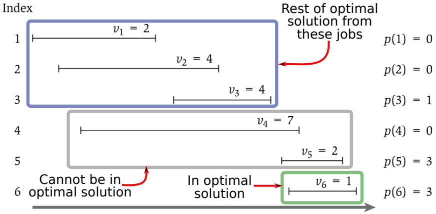
Sub-problems



- Let $\mathcal{O}$ be the optimal solution: it contains a subset of the input jobs. Two cases to consider. One of these cases must be true.
  - Case 1: job n is not in $\mathcal{O}$. $\mathcal{O}$ must be the optimal solution for jobs $\{1, 2, \ldots, n-1\}$.
  - Case 2: job n is in $\mathcal{O}$.
    - ★ $\mathcal{O}$ cannot use incompatible jobs $\{p(n)+1, p(n)+2, \ldots, n-1\}$.
    - ★ Remaining jobs in $\mathcal{O}$ must be the optimal solution for jobs $\{1, 2, \ldots, p(n)\}$.

Sub-problems





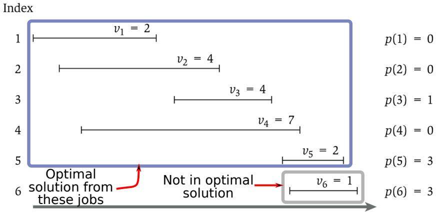
- Let $\mathcal{O}$ be the optimal solution: it contains a subset of the input jobs. Two cases to consider. One of these cases must be true.

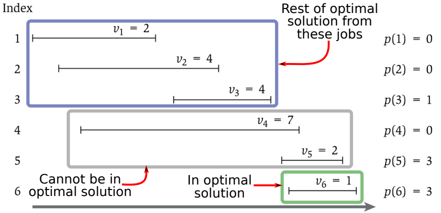
  Case 1: job n is not in $\mathcal{O}$. $\mathcal{O}$ must be the optimal solution for jobs $\{1, 2, \ldots, n-1\}$.
  Case 2: job n is in $\mathcal{O}$.

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

- $\mathcal{O}$ must be the best of these two choices!

Sub-problems





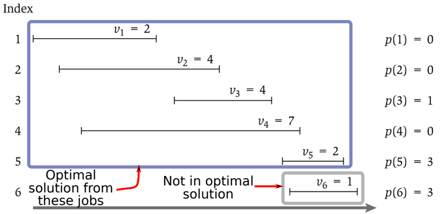
- Let $\mathcal{O}$ be the optimal solution: it contains a subset of the input jobs. Two cases to consider. One of these cases must be true.

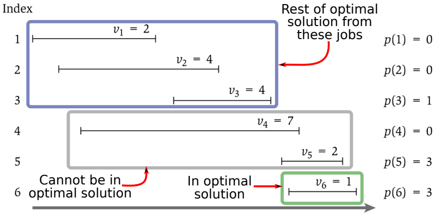
  Case 1: job n is not in $\mathcal{O}$. $\mathcal{O}$ must be the optimal solution for jobs $\{1, 2, \ldots, n-1\}$.

  Case 2: job n is in $\mathcal{O}$.

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

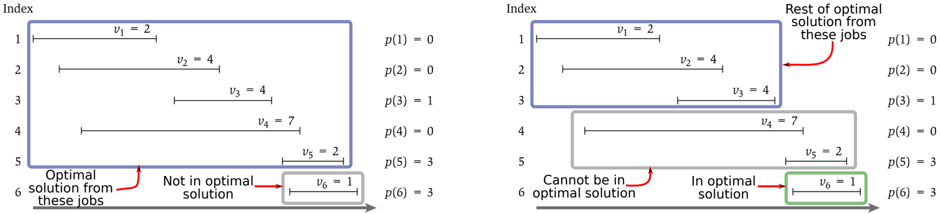
Recursion





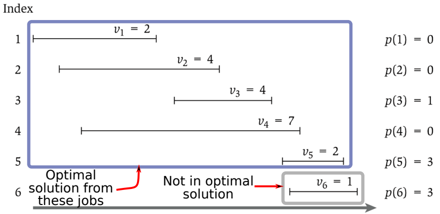
- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($\text{OPT}(0) = 0$).

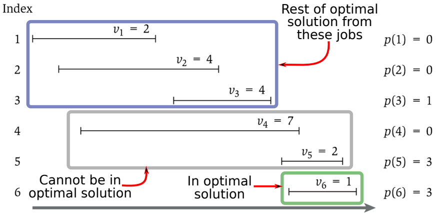
Recursion



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

Recursion

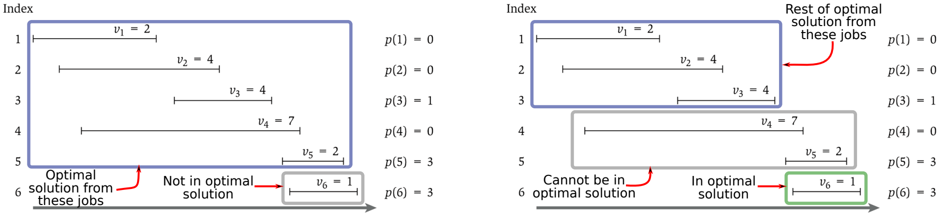




- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($\text{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\text{OPT}(n)$.
- To compute $\text{OPT}(j)$:

  Case 1 $j \notin \mathcal{O}_j$:

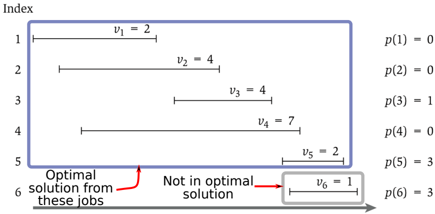
Recursion

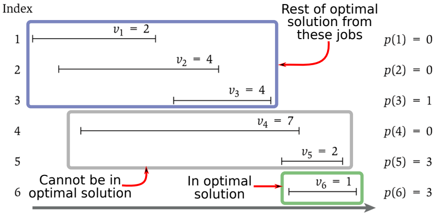


- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($\text{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\text{OPT}(n)$.
- To compute $\text{OPT}(j)$:

  Case 1 $j \notin \mathcal{O}_j$: $\text{OPT}(j) = \text{OPT}(j-1)$.

Recursion



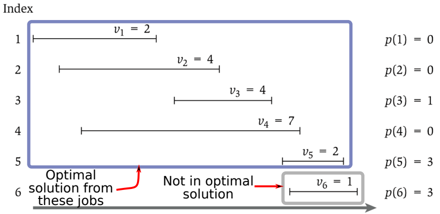


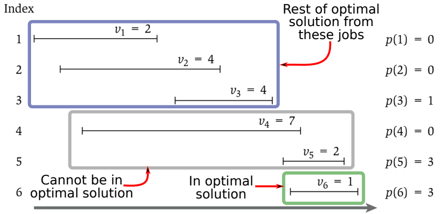
- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($\text{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\text{OPT}(n)$.
- To compute $\text{OPT}(j)$:

  Case 1 $j \notin \mathcal{O}_j$: $\text{OPT}(j) = \text{OPT}(j-1)$.

  Case 2 $j \in \mathcal{O}_j$:

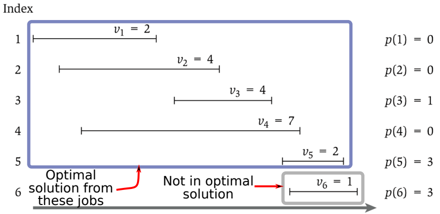
Recursion

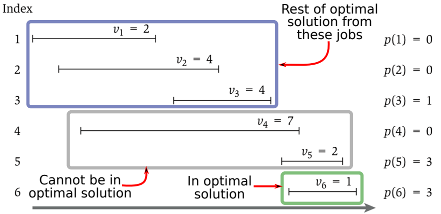




- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($\text{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\text{OPT}(n)$.
- To compute $\text{OPT}(j)$:
  - Case 1 $j \notin \mathcal{O}_j$: $\text{OPT}(j) = \text{OPT}(j-1)$.
  - Case 2 $j \in \mathcal{O}_j$: $\text{OPT}(j) = v_j + \text{OPT}(p(j))$

Recursion

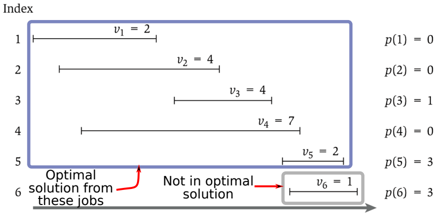


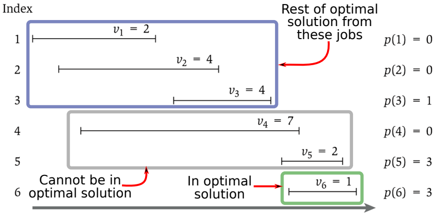


- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($\text{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\text{OPT}(n)$.
- To compute $\text{OPT}(j)$:
  - Case 1 $j \notin \mathcal{O}_j$: $\text{OPT}(j) = \text{OPT}(j-1)$.
  - Case 2 $j \in \mathcal{O}_j$: $\text{OPT}(j) = v_j + \text{OPT}(p(j))$

$$\text{OPT}(j) = \max\left(v_j + \text{OPT}(p(j)), \text{OPT}(j-1)\right)$$

Recursion





- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($\text{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\text{OPT}(n)$.
- To compute $\text{OPT}(j)$:

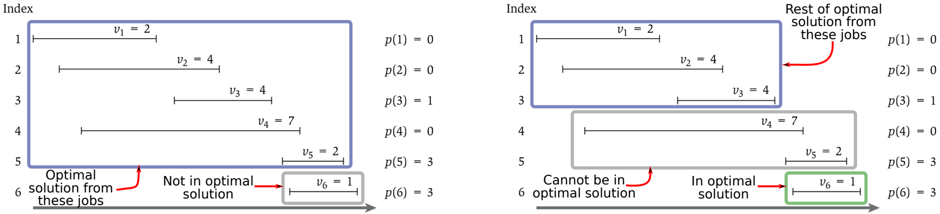
  Case 1 $j \notin \mathcal{O}_j$: $\text{OPT}(j) = \text{OPT}(j-1)$.

  Case 2 $j \in \mathcal{O}_j$: $\text{OPT}(j) = v_j + \text{OPT}(p(j))$

$$\text{OPT}(j) = \max\left(v_j + \text{OPT}(p(j)), \text{OPT}(j-1)\right)$$

- When does job j belong to $\mathcal{O}_j$? ▸ Dynamic Programming: Weighted Interval Scheduling: Optimal Solution

Recursion



- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution $(\text{OPT}(0) = 0)$.
- We are seeking $\mathcal{O}_n$ with a value of $\text{OPT}(n)$.
- To compute $\text{OPT}(j)$:

  Case 1 $j \notin \mathcal{O}_j$: $\text{OPT}(j) = \text{OPT}(j-1)$.

  Case 2 $j \in \mathcal{O}_j$: $\text{OPT}(j) = v_j + \text{OPT}(p(j))$

$$\text{OPT}(j) = \max\left(v_j + \text{OPT}(p(j)), \text{OPT}(j-1)\right)$$

- When does job j belong to $\mathcal{O}_j$? Dynamic Programming: Weighted Interval Scheduling: Optimal Solution If and only if $v_j + \text{OPT}(p(j)) \geq \text{OPT}(j-1)$.

Recursive Algorithm

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

```
Compute-Opt(j)
  If j = 0 then
    Return 0
  Else
    Return max(v_j+Compute-Opt(p(j)), Compute-Opt(j − 1))
  Endif
```

Recursive Algorithm

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

```
Compute-Opt(j)
  If j = 0 then
    Return 0
  Else
    Return max(v_j+Compute-Opt(p(j)), Compute-Opt(j − 1))
  Endif
```

- Correctness of algorithm follows by induction (see textbook for proof).

Example of Recursive Algorithm

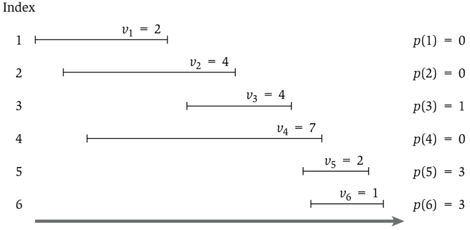


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

$\mathrm{OPT}(6) =$ ▸ Dynamic Programming: Weighted Interval Scheduling: Optimal Solution for Example

$\mathrm{OPT}(5) =$

$\mathrm{OPT}(4) =$

$\mathrm{OPT}(3) =$

$\mathrm{OPT}(2) =$

$\mathrm{OPT}(1) =$

$\mathrm{OPT}(0) = 0$

Example of Recursive Algorithm

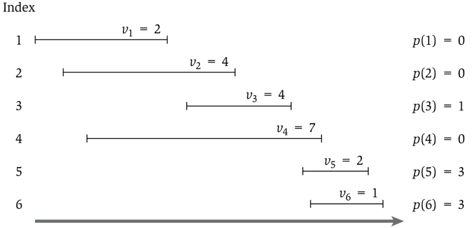


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

$\text{OPT}(5) =$

$\text{OPT}(4) =$

$\text{OPT}(3) =$

$\text{OPT}(2) =$

$\text{OPT}(1) =$

$\text{OPT}(0) = 0$

Example of Recursive Algorithm

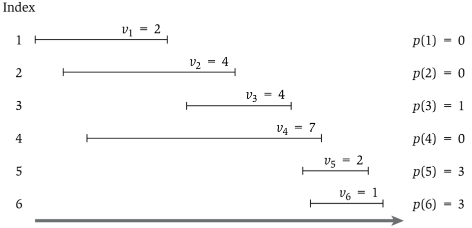


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

Example of Recursive Algorithm

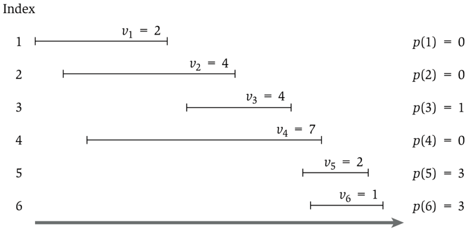


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

$$
\begin{aligned}
\mathrm{OPT}(6) &= \max(v_6 + \mathrm{OPT}(p(6)), \mathrm{OPT}(5)) = \max(1 + \mathrm{OPT}(3), \mathrm{OPT}(5)) \\
\mathrm{OPT}(5) &= \max(v_5 + \mathrm{OPT}(p(5)), \mathrm{OPT}(4)) = \max(2 + \mathrm{OPT}(3), \mathrm{OPT}(4)) \\
\mathrm{OPT}(4) &= \max(v_4 + \mathrm{OPT}(p(4)), \mathrm{OPT}(3)) = \max(7 + \mathrm{OPT}(0), \mathrm{OPT}(3)) \\
\mathrm{OPT}(3) &= \\
\mathrm{OPT}(2) &= \\
\mathrm{OPT}(1) &= \\
\mathrm{OPT}(0) &= 0
\end{aligned}
$$

Example of Recursive Algorithm

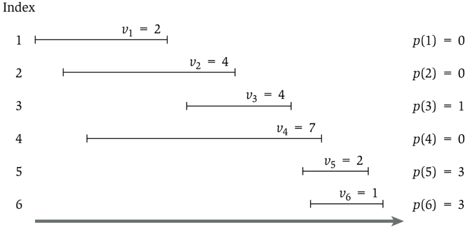


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

Example of Recursive Algorithm

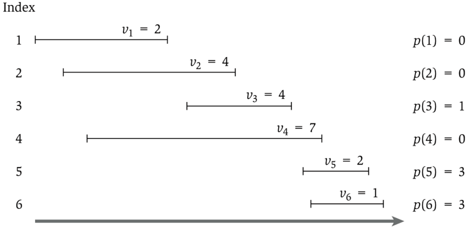


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

$$\begin{aligned}
\mathrm{OPT}(6) &= \max(v_6 + \mathrm{OPT}(p(6)), \mathrm{OPT}(5)) = \max(1 + \mathrm{OPT}(3), \mathrm{OPT}(5))\\
\mathrm{OPT}(5) &= \max(v_5 + \mathrm{OPT}(p(5)), \mathrm{OPT}(4)) = \max(2 + \mathrm{OPT}(3), \mathrm{OPT}(4))\\
\mathrm{OPT}(4) &= \max(v_4 + \mathrm{OPT}(p(4)), \mathrm{OPT}(3)) = \max(7 + \mathrm{OPT}(0), \mathrm{OPT}(3))\\
\mathrm{OPT}(3) &= \max(v_3 + \mathrm{OPT}(p(3)), \mathrm{OPT}(2)) = \max(4 + \mathrm{OPT}(1), \mathrm{OPT}(2))\\
\mathrm{OPT}(2) &= \max(v_2 + \mathrm{OPT}(p(2)), \mathrm{OPT}(1)) = \max(4 + \mathrm{OPT}(0), \mathrm{OPT}(1))\\
\mathrm{OPT}(1) &=\\
\mathrm{OPT}(0) &= 0
\end{aligned}$$

Example of Recursive Algorithm

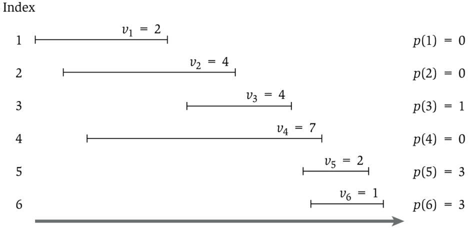


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

Example of Recursive Algorithm

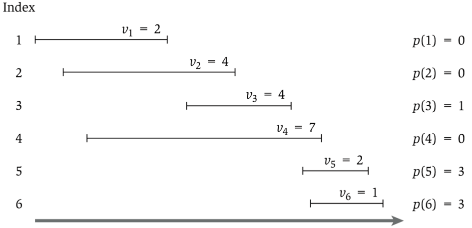


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

Example of Recursive Algorithm

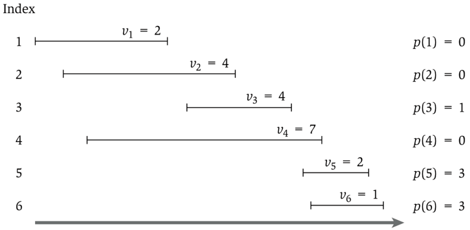


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

Example of Recursive Algorithm

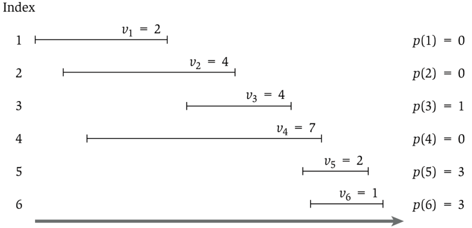


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

Example of Recursive Algorithm

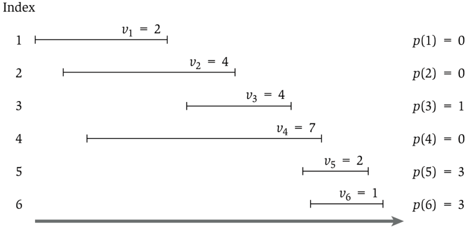


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

Example of Recursive Algorithm

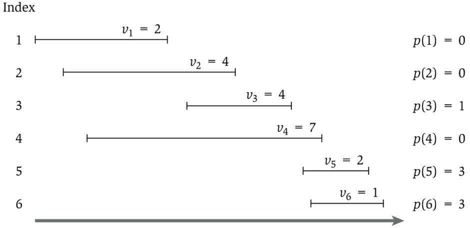


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

Example of Recursive Algorithm

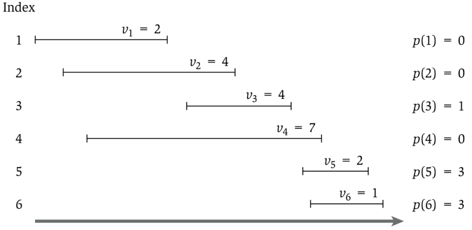


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

- Optimal solution is

Example of Recursive Algorithm

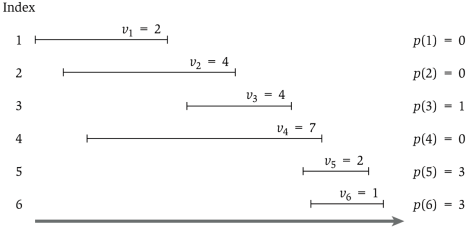


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

- 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+Compute-Opt(p(j)), Compute-Opt(j − 1))
  Endif
```

Running Time of Recursive Algorithm

```
Compute-Opt(j)
  If j = 0 then
    Return 0
  Else
    Return max(v_j+Compute-Opt(p(j)), 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+Compute-Opt(p(j)), 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+Compute-Opt(p(j)), 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 $j - 2$.

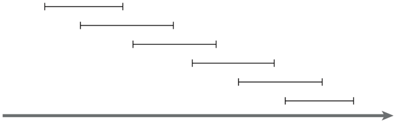


Figure 6.4 An instance of weighted interval scheduling on which the simple Compute-Opt recursion will take exponential time. The values of all intervals in this instance are 1.

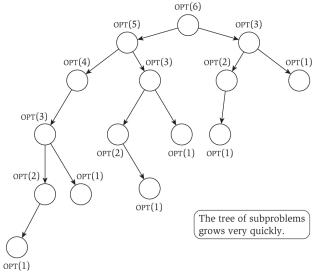


Figure 6.3 The tree of subproblems called by Compute-Opt on the problem instance of Figure 6.2.

Memoisation

- Store OPT(j) values in a cache and reuse them rather than recompute them.

Memoisation

- Store OPT(j) values in a cache and reuse them rather than recompute them.

```
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
```

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).

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?

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 $\mathcal{O}$ in Addition to OPT(n)

Computing $\mathcal{O}$ in Addition to OPT(n)

- Explicitly store $\mathcal{O}_j$ in addition to OPT(j).

Computing $\mathcal{O}$ in Addition to OPT(n)

- Explicitly store $\mathcal{O}_j$ in addition to OPT(j). Running time becomes $O(n^2)$.

Computing $\mathcal{O}$ in Addition to OPT(n)

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

Computing $\mathcal{O}$ in Addition to 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)] ≥ 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, ..., n
    M[j] = max(v_j + M[p(j)], M[j-1])
  Endfor
```

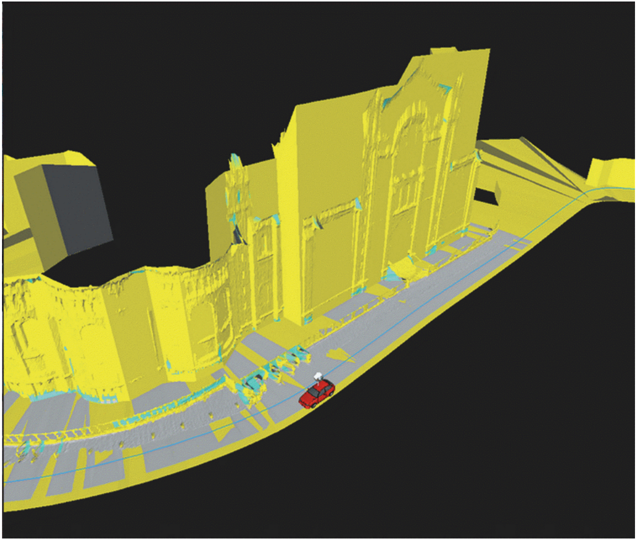
Basic Outline of Dynamic Programming

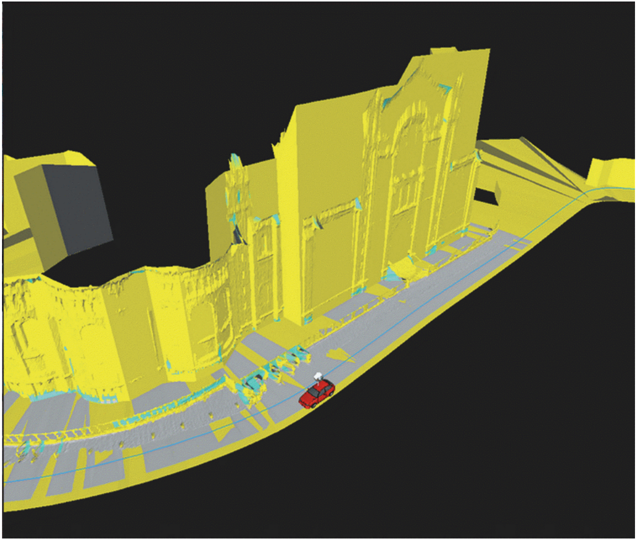
- To solve a problem, we need a collection of sub-problems that satisfy a few properties:
  1. There are a polynomial number of sub-problems.
  2. The solution to the problem can be computed easily from the solutions to the sub-problems.
  3. There is a natural ordering of the sub-problems from "smallest" 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)?

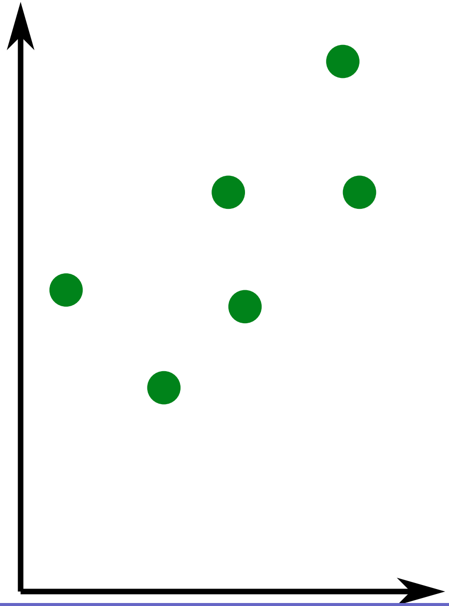

Google maps
Street View
google.com/streetview



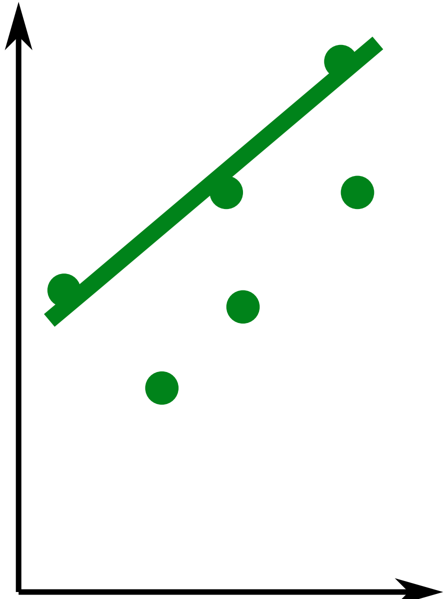


Imagery from street view vehicles is accompanied by laser range data, which is aggregated and simplified by robustly fitting it in a coarse mesh that models the dominant scene surfaces.

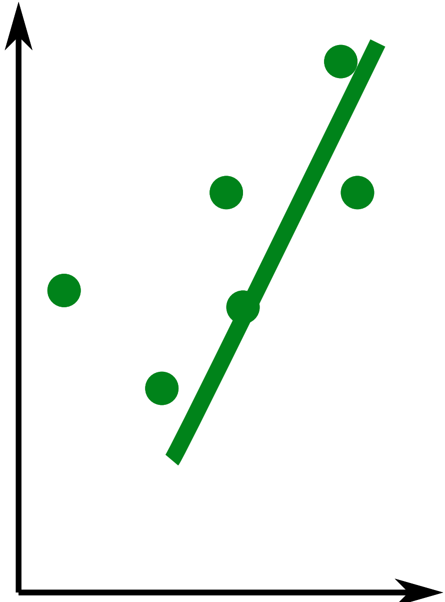
Fitting Lines



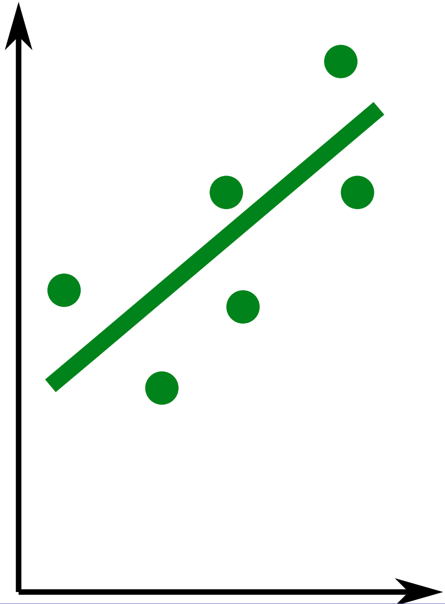
Fitting Lines



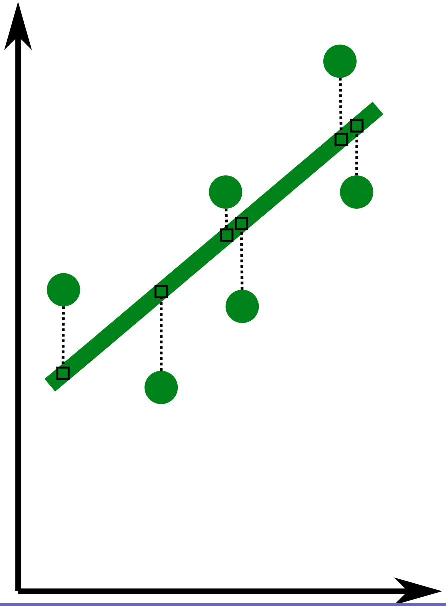
Fitting Lines



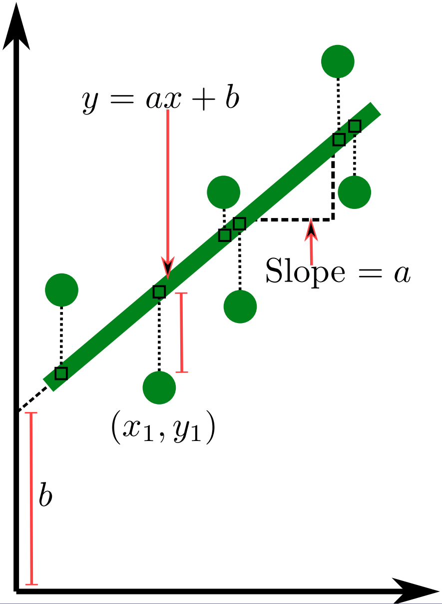
Fitting Lines



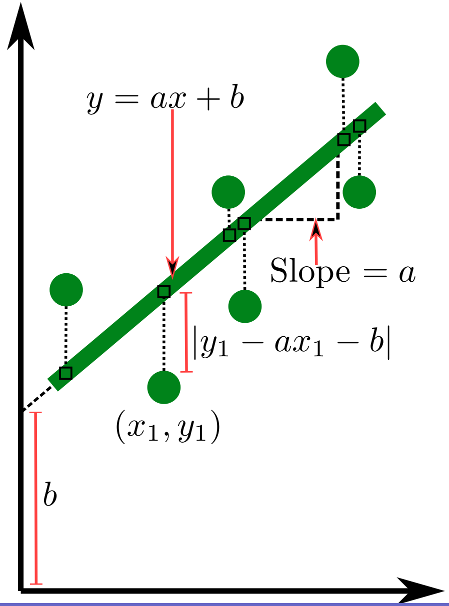
Fitting Lines



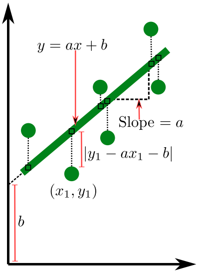
Fitting Lines



Fitting Lines

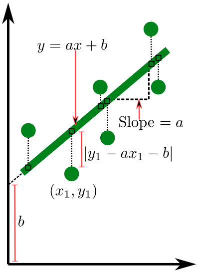


Least Squares Problem



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

Least Squares Problem



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

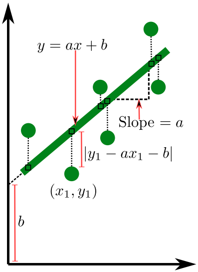
LEAST SQUARES REGRESSION

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



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

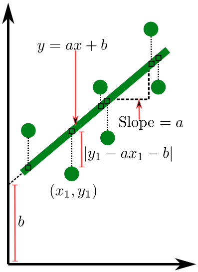
LEAST SQUARES REGRESSION
INSTANCE: Set $P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ of n points.
SOLUTION: Line $L : y = ax + b$ that minimises

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

- How many unknown parameters must we find values for?

▸ Dynamic Programming: Segmented Least Squares: Line Fitting

Least Squares Problem



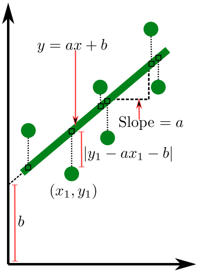
- Given scientific or statistical data plotted on two axes.
- Find the "best" line that "passes" through these points.

LEAST SQUARES REGRESSION
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

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

- How many unknown parameters must we find values for? Two: a and b.

Least Squares Problem



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

LEAST SQUARES REGRESSION

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

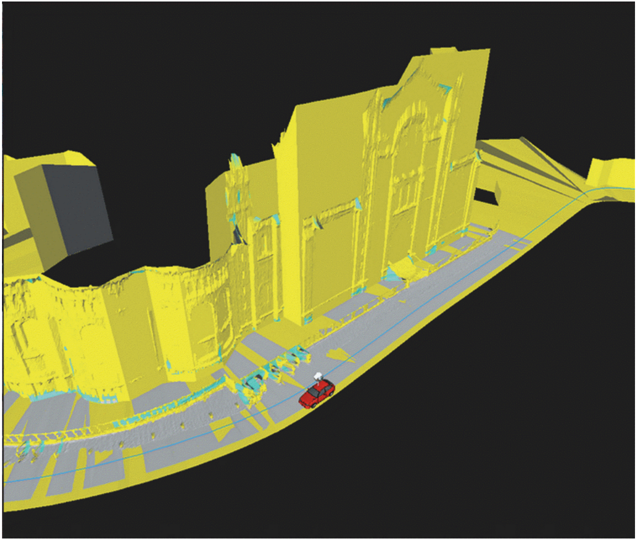
SOLUTION: Line $L : y = ax + b$ that minimises

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

- How many unknown parameters must we find values for? Two: a and b.
- Solution is achieved by

$$a = \frac{n \sum_i x_i y_i - \left(\sum_i x_i\right)\left(\sum_i y_i\right)}{n \sum_i x_i^2 - \left(\sum_i x_i\right)^2} \text{ and } b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

Segmented Least Squares



Segmented Least Squares

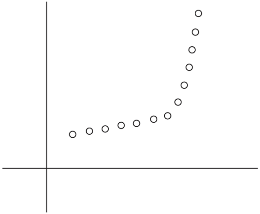


Figure 6.7 A set of points that lie approximately on two lines.

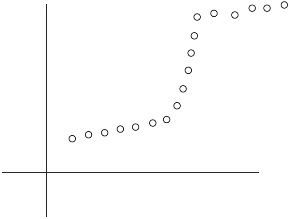
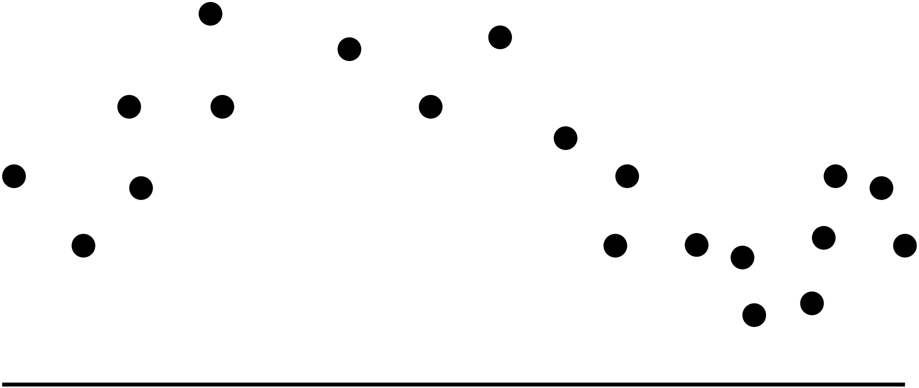


Figure 6.8 A set of points that lie approximately on three lines.

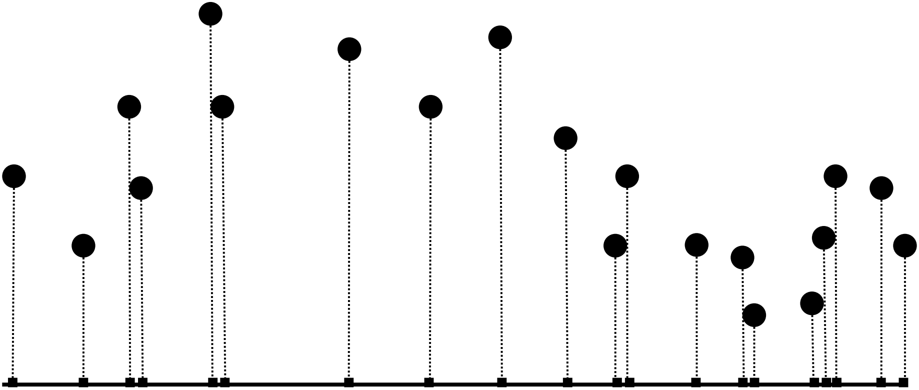
- Want to fit multiple lines through P.
- Each line must fit contiguous set of x-coordinates.
- Lines must minimise total error.

Example of Segmented Least Squares



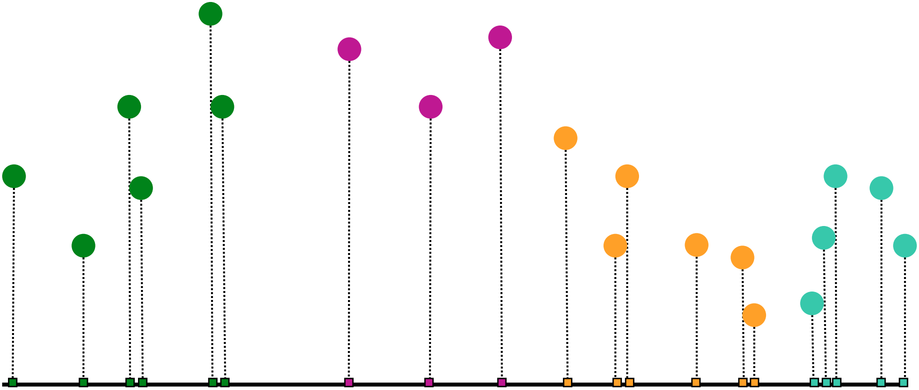
Input contains a set of two-dimensional points.

Example of Segmented Least Squares



Consider the sorted x-coordinates of the points in the input.

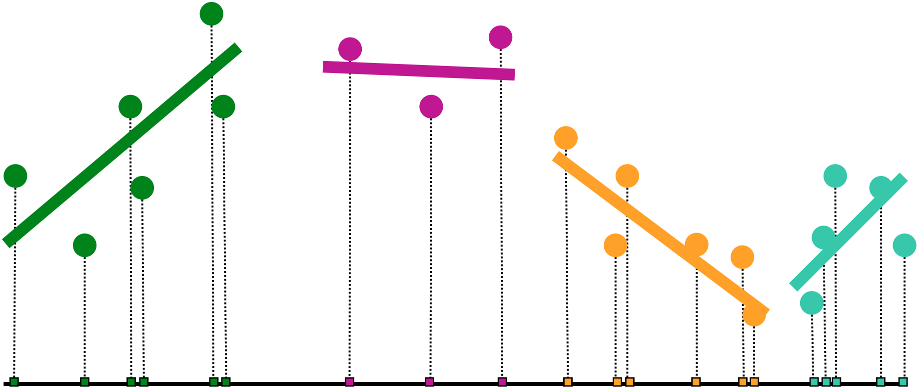
Example of Segmented Least Squares



Divide the points into segments; each *segment* contains consecutive points in the sorted order by x-coordinate.

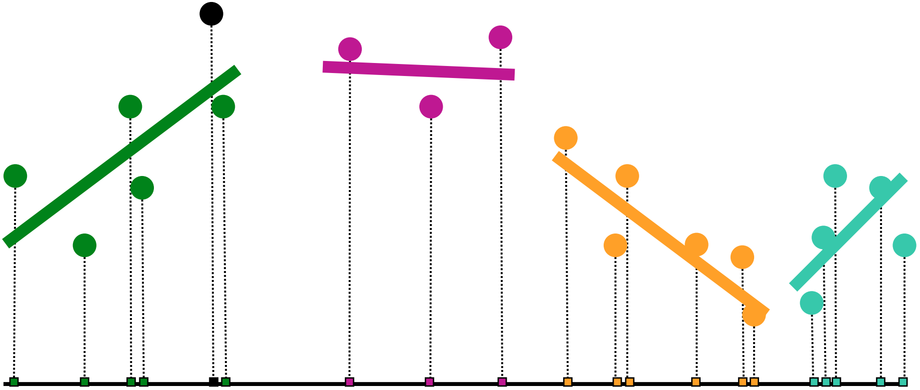
Here we are defining a meaning for "segment" that is specific to this problem.

Example of Segmented Least Squares



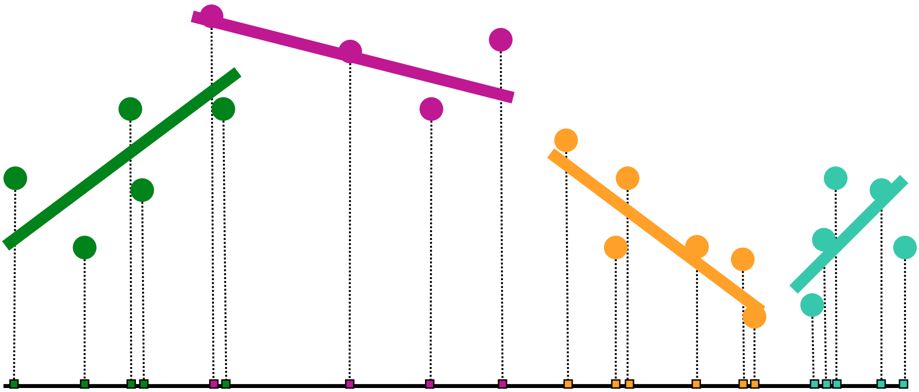
Fit the best line for each segment.

Example of Segmented Least Squares



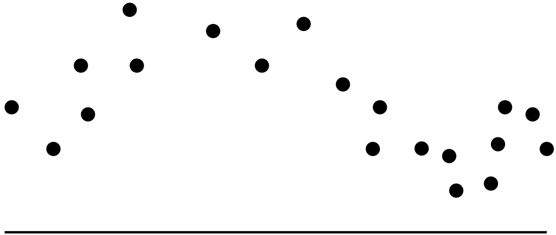
Illegal solution: black point is not in any segment.

Example of Segmented Least Squares



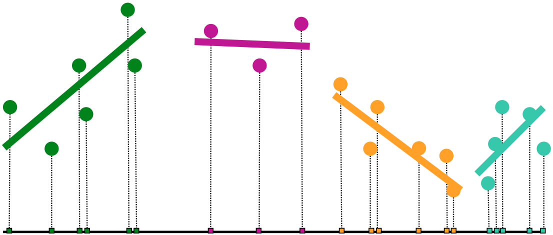
Illegal solution: leftmost purple point has x-coordinate between last two points in green segment.

Formulating Segmented Least Squares Problem



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:

Formulating Segmented Least Squares Problem



SEGMENTED LEAST SQUARES

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

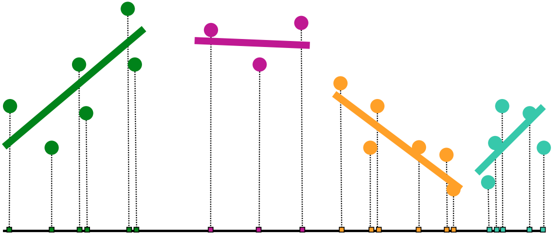
SOLUTION:

1. An integer k,
2. a partition of P into k segments $\{P_1, P_2, \ldots, P_k\}$, and
3. for each segment P_j, the best-fit line $L_j : y = a_j x + b_j, 1 \leq j \leq k$

that minimise the total error

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

Formulating Segmented Least Squares Problem



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 $C > 0$.

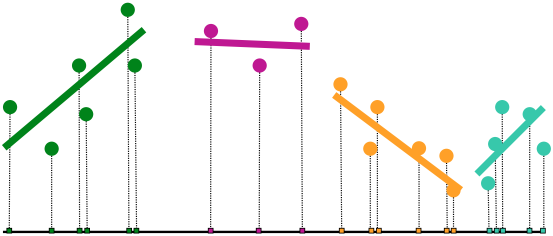
SOLUTION:

1. An integer k,
2. a partition of P into k segments $\{P_1, P_2, \ldots, P_k\}$, and
3. for each segment P_j, the best-fit line $L_j : y = a_j x + b_j, 1 \leq j \leq k$

that minimise the total error

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

Formulating Segmented Least Squares Problem



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:

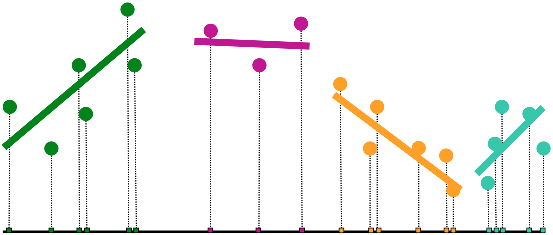
1. An integer k,
2. a partition of P into k segments $\{P_1, P_2, \ldots, P_k\}$, and
3. for each segment P_j, the best-fit line $L_j : y = a_j x + b_j, 1 \leq j \leq k$

that minimise the total error

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

- How many unknown parameters must we find? $2k$, and we must find k too!

Formulating Segmented Least Squares Problem



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:

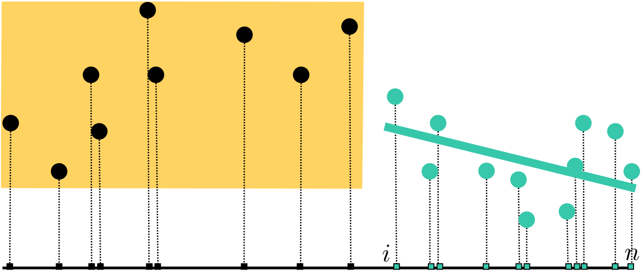
1. An integer k,
2. a partition of P into k segments $\{P_1, P_2, \ldots, P_k\}$, and
3. for each segment P_j, the best-fit line $L_j : y = a_j x + b_j, 1 \leq j \leq k$

that minimise the total error

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

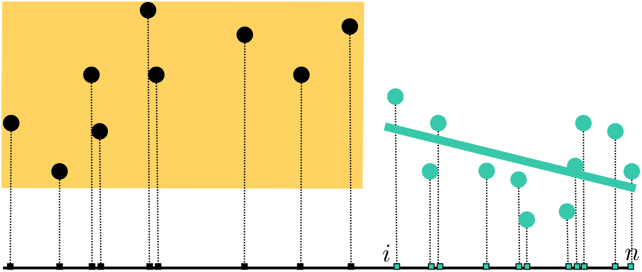
- How many unknown parameters must we find? $2k$, and we must find k too!
- Assume points in P are sorted in increasing order of x-coordinate.

Formulating the Recursion: Getting Intuition



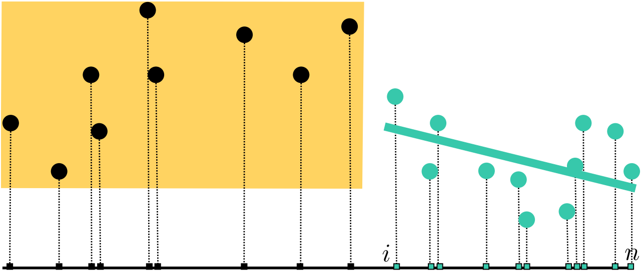
- Observation: Where does the last segment in the optimal solution end?

Formulating the Recursion: Getting Intuition



- Observation: Where does the last segment in the optimal solution end? p_n, and this segment starts at some point p_i. We don't know i yet!

Formulating the Recursion: Getting Intuition

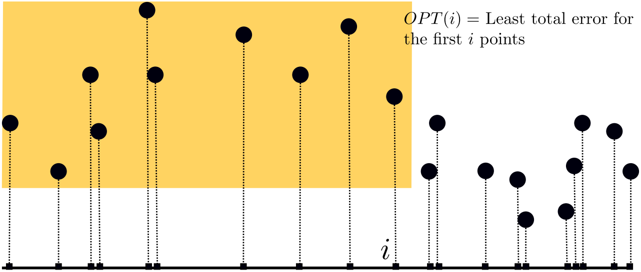


- Observation: Where does the last segment in the optimal solution end? p_n, and this segment starts at some point p_i. We don't know i yet!

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

optimal total error for n points = Error of the best line fitting $\{p_i, p_{i+1}, \ldots, p_n\}$ + C + optimal total error for the first $i - 1$ points.

Formulating the Recursion: Getting Intuition

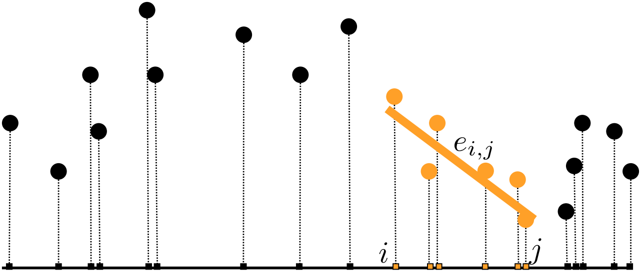


- Observation: Where does the last segment in the optimal solution end? p_n, and this segment starts at some point p_i. We don't know i yet!
- Let $OPT(i)$ be the optimal total error for the points $\{p_1, p_2, \ldots, p_i\}$.
- We want to compute OPT(n).

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

optimal total error for n points = Error of the best line fitting $\{p_i, p_{i+1}, \ldots, p_n\}$ + C + optimal total error for the first $i - 1$ points.

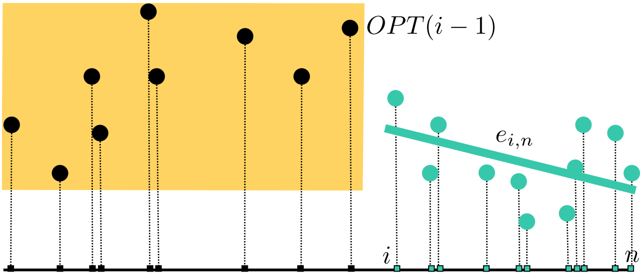
Formulating the Recursion: Getting Intuition



- Observation: Where does the last segment in the optimal solution end? p_n, and this segment starts at some point p_i. We don't know i yet!
- Let $OPT(i)$ be the optimal total error for the points $\{p_1, p_2, \ldots, p_i\}$.
- We want to compute $\text{OPT}(n)$.
- Let $e_{i,j}$ denote the minimum error of a (single) line that fits $\{p_i, p_2, \ldots, p_j\}$.
- If the last segment in the optimal partition is $\{p_i, p_{i+1}, \ldots, p_n\}$, then

optimal total error for n points = Error of the best line fitting $\{p_i, p_{i+1}, \ldots, p_n\}$ + C + optimal total error for the first $i-1$ points.

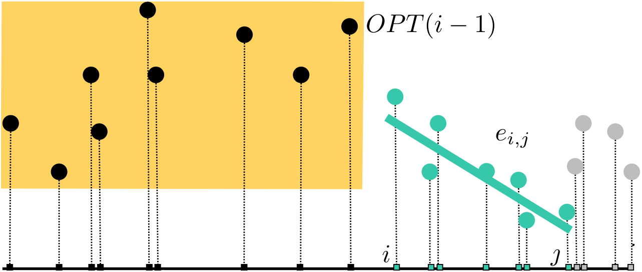
Formulating the Recursion: Getting Intuition



- Observation: Where does the last segment in the optimal solution end? p_n, and this segment starts at some point p_i. We don't know i yet!
- Let $OPT(i)$ be the optimal total error for the points $\{p_1, p_2, \ldots, p_i\}$.
- We want to compute $\text{OPT}(n)$.
- Let $e_{i,j}$ denote the minimum error of a (single) line that fits $\{p_i, p_2, \ldots, p_j\}$.
- If the last segment in the optimal partition is $\{p_i, p_{i+1}, \ldots, p_n\}$, then

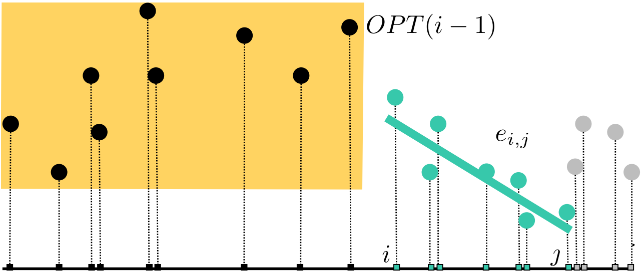
$$\text{OPT}(n) = e_{i,n} + C + \text{OPT}(i-1)$$

Formulating the Full Recursion



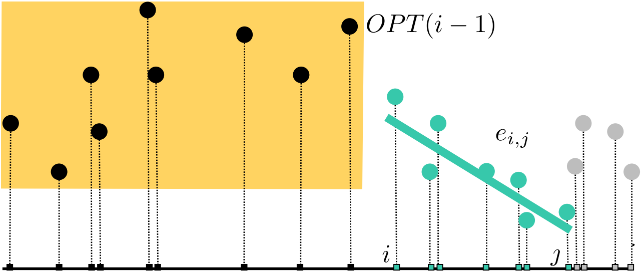
- In general, we want to solve sub-problem on the points $\{p_1, p_2, \ldots p_j\}$, i.e., we want to compute $\text{OPT}(j)$, where j lies between 1 and n.

Formulating the Full Recursion



- In general, we want to solve sub-problem on the points $\{p_1, p_2, \ldots p_j\}$, i.e., we want to compute $\text{OPT}(j)$, where j lies between 1 and n.
- If the last segment in the optimal partition is $\{p_i, p_{i+1}, \ldots, p_j\}$, then optimal total error for first j points $=$ Error of the best line fitting $\{p_i, p_{i+1}, \ldots, p_j\}$ $+ C +$ optimal total error for the first $i-1$ points.

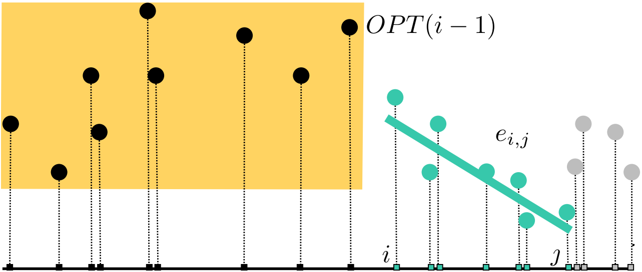
Formulating the Full Recursion



- In general, we want to solve sub-problem on the points $\{p_1, p_2, \ldots p_j\}$, i.e., we want to compute $\text{OPT}(j)$, where j lies between 1 and n.
- If the last segment in the optimal partition is $\{p_i, p_{i+1}, \ldots, p_j\}$, then

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

Formulating the Full Recursion

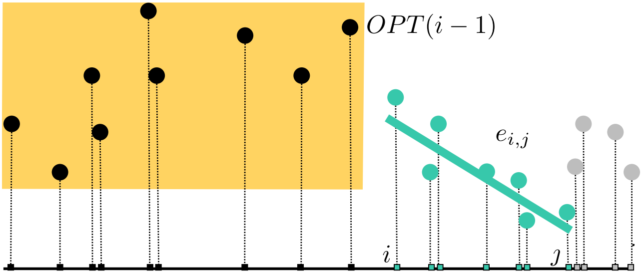


- In general, we want to solve sub-problem on the points $\{p_1, p_2, \ldots p_j\}$, i.e., we want to compute $\text{OPT}(j)$, where j lies between 1 and n.
- If the last segment in the optimal partition is $\{p_i, p_{i+1}, \ldots, p_j\}$, then

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

- We don't know i! Dynamic Programming: Segmented Least Squares: First j points

Formulating the Full Recursion



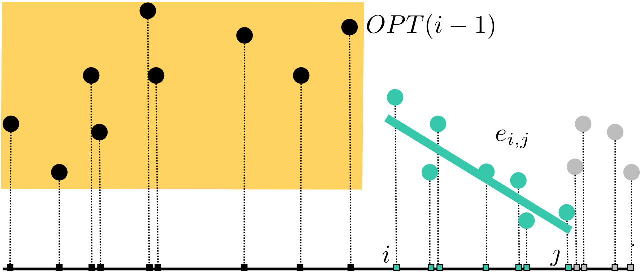
- In general, we want to solve sub-problem on the points $\{p_1, p_2, \ldots p_j\}$, i.e., we want to compute OPT(j), where j lies between 1 and n.
- If the last segment in the optimal partition is $\{p_i, p_{i+1}, \ldots, p_j\}$, then

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

- We don't know i! But i can take only j distinct values: $1, 2, \ldots, j-1, j$. Therefore,

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

Formulating the Full Recursion



- In general, we want to solve sub-problem on the points $\{p_1, p_2, \dots p_j\}$, i.e., we want to compute OPT(j), where j lies between 1 and n.
- If the last segment in the optimal partition is $\{p_i, p_{i+1}, \dots, p_j\}$, then

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

- We don't know i! But i can take only j distinct values: $1, 2, \dots, j-1, j$. Therefore,

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

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

Dynamic Programming Algorithm

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

```
Segmented-Least-Squares(n)
  Array M[0...n]
  Set M[0] = 0
  For all pairs i ≤ j
    Compute the least squares error e_{i,j} for the segment p_i, ..., p_j
  Endfor
  For j = 1, 2, ..., n
    Use the recurrence (6.7) to compute M[j]
  Endfor
  Return M[n]
```

Dynamic Programming Algorithm

$$\mathrm{OPT}(j) = \min_{1 \le i \le j} \left(e_{i,j} + C + \mathrm{OPT}(i-1) \right)$$

```
Segmented-Least-Squares(n)
  Array M[0...n]
  Set M[0] = 0
  For all pairs i ≤ j
    Compute the least squares error e_{i,j} for the segment p_i, ..., p_j
  Endfor
  For j = 1, 2, ..., n
    Use the recurrence (6.7) to compute M[j]
  Endfor
  Return M[n]
```

- We can find the segments in the optimal solution by backtracking.

Running Time

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

```
Segmented-Least-Squares(n)
  Array M[0 ... n]
  Set M[0] = 0
  For all pairs i ≤ j
    Compute the least squares error e_{i,j} for the segment p_i, ..., p_j
  Endfor
  For j = 1, 2, ..., n
    Use the recurrence (6.7) to compute M[j]
  Endfor
  Return M[n]
```

- Let $T(n)$ be the running time of this algorithm.

$$T(n) = \sum_{1 \le j \le n} \sum_{1 \le i \le j} O(j - i) =$$

Running Time

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

```
Segmented-Least-Squares(n)
  Array M[0...n]
  Set M[0] = 0
  For all pairs i ≤ j
    Compute the least squares error e_{i,j} for the segment p_i, ..., p_j
  Endfor
  For j = 1, 2, ..., n
    Use the recurrence (6.7) to compute M[j]
  Endfor
  Return M[n]
```

- Let $T(n)$ be the running time of this algorithm.

▸ Dynamic Programming: Segmented Least Squares: Running Time Part 1

$$T(n) = \sum_{1 \le j \le n} \sum_{1 \le i \le j} O(j - i) = ?$$

Running Time

$$\mathrm{OPT}(j) = \min_{1 \le i \le j} \left(e_{i,j} + C + \mathrm{OPT}(i-1)\right)$$

```
Segmented-Least-Squares(n)
  Array M[0...n]
  Set M[0]=0
  For all pairs i≤j
    Compute the least squares error e_{i,j} for the segment p_i,...,p_j
  Endfor
  For j=1,2,...,n
    Use the recurrence (6.7) to compute M[j]
  Endfor
  Return M[n]
```

- Let $T(n)$ be the running time of this algorithm.

▸ Dynamic Programming: Segmented Least Squares: Running Time Part 2

$$T(n) = \sum_{1 \le j \le n} \sum_{1 \le i \le j} O(j-i) = O(n^3)$$

Running Time

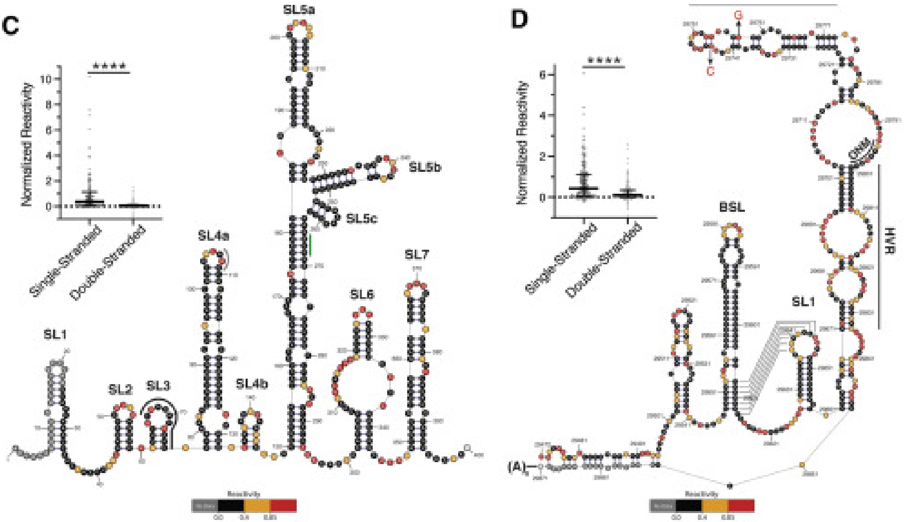
$$\text{OPT}(j) = \min_{1 \le i \le j} \left(e_{i,j} + C + \text{OPT}(i-1) \right)$$

```
Segmented-Least-Squares(n)
  Array M[0...n]
  Set M[0]=0
  For all pairs i≤j
    Compute the least squares error e_{i,j} for the segment p_i,...,p_j
  Endfor
  For j=1,2,...,n
    Use the recurrence (6.7) to compute M[j]
  Endfor
  Return M[n]
```

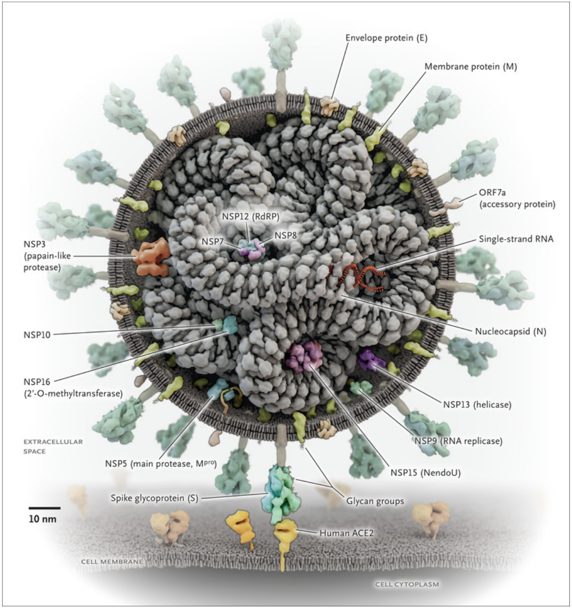
- Let $T(n)$ be the running time of this algorithm.

$$T(n) = \sum_{1 \le j \le n} \sum_{1 \le i \le j} O(j-i) = O(n^3)$$

- Running time is $O(n^3)$; can be improved to $O(n^2)$.



Dynamic Programming: Image


Envelope protein (E)
Membrane protein (M)
ORF7a
(accessory protein)
Single-strand RNA
NSP12 (RdRP)
NSP8
NSP7
NSP3
(papain-like
protease)
Nucleocapsid (N)
NSP10
NSP16
(2'-O-methyltransferase)
NSP13 (helicase)
NSP9 (RNA replicase)
EXTRACELLULAR
SPACE
NSP5 (main protease, Mpro)
NSP15 (NendoU)
Spike glycoprotein (S)
Glycan groups
10 nm
Human ACE2
CELL MEMBRANE
CELL CYTOPLASM

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 ≤ 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”.

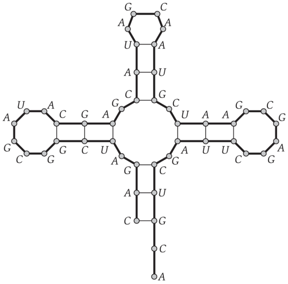


Figure 6.13 An RNA secondary structure. Thick lines connect adjacent elements of the sequence; thin lines indicate pairs of elements that are matched.

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 ≤ 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”.

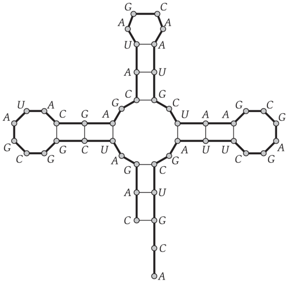


Figure 6.13 An RNA secondary structure. Thick lines connect adjacent elements of the sequence; thin lines indicate pairs of elements that are matched.

- 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 ≤ 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".

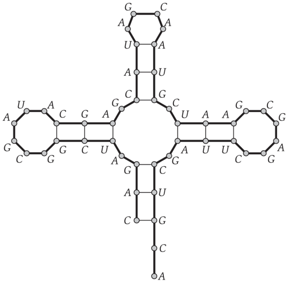


Figure 6.13 An RNA secondary structure. Thick lines connect adjacent elements of the sequence; thin lines indicate pairs of elements that are matched.

- 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

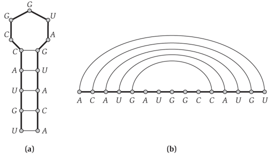


Figure 6.14 Two views of an RNA secondary structure. In the second view, (b), the string has been "stretched" lengthwise, and edges connecting matched pairs appear as noncrossing "bubbles" over the string.

- An *RNA molecule* is a string $B = b_1 b_2 \dots b_n$; 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

Formulating the Problem

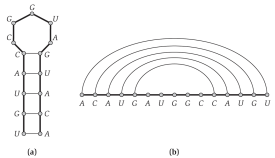
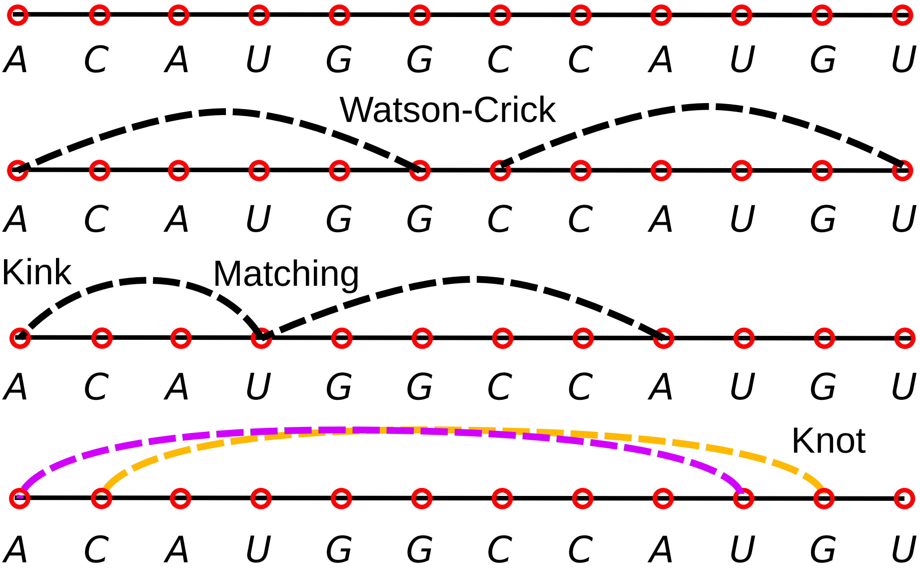


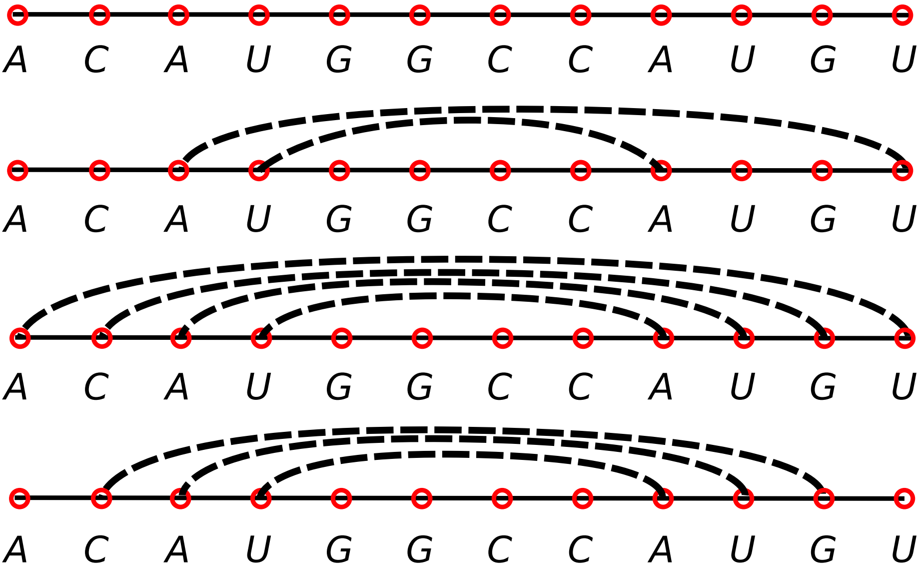
Figure 6.14 Two views of an RNA secondary structure. In the second view, (b), the string has been "stretched" lengthwise, and edges connecting matched pairs appear as noncrossing "bubbles" over the string.

- An *RNA molecule* is a string $B = b_1 b_2 \dots b_n$; 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



Legal Secondary Structures



Dynamic Programming Approach

- $OPT(j)$ is the maximum number of base pairs in a secondary structure for $b_1 b_2 \dots b_j$. ▸ Dynamic Programming: RNA Secondary Structure: Base cases 1

Dynamic Programming Approach

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

Dynamic Programming Approach

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

Dynamic Programming Approach

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

Dynamic Programming Approach

- $OPT(j)$ is the maximum number of base pairs in a secondary structure for $b_1 b_2 \dots b_j$. $\text{OPT}(j) = 0$, if $j \leq 5$.
- In the optimal secondary structure on $b_1 b_2 \dots b_j$
  1. if j is not a member of any pair, use $\text{OPT}(j-1)$.
  2. if j pairs with some $t < j - 4$,

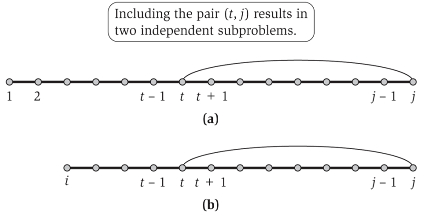


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

Dynamic Programming Approach

- *OPT*(j) is the maximum number of base pairs in a secondary structure for $b_1 b_2 \ldots b_j$. $\text{OPT}(j) = 0$, 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 $\text{OPT}(j-1)$.
  2. if j pairs with some $t < j - 4$, knot condition yields two independent sub-problems! ▸ Dynamic Programming: RNA Secondary Structure: Subproblems 1

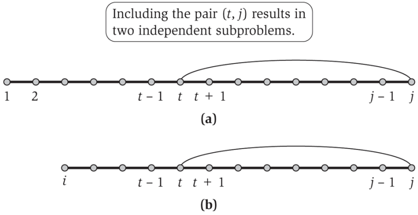


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

Dynamic Programming Approach

- *OPT*(j) is the maximum number of base pairs in a secondary structure for $b_1 b_2 \ldots b_j$. $\text{OPT}(j) = 0$, 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 $\text{OPT}(j-1)$.
  2. if j pairs with some $t < j - 4$, knot condition yields two independent sub-problems! $\text{OPT}(t-1)$ and ???

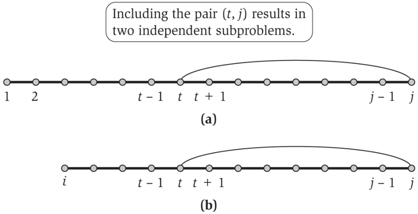


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

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.

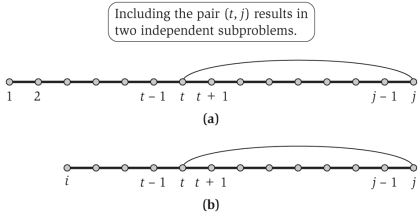


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

Correct Dynamic Programming Approach

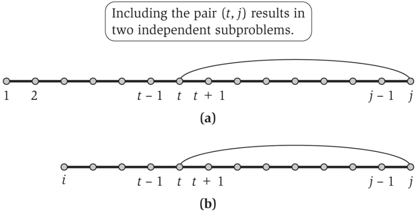


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

- $OPT(i, j)$ is the maximum number of base pairs in a secondary structure for $b_i b_{i+1} \ldots b_j$. ▸ Dynamic Programming: RNA Secondary Structure: Base cases 2

Correct Dynamic Programming Approach

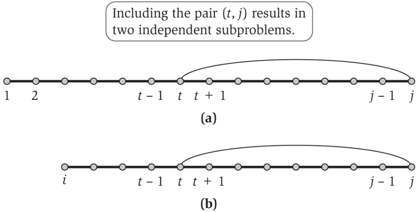


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

- $OPT(i, j)$ is the maximum number of base pairs in a secondary structure for $b_i b_{i+1} \ldots b_j$. $\text{OPT}(i, j) = 0$, if $i \geq j - 4$.

Correct Dynamic Programming Approach

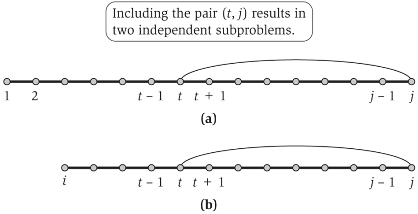


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

- $OPT(i, j)$ is the maximum number of base pairs in a secondary structure for $b_i b_{i+1} \dots b_j$. $\text{OPT}(i, j) = 0$, if $i \geq j - 4$.
- In the optimal secondary structure on $b_i b_{i+1} \dots b_j$

$$\text{OPT}(i, j) = \max \left(\qquad \right)$$

Correct Dynamic Programming Approach

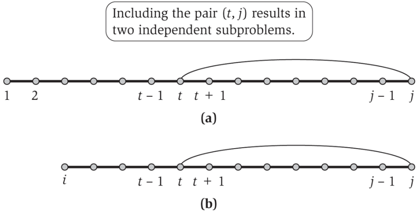


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

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

$$\text{OPT}(i, j) = \max\left(\text{OPT}(i, j-1), \qquad\qquad\qquad\qquad\qquad\qquad\right)$$

Correct Dynamic Programming Approach

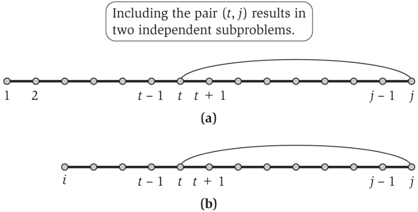


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

- $OPT(i, j)$ is the maximum number of base pairs in a secondary structure for $b_i b_{i+1} \ldots b_j$. $\text{OPT}(i, j) = 0$, if $i \geq j - 4$.
- In the optimal secondary structure on $b_i b_{i+1} \ldots b_j$
  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

▸ Dynamic Programming: RNA Secondary Structure: Subproblems 2

$$\text{OPT}(i, j) = \max\left(\text{OPT}(i, j-1), \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\right)$$

Correct Dynamic Programming Approach

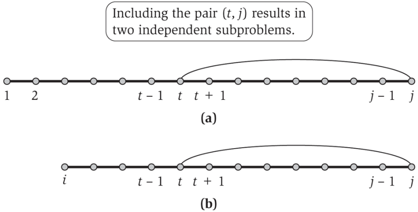


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

- $OPT(i, j)$ is the maximum number of base pairs in a secondary structure for $b_i b_{i+1} \dots b_j$. $\text{OPT}(i, j) = 0$, if $i \geq j - 4$.
- In the optimal secondary structure on $b_i b_{i+1} \dots b_j$
  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 $\text{OPT}(t + 1, j - 1)$.

$$\text{OPT}(i, j) = \max\left(\text{OPT}(i, j - 1), \qquad\qquad\qquad\qquad\qquad\qquad\right)$$

Correct Dynamic Programming Approach

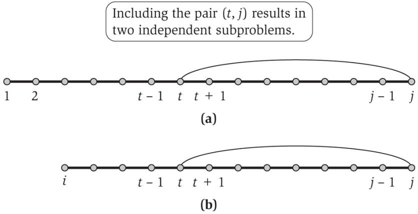


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

- $OPT(i,j)$ is the maximum number of base pairs in a secondary structure for $b_i b_{i+1} \ldots b_j$. $\text{OPT}(i,j) = 0$, if $i \geq j-4$.
- In the optimal secondary structure on $b_i b_{i+1} \ldots b_j$
  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

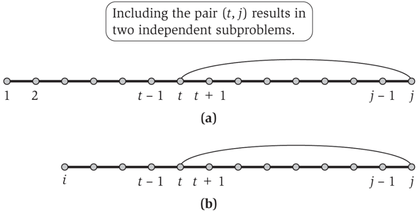


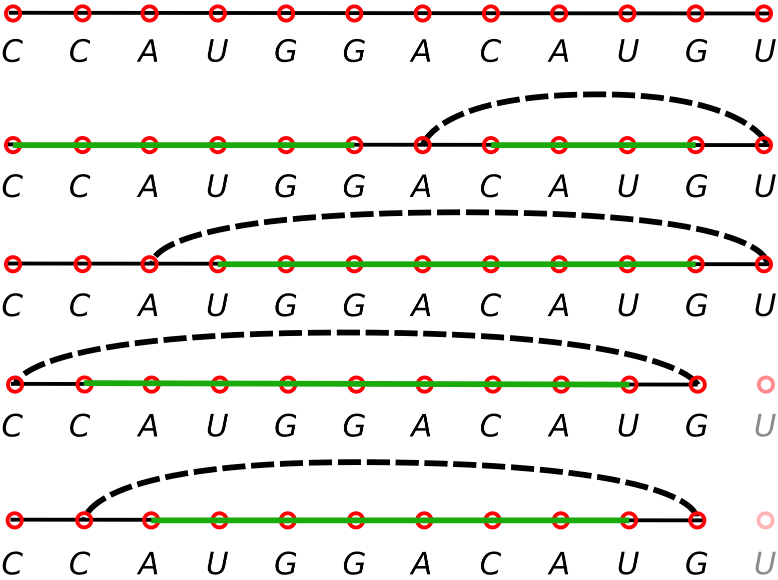
Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

- $OPT(i,j)$ is the maximum number of base pairs in a secondary structure for $b_i b_{i+1} \ldots b_j$. $\text{OPT}(i,j) = 0$, if $i \geq j - 4$.
- In the optimal secondary structure on $b_i b_{i+1} \ldots b_j$
    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 $j - 5$,

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

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

Example of Dynamic Programming Algorithm



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 ▸ Dynamic Programming: RNA Secondary Structure: Number of sub-problems sub-problems.

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"?

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"?
- Computing $\text{OPT}(i,j)$ involves sub-problems of the form $\text{OPT}(l,j-1)$.
- We should compute OPT() values in increasing order of the second argument.

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"?
- Computing $\text{OPT}(i,j)$ involves sub-problems of the form $\text{OPT}(l,j-1)$.
- We should compute OPT() values in increasing order of the second argument.

Initialise OPT(i,j)= 0 for every l,j such that $i \geq j-4$
for $j = 1, 2, \ldots, n-1, n$
    for $i = 1, 2, \ldots, j-6, j-5$
        Compute OPT(i,j) using the recurrence above.

- How long does it take to compute $\text{OPT}(i,j)$?

▸ Dynamic Programming: RNA Secondary Structure: Running time

- What is the running time of the algorithm?   .

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"?
- Computing $\text{OPT}(i,j)$ involves sub-problems of the form $\text{OPT}(l, j-1)$.
- We should compute OPT() values in increasing order of the second argument.

```
Initialise OPT(i,j)= 0 for every l,j such that i ≥ j − 4
for j = 1, 2, ..., n − 1, n
    for i = 1, 2, ..., j − 6, j − 5
        Compute OPT(i,j) using the recurrence above.
```

- How long does it take to compute $\text{OPT}(i,j)$? $O(j-i)$
- What is the running time of the algorithm? $O(n^3)$.

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 from source s to destination t with minimum total cost.
  2. Does G have a *negative cycle*? Application is to arbitrage opportunities.

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 from source s to destination t with minimum total cost.
  2. Does G have a *negative cycle*? Application is to arbitrage opportunities.

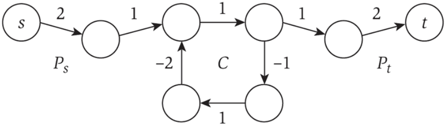


Figure 6.20 In this graph, one can find s-t paths of arbitrarily negative cost (by going around the cycle C many times).

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 from source s to destination t with minimum total cost.
  2. Does G have a *negative cycle*? Application is to arbitrage opportunities.

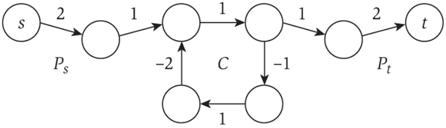


Figure 6.20 In this graph, one can find s-t paths of arbitrarily negative cost (by going around the cycle C many times).

Approaches for Shortest Path Algorithm

1. Run Dijsktra's algorithm.
   - Dynamic Programming: Shortest Paths: Example Graph (a)

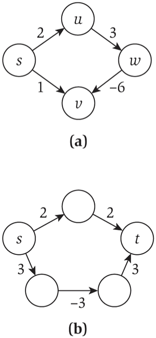


Figure 6.21 (a) With negative edge costs, Dijkstra's Algorithm can give the wrong answer for the Shortest-Path Problem. (b) Adding 3 to the cost of each edge will make all edges nonnegative, but it will change the identity of the shortest *s*-*t* path.

Approaches for Shortest Path Algorithm

1. Run Dijsktra's algorithm.
   Computes incorrect answers because it is greedy.

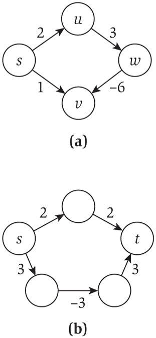


Figure 6.21 (a) With negative edge costs, Dijkstra's Algorithm can give the wrong answer for the Shortest-Path Problem. (b) Adding 3 to the cost of each edge will make all edges nonnegative, but it will change the identity of the shortest *s*-*t* path.

Approaches for Shortest Path Algorithm

1. Run Dijsktra's algorithm.
   Computes incorrect answers because it is greedy.
2. Add some large constant to each edge.

   ▸ Dynamic Programming: Shortest Paths: Example Graph (b)

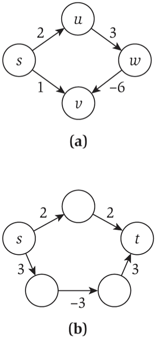


Figure 6.21 (a) With negative edge costs, Dijkstra's Algorithm can give the wrong answer for the Shortest-Path Problem. (b) Adding 3 to the cost of each edge will make all edges nonnegative, but it will change the identity of the shortest *s*-*t* path.

Approaches for Shortest Path Algorithm

1. Run Dijsktra'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.

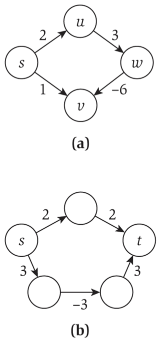


Figure 6.21 (a) With negative edge costs, Dijkstra's Algorithm can give the wrong answer for the Shortest-Path Problem. (b) Adding 3 to the cost of each edge will make all edges nonnegative, but it will change the identity of the shortest *s*-*t* path.

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: Shortest Paths: "Simple" Proof

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?

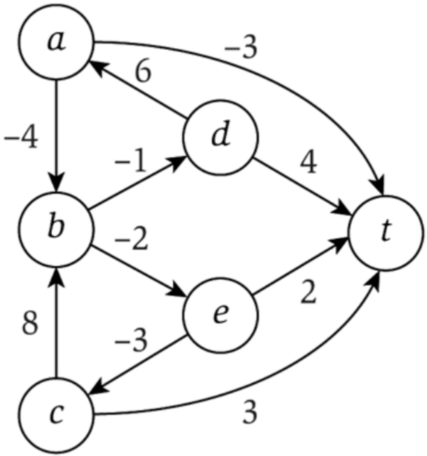
Dynamic Programming Approach

- 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 $\text{OPT}(n-1, s)$.

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 $\text{OPT}(n-1, s)$.

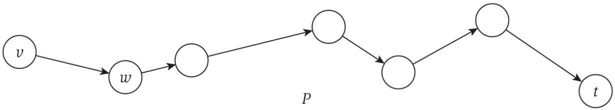


Figure 6.22 The minimum-cost path P from v to t using at most i edges.

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

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 $\text{OPT}(n-1, s)$.

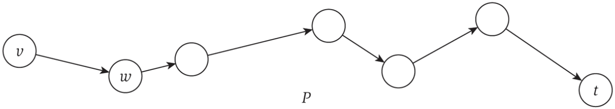


Figure 6.22 The minimum-cost path P from v to t using at most i edges.

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

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 $\text{OPT}(n-1, s)$.

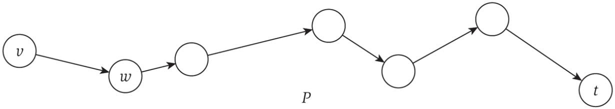


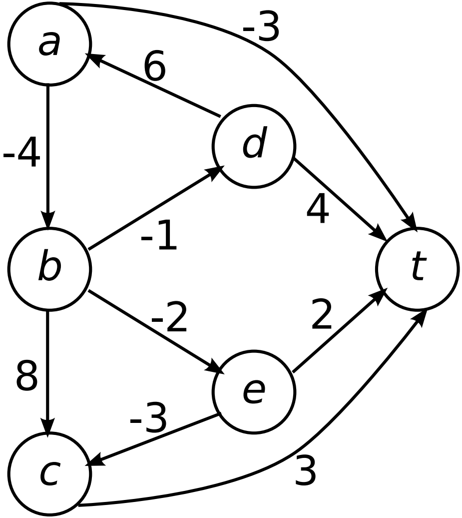
Figure 6.22 The minimum-cost path P from v to t using at most i edges.

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

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

Example of Dynamic Programming Recursion

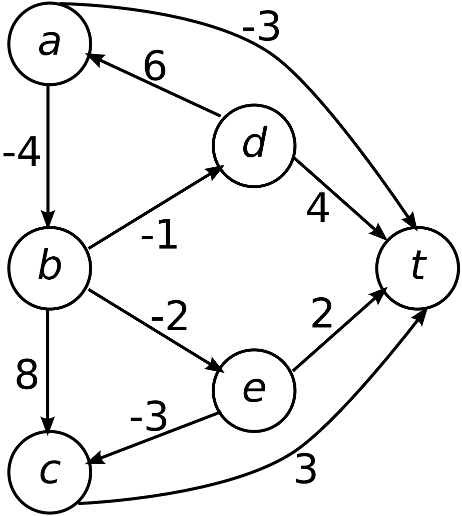
$$\mathrm{OPT}(i, v) = \min\left(\mathrm{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \mathrm{OPT}(i-1, w)\right)\right)$$



	0	1	2	3	4	5
t						
a						
b						
c						
d						
e						

Example of Dynamic Programming Recursion

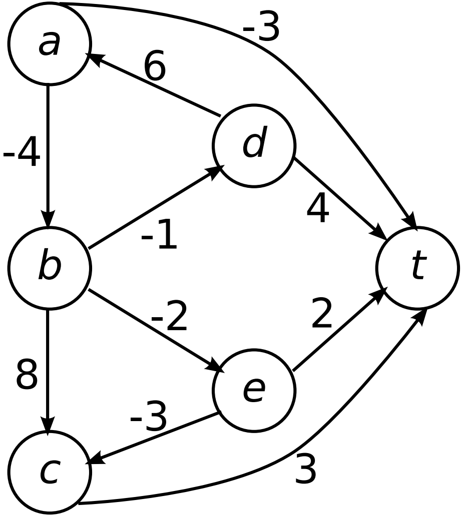
$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$



	0	1	2	3	4	5
t						
a						
b						
c						
d						
e						

Example of Dynamic Programming Recursion

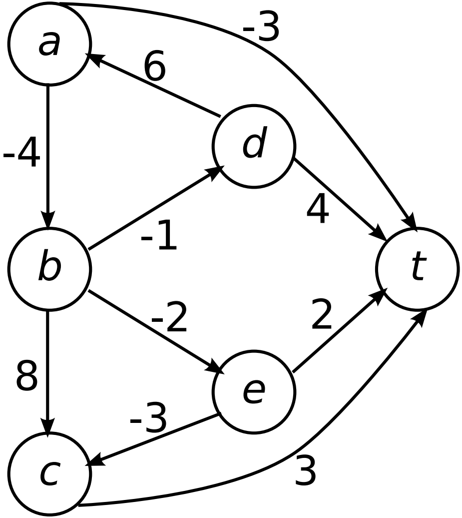
$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞					
b	∞					
c	∞					
d	∞					
e	∞					

Example of Dynamic Programming Recursion

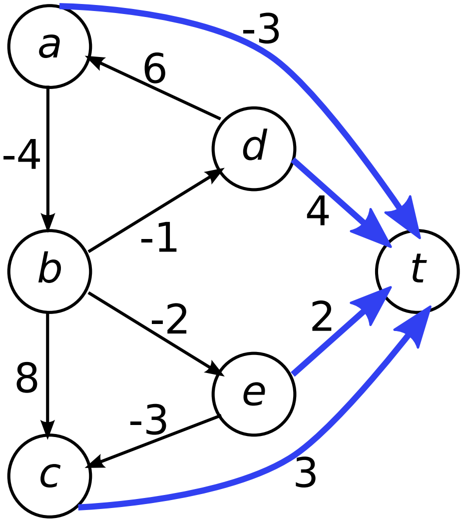
$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞					
b	∞					
c	∞					
d	∞					
e	∞					

Example of Dynamic Programming Recursion

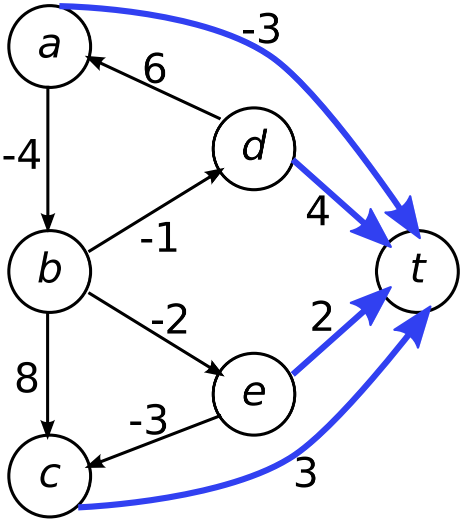
$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3				
b	∞	∞				
c	∞	3				
d	∞	4				
e	∞	2				

Example of Dynamic Programming Recursion

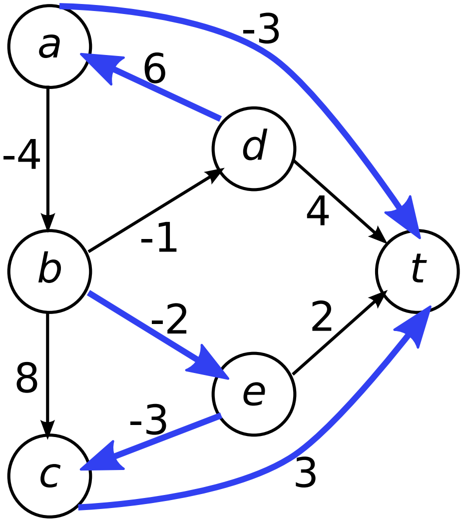
$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3				
b	∞	∞				
c	∞	3				
d	∞	4				
e	∞	2				

Example of Dynamic Programming Recursion

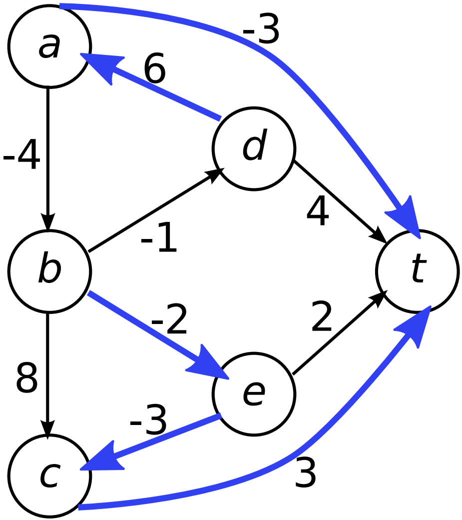
$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3			
b	∞	∞	0			
c	∞	3	3			
d	∞	4	3			
e	∞	2	0			

Example of Dynamic Programming Recursion

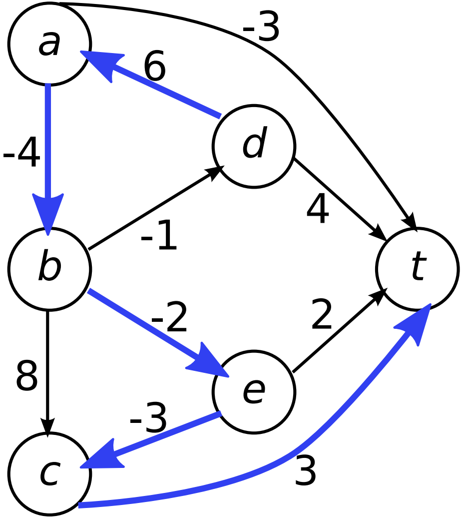
$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3			
b	∞	∞	0			
c	∞	3	3			
d	∞	4	3			
e	∞	2	0			

Example of Dynamic Programming Recursion

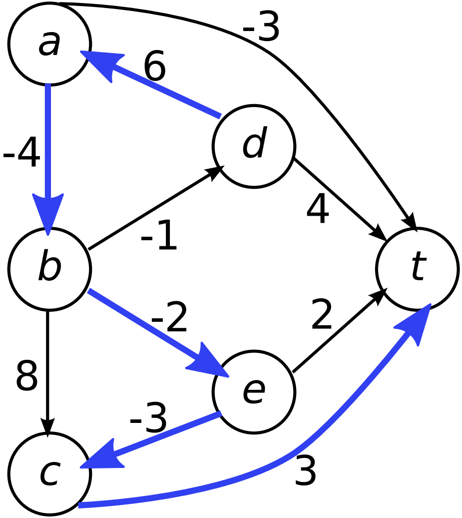
$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4		
b	∞	∞	0	-2		
c	∞	3	3	3		
d	∞	4	3	3		
e	∞	2	0	0		

Example of Dynamic Programming Recursion

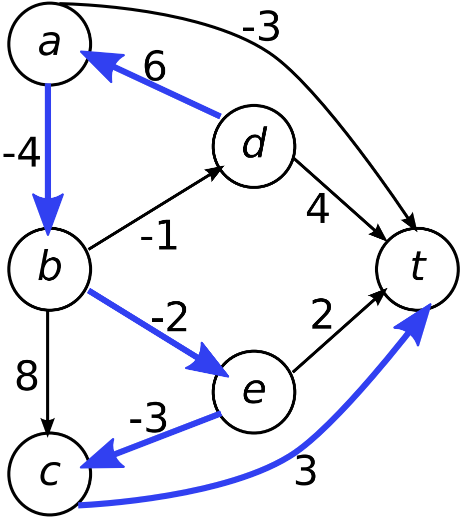
$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4		
b	∞	∞	0	-2		
c	∞	3	3	3		
d	∞	4	3	3		
e	∞	2	0	0		

Example of Dynamic Programming Recursion

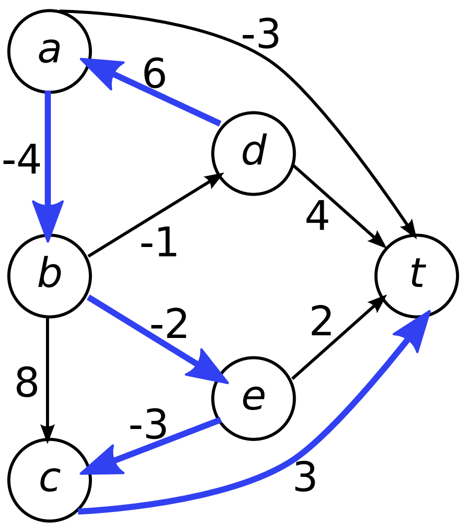
$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	
b	∞	∞	0	-2	-2	
c	∞	3	3	3	3	
d	∞	4	3	3	2	
e	∞	2	0	0	0	

Example of Dynamic Programming Recursion

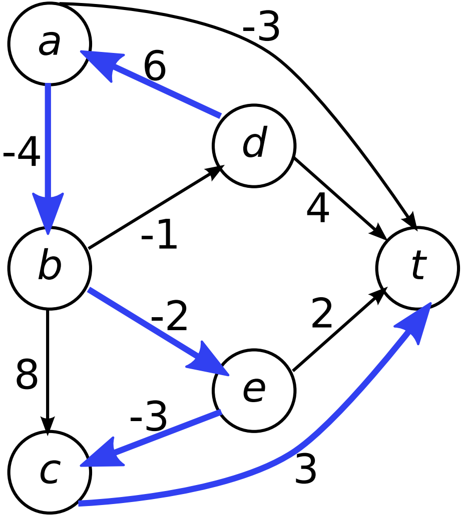
$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	
b	∞	∞	0	-2	-2	
c	∞	3	3	3	3	
d	∞	4	3	3	2	
e	∞	2	0	0	0	

Example of Dynamic Programming Recursion

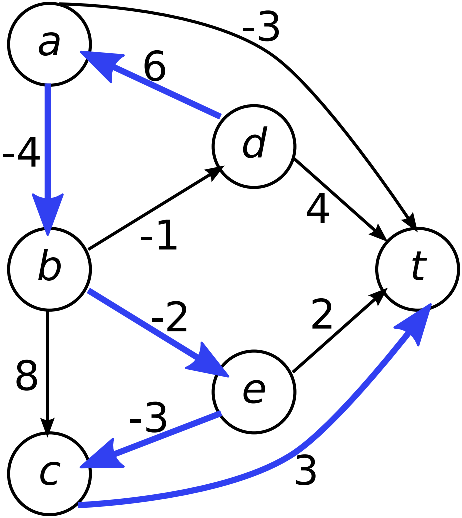
$$\mathrm{OPT}(i, v) = \min\left(\mathrm{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \mathrm{OPT}(i-1, w)\right)\right)$$



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

Example of Dynamic Programming Recursion

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



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

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

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

Alternate Dynamic Programming Formulation

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

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

- Let P be the optimal path whose cost is $\mathrm{OPT}_=(\mathrm{i}, \mathrm{v})$.

Alternate Dynamic Programming Formulation

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

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

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

Alternate Dynamic Programming Formulation

- *$OPT_=(i, v)$*: minimum cost of a v-t path that uses exactly i edges. Goal is to compute
$$\min_{i=1}^{n-1} \mathrm{OPT}_=(\mathrm{i, s}).$$
- Let P be the optimal path whose cost is $\mathrm{OPT}_=(\mathrm{i, v})$.
  - If first node on P is w, then $\mathrm{OPT}_=(\mathrm{i, v}) = c_{vw} + \mathrm{OPT}_=(\mathrm{i - 1, w})$.
$$\mathrm{OPT}_=(\mathrm{i, v}) = \min_{w \in V} \left(c_{vw} + \mathrm{OPT}_=(\mathrm{i - 1, w}) \right)$$

Alternate Dynamic Programming Formulation

- *$OPT_{=}(i, v)$*: minimum cost of a v-t path that uses exactly i edges. Goal is to compute

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

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

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

- Compare the two desired solutions:

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

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

Bellman-Ford Algorithm

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

```
Shortest-Path(G, s, t)
  n = number of nodes in G
  Array M[0 ... n − 1, V]
  Define M[0, t] = 0 and M[0, v] = ∞ for all other v ∈ V
  For i = 1, ..., n − 1
    For v ∈ V in any order
      Compute M[i, v] using the recurrence (6.23)
    Endfor
  Endfor
  Return M[n − 1, s]
```

Bellman-Ford Algorithm

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

```
Shortest-Path(G, s, t)
  n = number of nodes in G
  Array M[0...n-1, V]
  Define M[0,t]=0 and M[0,v]=∞ for all other v ∈ V
  For i=1,...,n-1
    For v ∈ V in any order
      Compute M[i,v] using the recurrence (6.23)
    Endfor
  Endfor
  Return M[n-1, s]
```

▸ Dynamic Programming: Shortest Paths: Space and Running Time

- Space used is ??. Running time is ??.

Bellman-Ford Algorithm

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

```
Shortest-Path(G, s, t)
  n = number of nodes in G
  Array M[0...n−1, V]
  Define M[0, t] = 0 and M[0, v] = ∞ for all other v ∈ V
  For i = 1, ..., n−1
    For v ∈ 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?

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

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?

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

- w only needs to range over outgoing neighbours N_v of v.
- If $n_v = |N_v|$ is the number of outgoing 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?

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

- w only needs to range over outgoing neighbours N_v of v.
- If $n_v = |N_v|$ is the number of outgoing 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

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

- The algorithm uses $O(n^2)$ space to store the array M.

Improving the Memory Requirements

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

- 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

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

- 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

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

Improving the Memory Requirements

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

- 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

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

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

Computing the Shortest Path: Algorithm

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

- How can we recover the shortest path that has cost $M[v]$?

Computing the Shortest Path: Algorithm

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

- How can we recover the shortest path that has cost $M[v]$?
- For each node v, compute and update $f(v)$, the first node after v in the current shortest path from v to t.
- Updating $f(v)$:

Computing the Shortest Path: Algorithm

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

- How can we recover the shortest path that has cost $M[v]$?
- For each node v, compute and update $f(v)$, the first node after v in the current shortest path from v to t.
- Updating $f(v)$: If x is the node that attains the minimum in $\min_{w \in N_v}\left(c_{vw} + M'[w]\right)$ and $M'[v] > c_{vx} + M'[x]$,

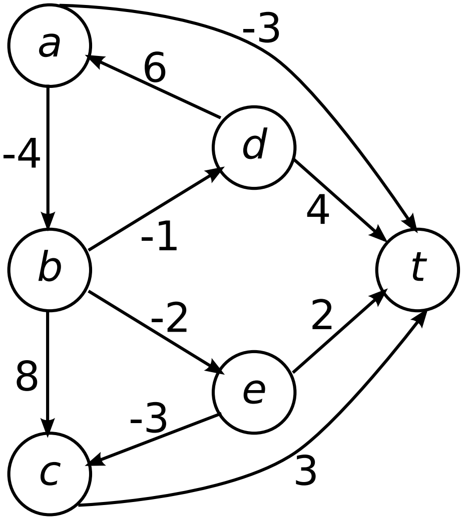
Computing the Shortest Path: Algorithm

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

- How can we recover the shortest path that has cost $M[v]$?
- For each node v, compute and update $f(v)$, the first node after v in the current shortest path from v to t.
- Updating $f(v)$: If x is the node that attains the minimum in $\min_{w \in N_v}\left(c_{vw} + M'[w]\right)$ and $M'[v] > c_{vx} + M'[x]$, then
  - set $M[v] = c_{vx} + M'[x]$ and
  - set $f(v) = x$.
- At the end, follow $f(v)$ pointers from s to t (and hope for the best).

Example of Maintaining Pointers

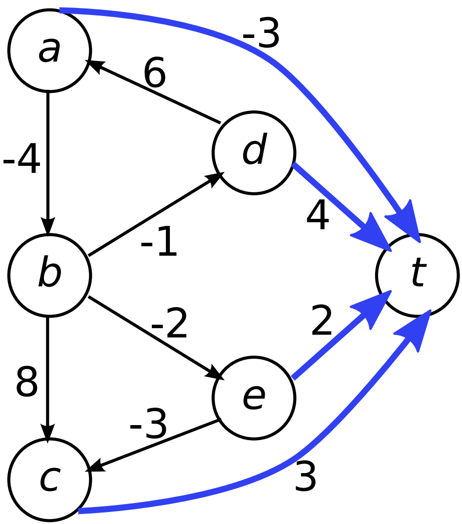
$$M[v] = \min\left(M'[v], \min_{w \in N_v}\left(c_{vw} + M'[w]\right)\right)$$



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞					
b	∞					
c	∞					
d	∞					
e	∞					

Example of Maintaining Pointers

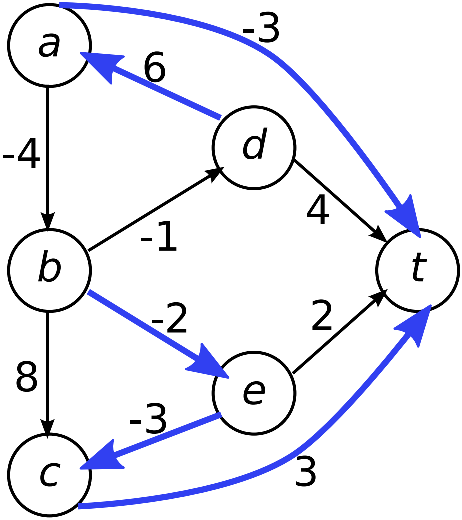
$$M[v] = \min\left(M'[v], \min_{w \in N_v}\left(c_{vw} + M'[w]\right)\right)$$



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3				
b	∞	∞				
c	∞	3				
d	∞	4				
e	∞	2				

Example of Maintaining Pointers

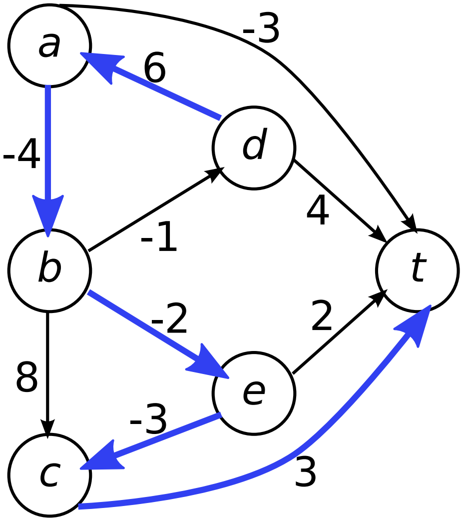
$$M[v] = \min \left(M'[v], \min_{w \in N_v} \left(c_{vw} + M'[w] \right) \right)$$



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3			
b	∞	∞	0			
c	∞	3	3			
d	∞	4	3			
e	∞	2	0			

Example of Maintaining Pointers

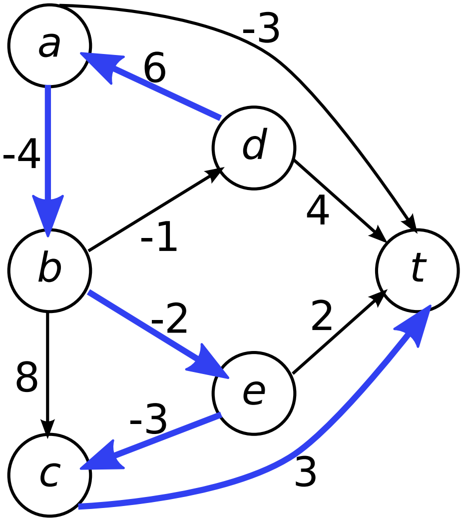
$$M[v] = \min\left(M'[v], \min_{w \in N_v}\left(c_{vw} + M'[w]\right)\right)$$



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4		
b	∞	∞	0	-2		
c	∞	3	3	3		
d	∞	4	3	3		
e	∞	2	0	0		

Example of Maintaining Pointers

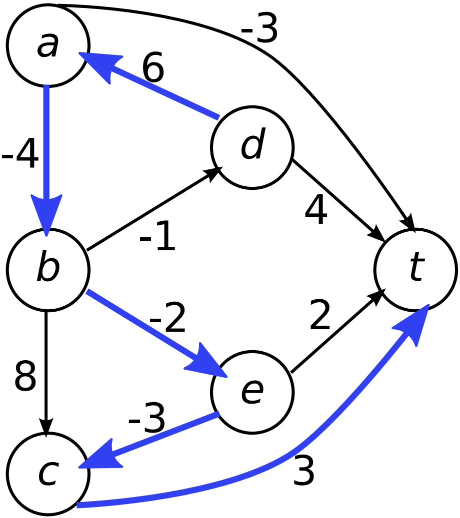
$$M[v] = \min\left(M'[v], \min_{w \in N_v}\left(c_{vw} + M'[w]\right)\right)$$



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	
b	∞	∞	0	-2	-2	
c	∞	3	3	3	3	
d	∞	4	3	3	2	
e	∞	2	0	0	0	

Example of Maintaining Pointers

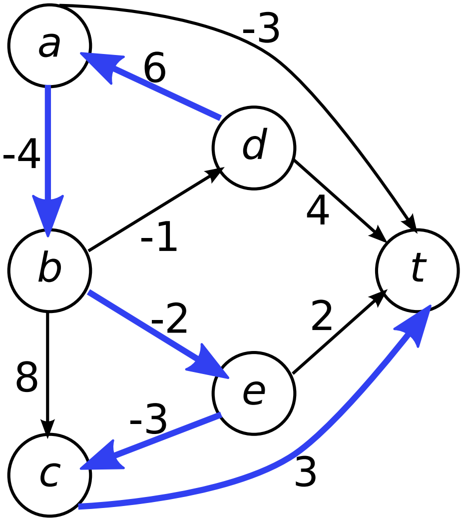
$$M[v] = \min\left(M'[v], \min_{w \in N_v}\left(c_{vw} + M'[w]\right)\right)$$



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

Example of Maintaining Pointers

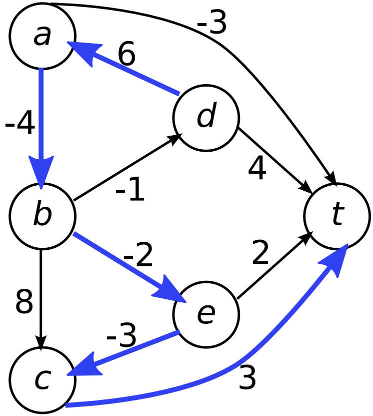
$$M[v] = \min\left(M'[v], \min_{w \in N_v}\left(c_{vw} + M'[w]\right)\right)$$



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

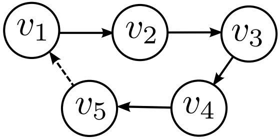
Computing the Shortest Path: Correctness

- *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?



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

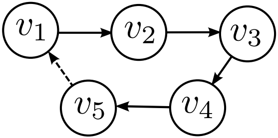
Computing the Shortest Path: Cycles in P



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

- Claim: If P has a cycle C, then C has negative cost.

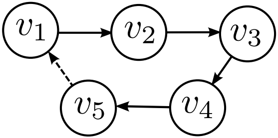
Computing the Shortest Path: Cycles in P



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

- Claim: If P has a cycle C, then C has negative cost.
  - Suppose we set $f(v) = w$. At this instant, $M[v] = c_{vw} + M'[w]$.

Computing the Shortest Path: Cycles in P

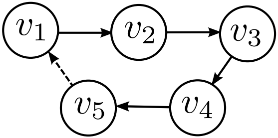


$$M[v] = \min\left(M'[v], \min_{w \in N_v}\left(c_{vw} + M'[w]\right)\right)$$

- Claim: If P has a cycle C, then C has negative cost.
  - Suppose we set $f(v) = w$. At this instant, $M[v] = c_{vw} + M'[w]$.
  - Comparing $M[w]$ and $M'[w]$, Dynamic Programming: Shortest Paths: $M[w]$ and $M'[w]$

.

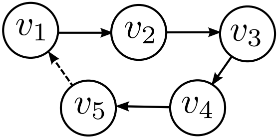
Computing the Shortest Path: Cycles in P



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

- Claim: If P has a cycle C, then C has negative cost.
  - Suppose we set $f(v) = w$. At this instant, $M[v] = c_{vw} + M'[w]$.
  - Comparing $M[w]$ and $M'[w]$, we know that $M[w] \leq M'[w]$.
  - Between this assignment and the assignment of $f(v)$ to some other node, $M[w]$ may itself further decrease. Hence, $M[v] \geq c_{vw} + M[w]$, in general.

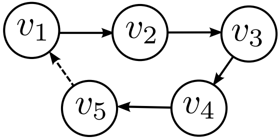
Computing the Shortest Path: Cycles in P



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

- Claim: If P has a cycle C, then C has negative cost.
  - Suppose we set $f(v) = w$. At this instant, $M[v] = c_{vw} + M'[w]$.
  - Comparing $M[w]$ and $M'[w]$, we know that $M[w] \leq M'[w]$.
  - Between this assignment and the assignment of $f(v)$ to some other node, $M[w]$ may itself further decrease. Hence, $M[v] \geq c_{vw} + M[w]$, in general.
  - 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?

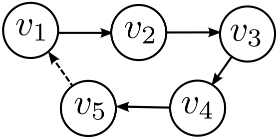
Computing the Shortest Path: Cycles in P



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

- Claim: If P has a cycle C, then C has negative cost.
  - Suppose we set $f(v) = w$. At this instant, $M[v] = c_{vw} + M'[w]$.
  - Comparing $M[w]$ and $M'[w]$, we know that $M[w] \leq M'[w]$.
  - Between this assignment and the assignment of $f(v)$ to some other node, $M[w]$ may itself further decrease. Hence, $M[v] \geq c_{vw} + M[w]$, in general.
  - 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 $1 \leq i < k - 1$.

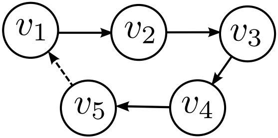
Computing the Shortest Path: Cycles in P



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

- Claim: If P has a cycle C, then C has negative cost.
  - Suppose we set $f(v) = w$. At this instant, $M[v] = c_{vw} + M'[w]$.
  - Comparing $M[w]$ and $M'[w]$, we know that $M[w] \leq M'[w]$.
  - Between this assignment and the assignment of $f(v)$ to some other node, $M[w]$ may itself further decrease. Hence, $M[v] \geq c_{vw} + M[w]$, in general.
  - 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 $1 \leq i < k - 1$.
  - Dynamic Programming: Shortest Paths: Bound on $M[v_k] - M[v_1]$

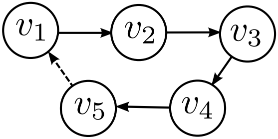
Computing the Shortest Path: Cycles in P



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

- Claim: If P has a cycle C, then C has negative cost.
  - Suppose we set $f(v) = w$. At this instant, $M[v] = c_{vw} + M'[w]$.
  - Comparing $M[w]$ and $M'[w]$, we know that $M[w] \leq M'[w]$.
  - Between this assignment and the assignment of $f(v)$ to some other node, $M[w]$ may itself further decrease. Hence, $M[v] \geq c_{vw} + M[w]$, in general.
  - 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 $1 \leq i < k - 1$.
  - $M[v_k] - M[v_1] > c_{v_k v_1}$.

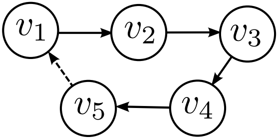
Computing the Shortest Path: Cycles in P



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

- Claim: If P has a cycle C, then C has negative cost.
  - Suppose we set $f(v) = w$. At this instant, $M[v] = c_{vw} + M'[w]$.
  - Comparing $M[w]$ and $M'[w]$, we know that $M[w] \leq M'[w]$.
  - Between this assignment and the assignment of $f(v)$ to some other node, $M[w]$ may itself further decrease. Hence, $M[v] \geq c_{vw} + M[w]$, in general.
  - 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, $0 > \sum_{i=1}^{k-1} c_{v_i v_{i+1}} + c_{v_k v_1} =$ cost of C.

Computing the Shortest Path: Cycles in P



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

- Claim: If P has a cycle C, then C has negative cost.
  - Suppose we set $f(v) = w$. At this instant, $M[v] = c_{vw} + M'[w]$.
  - Comparing $M[w]$ and $M'[w]$, we know that $M[w] \leq M'[w]$.
  - Between this assignment and the assignment of $f(v)$ to some other node, $M[w]$ may itself further decrease. Hence, $M[v] \geq c_{vw} + M[w]$, in general.
  - 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, $0 > \sum_{i=1}^{k-1} c_{v_i v_{i+1}} + c_{v_k v_1} =$ 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. ▸ Dynamic Programming: Shortest Paths: Last node of P_v

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.

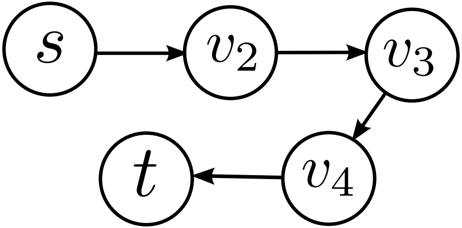
Bellman-Ford Algorithm: One Array

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

- We can prove algorithm's correctness in this case as well.

Bellman-Ford Algorithm: Early Termination

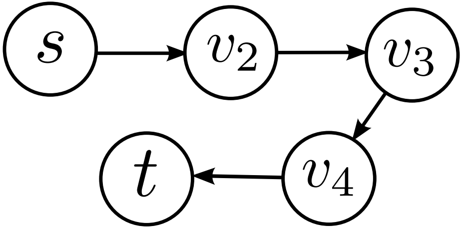
$$M[v] = \min \left(M[v], \min_{w \in N_v} \left(c_{vw} + M[w] \right) \right)$$



- In general, after i iterations, the path whose length is $M[v]$ may have many more than i edges.

Bellman-Ford Algorithm: Early Termination

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



- 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.