Dynamic Programming

T. M. Murali

September 26, October 1, 3, 8, 2018

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.

EST	6:00 PM	6:30 PM	7:00 PM	7:30 PM	8:00 PM	8:30 PM	9:00 PM	9:30 PM	10:00 PM	10:30 PM	11:00 PM	11:30 PM
21 FREEFRM	**The Notebook** (2004)				**Shadowhunters** *How Are Thou Fallen* NEW		**Beyond** *Last Action Hero* NEW		**Shadowhunters** *How Are Thou Fallen*		**The 700 Club**	
22 FX	**This Is 40** (2012)				**The Internship** (2013)					**The Internship** (2013)		
23 CNN	**The Situation Room With Wolf Blitzer** NEW		**Erin Burnett OutFront** NEW		**Anderson Cooper 360** NEW		**Anderson Cooper 360** NEW		**CNN Tonight With Don Lemon** NEW		**CNN Tonight With Don Lemon** NEW	
24 HLN	**Forensic Files** *Partners in Crime*	**Forensic Files** *Within a Hair*	**Forensic Files** *Elephant Tracks*	**Forensic Files** *A Bag of Evidence*	**Primetime Justice With Ashleigh Banfield** LIVE		**How It Really Happened** *Prince, The End*		**How It Really Happened** *The OJ Simpson Case: Other Killer Theories*			
25 Comedy	**The Wedding Ringer** (2015)				**South Park** *Butters' Bottom Bitch*	**South Park** *Raisins*	**Archer** *The Figgis Agency*	**Archer** *The Handoff*	**South Park** *Cartman Finds Love*	**South Park** *Tweek x Craig*	**The Daily Show With Elaine** *Welteroth & Phillip Picardi* NEW	**At Midnight With Chris** *Kyle Kinane; Tom Lennon; Milana Vayntrub* LIVE
26 Spike	**Cops** *Fort Worth*	**Cops** *Fort Worth, Chattanooga, Des Moines: Coast to Coast*	**Cops**	**Cops** *Street Crimes Special Edition*	**Cops** *Eye in the Sky*	**Cops**	**Cops** *Crying Over Spilled Milk*	**Cops** *Atlanta*	**Cops** *One, Two, Tree*	**Cops**	**Cops** *Trouble in Paradise*	**Cops** *Texas*
27 TBS	**Family Guy** *Roasted Guy*	**Family Guy** *Fighting Irish*	**Family Guy** *Take My Wife*	**Family Guy** *Pilling Them Softly*	**Family Guy** *Papa Has a Rollin' Son*	**American Dad** *The Life Aquatic with Steve Smith*	**American Dad** *Hayley Smith, Seal Team Six*	**Family Guy** *Guy Robot*	**Family Guy** *Peter, Chris & Brian*	**Family Guy** *Peter's Sister*	**Conan** *David Oyelowo; Louie Anderson; Angel Olsen* NEW	
28 ESPN	**SportsCenter With Michael and Jemele** LIVE		**College Basketball** *Louisville at Syracuse* LIVE				**College Basketball** *West Virginia at Kansas* LIVE				**SportsCenter** LIVE	
29 ESPN 2	**Around the Horn** NEW	**Pardon the Interruption** NEW	**Women's College Basketball** *Texas at Florida State* LIVE				**Women's College Basketball** *South Carolina at Connecticut* LIVE				**College Basketball** LIVE	**Women's College** *Georgia at Florida*
30 Lifetime	**My Best Friend's Wedding** (1997)				**Valentine's Day** (2010)						**Project Runway: Junior** *Race to the Finale*	
Syfy	**Source Code**		**Underworld: Evolution**				**Underworld: Rise of the Lycans**				**My Soul to Take**	

- Input: Start and end time of each movie.
- Constraint: Only one TV $\Rightarrow$ cannot watch two overlapping movies at the same time.
- Goal: Compute the largest number of movies we can watch.

Interval Scheduling

INTERVAL SCHEDULING

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

SOLUTION: The largest subset of mutually compatible jobs.

- Two jobs are *compatible* if they do not overlap.
- This problem models the situation where you have a resource, a set of fixed jobs, and you want to schedule as many jobs as possible.
- For any input set of jobs, algorithm must provably compute the largest set of compatible jobs.

Template for Greedy Algorithm

- Process jobs in some order. Add next job to the result if it is compatible with the jobs already in the result.
- Key question: in what order should we process the jobs?

Template for Greedy Algorithm

- Process jobs in some order. Add next job to the result if it is compatible with the jobs already in the result.
- Key question: in what order should we process the jobs?

  Earliest start time Increasing order of start time $s(i)$.

  Earliest finish time Increasing order of finish time $f(i)$.

  Shortest interval Increasing order of length $f(i) - s(i)$.

  Fewest conflicts Increasing order of the number of conflicting jobs. How fast can you compute the number of conflicting jobs for each job?

Greedy Ideas that Do Not Work

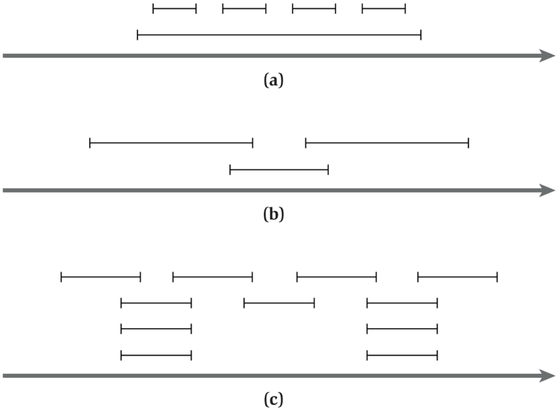


Figure 4.1 Some instances of the Interval Scheduling Problem on which natural greedy algorithms fail to find the optimal solution. In (a), it does not work to select the interval that starts earliest; in (b), it does not work to select the shortest interval; and in (c), it does not work to select the interval with the fewest conflicts.

Interval Scheduling Algorithm: Earliest Finish Time

- Schedule jobs in order of earliest finish time (EFT).

```
Initially let R be the set of all requests, and let A be empty
While R is not yet empty
  Choose a request i ∈ R that has the smallest finishing time
  Add request i to A
  Delete all requests from R that are not compatible with request i
EndWhile
Return the set A as the set of accepted requests
```

Interval Scheduling Algorithm: Earliest Finish Time

- Schedule jobs in order of earliest finish time (EFT).

```
Initially let R be the set of all requests, and let A be empty
While R is not yet empty
  Choose a request i ∈ R that has the smallest finishing time
  Add request i to A
  Delete all requests from R that are not compatible with request i
EndWhile
Return the set A as the set of accepted requests
```

- Claim: A is a compatible set of jobs.

Interval Scheduling Algorithm: Earliest Finish Time

- Schedule jobs in order of earliest finish time (EFT).

```
Initially let R be the set of all requests, and let A be empty
While R is not yet empty
  Choose a request i ∈ R that has the smallest finishing time
  Add request i to A
  Delete all requests from R that are not compatible with request i
EndWhile
Return the set A as the set of accepted requests
```

- Claim: A is a compatible set of jobs. Proof follows by construction, i.e., the algorithm computes a compatible set of jobs.

Ideas for Analysing the EFT Algorithm

- We need to prove that $|A|$ (the number of jobs in A) is the largest possible in *any* set of mutually compatible jobs.

Ideas for Analysing the EFT Algorithm

- We need to prove that $|A|$ (the number of jobs in A) is the largest possible in *any* set of mutually compatible jobs.
- Proof idea 1: algorithm makes the best choice at each step, so it must choose the largest number of mutually compatible jobs.

Ideas for Analysing the EFT Algorithm

- We need to prove that $|A|$ (the number of jobs in A) is the largest possible in *any* set of mutually compatible jobs.
- Proof idea 1: algorithm makes the best choice at each step, so it must choose the largest number of mutually compatible jobs.
  - What does "best" mean?
  - This idea is too generic. It can be applied even to algorithms that we know do not work correctly.

Ideas for Analysing the EFT Algorithm

- We need to prove that $|A|$ (the number of jobs in A) is the largest possible in *any* set of mutually compatible jobs.
- Proof idea 1: algorithm makes the best choice at each step, so it must choose the largest number of mutually compatible jobs.
  - What does "best" mean?
  - This idea is too generic. It can be applied even to algorithms that we know do not work correctly.
- Proof idea 2: at each step, can we show algorithm has the "better" solution than any other answer?
  - What does "better" mean?
  - How do we measure progress of the algorithm?

Ideas for Analysing the EFT Algorithm

- We need to prove that $|A|$ (the number of jobs in A) is the largest possible in *any* set of mutually compatible jobs.
- Proof idea 1: algorithm makes the best choice at each step, so it must choose the largest number of mutually compatible jobs.
  - What does "best" mean?
  - This idea is too generic. It can be applied even to algorithms that we know do not work correctly.
- Proof idea 2: at each step, can we show algorithm has the "better" solution than any other answer?
  - What does "better" mean?
  - How do we measure progress of the algorithm?
- Basic idea of proof:
  - We can sort jobs in any solution in increasing order of their finishing time.
  - Finishing time of job number r selected by $A \leq$ finishing time of job number r selected by any other algorithm.

Analysing the EFT Algorithm

- Let O be an optimal set of jobs. We will show that $|A| = |O|$.
- Let $i_1, i_2, \ldots, i_k$ be the set of jobs in A in order.
- Let $j_1, j_2, \ldots, j_m$ be the set of jobs in O in order, $m \geq k$.
- Claim: For all indices $r \leq k$, $f(i_r) \leq f(j_r)$.

Analysing the EFT Algorithm

- Let O be an optimal set of jobs. We will show that $|A| = |O|$.
- Let $i_1, i_2, \ldots, i_k$ be the set of jobs in A in order.
- Let $j_1, j_2, \ldots, j_m$ be the set of jobs in O in order, $m \geq k$.
- Claim: For all indices $r \leq k$, $f(i_r) \leq f(j_r)$. Prove by induction on r.

Analysing the EFT Algorithm

- Let O be an optimal set of jobs. We will show that $|A| = |O|$.
- Let $i_1, i_2, \ldots, i_k$ be the set of jobs in A in order.
- Let $j_1, j_2, \ldots, j_m$ be the set of jobs in O in order, $m \geq k$.
- Claim: For all indices $r \leq k$, $f(i_r) \leq f(j_r)$. Prove by induction on r.

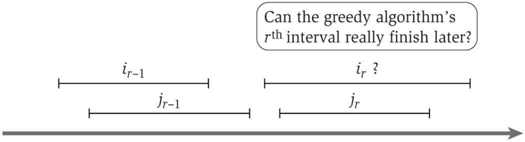


Figure 4.3 The inductive step in the proof that the greedy algorithm stays ahead.

Analysing the EFT Algorithm

- Let O be an optimal set of jobs. We will show that $|A| = |O|$.
- Let $i_1, i_2, \ldots, i_k$ be the set of jobs in A in order.
- Let $j_1, j_2, \ldots, j_m$ be the set of jobs in O in order, $m \geq k$.
- Claim: For all indices $r \leq k$, $f(i_r) \leq f(j_r)$. Prove by induction on r.

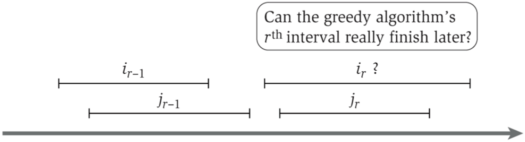


Figure 4.3 The inductive step in the proof that the greedy algorithm stays ahead.

- Claim: $m = k$.

Analysing the EFT Algorithm

- Let O be an optimal set of jobs. We will show that $|A| = |O|$.
- Let $i_1, i_2, \ldots, i_k$ be the set of jobs in A in order.
- Let $j_1, j_2, \ldots, j_m$ be the set of jobs in O in order, $m \geq k$.
- Claim: For all indices $r \leq k$, $f(i_r) \leq f(j_r)$. Prove by induction on r.

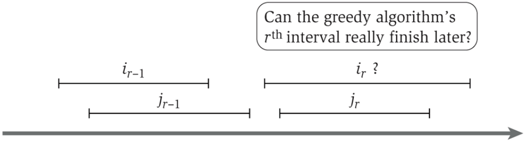


Figure 4.3 The inductive step in the proof that the greedy algorithm stays ahead.

- Claim: $m = k$.
- Claim: The greedy algorithm returns an optimal set A.

Implementing the EFT Algorithm

1. Reorder jobs so that they are in increasing order of finish time.
2. Store starting time of jobs in an array S.
3. $k = 1$.
4. While $k \leq |S|$,
   1. Output job k.
   2. Let finish time of job k be f.
   3. Iterate over S from index k onwards to find the first index i such that $S[i] \geq f$.
   4. $k = i$

Implementing the EFT Algorithm

1. Reorder jobs so that they are in increasing order of finish time.
2. Store starting time of jobs in an array S.
3. $k = 1$.
4. While $k \leq |S|$,
    1. Output job k.
    2. Let finish time of job k be f.
    3. Iterate over S from index k onwards to find the first index i such that $S[i] \geq f$.
    4. $k = i$

- Must be careful to iterate over S such that we never scan same index more than once.
- Running time is $O(n \log n)$, dominated by sorting.

Weighted Interval Scheduling

WEIGHTED INTERVAL SCHEDULING

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

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

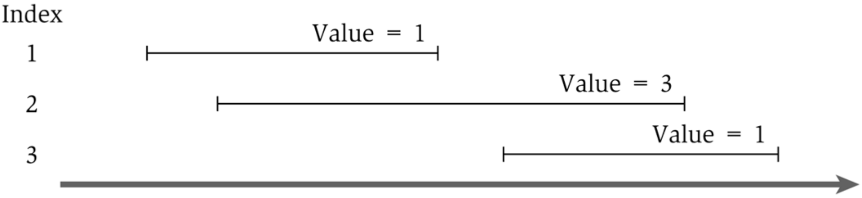


Figure 6.1 A simple instance of weighted interval scheduling.

Weighted Interval Scheduling

Weighted Interval Scheduling

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

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

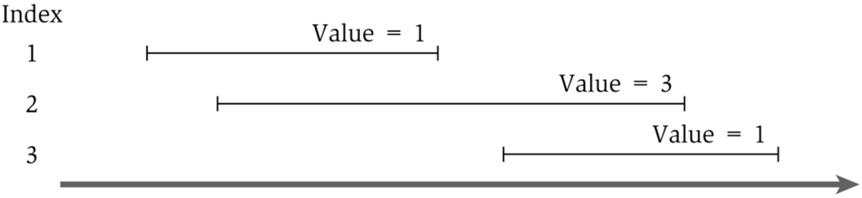
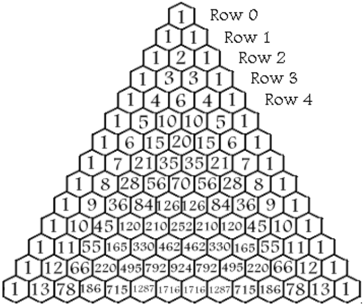


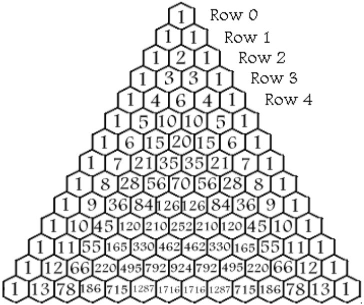
Figure 6.1 A simple instance of weighted interval scheduling.

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

Detour: a Binomial Identity

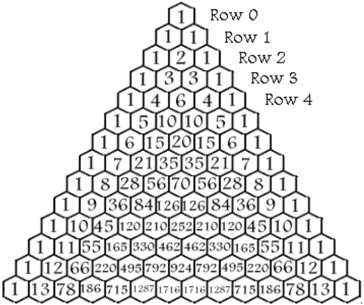


Detour: a Binomial Identity



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

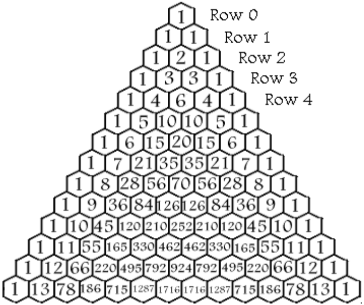
Detour: a Binomial Identity



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

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

Detour: a Binomial Identity



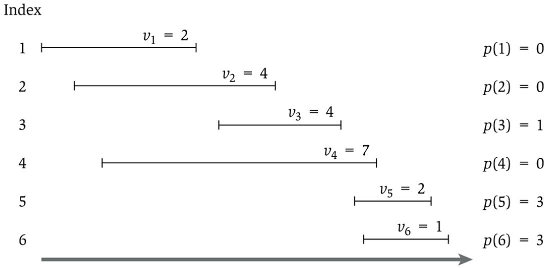
- Pascal's triangle:
  - Each element is a binomial co-efficient.
  - Each element is the sum of the two elements above it.

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

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

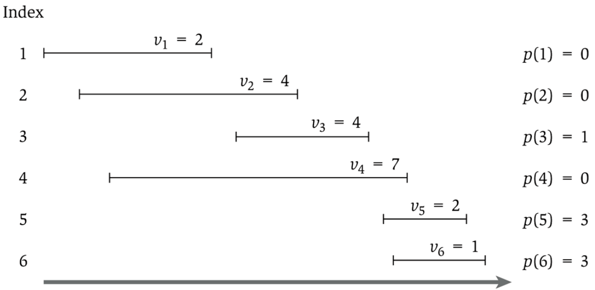
Approach

- 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.
- 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

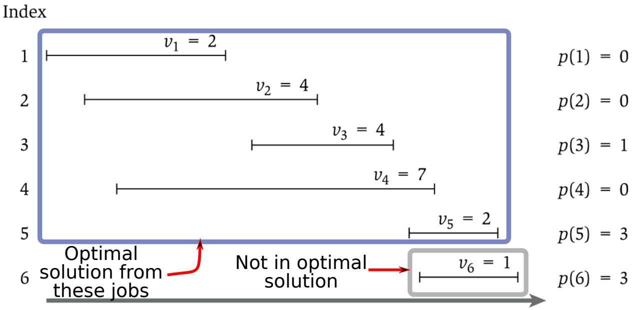


- Let $\mathcal{O}$ be the optimal solution: it contains a subset of the input jobs. Two cases to consider. One of these cases must be true.

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

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

Sub-problems

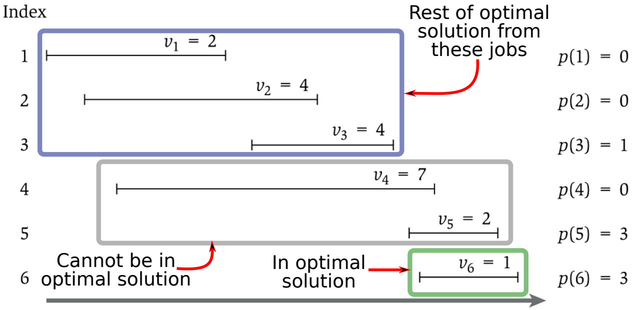


- Let $\mathcal{O}$ be the optimal solution: it contains a subset of the input jobs. Two cases to consider. One of these cases must be true.

  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

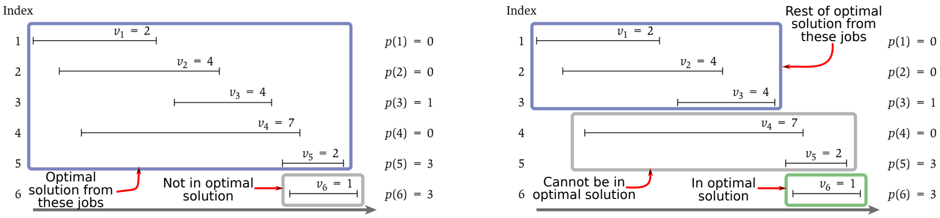


- Let $\mathcal{O}$ be the optimal solution: it contains a subset of the input jobs. Two cases to consider. One of these cases must be true.

  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.

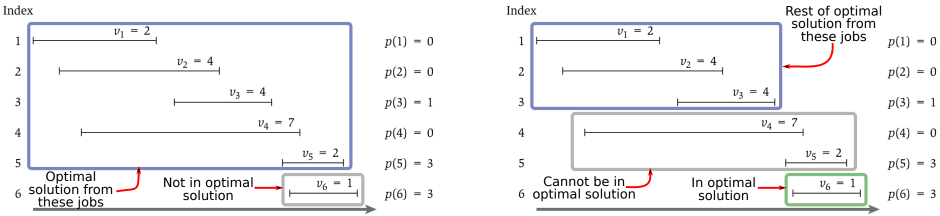
  Case 1: job n is not in $\mathcal{O}$. $\mathcal{O}$ must be the optimal solution for jobs $\{1, 2, \ldots, n-1\}$.

  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

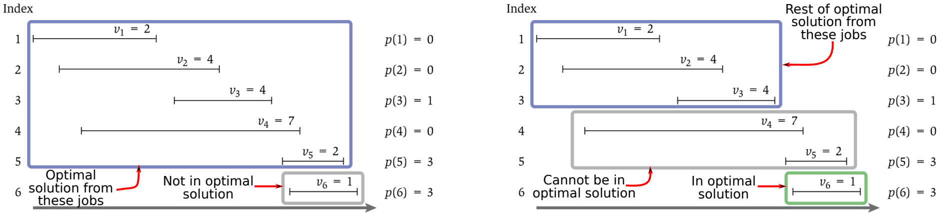


- Let $\mathcal{O}$ be the optimal solution: it contains a subset of the input jobs. Two cases to consider. One of these cases must be true.

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

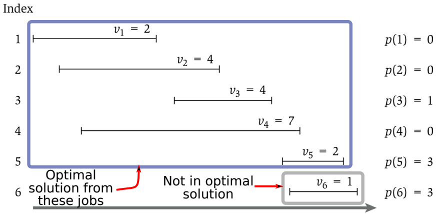
  Case 2: job n is in $\mathcal{O}$.
  - $\mathcal{O}$ cannot use incompatible jobs $\{p(n)+1, p(n)+2, \ldots, n-1\}$.
  - Remaining jobs in $\mathcal{O}$ must be the optimal solution for jobs $\{1, 2, \ldots, p(n)\}$.
- $\mathcal{O}$ must be the best of these two choices!
- Suggests finding optimal solution for sub-problems consisting of jobs $\{1, 2, \ldots, j-1, j\}$, for all values of j.

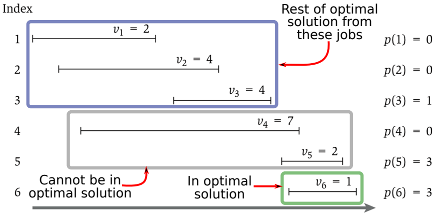
Recursion



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

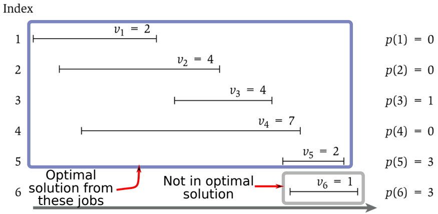
Recursion

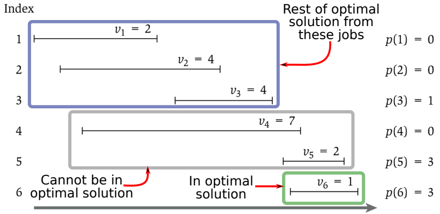




- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($\text{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\text{OPT}(n)$.

Recursion

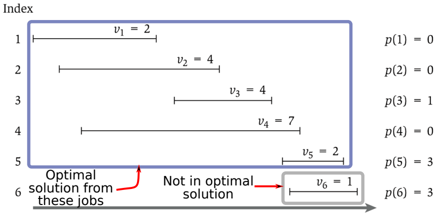


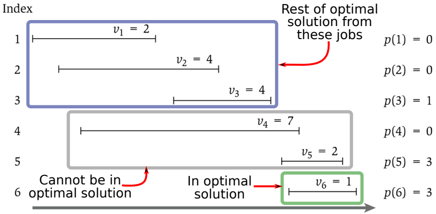


- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($\text{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\text{OPT}(n)$.
- To compute $\text{OPT}(j)$:

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

Recursion

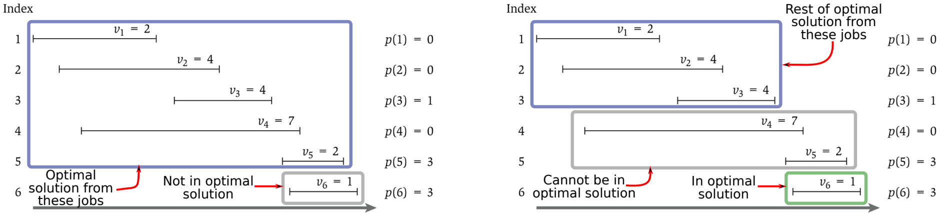




- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($\text{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\text{OPT}(n)$.
- To compute $\text{OPT}(j)$:

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

Recursion

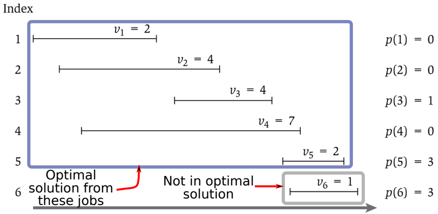


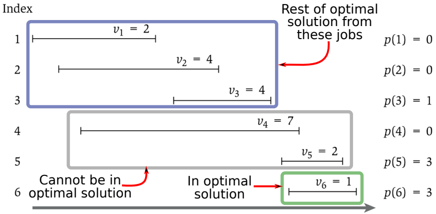
- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($\text{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\text{OPT}(n)$.
- To compute $\text{OPT}(j)$:

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

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

Recursion



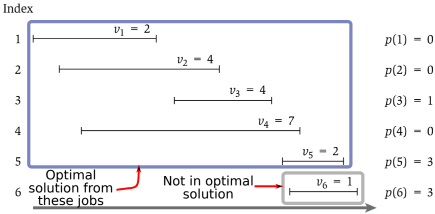


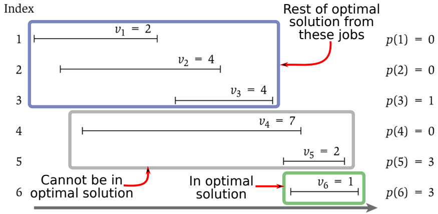
- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($\text{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\text{OPT}(n)$.
- To compute $\text{OPT}(j)$:

  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

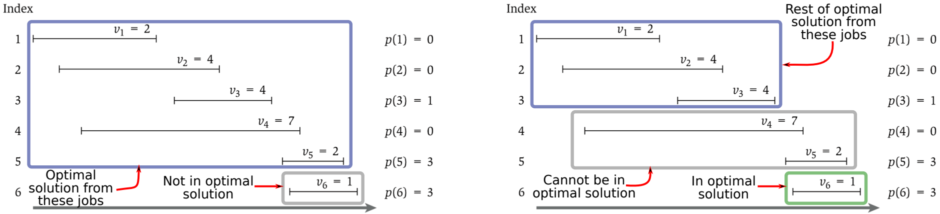




- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($\text{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\text{OPT}(n)$.
- To compute $\text{OPT}(j)$:
  - Case 1 $j \notin \mathcal{O}_j$: $\text{OPT}(j) = \text{OPT}(j-1)$.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

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

Recursion



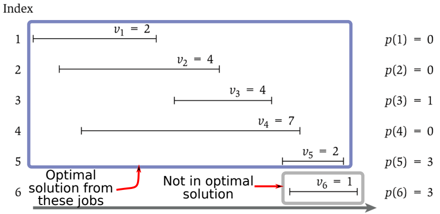
- Let $\mathcal{O}_j$ be the optimal solution for jobs $\{1, 2, \ldots, j\}$ and $OPT(j)$ be the value of this solution ($\text{OPT}(0) = 0$).
- We are seeking $\mathcal{O}_n$ with a value of $\text{OPT}(n)$.
- To compute $\text{OPT}(j)$:

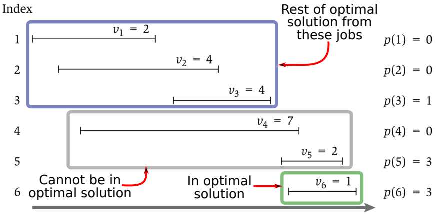
  Case 1 $j \notin \mathcal{O}_j$: $\text{OPT}(j) = \text{OPT}(j-1)$.
  Case 2 $j \in \mathcal{O}_j$: $\text{OPT}(j) = v_j + \text{OPT}(p(j))$

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

- When does job j belong to $\mathcal{O}_j$?

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

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

- When does job j belong to $\mathcal{O}_j$? If and only if $v_j + \text{OPT}(p(j)) \geq \text{OPT}(j-1)$.

Recursive Algorithm

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

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

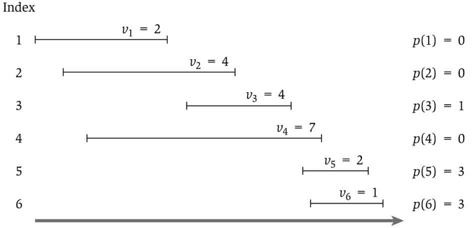


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

$\text{OPT}(6) =$
$\text{OPT}(5) =$
$\text{OPT}(4) =$
$\text{OPT}(3) =$
$\text{OPT}(2) =$
$\text{OPT}(1) =$
$\text{OPT}(0) = 0$

Example of Recursive Algorithm

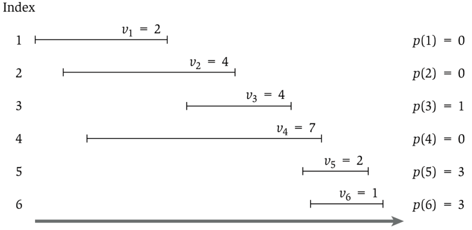


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

Example of Recursive Algorithm

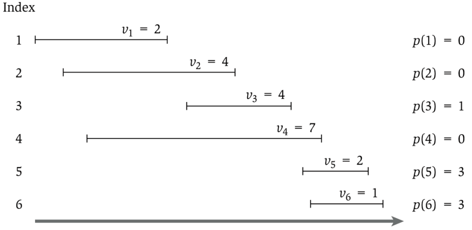


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

Example of Recursive Algorithm

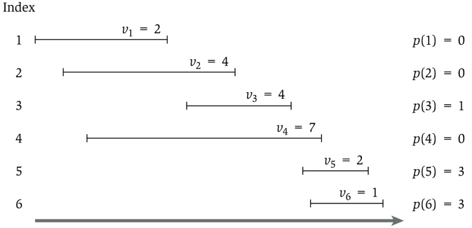


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

Example of Recursive Algorithm

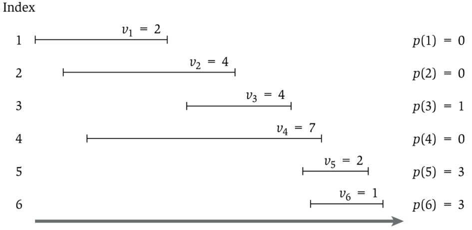


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

Example of Recursive Algorithm

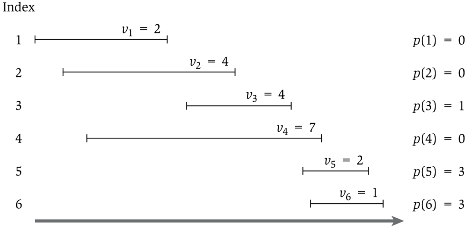


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

Example of Recursive Algorithm

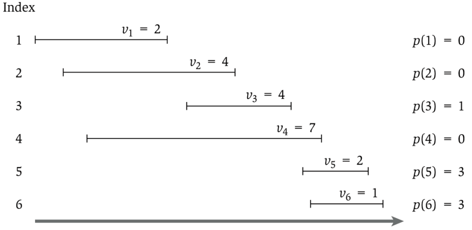


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

Example of Recursive Algorithm

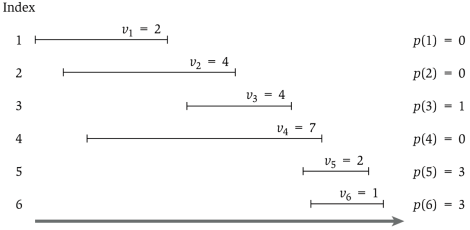


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

Example of Recursive Algorithm

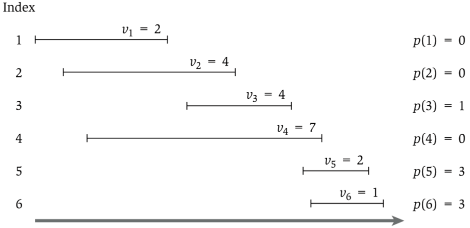


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

Example of Recursive Algorithm

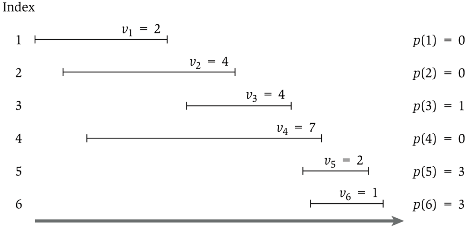


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

Example of Recursive Algorithm

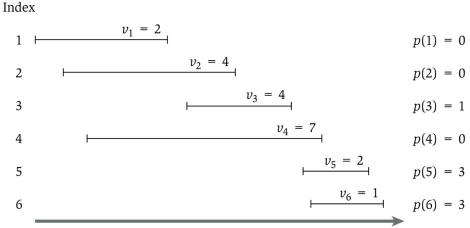


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

Example of Recursive Algorithm

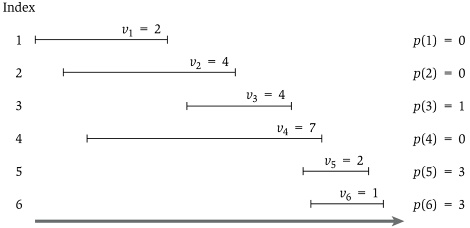


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

Example of Recursive Algorithm

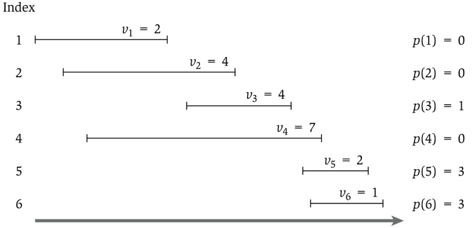


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

- Optimal solution is

Example of Recursive Algorithm

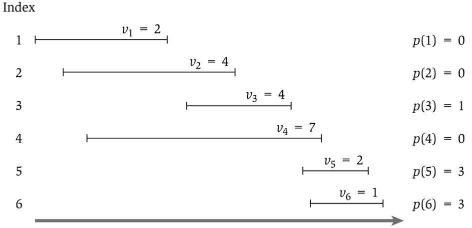


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

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

- 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 $j - 2$.

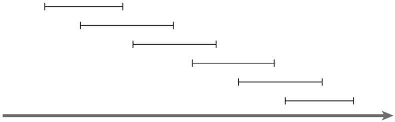


Figure 6.4 An instance of weighted interval scheduling on which the simple Compute-Opt recursion will take exponential time. The values of all intervals in this instance are 1.

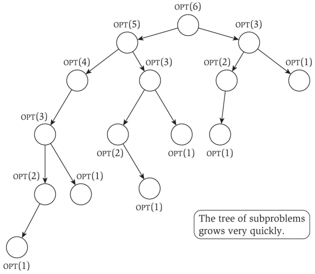


Figure 6.3 The tree of subproblems called by Compute-Opt on the problem instance of Figure 6.2.

Memoisation

- Store $\text{OPT}(j)$ 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
```

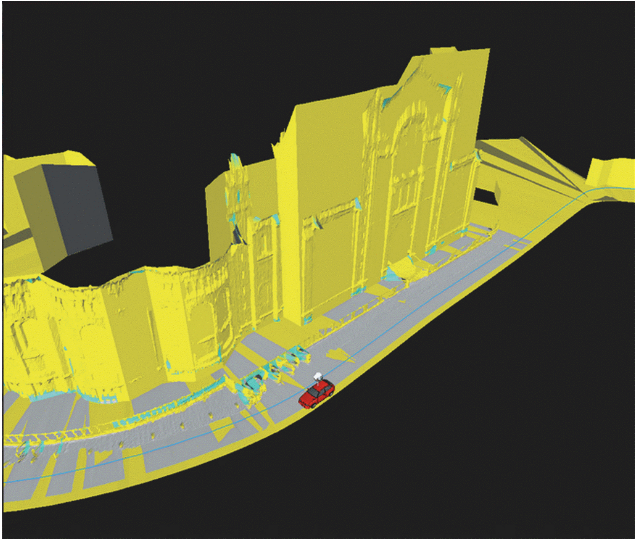
Basic Outline of Dynamic Programming

- 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

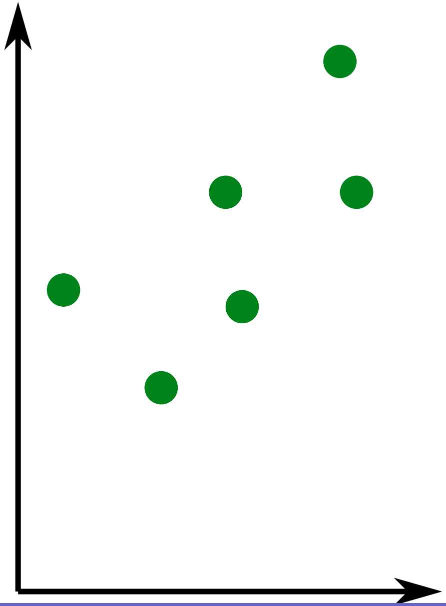
- To solve a problem, we need a collection of sub-problems that satisfy a few properties:
  1. There are a polynomial number of sub-problems.
  2. The solution to the problem can be computed easily from the solutions to the sub-problems.
  3. There is a natural ordering of the sub-problems from “smallest”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

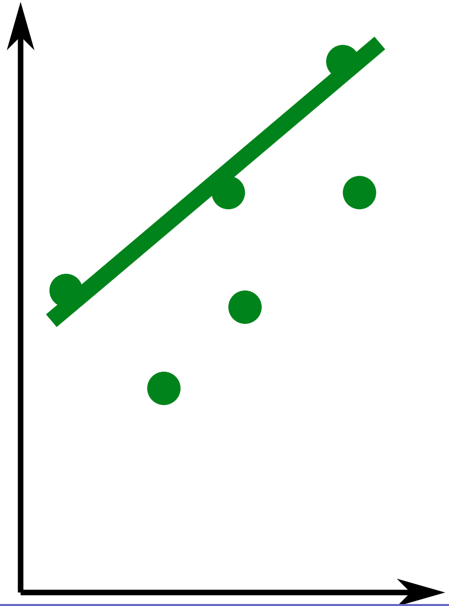


Imagery from new street view vehicles is accompanied by laser range data, which is aggregated and simplified by robustly fitting it in a coarse mesh that models the dominant scene surfaces.

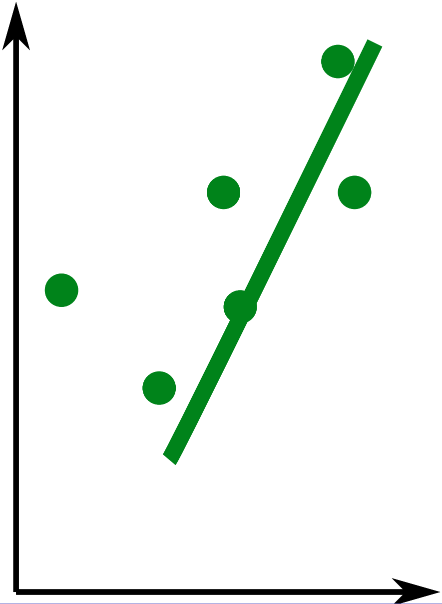
Fitting Lines



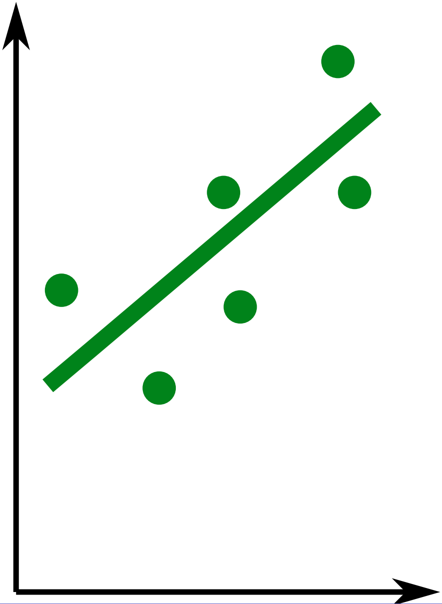
Fitting Lines



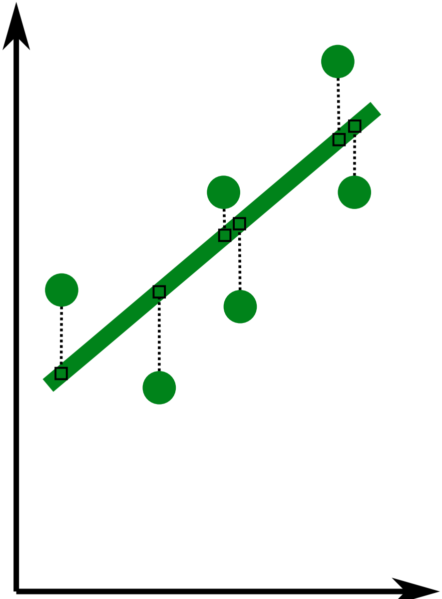
Fitting Lines



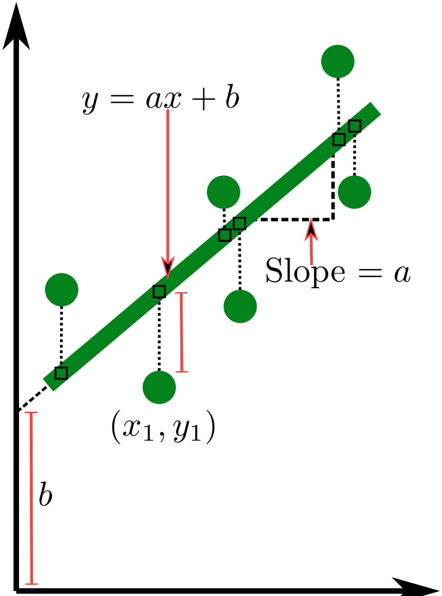
Fitting Lines



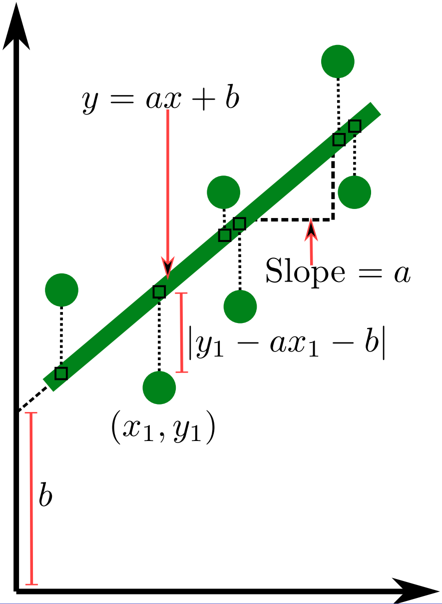
Fitting Lines



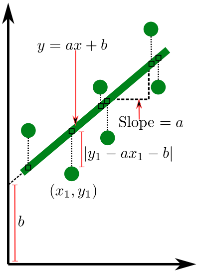
Fitting Lines



Fitting Lines

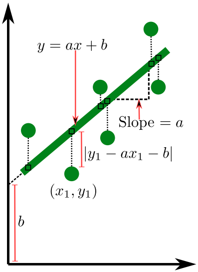


Least Squares Problem



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

Least Squares Problem

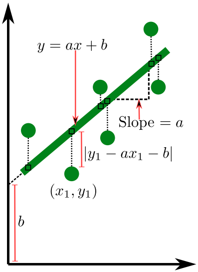


- Given scientific or statistical data plotted on two axes.
- Find the "best" line that "passes" through these points.

Least Squares
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



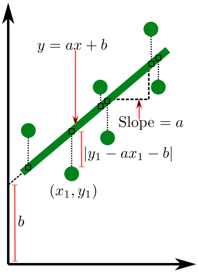
- Given scientific or statistical data plotted on two axes.
- Find the "best" line that "passes" through these points.

LEAST SQUARES
INSTANCE: Set $P = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$ 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?

Least Squares Problem



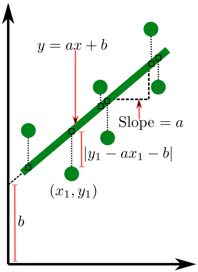
- Given scientific or statistical data plotted on two axes.
- Find the "best" line that "passes" through these points.

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

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

- 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.

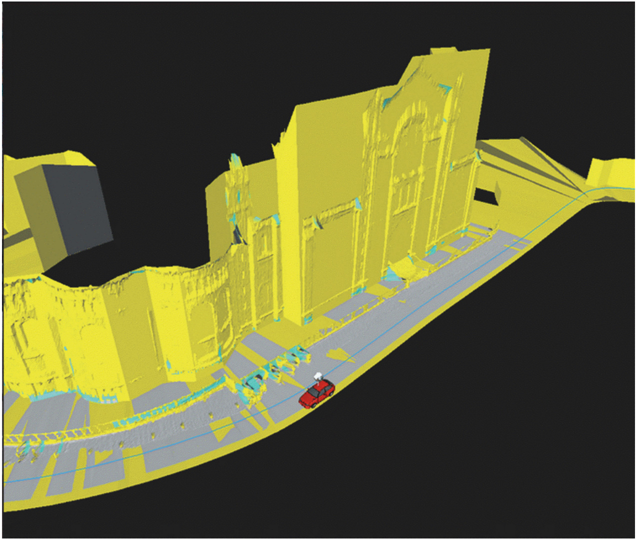
Least Squares
INSTANCE: Set $P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ of n points.
SOLUTION: Line $L : y = ax + b$ that minimises

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

- 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

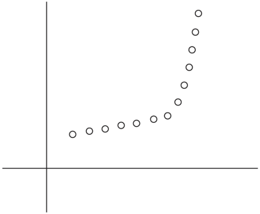


Figure 6.7 A set of points that lie approximately on two lines.

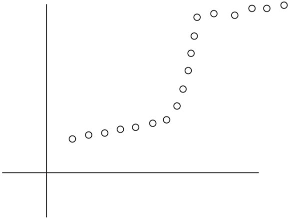
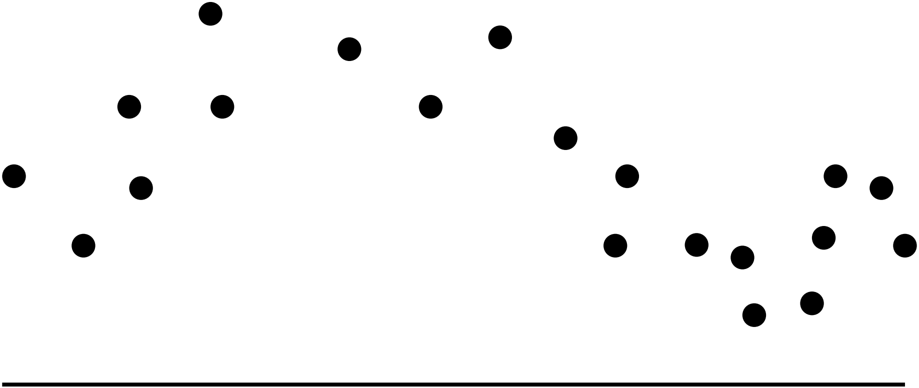


Figure 6.8 A set of points that lie approximately on three lines.

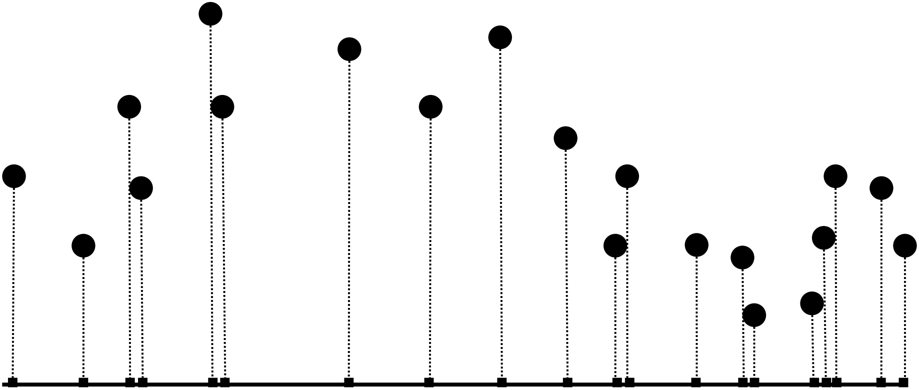
- Want to fit multiple lines through P.
- Each line must fit contiguous set of x-coordinates.
- Lines must minimise total error.

Example of Segmented Least Squares



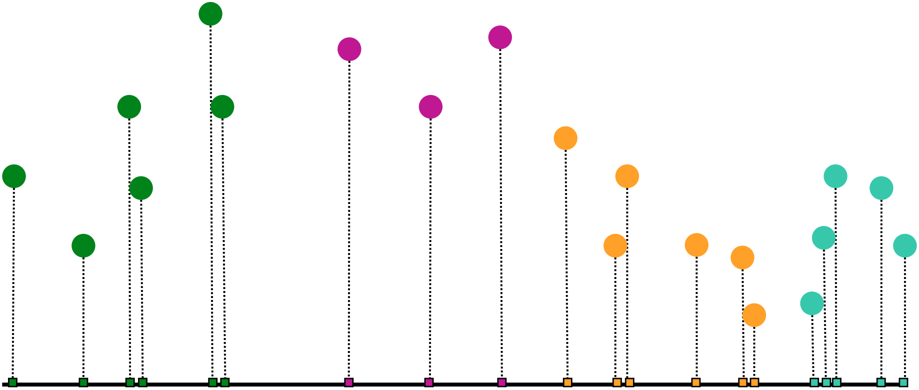
Input contains a set of two-dimensional points.

Example of Segmented Least Squares



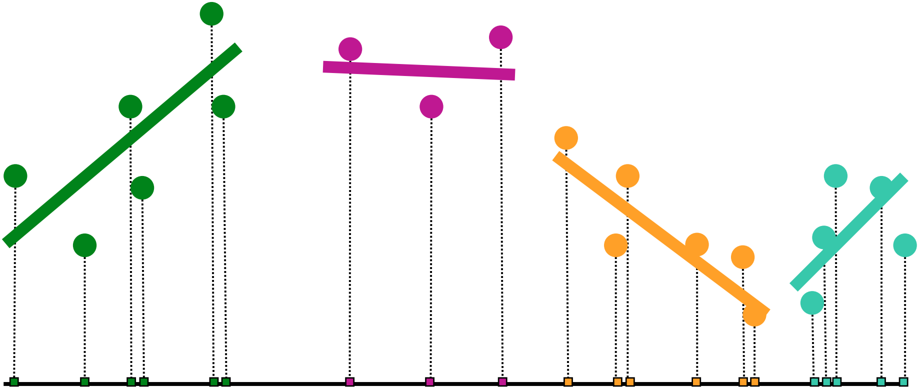
Consider the x-coordinates of the points in the input.

Example of Segmented Least Squares



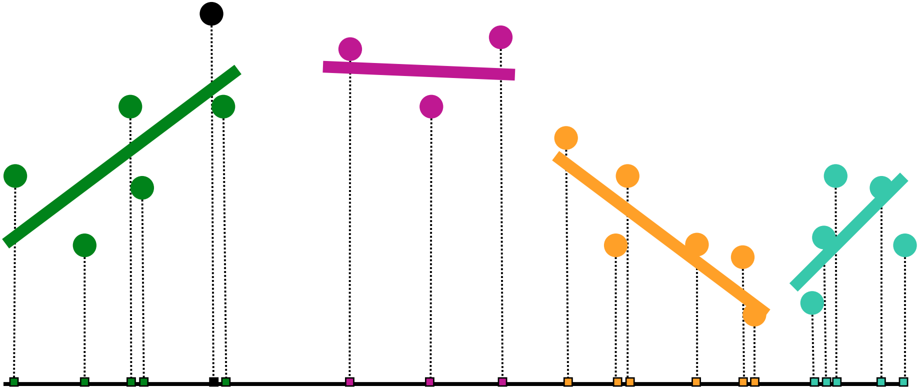
Divide the points into segments; each *segment* contains consecutive points in the sorted order by x-coordinate.

Example of Segmented Least Squares



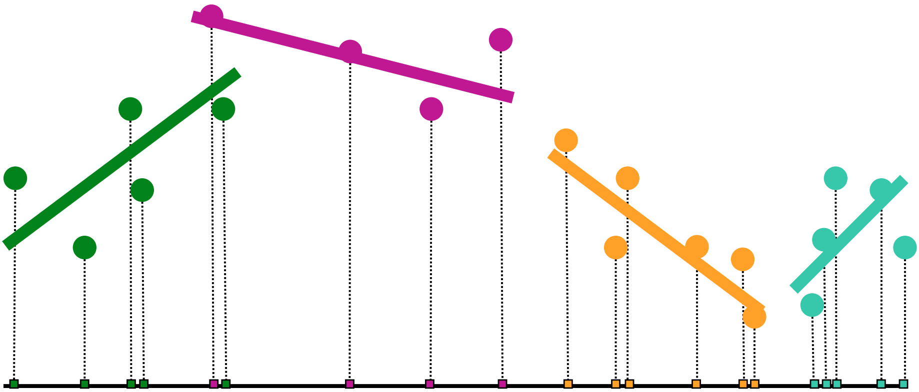
Fit the best line for each segment.

Example of Segmented Least Squares



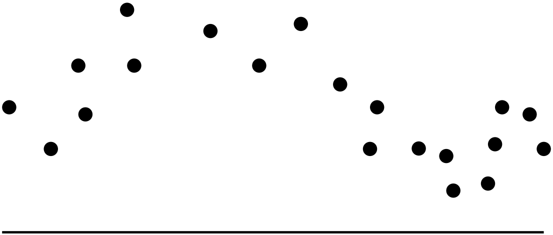
Illegal solution: black point is not in any segment.

Example of Segmented Least Squares



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

Segmented Least Squares

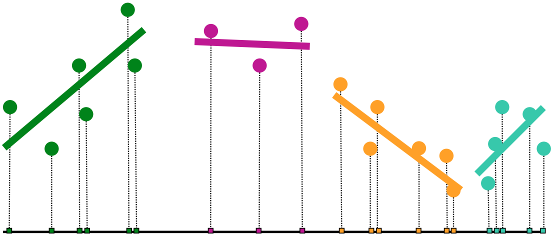


Segmented Least Squares

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:

Segmented Least Squares



SEGMENTED LEAST SQUARES

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

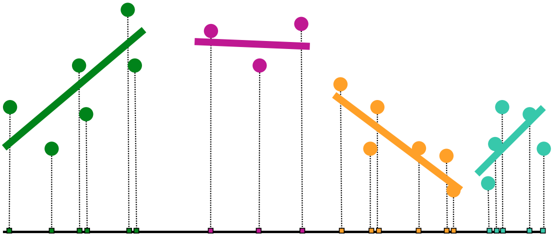
SOLUTION:

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

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

Segmented Least Squares



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 $C > 0$.

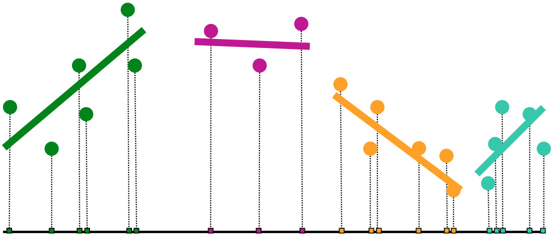
SOLUTION:

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

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

Segmented Least Squares



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:

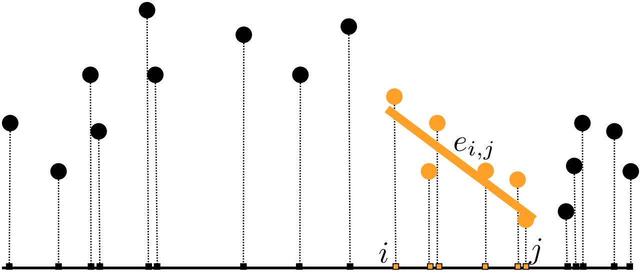
1. An integer k,
2. a partition of P into k segments $\{P_1, P_2, \ldots, P_k\}$, and
3. for each segment P_j, the best-fit line $L_j : y = a_j x + b_j, 1 \leq j \leq k$

that minimise the total error

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

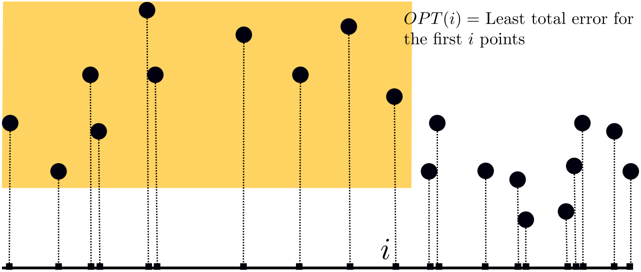
- How many unknown parameters must we find? $2k$, and we must find k too!

Formulating the Recursion I



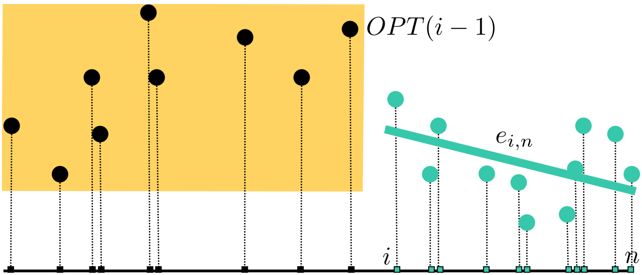
- Let $e_{i,j}$ denote the minimum error of a (single) line that fits $\{p_i, p_2, \ldots, p_j\}$.
- Let $OPT(i)$ be the optimal total error for the points $\{p_1, p_2, \ldots, p_i\}$.
- We want to compute OPT(n).

Formulating the Recursion I



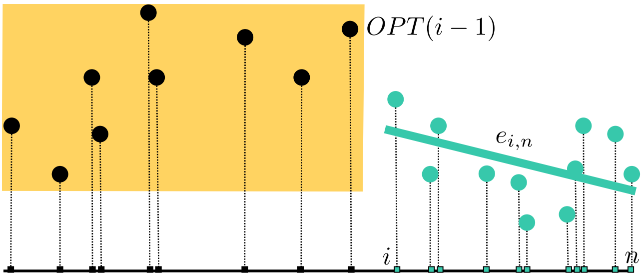
- Let $e_{i,j}$ denote the minimum error of a (single) line that fits $\{p_i, p_2, \ldots, p_j\}$.
- Let $OPT(i)$ be the optimal total error for the points $\{p_1, p_2, \ldots, p_i\}$.
- We want to compute OPT(n).

Formulating the Recursion I



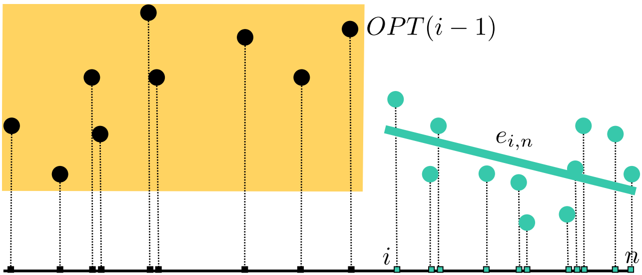
- Let $e_{i,j}$ denote the minimum error of a (single) line that fits $\{p_i, p_2, \ldots, p_j\}$.
- Let $OPT(i)$ be the optimal total error for the points $\{p_1, p_2, \ldots, p_i\}$.
- We want to compute OPT(n).
- Observation: Where does the last segment in the optimal solution end?

Formulating the Recursion I



- Let $e_{i,j}$ denote the minimum error of a (single) line that fits $\{p_i, p_2, \ldots, p_j\}$.
- Let $OPT(i)$ be the optimal total error for the points $\{p_1, p_2, \ldots, p_i\}$.
- We want to compute OPT(n).
- Observation: Where does the last segment in the optimal solution end? p_n, and this segment starts at some point p_i.

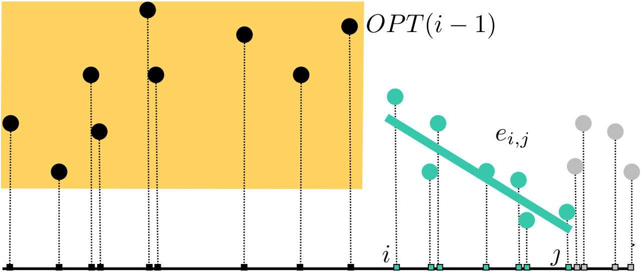
Formulating the Recursion I



- Let $e_{i,j}$ denote the minimum error of a (single) line that fits $\{p_i, p_2, \ldots, p_j\}$.
- Let $OPT(i)$ be the optimal total error for the points $\{p_1, p_2, \ldots, p_i\}$.
- We want to compute OPT(n).
- Observation: Where does the last segment in the optimal solution end? p_n, and this segment starts at some point p_i.
- If the last segment in the optimal partition is $\{p_i, p_{i+1}, \ldots, p_n\}$, then

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

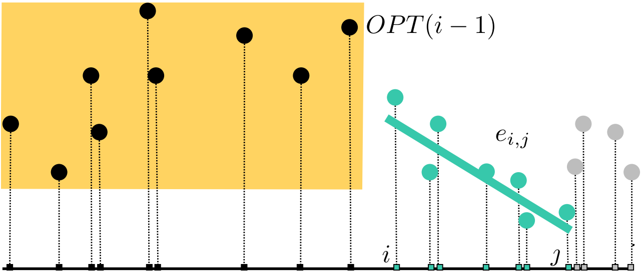
Formulating the Recursion II



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

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

Formulating the Recursion II



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

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

- But i can take only j distinct values: $1, 2, \ldots, j-1, j$. Therefore,

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

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

Dynamic Programming Algorithm

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

```
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

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

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

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

Running Time

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

```
Segmented-Least-Squares(n)
  Array M[0...n]
  Set M[0]=0
  For all pairs i≤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.

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

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.

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

Running Time

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

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

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

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".

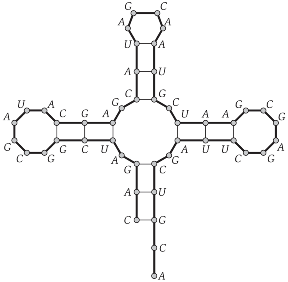


Figure 6.13 An RNA secondary structure. Thick lines connect adjacent elements of the sequence; thin lines indicate pairs of elements that are matched.

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".

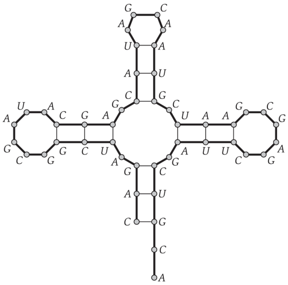


Figure 6.13 An RNA secondary structure. Thick lines connect adjacent elements of the sequence; thin lines indicate pairs of elements that are matched.

- 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".

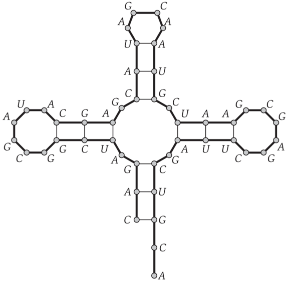


Figure 6.13 An RNA secondary structure. Thick lines connect adjacent elements of the sequence; thin lines indicate pairs of elements that are matched.

- 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

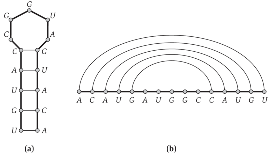


Figure 6.14 Two views of an RNA secondary structure. In the second view, (b), the string has been "stretched" lengthwise, and edges connecting matched pairs appear as noncrossing "bubbles" over the string.

- 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

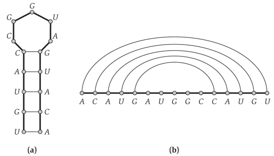
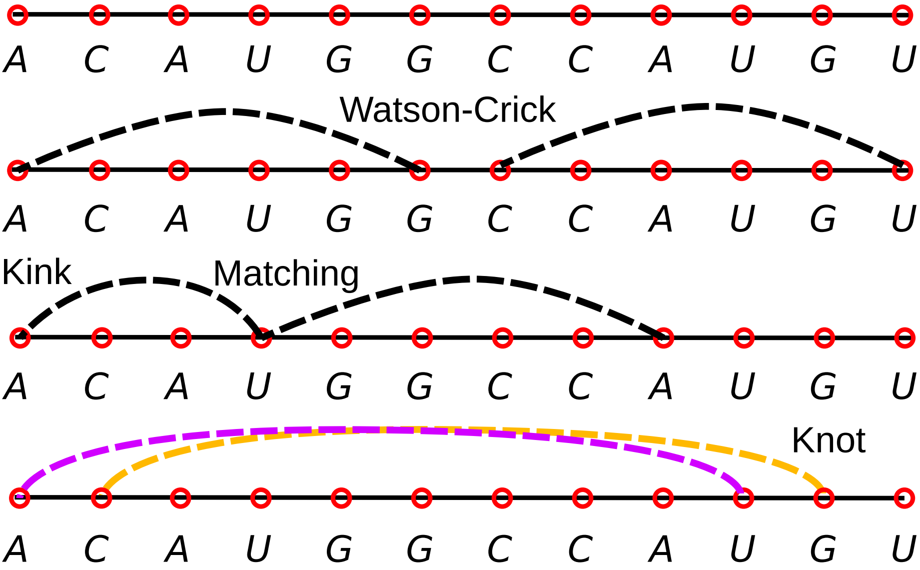


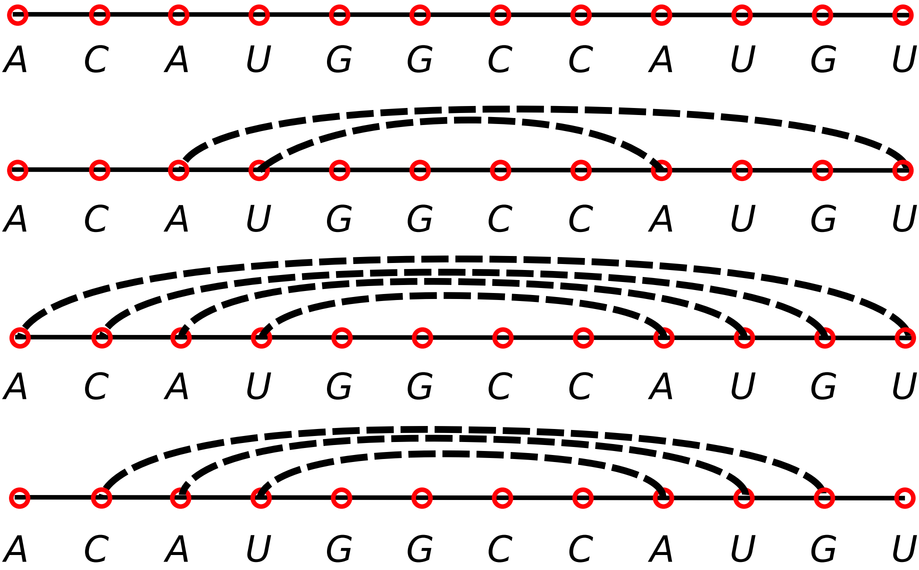
Figure 6.14 Two views of an RNA secondary structure. In the second view, (b), the string has been "stretched" lengthwise, and edges connecting matched pairs appear as noncrossing "bubbles" over the string.

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

Illegal Secondary Structures



Legal Secondary Structures



Dynamic Programming Approach

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

Dynamic Programming Approach

- $OPT(j)$ is the maximum number of base pairs in a secondary structure for $b_1 b_2 \ldots b_j$. $\text{OPT}(j) = 0$, 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 $b_1 b_2 \ldots b_j$. $\text{OPT}(j) = 0$, 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 $\text{OPT}(j-1)$.

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 $t < j - 4$,

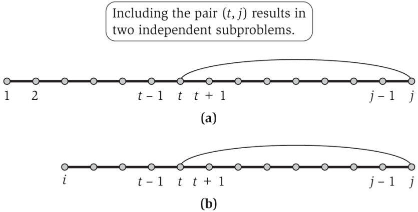


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

Dynamic Programming Approach

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

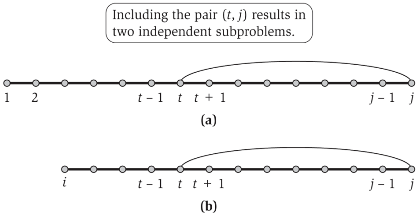


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

Dynamic Programming Approach

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

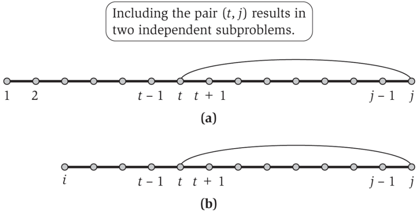


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

Dynamic Programming Approach

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

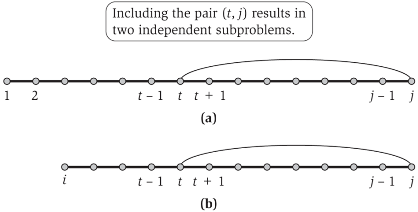


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

Correct Dynamic Programming Approach

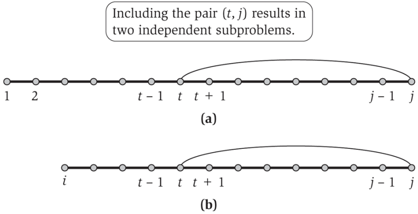


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

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

Correct Dynamic Programming Approach

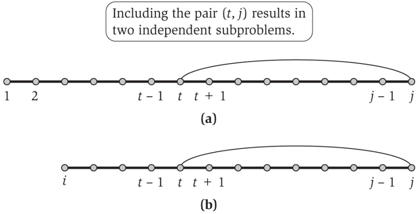


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

- $OPT(i, j)$ 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 $i \geq j - 4$.

Correct Dynamic Programming Approach

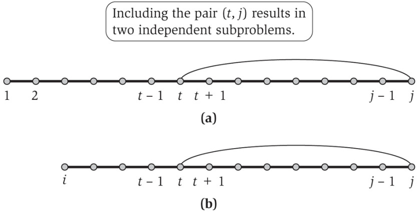


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

- $OPT(i, j)$ 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 $b_i b_{i+2} \ldots b_j$

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

Correct Dynamic Programming Approach

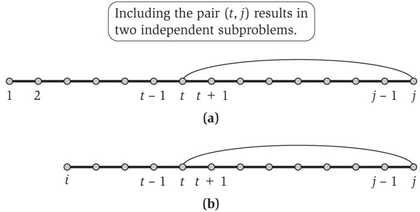


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

- *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 $b_i b_{i+2} \ldots b_j$
  1. if j is not a member of any pair, compute $\text{OPT}(i, j-1)$.

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

Correct Dynamic Programming Approach

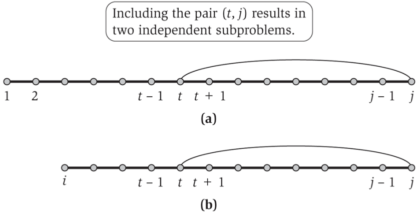


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

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

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

Correct Dynamic Programming Approach

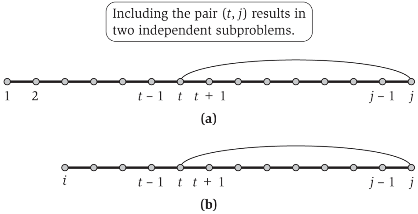


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

- $OPT(i,j)$ 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 $b_i b_{i+2} \ldots b_j$
  1. if j is not a member of any pair, compute $\text{OPT}(i, j-1)$.
  2. if j pairs with some $t < j - 4$, compute $\text{OPT}(i, t-1)$ and $\text{OPT}(t+1, j-1)$.
- Since t can range from i to $j - 5$,

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

Correct Dynamic Programming Approach

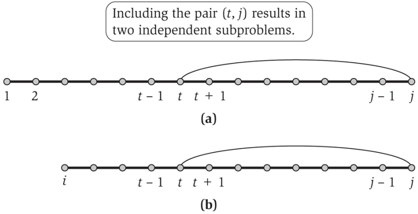


Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

- $OPT(i,j)$ 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 $b_i b_{i+2} \ldots b_j$
  1. if j is not a member of any pair, compute $\text{OPT}(i, j-1)$.
  2. if j pairs with some $t < j - 4$, compute $\text{OPT}(i, t-1)$ and $\text{OPT}(t+1, j-1)$.
- Since t can range from i to $j - 5$,

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

Correct Dynamic Programming Approach

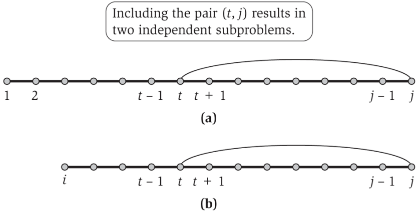


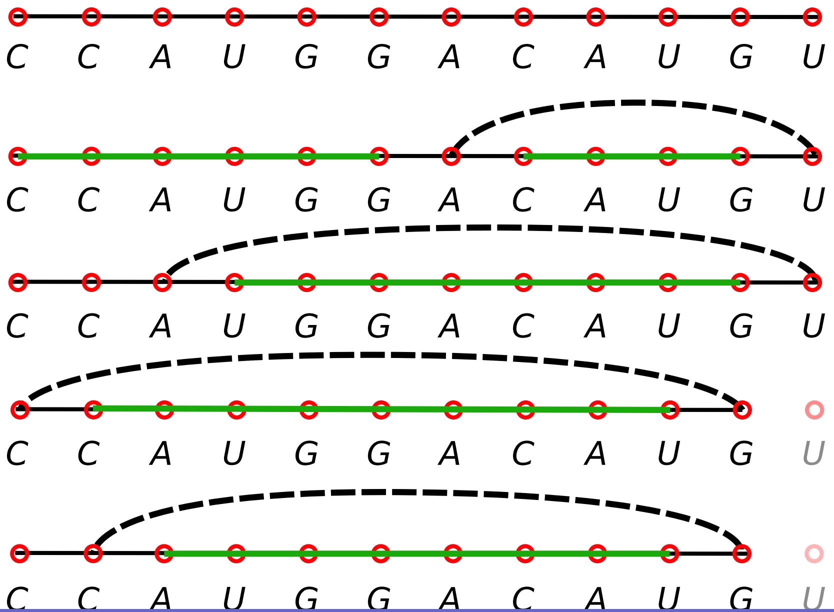
Figure 6.15 Schematic views of the dynamic programming recurrence using (a) one variable, and (b) two variables.

- $OPT(i,j)$ 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 $b_i b_{i+2} \ldots b_j$
  1. if j is not a member of any pair, compute $\text{OPT}(i, j-1)$.
  2. if j pairs with some $t < j - 4$, compute $\text{OPT}(i, t-1)$ and $\text{OPT}(t+1, j-1)$.
- Since t can range from i to $j - 5$,

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

- 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 $O(n^2)$ 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"?
- Note that computing $\text{OPT}(i, j)$ involves sub-problems $\text{OPT}(l, m)$ where $m - l < j - i$.

Dynamic Programming Algorithm

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

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

```
Initialize OPT(i, j) = 0 whenever i ≥ j − 4
For k = 5, 6, ..., n − 1
  For i = 1, 2, ... n − k
    Set j = i + k
    Compute OPT(i, j) using the recurrence in (6.13)
  Endfor
Endfor
Return OPT(1, n)
```

Dynamic Programming Algorithm

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

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

```
Initialize OPT(i, j) = 0 whenever i ≥ j − 4
For k = 5, 6, . . . , n − 1
  For i = 1, 2, . . . n − k
    Set j = i + k
    Compute OPT(i, j) using the recurrence in (6.13)
  Endfor
Endfor
Return OPT(1, n)
```

- Running time of the algorithm is $O(n^3)$.

Example of Algorithm

RNA sequence *ACCGGUAGU*

i \ j	6	7	8	9
4	0	0	0	
3	0	0		
2	0			
$i=1$				

Initial values

i \ j	6	7	8	9
4	0	0	0	0
3	0	0	1	
2	0	0		
$i=1$	1			

Filling in the values for $k = 5$

i \ j	6	7	8	9
4	0	0	0	0
3	0	0	1	1
2	0	0	1	
$i=1$	1	1		

Filling in the values for $k = 6$

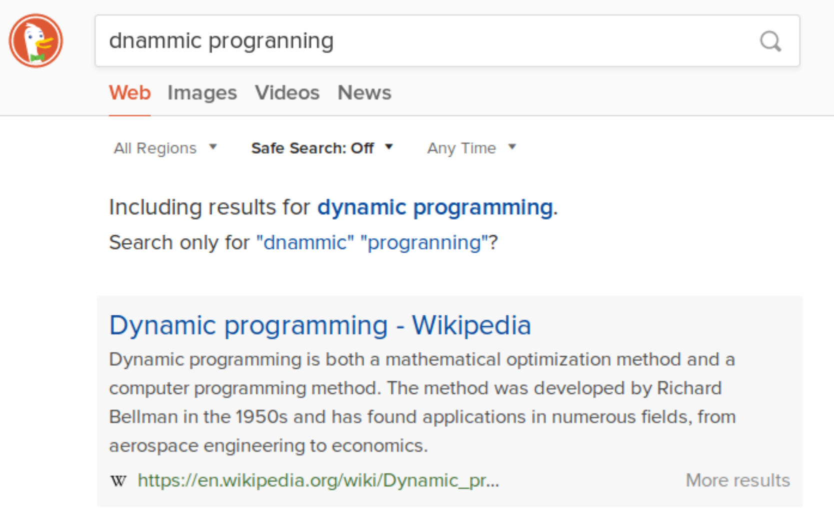
i \ j	6	7	8	9
4	0	0	0	0
3	0	0	1	1
2	0	0	1	1
$i=1$	1	1	1	

Filling in the values for $k = 7$

i \ j	6	7	8	9
4	0	0	0	0
3	0	0	1	1
2	0	0	1	1
$i=1$	1	1	1	2

Filling in the values for $k = 8$

Web Search for "dnammic programning"



- How do they know "Dynamic" and "Dymanic" are similar?

Sequence Similarity

- Given two strings, measure how similar they are.
- Given a database of strings and a query string, compute the string most similar to query in the database.
- Applications:
  - Online searches (Web, dictionary).
  - Spell-checkers.
  - Computational biology
  - Speech recognition.
  - Basis for Unix `diff`.

Defining Sequence Similarity

- "ocurrance" (wrong) vs "occurrence" (right).

```
o-currance
occurrence
```

```
o-curr-ance
occurre-nce
```

Defining Sequence Similarity

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

```
o-currance
occurrence
```

```
o-curr-ance
occurre-nce
```

```
abbbaa--bbbbaab
ababaaabbbbba-b
```

Defining Sequence Similarity

- "ocurrance" (wrong) vs "occurrence" (right).

```
o-currance
occurrence
```

```
o-curr-ance
occurre-nce
```

```
abbbaa--bbbbaab
ababaaabbbbba-b
```

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

Edit Distance

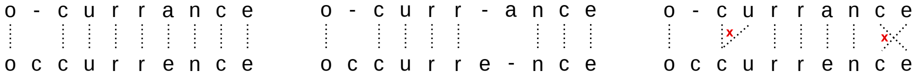
- Proposed by Needleman and Wunsch in the early 1970s.
- Input: two strings $x = x_1 x_2 x_3 \ldots x_m$ and $y = y_1 y_2 \ldots y_n$.
- Indices $\{1, 2, \ldots, m\}$ and $\{1, 2, \ldots, n\}$ represent positions in x and y.

Edit Distance

```
o - c u r r a n c e     o - c u r r - a n c e
:   : : : : : : : :     :   : : : :   : : :
o c c u r r e n c e     o c c u r r e - n c e
```

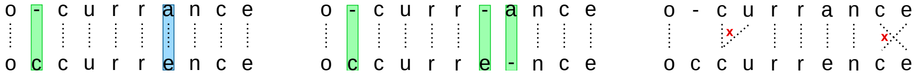
- Proposed by Needleman and Wunsch in the early 1970s.
- Input: two strings $x = x_1x_2x_3 \ldots x_m$ and $y = y_1y_2 \ldots y_n$.
- Indices $\{1, 2, \ldots, m\}$ and $\{1, 2, \ldots, n\}$ represent positions in x and y.
- A *matching* of x and y is a set M of ordered pairs such that
  1. in each pair (i, j), $1 \leq i \leq m$ and $1 \leq j \leq n$ and
  2. no index from x (respectively, from y) appears as the first (respectively, second) element in more than one ordered pair.

Edit Distance



- Proposed by Needleman and Wunsch in the early 1970s.
- Input: two strings $x = x_1 x_2 x_3 \dots x_m$ and $y = y_1 y_2 \dots y_n$.
- Indices $\{1, 2, \dots, m\}$ and $\{1, 2, \dots, n\}$ represent positions in x and y.
- A *matching* of x and y is a set M of ordered pairs such that
  1. in each pair (i, j), $1 \leq i \leq m$ and $1 \leq j \leq n$ and
  2. no index from x (respectively, from y) appears as the first (respectively, second) element in more than one ordered pair.
- A matching M is an *alignment* if there are no "crossing pairs" in M: if $(i, j) \in M$ and $(i', j') \in M$ and $i < i'$ then $j < j'$.

Edit Distance

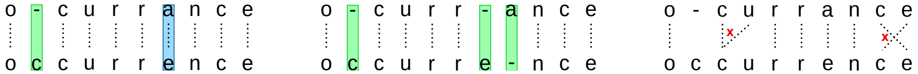


- Proposed by Needleman and Wunsch in the early 1970s.
- Input: two strings $x = x_1 x_2 x_3 \dots x_m$ and $y = y_1 y_2 \dots y_n$.
- Indices $\{1, 2, \dots, m\}$ and $\{1, 2, \dots, n\}$ represent positions in x and y.
- A *matching* of x and y is a set M of ordered pairs such that
  1. in each pair (i, j), $1 \le i \le m$ and $1 \le j \le n$ and
  2. no index from x (respectively, from y) appears as the first (respectively, second) element in more than one ordered pair.
- A matching M is an *alignment* if there are no "crossing pairs" in M: if $(i, j) \in M$ and $(i', j') \in M$ and $i < i'$ then $j < j'$.
- An index is *not matched* if it does not appear in the matching.
- *Cost* of an alignment is the sum of gap and mismatch penalties:

  Gap penalty   Penalty $\delta > 0$ for every unmatched index.

  Mismatch penalty   Penalty $\alpha_{x_i y_j} > 0$ if $(i, j) \in M$ and $x_i \neq y_j$.

Edit Distance



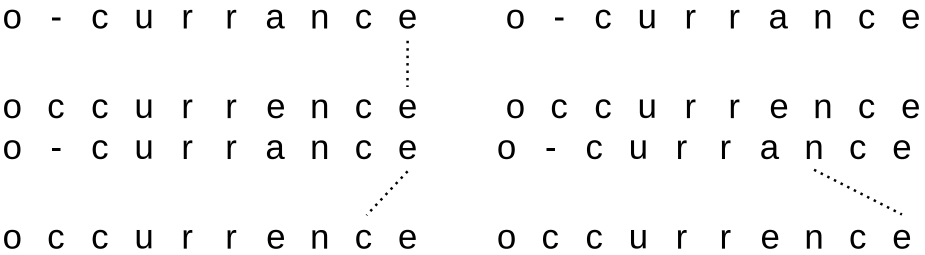
- Proposed by Needleman and Wunsch in the early 1970s.
- Input: two strings $x = x_1 x_2 x_3 \ldots x_m$ and $y = y_1 y_2 \ldots y_n$.
- Indices $\{1, 2, \ldots, m\}$ and $\{1, 2, \ldots, n\}$ represent positions in x and y.
- A *matching* of x and y is a set M of ordered pairs such that
  1. in each pair (i, j), $1 \leq i \leq m$ and $1 \leq j \leq n$ and
  2. no index from x (respectively, from y) appears as the first (respectively, second) element in more than one ordered pair.
- A matching M is an *alignment* if there are no "crossing pairs" in M: if $(i, j) \in M$ and $(i', j') \in M$ and $i < i'$ then $j < j'$.
- An index is *not matched* if it does not appear in the matching.
- *Cost* of an alignment is the sum of gap and mismatch penalties:
  - Gap penalty: Penalty $\delta > 0$ for every unmatched index.
  - Mismatch penalty: Penalty $\alpha_{x_i y_j} > 0$ if $(i, j) \in M$ and $x_i \neq y_j$.
- Output: compute an alignment of minimal cost.

Developing Intuition for Dynamic Programming

- How do we start formulating the dynamic program?
- Consider index $m \in x$ and index $n \in y$. What are the possibilities?

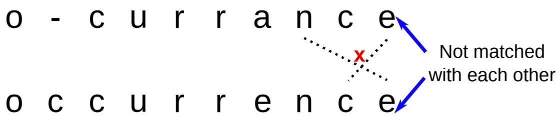
Developing Intuition for Dynamic Programming

- How do we start formulating the dynamic program?
- Consider index $m \in x$ and index $n \in y$. What are the possibilities?
  - (m, n) could be paired in the matching M.
  - Neither n nor m may be matched.
  - Only m may not be matched.
  - Only n may not be matched.



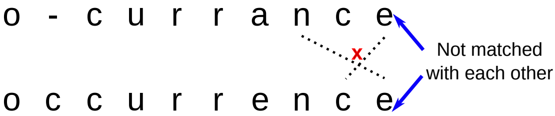
Developing Intuition for Dynamic Programming

- How do we start formulating the dynamic program?
- Consider index $m \in x$ and index $n \in y$. What are the possibilities?
  - (m, n) could be paired in the matching M.
  - Neither n nor m may be matched.
  - Only m may not be matched.
  - Only n may not be matched.
  - (m, i) could be paired and (j, n) could be paired where $i < n$ and $j < m$. Not possible in an alignment!

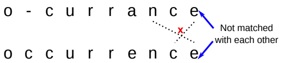


Developing Intuition for Dynamic Programming

- How do we start formulating the dynamic program?
- Consider index $m \in x$ and index $n \in y$. What are the possibilities?
  - (m, n) could be paired in the matching M.
  - Neither n nor m may be matched.
  - Only m may not be matched.
  - Only n may not be matched.
  - (m, i) could be paired and (j, n) could be paired where $i < n$ and $j < m$. Not possible in an alignment!
- Claim: $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.

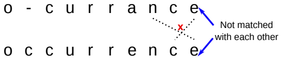


Dynamic Programming Approach



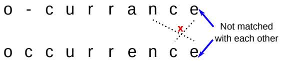
- $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.
- How should we define sub-problems?

Dynamic Programming Approach



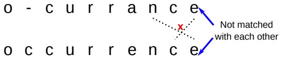
- $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.
- How should we define sub-problems?
- $OPT(i, j)$: cost of optimal alignment between $x = x_1 x_2 x_3 \ldots x_i$ and $y = y_1 y_2 \ldots y_j$.
  - $(i, j) \in M$:

Dynamic Programming Approach



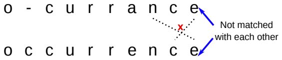
- $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.
- How should we define sub-problems?
- $OPT(i, j)$: cost of optimal alignment between $x = x_1 x_2 x_3 \ldots x_i$ and $y = y_1 y_2 \ldots y_j$.
  - $(i, j) \in M$: $\text{OPT}(i, j) = \alpha_{x_i y_j} + \text{OPT}(i - 1, j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

Dynamic Programming Approach



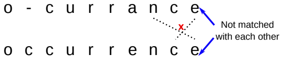
- $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.
- How should we define sub-problems?
- $OPT(i, j)$: cost of optimal alignment between $x = x_1 x_2 x_3 \ldots x_i$ and $y = y_1 y_2 \ldots y_j$.
  - $(i, j) \in M$: $\text{OPT}(i, j) = \alpha_{x_i y_j} + \text{OPT}(i - 1, j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

Dynamic Programming Approach



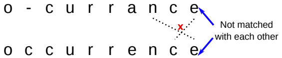
- $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.
- How should we define sub-problems?
- $OPT(i, j)$: cost of optimal alignment between $x = x_1 x_2 x_3 \ldots x_i$ and $y = y_1 y_2 \ldots y_j$.
  - $(i, j) \in M$: $\text{OPT}(i, j) = \alpha_{x_i y_j} + \text{OPT}(i - 1, j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

Dynamic Programming Approach



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

Dynamic Programming Approach

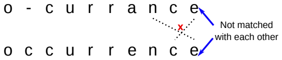


- $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.
- How should we define sub-problems?
- $OPT(i, j)$: cost of optimal alignment between $x = x_1 x_2 x_3 \ldots x_i$ and $y = y_1 y_2 \ldots y_j$.
  - $(i, j) \in M$: $\text{OPT}(i, j) = \alpha_{x_i y_j} + \text{OPT}(i-1, j-1)$.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

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

  - $(i, j) \in M$ if and only if minimum is achieved by the first term.
- What are the base cases?

Dynamic Programming Approach



- $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.
- How should we define sub-problems?
- $OPT(i, j)$: cost of optimal alignment between $x = x_1x_2x_3 \dots x_i$ and $y = y_1y_2 \dots y_j$.
  - $(i, j) \in M$: $\text{OPT}(i, j) = \alpha_{x_iy_j} + \text{OPT}(i-1, j-1)$.
  - i not matched: $\text{OPT}(i, j) = \delta + \text{OPT}(i-1, j)$.
  - j not matched: $\text{OPT}(i, j) = \delta + \text{OPT}(i, j-1)$.

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

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

Dynamic Programming Algorithm

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

Alignment(X,Y)
  Array $A[0\ldots m, 0\ldots n]$
  Initialize $A[i,0] = i\delta$ for each i
  Initialize $A[0,j] = j\delta$ for each j
  For $j = 1,\ldots,n$
    For $i = 1,\ldots,m$
      Use the recurrence (6.16) to compute $A[i,j]$
    Endfor
  Endfor
  Return $A[m,n]$

Dynamic Programming Algorithm

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

```
Alignment(X,Y)
  Array A[0...m, 0...n]
  Initialize A[i,0]=iδ for each i
  Initialize A[0,j]=jδ for each j
  For j=1,...,n
     For i=1,...,m
        Use the recurrence (6.16) to compute A[i,j]
     Endfor
  Endfor
  Return A[m,n]
```

- Running time is $O(mn)$. Space used in $O(mn)$.
- (i,j) is in the optimal alignment if the first term is the smallest.

Improving the Running Time

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

- Key observation: Computing entry (i,j) requires values only in previous row/column or in previous row in current column.

Improving the Running Time

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

- Key observation: Computing entry (i,j) requires values only in previous row/column or in previous row in current column.

```
Space-Efficient-Alignment(X,Y)
  Array B[0...m, 0...1]
  Initialize B[i,0] = iδ for each i (just as in column 0 of A)
  For j = 1,...,n
     B[0,1] = jδ (since this corresponds to entry A[0,j])
     For i = 1,...,m
         B[i,1] = min[α_{x_i y_j} + B[i-1,0],
                        δ + B[i-1,1],  δ + B[i,0]]
     Endfor
     Move column 1 of B to column 0 to make room for next iteration:
         Update B[i,0] = B[i,1] for each i
  Endfor
```

- Can compute $\text{OPT}(m,n)$ in $O(mn)$ time and $O(m+n)$ space.

Improving the Running Time

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

- Key observation: Computing entry (i,j) requires values only in previous row/column or in previous row in current column.

```
Space-Efficient-Alignment(X,Y)
  Array B[0...m,0...1]
  Initialize B[i,0]=iδ for each i (just as in column 0 of A)
  For j=1,...,n
     B[0,1]=jδ (since this corresponds to entry A[0,j])
     For i=1,...,m
         B[i,1]=min[α_{x_i y_j} + B[i-1,0],
                       δ + B[i-1,1],  δ + B[i,0]]
     Endfor
     Move column 1 of B to column 0 to make room for next iteration:
         Update B[i,0]=B[i,1] for each i
  Endfor
```

- Can compute $\text{OPT}(m,n)$ in $O(mn)$ time and $O(m+n)$ space.
- Problem: How do we compute matched pairs in the optimal alignment?

Improving the Running Time

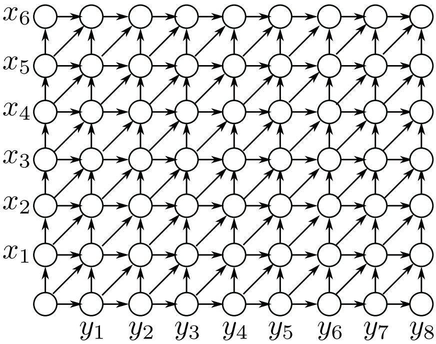
$$\mathrm{OPT}(i,j) = \min\left(\alpha_{x_i y_j} + \mathrm{OPT}(i-1,j-1), \delta + \mathrm{OPT}(i-1,j), \delta + \mathrm{OPT}(i,j-1)\right)$$

- Key observation: Computing entry (i,j) requires values only in previous row/column or in previous row in current column.

```
Space-Efficient-Alignment(X,Y)
  Array B[0...m, 0...1]
  Initialize B[i,0] = iδ for each i (just as in column 0 of A)
  For j = 1,...,n
     B[0,1] = jδ (since this corresponds to entry A[0,j])
     For i = 1,...,m
         B[i,1] = min[α_{x_i y_j} + B[i-1,0],
                               δ + B[i-1,1],  δ + B[i,0]]
     Endfor
     Move column 1 of B to column 0 to make room for next iteration:
         Update B[i,0] = B[i,1] for each i
  Endfor
```

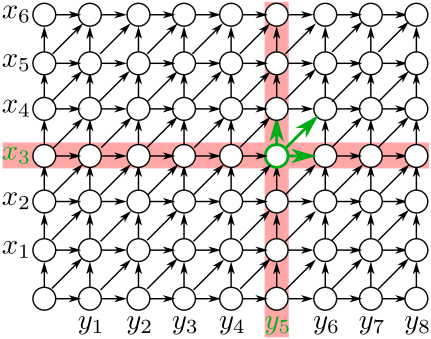
- Can compute $\mathrm{OPT}(m,n)$ in $O(mn)$ time and $O(m+n)$ space.
- Problem: How do we compute matched pairs in the optimal alignment? Requires new ideas: combine divide and conquer with dynamic programming!

Graph-theoretic View of Sequence Alignment



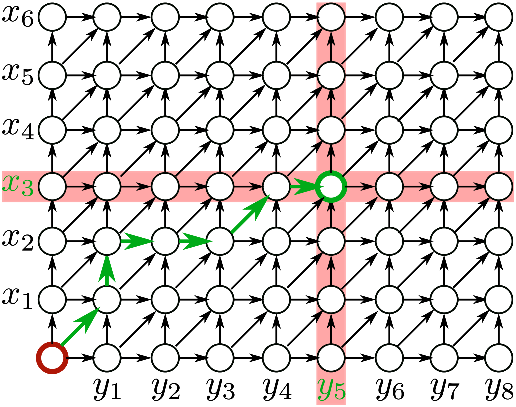
- Grid graph G_{xy}:
  - $m + 1$ rows numbered from 0 to m (corresponding to x).
  - $n + 1$ rows numbered from 0 to n (corresponding to y).
  - Rows labelled by symbols in x and columns labelled by symbols in y.

Graph-theoretic View of Sequence Alignment



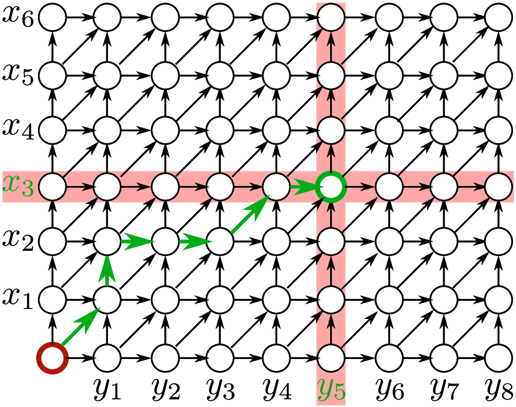
- Grid graph G_{xy}:
  - $m+1$ rows numbered from 0 to m (corresponding to x).
  - $n+1$ rows numbered from 0 to n (corresponding to y).
  - Rows labelled by symbols in x and columns labelled by symbols in y.
  - Node (i, j) has three outgoing edges to $(i, j+1)$), to $(i+1, j)$, and to $(i+1, j+1)$.
  - Edges directed upward or to the right have cost δ.
  - Edge directed from (i, j) to $(i+1, j+1)$ has cost $\alpha_{x_{i+1}y_{j+1}}$.

Shortest Paths in G_{xy}



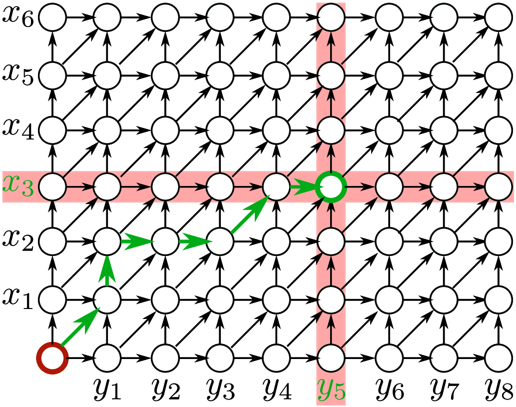
- For every i, j, $f(i, j) =$ minimum cost of a path in G_{XY} from $(0, 0)$ to (i, j).

Shortest Paths in G_{xy}



- For every i, j, $f(i,j) =$ minimum cost of a path in G_{XY} from $(0,0)$ to (i,j).
- Claim: $f(i,j) = \text{OPT}(i,j)$.

Shortest Paths in G_{xy}

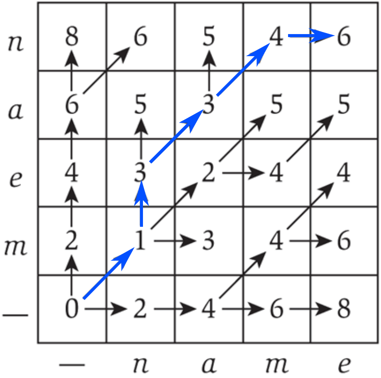


- For every i, j, $f(i,j) =$ minimum cost of a path in G_{XY} from $(0,0)$ to (i,j).
- Claim: $f(i,j) = \text{OPT}(i,j)$.
- Proof by induction on $i + j$: Use the fact that the last edge on the shortest path to (i,j) must be either from $(i-1, j-1)$, $(i-1, j)$ or $(i, j-1)$.

Shortest Paths in G_{xy}

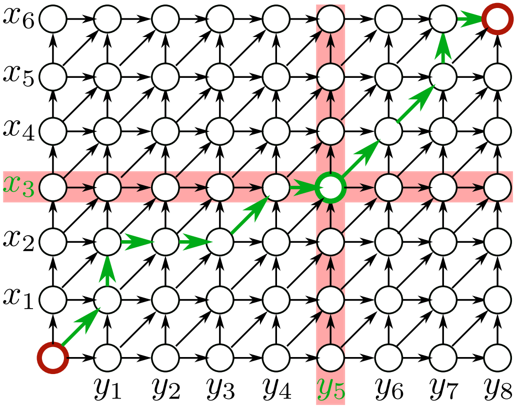
$\delta = 2$
$\alpha(\text{cons},\text{cons}) = 1$
$\alpha(\text{vow},\text{vow}) = 1$
$\alpha(\text{cons},\text{vow}) = 3$

```
m e a n -
n - a m e
```



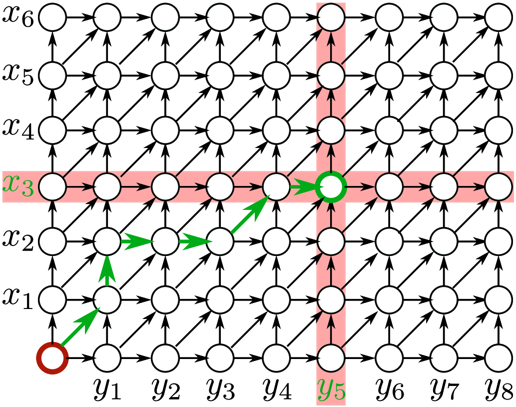
- For every i, j, $f(i,j) =$ minimum cost of a path in G_{XY} from $(0,0)$ to (i,j).
- Claim: $f(i,j) = \text{OPT}(i,j)$.
- Proof by induction on $i + j$: Use the fact that the last edge on the shortest path to (i,j) must be either from $(i-1, j-1)$, $(i-1, j)$ or $(i, j-1)$.
- Diagonal edges in the shortest path are the matched pairs in the alignment

Shortest Paths Through (i, j) in G_{xy}



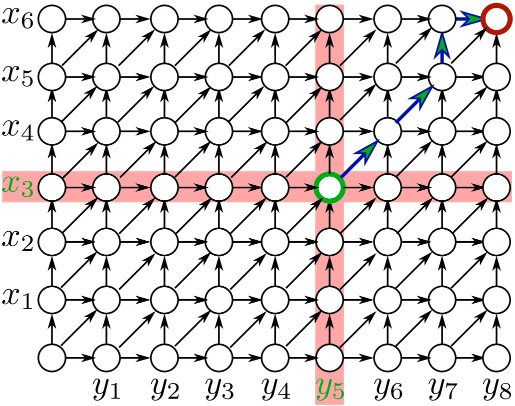
- *Corner-to-corner path*: path from $(0, 0)$ to (n, m).
- Given i and j, what is the length $l(i, j)$ of the shortest corner-to-corner path through (i, j)?

Shortest Paths Through (i, j) in G_{xy}



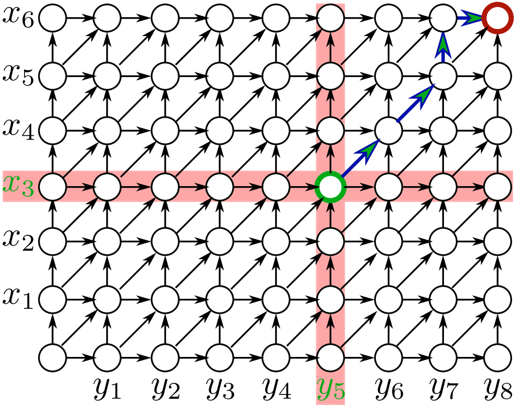
- *Corner-to-corner path*: path from $(0, 0)$ to (n, m).
- Given i and j, what is the length $l(i, j)$ of the shortest corner-to-corner path through (i, j)?
  - One segment is the shortest path from $(0, 0)$ to (i, j) with cost $f(i, j)$.

Shortest Paths Through (i, j) in G_{xy}



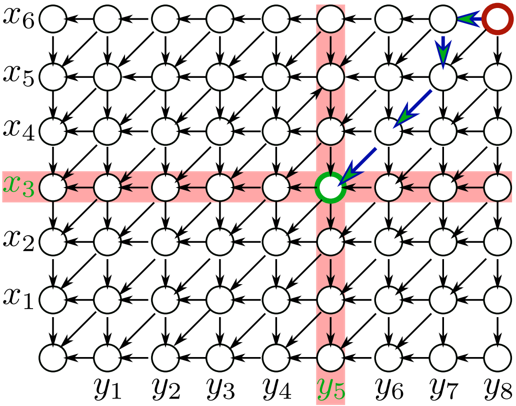
- *Corner-to-corner path*: path from $(0, 0)$ to (n, m).
- Given i and j, what is the length $l(i, j)$ of the shortest corner-to-corner path through (i, j)?
  - One segment is the shortest path from $(0, 0)$ to (i, j) with cost $f(i, j)$.
  - The other segment is the shortest path from (i, j) to (m, n) with some cost. How can we compute the cost of this path?

Shortest Paths Through (i, j) in G_{xy}



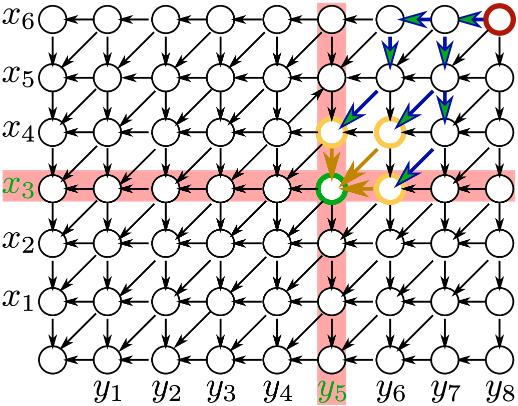
- Define $g(i, j)$ as cost of the shortest path from (i, j) to (m, n).

Shortest Paths Through (i, j) in G_{xy}



- Define $g(i, j)$ as cost of the shortest path from (i, j) to (m, n).

Shortest Paths Through (i, j) in G_{xy}

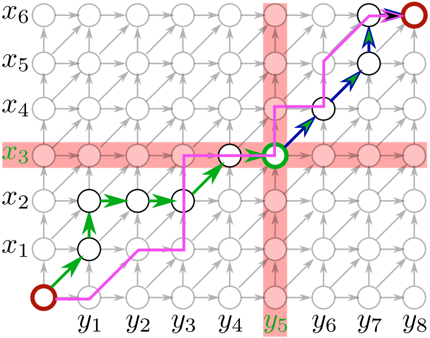


- Define $g(i, j)$ as cost of the shortest path from (i, j) to (m, n).

$$g(i, j) = \min\left(\alpha_{x_{i+1} y_{j+1}} + \text{OPT}(i+1, j+1), \delta + \text{OPT}(i+1, j), \delta + \text{OPT}(i, j+1)\right)$$

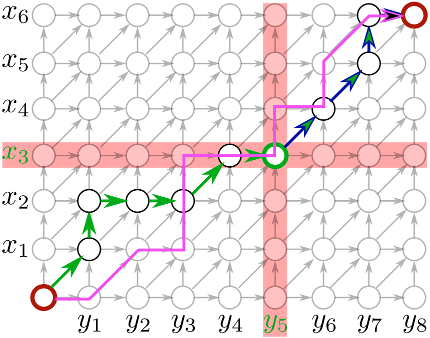
- We can compute $g(i, j)$ for every i and j in $O(mn)$ time and $O(m+n)$ space using `Backward-Space-Efficient-Alignment`.

Shortest Path Through (i, j) in G_{xy}



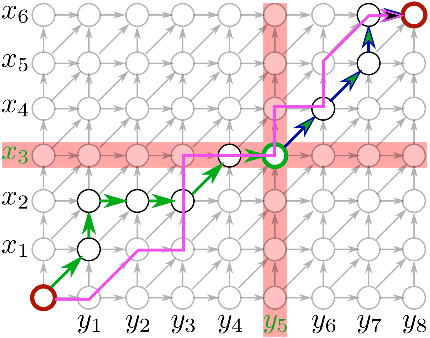
- Claim: $l(i, j) = f(i, j) + g(i, j)$.

Shortest Path Through (i, j) in G_{xy}



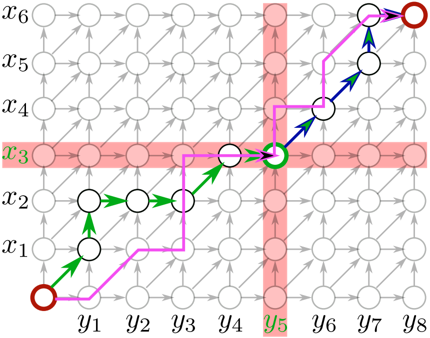
- Claim: $l(i,j) = f(i,j) + g(i,j)$.
- Shortest corner-to-corner path through (i,j) must go from $(0,0)$ to (i,j) and then from (i,j) to (m,n).

Shortest Path Through (i, j) in G_{xy}



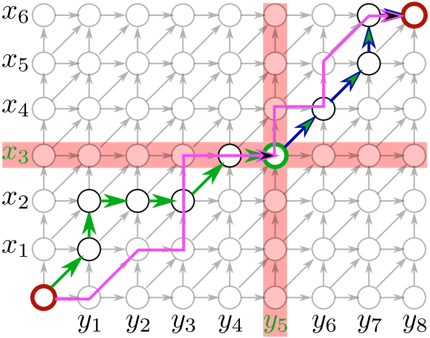
- Claim: $l(i,j) = f(i,j) + g(i,j)$.
- Shortest corner-to-corner path through (i, j) must go from $(0, 0)$ to (i, j) and then from (i, j) to (m, n).
- Therefore, $l(i,j) \geq f(i,j) + g(i,j)$.

Shortest Path Through (i, j) in G_{xy}



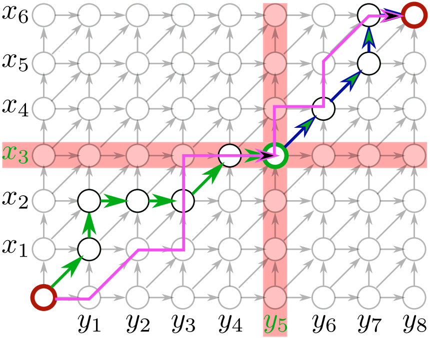
- Claim: $l(i, j) = f(i, j) + g(i, j)$.
- Shortest corner-to-corner path through (i, j) must go from $(0, 0)$ to (i, j) and then from (i, j) to (m, n).
- Therefore, $l(i, j) \geq f(i, j) + g(i, j)$.
- Now consider the following corner-to-corner path: Shortest path from $(0, 0)$ to (i, j) followed by the shortest path from (i, j) to (m, n). What is its length? .

Shortest Path Through (i, j) in G_{xy}



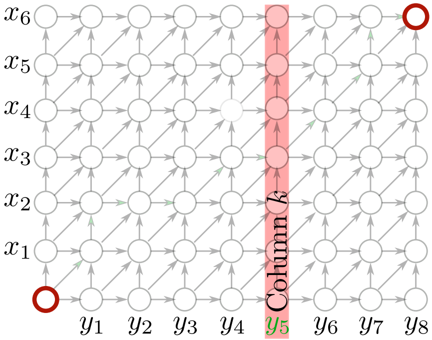
- Claim: $l(i, j) = f(i, j) + g(i, j)$.
- Shortest corner-to-corner path through (i, j) must go from $(0, 0)$ to (i, j) and then from (i, j) to (m, n).
- Therefore, $l(i, j) \geq f(i, j) + g(i, j)$.
- Now consider the following corner-to-corner path: Shortest path from $(0, 0)$ to (i, j) followed by the shortest path from (i, j) to (m, n). What is its length? $f(i, j) + g(i, j)$.

Shortest Path Through (i, j) in G_{xy}



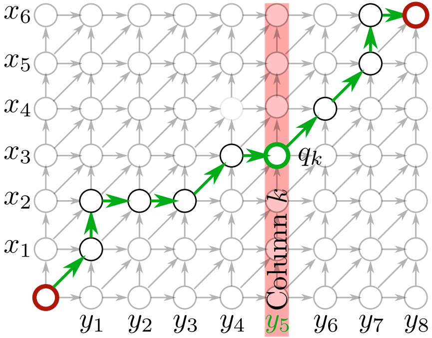
- Claim: $l(i, j) = f(i, j) + g(i, j)$.
- Shortest corner-to-corner path through (i, j) must go from $(0, 0)$ to (i, j) and then from (i, j) to (m, n).
- Therefore, $l(i, j) \geq f(i, j) + g(i, j)$.
- Now consider the following corner-to-corner path: Shortest path from $(0, 0)$ to (i, j) followed by the shortest path from (i, j) to (m, n). What is its length? $f(i, j) + g(i, j)$.
- Therefore, $l(i, j) \leq f(i, j) + g(i, j)$.

Shortest Paths Through Column k in G_{xy}



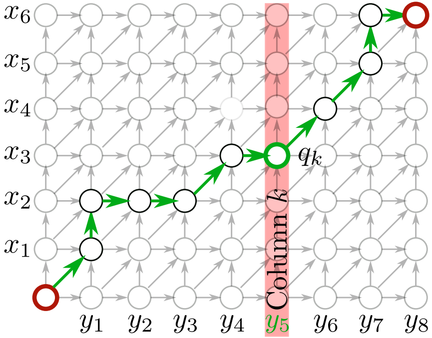
- Fix arbitrary k between 0 and n.
- Does the shortest corner-to-corner path pass through a node in column k?

Shortest Paths Through Column k in G_{xy}



- Fix arbitrary k between 0 and n.
- Does the shortest corner-to-corner path pass through a node in column k?
  - Yes, it must pass through exactly one such node, say (q_k, k).
  - How can we compute q_k given values of $f(i, k)$ and $g(i, k)$ for every i?

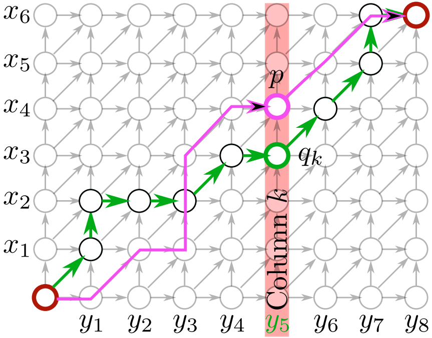
Shortest Paths Through Column k in G_{xy}



- Fix arbitrary k between 0 and n.
- Does the shortest corner-to-corner path pass through a node in column k?
  - Yes, it must pass through exactly one such node, say (q_k, k).
  - How can we compute q_k given values of $f(i, k)$ and $g(i, k)$ for every i?

$$q_k = \arg\min_{0 \leq i \leq m} \big(f(i, k) + g(i, k)\big)$$

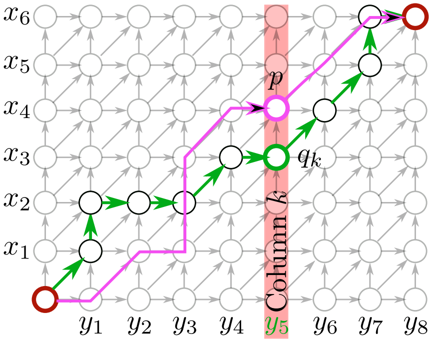
Shortest Paths Through Column k in G_{xy}



$$q_k = \arg\min_{0 \leq i \leq m} \left(f(i,k) + g(i,k) \right)$$

- Why should there be a shortest corner-to-corner path that passes through node (q_k, k)? Proof is very similar to previous proof.

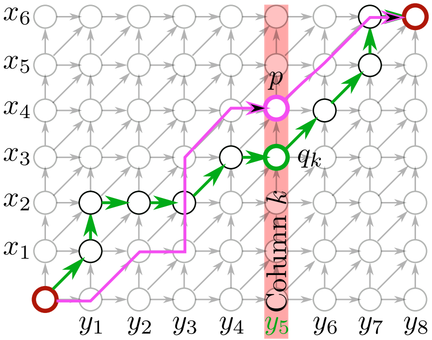
Shortest Paths Through Column k in G_{xy}



$$q_k = \arg\min_{0 \le i \le m} \big(f(i,k) + g(i,k)\big)$$

- Why should there be a shortest corner-to-corner path that passes through node (q_k, k)? Proof is very similar to previous proof.
  - Let l^* be the length of the shortest corner-to-corner path.

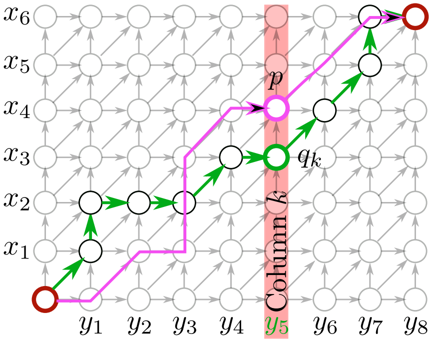
Shortest Paths Through Column k in G_{xy}



$$q_k = \arg \min_{0 \leq i \leq m} \left(f(i,k) + g(i,k) \right)$$

- Why should there be a shortest corner-to-corner path that passes through node (q_k, k)? Proof is very similar to previous proof.
  - Let l^* be the length of the shortest corner-to-corner path.
  - $l^* \leq f(q_k, k) + g(q_k, k)$. Why?

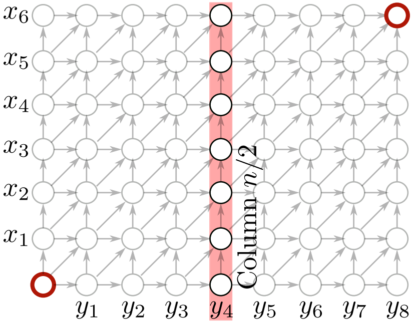
Shortest Paths Through Column k in G_{xy}



$$q_k = \arg\min_{0 \le i \le m} \left(f(i,k) + g(i,k) \right)$$

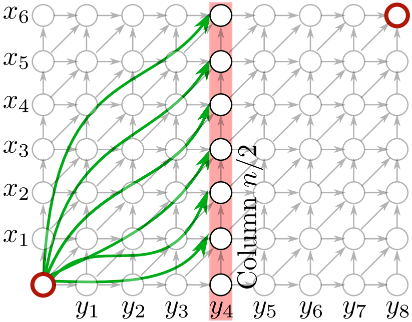
- Why should there be a shortest corner-to-corner path that passes through node (q_k, k)? Proof is very similar to previous proof.
  - Let l^* be the length of the shortest corner-to-corner path.
  - $l^* \le f(q_k, k) + g(q_k, k)$. Why?
  - Shortest corner-to-corner path must use some node p in column k. Therefore, $l^* = f(p,k) + g(p,k) \ge \min_{0 \le i \le m} \left(f(i,k) + g(i,k) \right) = f(q_k,k) + g(q_k,k)$.

Divide and Conquer Intuition



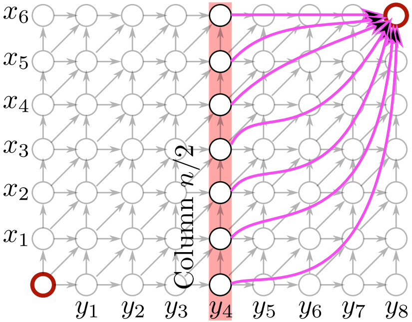
- Divide G_{xy} into two: columns 0 to $n/2$ and columns $n/2$ to n.
- Determine $q_{n/2}$.

Divide and Conquer Intuition



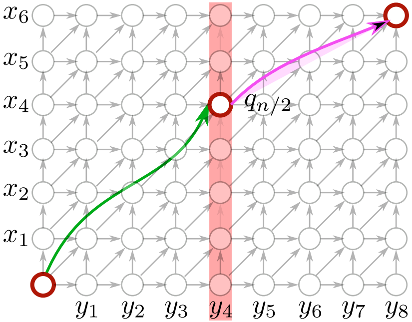
- Divide G_{xy} into two: columns 0 to $n/2$ and columns $n/2$ to n.
- Determine $q_{n/2}$.
  - Compute $f(i, n/2)$ for every i: use `Space-Efficient-Alignment`.

Divide and Conquer Intuition



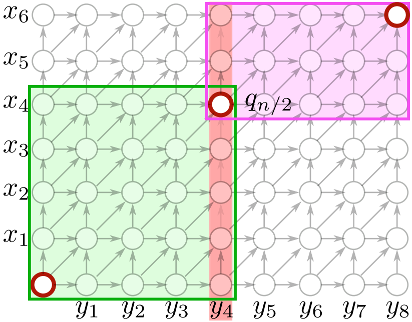
- Divide G_{xy} into two: columns 0 to $n/2$ and columns $n/2$ to n.
- Determine $q_{n/2}$.
  - Compute $f(i, n/2)$ for every i: use `Space-Efficient-Alignment`.
  - Compute $g(i, n/2)$ for every i: use `Backward-Space-Efficient-Alignment`.

Divide and Conquer Intuition



- Divide G_{xy} into two: columns 0 to $n/2$ and columns $n/2$ to n.
- Determine $q_{n/2}$.
  - Compute $f(i, n/2)$ for every i: use `Space-Efficient-Alignment`.
  - Compute $g(i, n/2)$ for every i: use `Backward-Space-Efficient-Alignment`.
  - Compute $q_{n/2}$. All these steps take $O(mn)$ time and $O(m + n)$ space.
- Store $(q_{n/2}, n/2)$ in a global list. There must be a shortest corner-to-corner path through this node.

Divide and Conquer Intuition



- Divide G_{xy} into two: columns 0 to $n/2$ and columns $n/2$ to n.
- Determine $q_{n/2}$.
  - Compute $f(i, n/2)$ for every i: use `Space-Efficient-Alignment`.
  - Compute $g(i, n/2)$ for every i: use `Backward-Space-Efficient-Alignment`.
  - Compute $q_{n/2}$. All these steps take $O(mn)$ time and $O(m + n)$ space.
- Store $(q_{n/2}, n/2)$ in a global list. There must be a shortest corner-to-corner path through this node.
- Recursively compute the nodes in the shortest path from $(0, 0)$ to $(q_{n/2}, n)$.
- Recursively compute the nodes in the shortest path from $(0, 0)$ to $(q_{n/2}, n)$.

Divide and Conquer Algorithm

```
Divide-and-Conquer-Alignment(X,Y)
  Let m be the number of symbols in X
  Let n be the number of symbols in Y
  If m ≤ 2 or n ≤ 2 then
     Compute optimal alignment using Alignment(X,Y)
  Call Space-Efficient-Alignment(X,Y[1:n/2])
  Call Backward-Space-Efficient-Alignment(X,Y[n/2+1:n])
  Let q be the index minimizing f(q,n/2)+g(q,n/2)
  Add (q,n/2) to global list P
  Divide-and-Conquer-Alignment(X[1:q],Y[1:n/2])
  Divide-and-Conquer-Alignment(X[q+1:n],Y[n/2+1:n])
  Return P
```

Running Time of Divide and Conquer Algorithm

Divide-and-Conquer-Alignment(X,Y)
  Let m be the number of symbols in X
  Let n be the number of symbols in Y
  If $m \le 2$ or $n \le 2$ then
    Compute optimal alignment using Alignment(X,Y)
  Call Space-Efficient-Alignment$(X,Y[1:n/2])$
  Call Backward-Space-Efficient-Alignment$(X,Y[n/2+1:n])$
  Let q be the index minimizing $f(q,n/2)+g(q,n/2)$
  Add $(q,n/2)$ to global list P
  Divide-and-Conquer-Alignment$(X[1:q],Y[1:n/2])$
  Divide-and-Conquer-Alignment$(X[q+1:n],Y[n/2+1:n])$
  Return P

Running Time of Divide and Conquer Algorithm

```
Divide-and-Conquer-Alignment(X,Y)
  Let m be the number of symbols in X
  Let n be the number of symbols in Y
  If m ≤ 2 or n ≤ 2 then
     Compute optimal alignment using Alignment(X,Y)
  Call Space-Efficient-Alignment(X,Y[1:n/2])
  Call Backward-Space-Efficient-Alignment(X,Y[n/2+1:n])
  Let q be the index minimizing f(q,n/2)+g(q,n/2)
  Add (q,n/2) to global list P
  Divide-and-Conquer-Alignment(X[1:q],Y[1:n/2])
  Divide-and-Conquer-Alignment(X[q+1:n],Y[n/2+1:n])
  Return P
```

- Let $T(m, n)$ be the worst-case running time of this algorithm on strings on input m and n, respectively.

Running Time of Divide and Conquer Algorithm

```
Divide-and-Conquer-Alignment(X,Y)
  Let m be the number of symbols in X
  Let n be the number of symbols in Y
  If m ≤ 2 or n ≤ 2 then
     Compute optimal alignment using Alignment(X,Y)
  Call Space-Efficient-Alignment(X,Y[1:n/2])
  Call Backward-Space-Efficient-Alignment(X,Y[n/2+1:n])
  Let q be the index minimizing f(q,n/2)+g(q,n/2)
  Add (q,n/2) to global list P
  Divide-and-Conquer-Alignment(X[1:q],Y[1:n/2])
  Divide-and-Conquer-Alignment(X[q+1:n],Y[n/2+1:n])
  Return P
```

- Let $T(m, n)$ be the worst-case running time of this algorithm on strings on input m and n, respectively.

$$T(m, n) \leq T(q, n/2) + T(m - q, n/2) + cmn$$
$$T(m, 2) \leq cm$$
$$T(2, n) \leq cn$$

Running Time of Divide and Conquer Algorithm

```
Divide-and-Conquer-Alignment(X,Y)
  Let m be the number of symbols in X
  Let n be the number of symbols in Y
  If m ≤ 2 or n ≤ 2 then
     Compute optimal alignment using Alignment(X,Y)
  Call Space-Efficient-Alignment(X,Y[1:n/2])
  Call Backward-Space-Efficient-Alignment(X,Y[n/2+1:n])
  Let q be the index minimizing f(q,n/2)+g(q,n/2)
  Add (q,n/2) to global list P
  Divide-and-Conquer-Alignment(X[1:q],Y[1:n/2])
  Divide-and-Conquer-Alignment(X[q+1:n],Y[n/2+1:n])
  Return P
```

- Let $T(m, n)$ be the worst-case running time of this algorithm on strings on input m and n, respectively.
- Challenges: Function of both m and n and q depends on the input.

$$T(m,n) \leq T(q, n/2) + T(m-q, n/2) + cmn$$
$$T(m,2) \leq cm$$
$$T(2,n) \leq cn$$

Solving the Recurrence

- Consider a special case first. Assume $n = m$ and $q = m/2$.

$$T(n) \leq 2T(n/2) + cn^2$$
$$T(2) \leq 2c$$

Solving the Recurrence

- Consider a special case first. Assume $n = m$ and $q = m/2$.

$$T(n) \leq 2T(n/2) + cn^2$$
$$T(2) \leq 2c$$

- We can prove by induction that $T(n) = O(n^2)$.

Solving the Recurrence

- Consider a special case first. Assume $n = m$ and $q = m/2$.

$$T(n) \leq 2T(n/2) + cn^2$$
$$T(2) \leq 2c$$

- We can prove by induction that $T(n) = O(n^2)$.
- Therefore, let us guess that the original recurrence has the solution $T(m, n) \leq kmn$.

Solving the Recurrence

- Consider a special case first. Assume $n = m$ and $q = m/2$.

$$T(n) \leq 2T(n/2) + cn^2$$
$$T(2) \leq 2c$$

- We can prove by induction that $T(n) = O(n^2)$.
- Therefore, let us guess that the original recurrence has the solution $T(m, n) \leq kmn$.
  - Base case: $m \leq 2, n \leq 2$. Holds if $k \geq c/2$.

Solving the Recurrence

- Consider a special case first. Assume $n = m$ and $q = m/2$.

$$T(n) \leq 2T(n/2) + cn^2$$
$$T(2) \leq 2c$$

- We can prove by induction that $T(n) = O(n^2)$.
- Therefore, let us guess that the original recurrence has the solution $T(m, n) \leq kmn$.
  - Base case: $m \leq 2, n \leq 2$. Holds if $k \geq c/2$.
  - Inductive hypothesis:

Solving the Recurrence

- Consider a special case first. Assume $n = m$ and $q = m/2$.

$$T(n) \le 2T(n/2) + cn^2$$
$$T(2) \le 2c$$

- We can prove by induction that $T(n) = O(n^2)$.
- Therefore, let us guess that the original recurrence has the solution $T(m, n) \le kmn$.
  - Base case: $m \le 2, n \le 2$. Holds if $k \ge c/2$.
  - Inductive hypothesis: For every $m' < m, n' < n$, $T(m', n') \le km'n'$.

Solving the Recurrence

- Consider a special case first. Assume $n = m$ and $q = m/2$.

$$T(n) \leq 2T(n/2) + cn^2$$
$$T(2) \leq 2c$$

- We can prove by induction that $T(n) = O(n^2)$.
- Therefore, let us guess that the original recurrence has the solution $T(m,n) \leq kmn$.
  - Base case: $m \leq 2, n \leq 2$. Holds if $k \geq c/2$.
  - Inductive hypothesis: For every $m' < m, n' < n$, $T(m', n') \leq km'n'$.
  - Inductive step: Need to prove $T(m,n) \leq kmn$.

Solving the Recurrence

- Consider a special case first. Assume $n = m$ and $q = m/2$.

$$T(n) \leq 2T(n/2) + cn^2$$
$$T(2) \leq 2c$$

- We can prove by induction that $T(n) = O(n^2)$.
- Therefore, let us guess that the original recurrence has the solution $T(m, n) \leq kmn$.
  - Base case: $m \leq 2, n \leq 2$. Holds if $k \geq c/2$.
  - Inductive hypothesis: For every $m' < m, n' < n$, $T(m', n') \leq km'n'$.
  - Inductive step: Need to prove $T(m, n) \leq kmn$.

$$\begin{aligned} T(m, n) &\leq T(q, n/2) + T(m - q, n/2) + cmn, \text{ from the recurrence} \\ &\leq kqn/2 + k(m - q)n/2 + cmn, \text{ from the inductive hypothesis} \\ &\leq (k/2 + c)mn \\ &\leq kmn, \text{ if } k \geq 2c. \end{aligned}$$

Analysing the Space Used by the Algorithm

```
Divide-and-Conquer-Alignment(X,Y)
  Let m be the number of symbols in X
  Let n be the number of symbols in Y
  If m ≤ 2 or n ≤ 2 then
     Compute optimal alignment using Alignment(X,Y)
  Call Space-Efficient-Alignment(X,Y[1:n/2])
  Call Backward-Space-Efficient-Alignment(X,Y[n/2+1:n])
  Let q be the index minimizing f(q,n/2)+g(q,n/2)
  Add (q,n/2) to global list P
  Divide-and-Conquer-Alignment(X[1:q],Y[1:n/2])
  Divide-and-Conquer-Alignment(X[q+1:n],Y[n/2+1:n])
  Return P
```

- At most one call to `Space-Efficient-Alignment` or `Backward-Space-Efficient-Alignment` executing at any time.
- Input to any invocation of these procedures has size at most $m + n$.
- Size of P is at most n.
- Therefore, total space used is $O(m + n)$.

Motivation

- Computational finance:
  - Each node is a financial agent.
  - The cost c_{uv} of an edge (u, v) is the cost of a transaction in which we buy from agent u and sell to agent v.
  - Negative cost corresponds to a profit.
- Internet routing protocols
  - Dijkstra's algorithm needs knowledge of the entire network.
  - Routers only know which other routers they are connected to.
  - Algorithm for shortest paths with negative edges is decentralised.
  - We will not study this algorithm in the class. See Chapter 6.9.

Problem Statement

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

Problem Statement

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

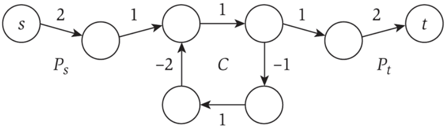


Figure 6.20 In this graph, one can find s-t paths of arbitrarily negative cost (by going around the cycle C many times).

Approaches for Shortest Path Algorithm

1. Dijsktra's algorithm.

2. Add some large constant to each edge.

Approaches for Shortest Path Algorithm

1. 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.

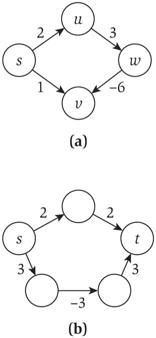


Figure 6.21 (a) With negative edge costs, Dijkstra's Algorithm can give the wrong answer for the Shortest-Path Problem. (b) Adding 3 to the cost of each edge will make all edges nonnegative, but it will change the identity of the shortest *s*-*t* path.

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 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

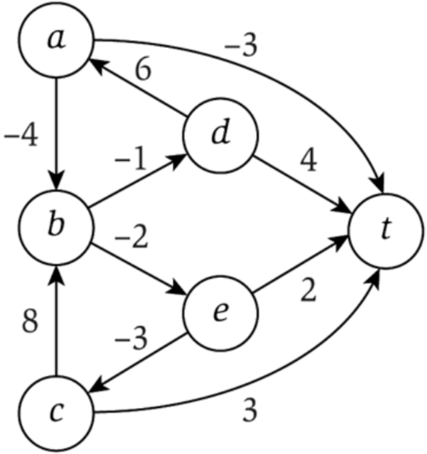
- Assume G has no negative cycles.
- Claim: There is a shortest path from s to t that is *simple* (does not repeat a node) and hence has at most $n-1$ edges.

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

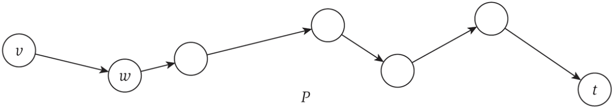


Figure 6.22 The minimum-cost path P from v to t using at most i edges.

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

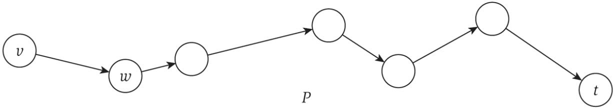


Figure 6.22 The minimum-cost path P from v to t using at most i edges.

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

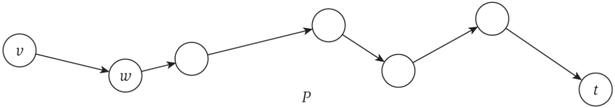


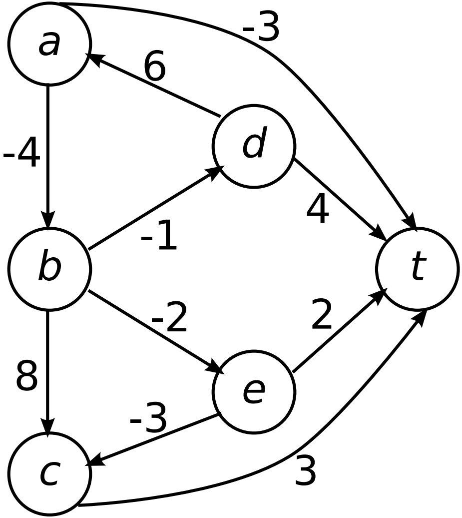
Figure 6.22 The minimum-cost path P from v to t using at most i edges.

- 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

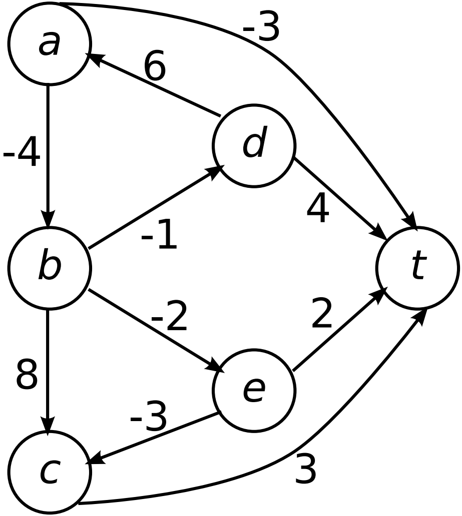
$$\mathrm{OPT}(i, v) = \min\left(\mathrm{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \mathrm{OPT}(i-1, w)\right)\right)$$



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

Example of Dynamic Programming Recursion

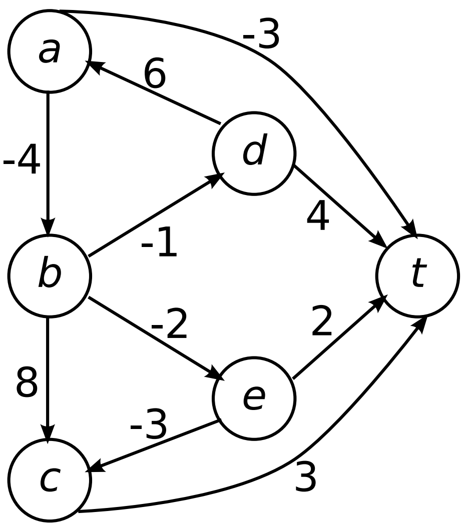
$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$



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

Example of Dynamic Programming Recursion

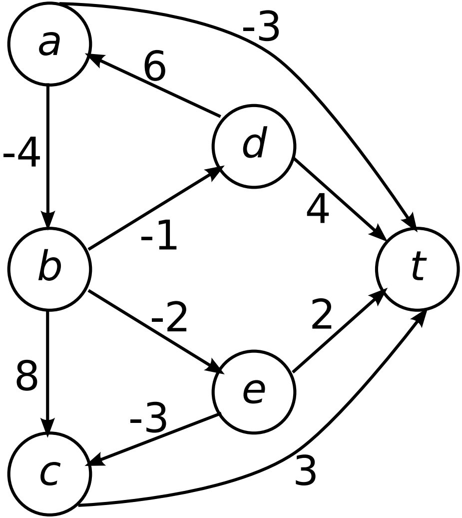
$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$



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

Example of Dynamic Programming Recursion

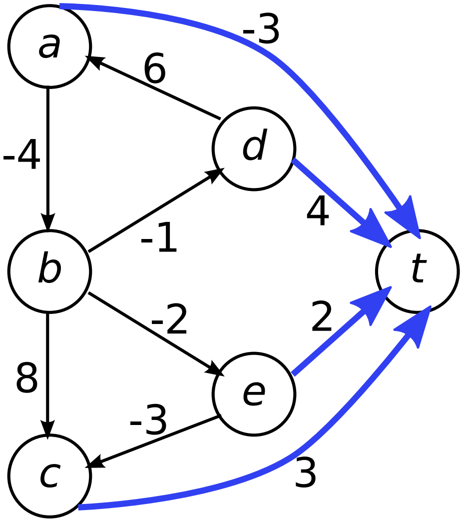
$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$



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

Example of Dynamic Programming Recursion

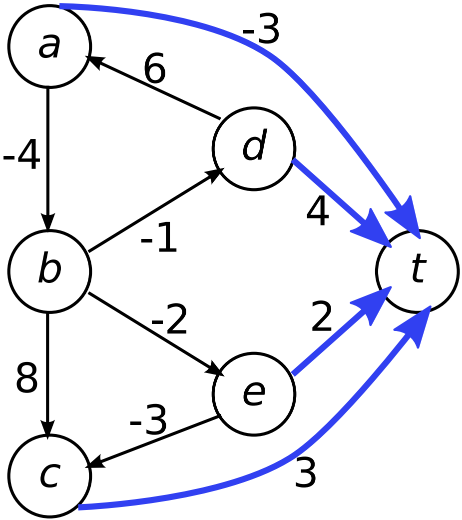
$$\text{OPT}(i, v) = \min \left(\text{OPT}(i-1, v), \min_{w \in V} \left(c_{vw} + \text{OPT}(i-1, w) \right) \right)$$



	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

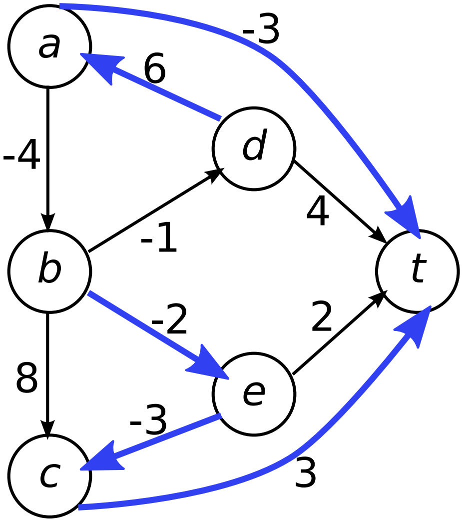
$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$



	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

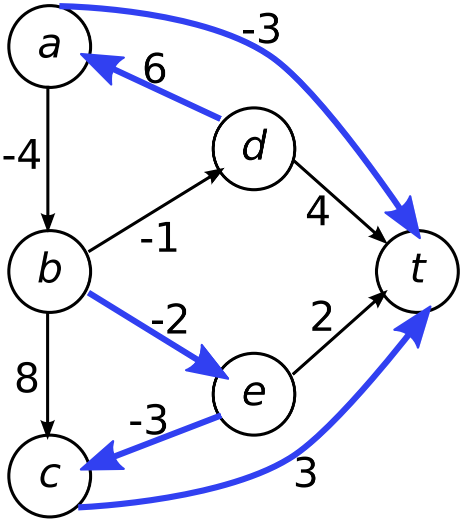
$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$



	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

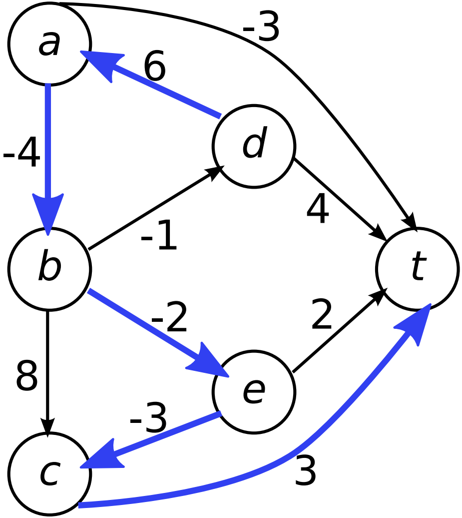
$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$



	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

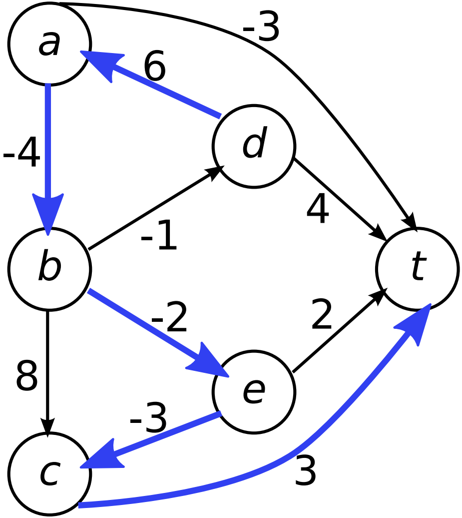
$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$



	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

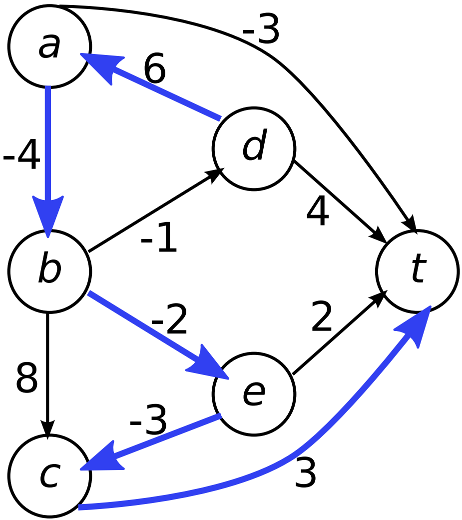
$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$



	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

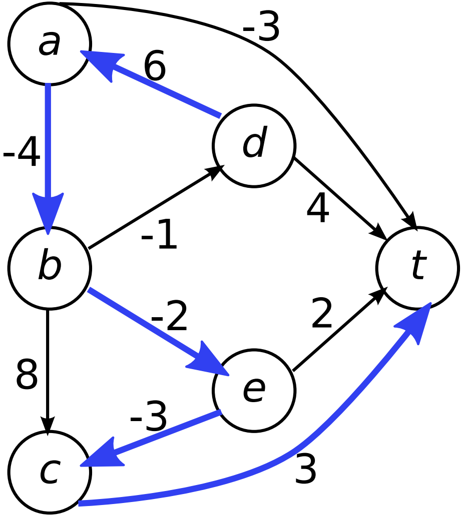
$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$



	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

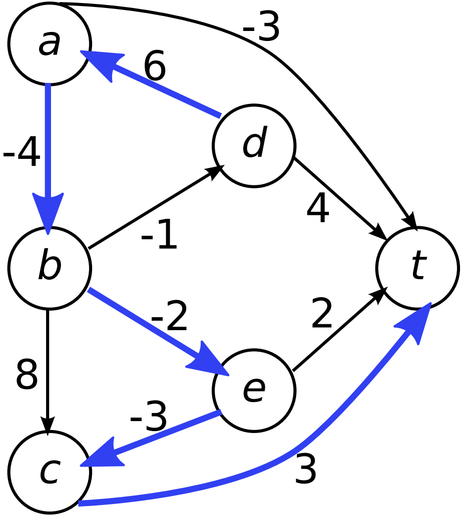
$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$



	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

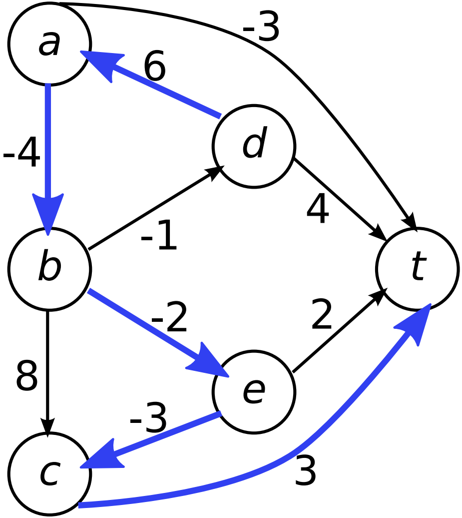
$$\text{OPT}(i, v) = \min\left(\text{OPT}(i-1, v), \min_{w \in V}\left(c_{vw} + \text{OPT}(i-1, w)\right)\right)$$



	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

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

- *$\text{OPT}_=(i, v)$*: 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 $\text{OPT}_=(\text{i, v})$.

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

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

```
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]
```

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 $\text{OPT}(i-1,v)$ for all nodes $v \in V$.
- Space used is $O(n)$.

Computing the Shortest Path: Algorithm

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

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

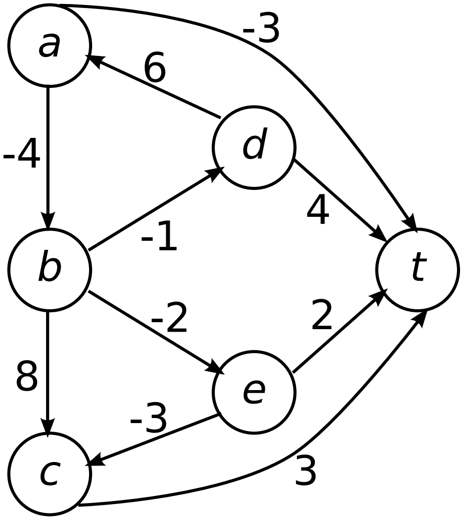
Computing the Shortest Path: Algorithm

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

- 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 $\min_{w \in N_v} \left(c_{vw} + M'[w]\right)$, set
  - $M[v] = c_{vx} + M'[x]$ and
  - $f(v) = x$.
- At the end, follow $f(v)$ pointers from s to t.

Example of Maintaining Pointers

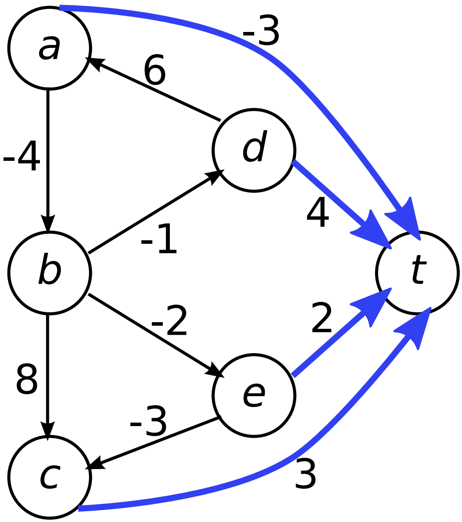
$$M[v] = \min\left(M'[v], \min_{w \in N_v}\left(c_{vw} + M'[w]\right)\right)$$



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

Example of Maintaining Pointers

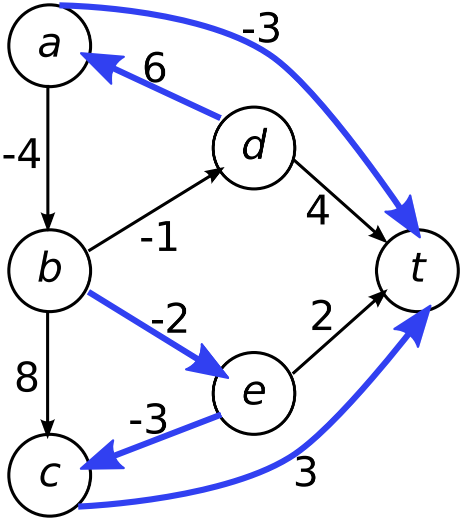
$$M[v] = \min\left(M'[v], \min_{w \in N_v}\left(c_{vw} + M'[w]\right)\right)$$



	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

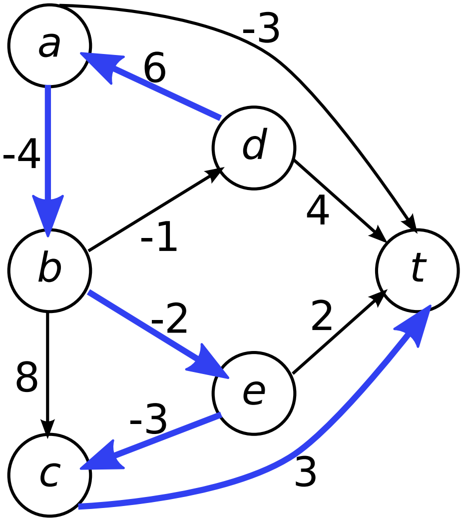
$$M[v] = \min\left(M'[v], \min_{w \in N_v}\left(c_{vw} + M'[w]\right)\right)$$



	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

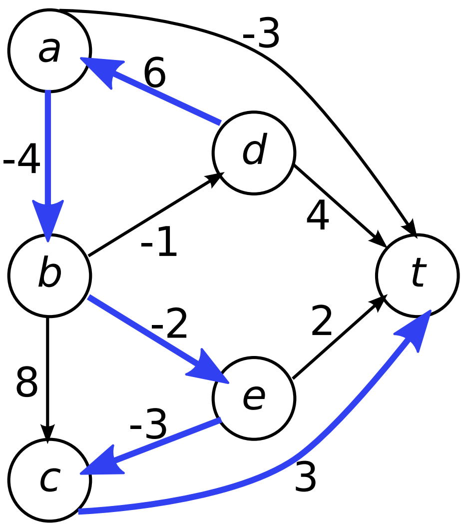
$$M[v] = \min\left(M'[v], \min_{w \in N_v}\left(c_{vw} + M'[w]\right)\right)$$



	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

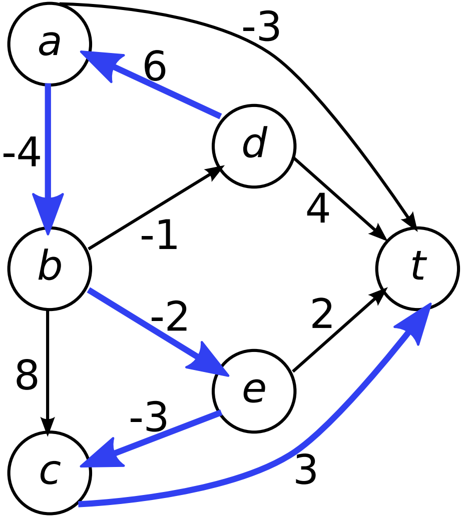
$$M[v] = \min\left(M'[v], \min_{w \in N_v}\left(c_{vw} + M'[w]\right)\right)$$



	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

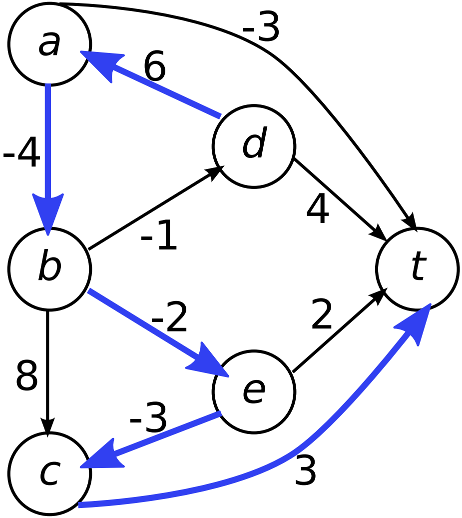
$$M[v] = \min \left(M'[v], \min_{w \in N_v} \left(c_{vw} + M'[w] \right) \right)$$



	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

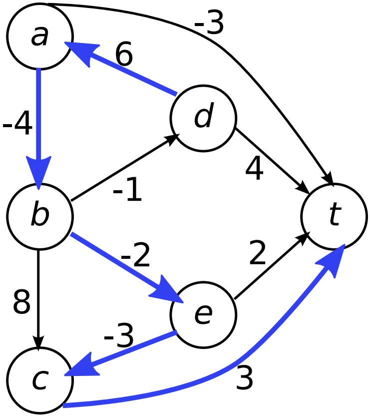
$$M[v] = \min\left(M'[v], \min_{w \in N_v}\left(c_{vw} + M'[w]\right)\right)$$



	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

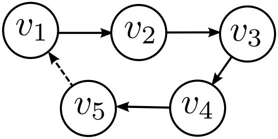
Computing the Shortest Path: Correctness

- *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

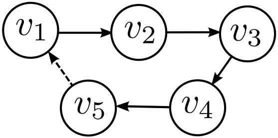
Computing the Shortest Path: Cycles in P



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

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

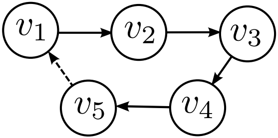
Computing the Shortest Path: Cycles in P



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

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

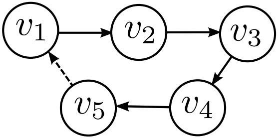
Computing the Shortest Path: Cycles in P



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

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

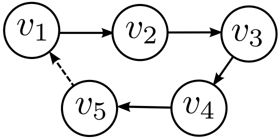
Computing the Shortest Path: Cycles in P



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

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

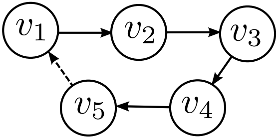
Computing the Shortest Path: Cycles in P



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

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

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.

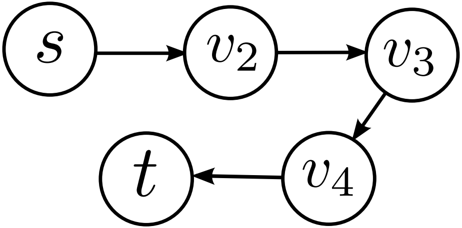
Bellman-Ford Algorithm: One Array

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

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

Bellman-Ford Algorithm: Early Termination

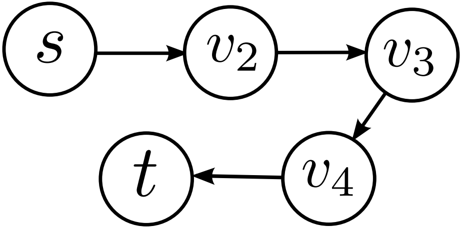
$$M[v] = \min\left(M[v], \min_{w \in N_v}\left(c_{vw} + M[w]\right)\right)$$



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

Bellman-Ford Algorithm: Early Termination

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



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