Dynamic Programming

Textbook Reading

Chapters 15, 24 & 25

Overview

Design principle

- Recursively break the problem into smaller subproblems.
- Avoid repeatedly solving the same subproblems by caching their solutions.

Important tool

- Recurrence relations

Problems

- Weighted interval scheduling
- Sequence alignment
- Optimal binary search trees
- Shortest paths

Weighted Interval Scheduling

Given:

A set of activities competing for time intervals on a certain resource
(E.g., classes to be scheduled competing for a classroom)

Goal:

Schedule non-conflicting activities so that the total time the resource is in use is maximized.

W. I. S.: A Naïve Solution

- Try all possible subsets.
- Check each subset for conflicts.
- Out of the non-conflicting ones, remember the one with maximal total length.

W. I. S.: A Naïve Solution

- Try all possible subsets.
- Check each subset for conflicts.
- Out of the non-conflicting ones, remember the one with maximal total length.

Cost:

W. I. S.: A Naïve Solution

- Try all possible subsets.
- Check each subset for conflicts.
- Out of the non-conflicting ones, remember the one with maximal total length.

Cost: $O(2^n \cdot n^2)$

W. I. S.: Towards a Better Solution

General idea:

- Try to make one choice at a time, just as in a greedy algorithm.
- In each step, what are the options we can choose from?
- What can we say about the subproblem we obtain after choosing each option?

W. I. S.: Towards a Better Solution

General idea:

- Try to make one choice at a time, just as in a greedy algorithm.
- In each step, what are the options we can choose from?
- What can we say about the subproblem we obtain after choosing each option?

What options do we have?

W. I. S.: Towards a Better Solution

General idea:

- Try to make one choice at a time, just as in a greedy algorithm.
- In each step, what are the options we can choose from?
- What can we say about the subproblem we obtain after choosing each option?

What options do we have?

An interval is in the optimal solution or it isn't.

W. I. S.: Towards a Better Solution

General idea:

- Try to make one choice at a time, just as in a greedy algorithm.
- In each step, what are the options we can choose from?
- What can we say about the subproblem we obtain after choosing each option?

What options do we have?

An interval is in the optimal solution or it isn't.

Towards a recurrence for the cost of an optimal solution:

W. I. S.: Towards a Better Solution

General idea:

- Try to make one choice at a time, just as in a greedy algorithm.
- In each step, what are the options we can choose from?
- What can we say about the subproblem we obtain after choosing each option?

What options do we have?

An interval is in the optimal solution or it isn't.

Towards a recurrence for the cost of an optimal solution:

If the maximal-length subset of $\{I_1, I_2, \ldots, I_n\}$ does not include I_n, then it must be a maximal-length subset of $\{I_1, I_2, \ldots, I_{n-1}\}$.

W. I. S.: Towards a Better Solution

General idea:

- Try to make one choice at a time, just as in a greedy algorithm.
- In each step, what are the options we can choose from?
- What can we say about the subproblem we obtain after choosing each option?

What options do we have?

An interval is in the optimal solution or it isn't.

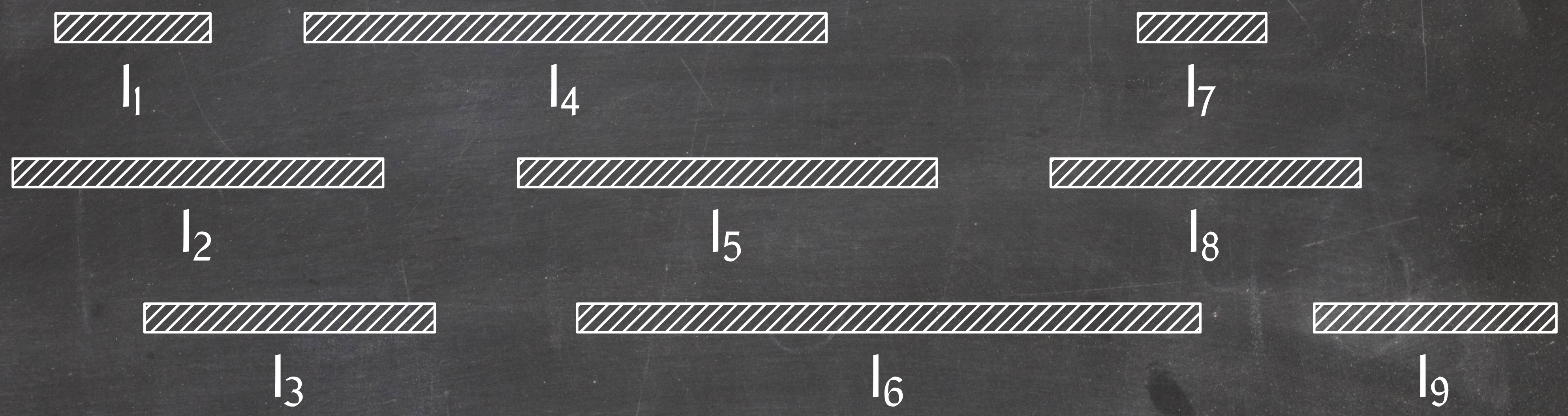
Towards a recurrence for the cost of an optimal solution:

If the maximal-length subset of $\{I_1, I_2, \dots, I_n\}$ does not include I_n, then it must be a maximal-length subset of $\{I_1, I_2, \dots, I_{n-1}\}$.

If the maximal-length subset of $\{I_1, I_2, \dots, I_n\}$ includes I_n, then it must be $O \cup \{I_n\}$, where O is a maximal-length subset of all intervals in $\{I_1, I_2, \dots, I_n\}$ that do not overlap I_n.

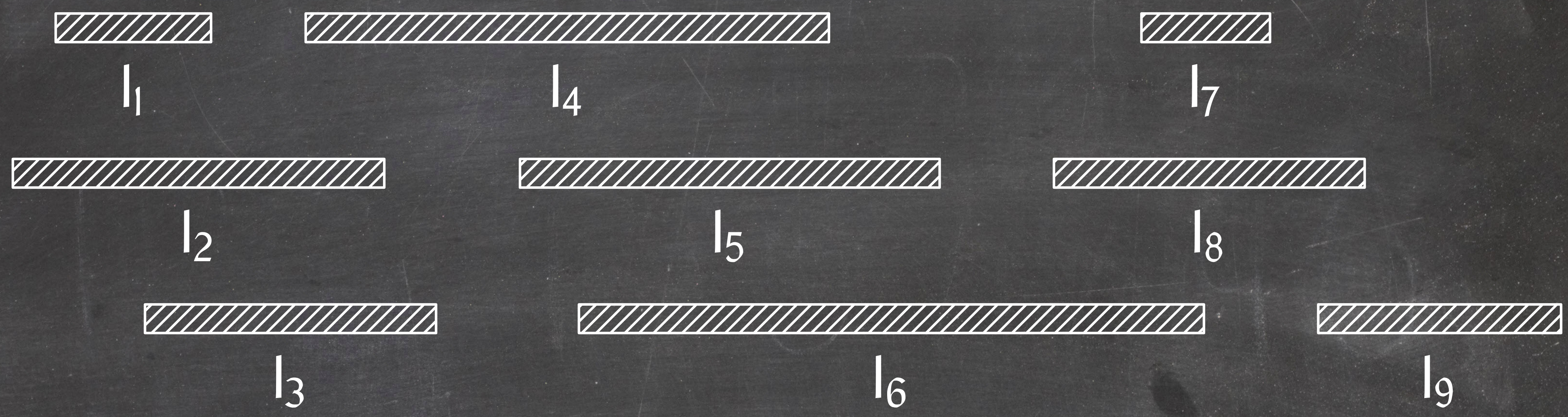
W. I. S.: Cleaning Up the Model

Number the intervals by increasing ending times:



W. I. S.: Cleaning Up the Model

Number the intervals by increasing ending times:

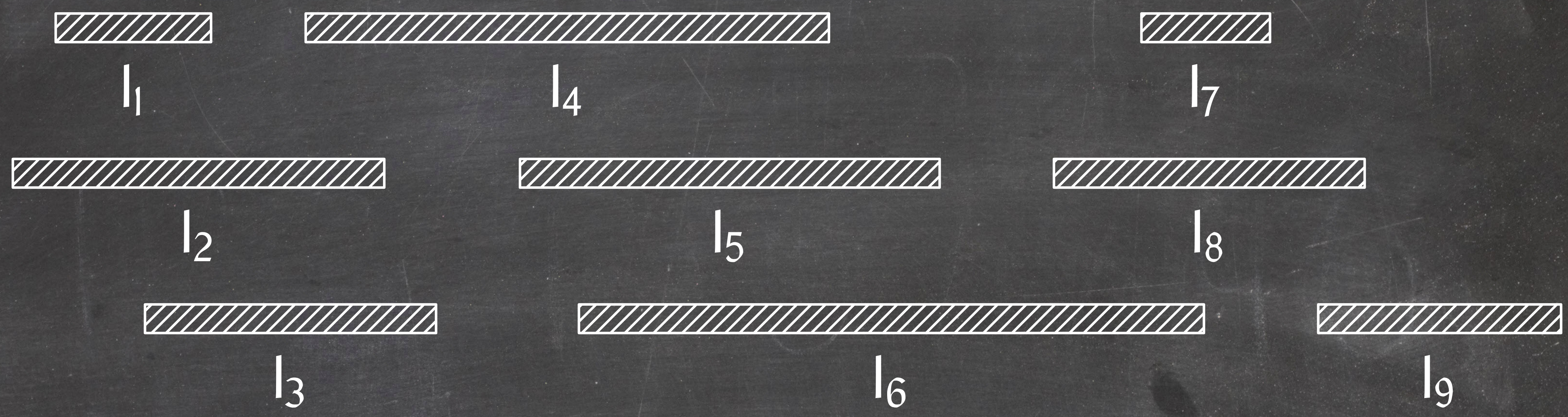


For $1 \leq j \leq n$, let $p_j = \max(\{0\} \cup \{k \mid 1 \leq k < j \text{ and } I_k \text{ does not overlap } I_j\})$.

j	1	2	3	4	5	6	7	8	9
p_j	0	0	0	1	3	3	5	5	7

W. I. S.: Cleaning Up the Model

Number the intervals by increasing ending times:



For $1 \leq j \leq n$, let $p_j = \max(\{0\} \cup \{k \mid 1 \leq k < j \text{ and } I_k \text{ does not overlap } I_j\})$.

j	1	2	3	4	5	6	7	8	9
p_j	0	0	0	1	3	3	5	5	7

If the maximal-length subset of $\{I_1, I_2, \ldots, I_n\}$ includes I_n, then it is O_{p_n}, where O_{p_n} is a maximal-length subset of the intervals $\{I_1, I_2, \ldots, I_{p_n}\}$.

W. I. S.: A Recurrence for the Optimal Solution

Let $|I_j|$ be the length of interval I_j.

W. I. S.: A Recurrence for the Optimal Solution

Let $|I_j|$ be the length of interval I_j.

Let $\ell(j)$ be maximal total length of any subset of non-overlapping intervals in $\{I_1, I_2, \ldots, I_j\}$.

W. I. S.: A Recurrence for the Optimal Solution

Let $|I_j|$ be the length of interval I_j.

Let $\ell(j)$ be maximal total length of any subset of non-overlapping intervals in $\{I_1, I_2, \ldots, I_j\}$.

What we're interested in is $\ell(n)$!

W. I. S.: A Recurrence for the Optimal Solution

Let $|I_j|$ be the length of interval I_j.

Let $\ell(j)$ be maximal total length of any subset of non-overlapping intervals in $\{I_1, I_2, \ldots, I_j\}$.

What we're interested in is $\ell(n)$!

$$\ell(j) = \begin{cases} 0 & j = 0 \\ \max(\ell(j-1), |I_j| + \ell(p_j)) & j > 0 \end{cases}$$

W. I. S.: A Recursive Algorithm

FindScheduleLength(I, p, j)

```
if j = 0
   then return 0
   else  return max(FindScheduleLength(I, p, p[j]) + |I[j]|,
                    FindScheduleLength(I, p, j − 1))
```

W. I. S.: A Recursive Algorithm

FindScheduleLength(I, p, j)

```
if j = 0
    then return 0
    else return max(FindScheduleLength(I, p, p[j]) + |I[j]|,
                    FindScheduleLength(I, p, j − 1))
```

Running time:

W. I. S.: A Recursive Algorithm

FindScheduleLength(I, p, j)

```
if j = 0
    then return 0
    else return max(FindScheduleLength(I, p, p[j]) + |I[j]|,
                    FindScheduleLength(I, p, j − 1))
```

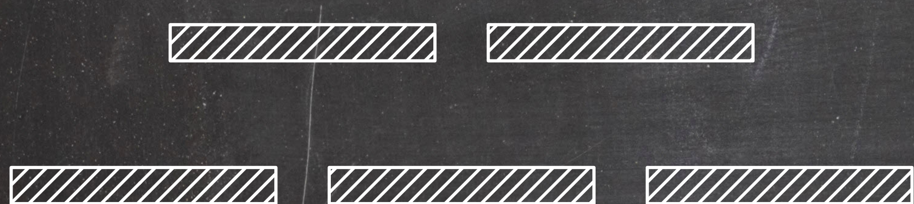
Running time: $O(2^n)$

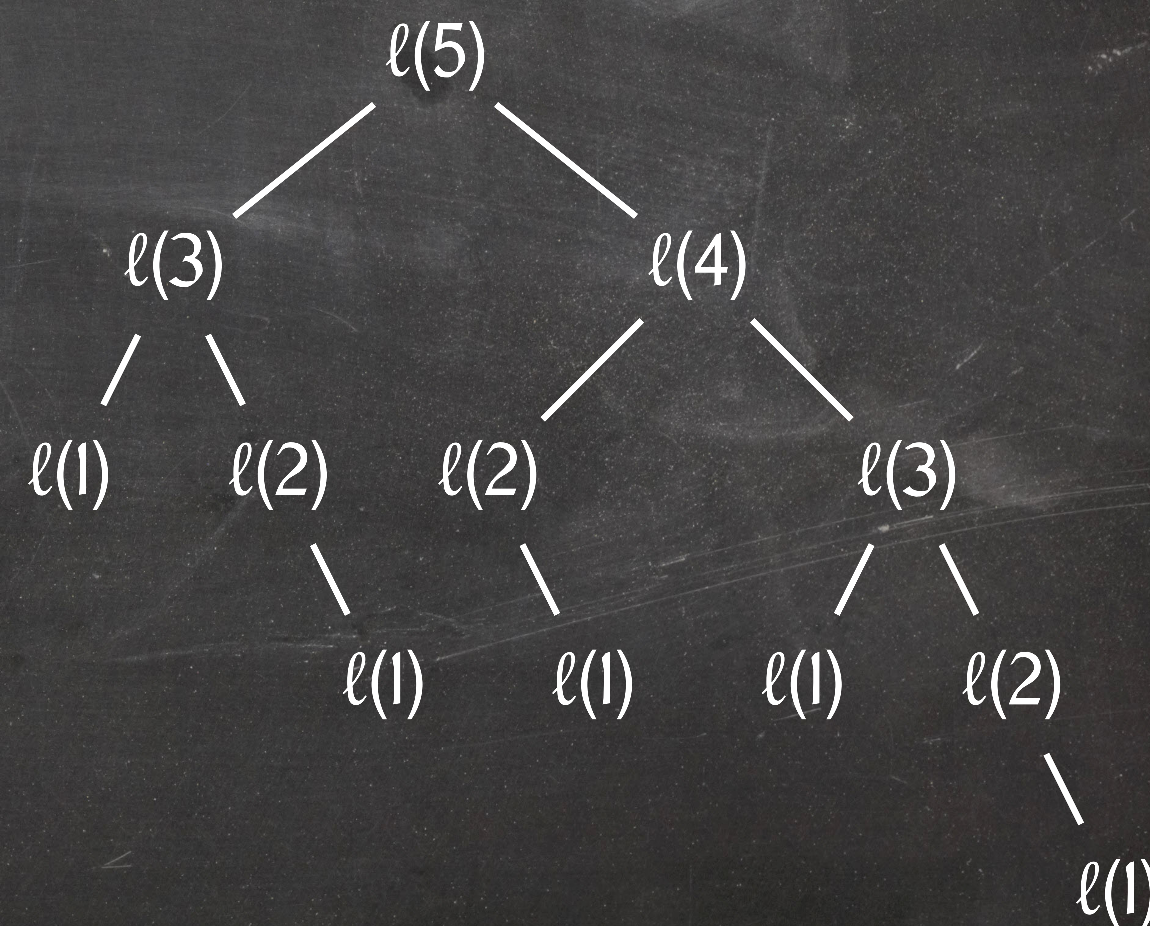
W. I. S.: A Recursive Algorithm

FindScheduleLength(I, p, j)

```
if j = 0
    then return 0
    else return max(FindScheduleLength(I, p, p[j]) + |I[j]|,
                    FindScheduleLength(I, p, j − 1))
```

Running time: $O(2^n)$



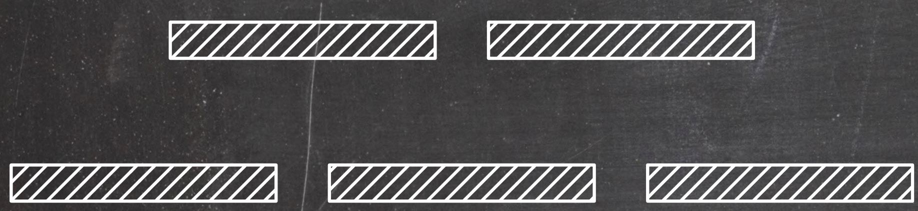


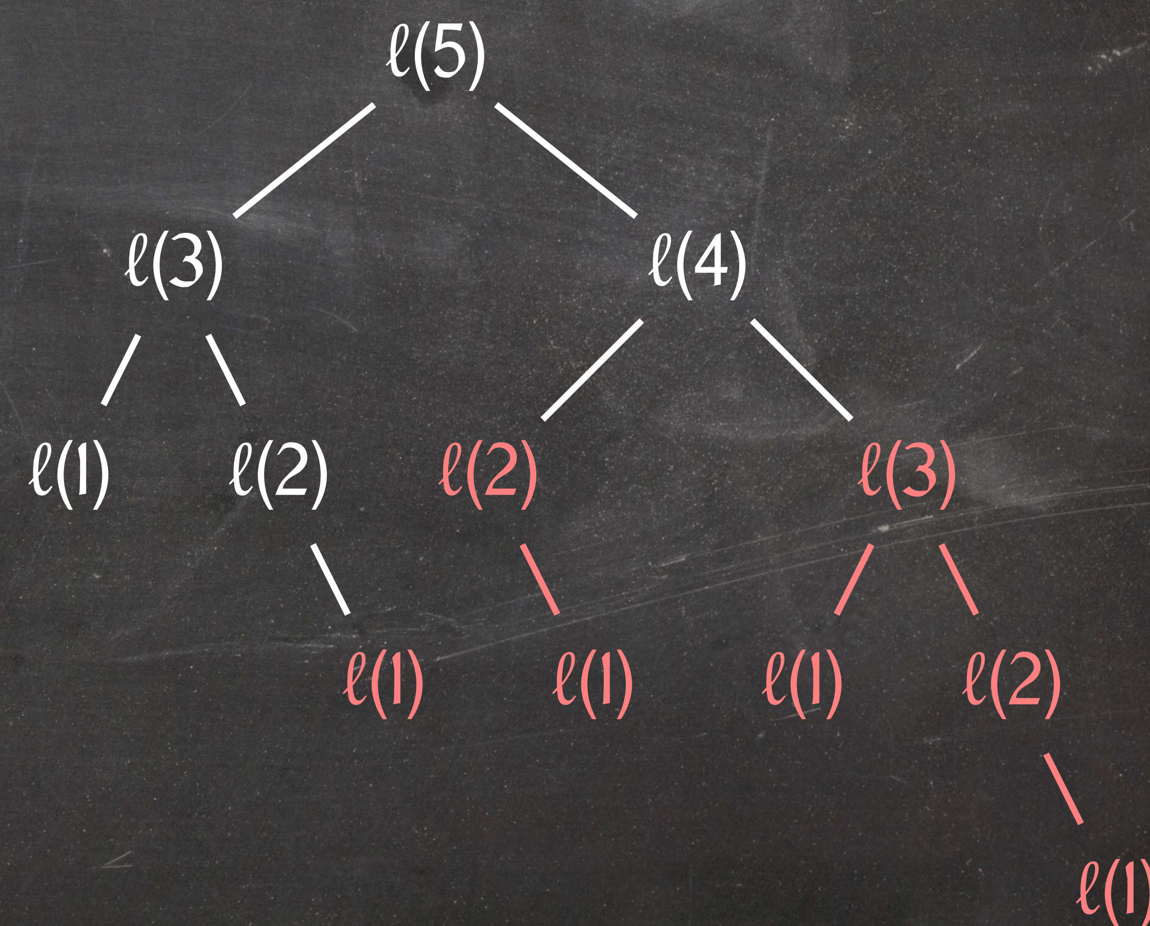
W. I. S.: A Recursive Algorithm

FindScheduleLength(I, p, j)

```
if j = 0
   then return 0
   else return max(FindScheduleLength(I, p, p[j]) + |I[j]|,
                   FindScheduleLength(I, p, j − 1))
```

Running time: $O(2^n)$





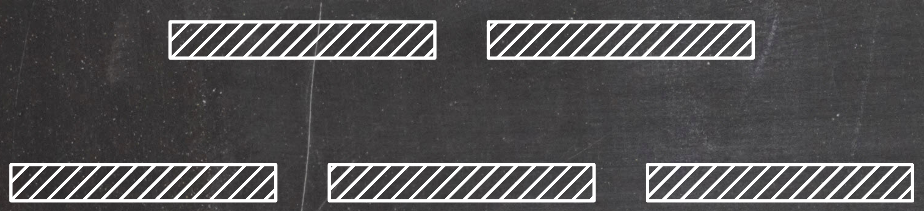
The recursive algorithm computes many values repeatedly.

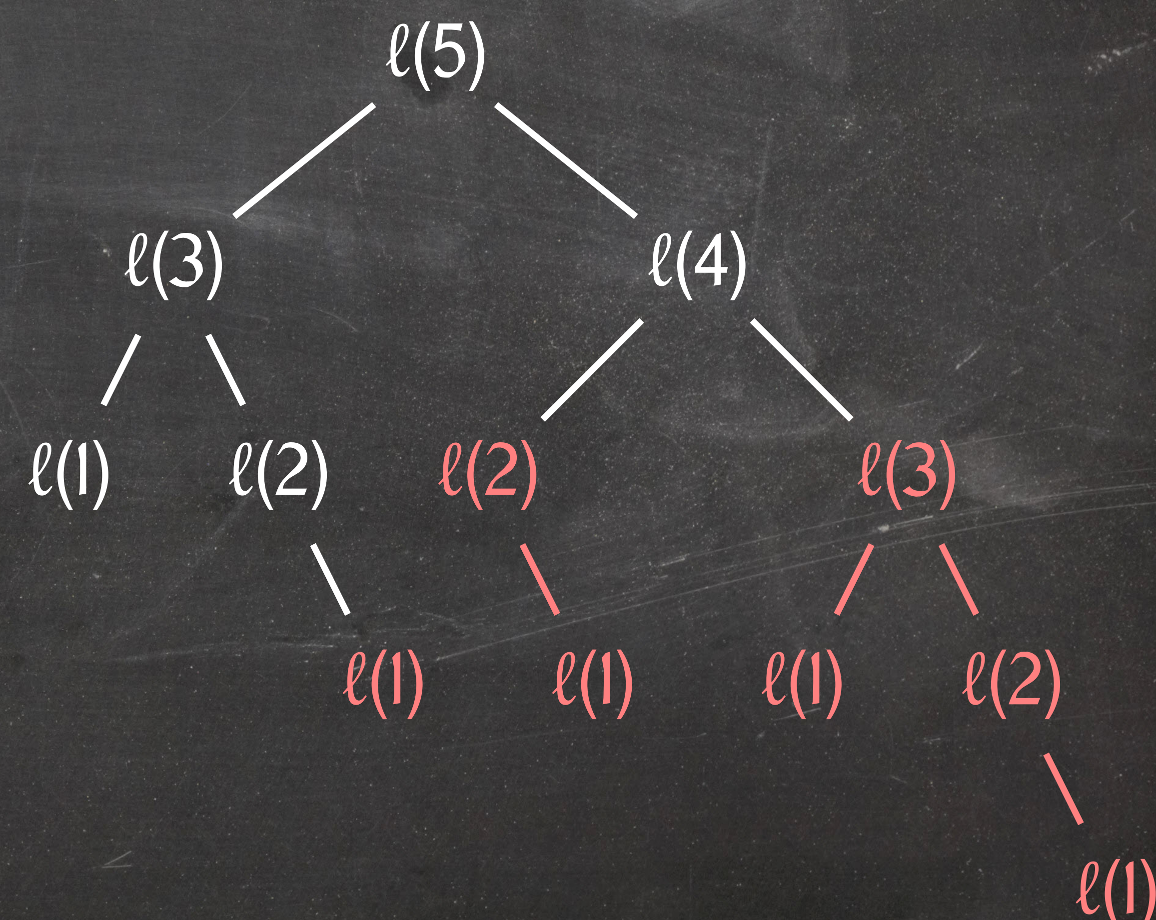
W. I. S.: A Recursive Algorithm

FindScheduleLength(I, p, j)

```
if j = 0
    then return 0
    else return max(FindScheduleLength(I, p, p[j]) + |I[j]|,
                    FindScheduleLength(I, p, j − 1))
```

Running time: $O(2^n)$





The recursive algorithm computes many values repeatedly.

There are only n values to compute!

W. I. S.: Memoizing the Recursive Algorithm

Memoization: Store already computed values in a table to avoid recomputing them.

W. I. S.: Memoizing the Recursive Algorithm

Memoization: Store already computed values in a table to avoid recomputing them.

Here, initialize a table ℓ where $\ell[j]$ is the length of the optimal schedule for $\{I_1, I_2, \ldots, I_j\}$.

Initially, $\ell[j] = -\infty$ for all j.

W. I. S.: Memoizing the Recursive Algorithm

Memoization: Store already computed values in a table to avoid recomputing them.

Here, initialize a table ℓ where $\ell[j]$ is the length of the optimal schedule for $\{I_1, I_2, \ldots, I_j\}$.

Initially, $\ell[j] = -\infty$ for all j.

FindScheduleLength(I, ℓ, p, j)

```
if j = 0
   then return 0
   else if ℓ[j] < 0
           then ℓ[j] = max(FindScheduleLength(I, p, p[j]) + |I[j]|,
                           FindScheduleLength(I, p, j − 1))
        return ℓ[j]
```

W. I. S.: Memoizing the Recursive Algorithm

Memoization: Store already computed values in a table to avoid recomputing them.

Here, initialize a table ℓ where $\ell[j]$ is the length of the optimal schedule for $\{I_1, I_2, \ldots, I_j\}$.

Initially, $\ell[j] = -\infty$ for all j.

FindScheduleLength(I, ℓ, p, j)

```
if j = 0
    then return 0
    else if ℓ[j] < 0
            then ℓ[j] = max(FindScheduleLength(I, p, p[j]) + |I[j]|,
                            FindScheduleLength(I, p, j − 1))
         return ℓ[j]
```

Running time: O(n)

W. I. S.: Iterative Table Fill-In

FindScheduleLength(I, p)

```
ℓ[0] = 0
for j = 1 to n
    do ℓ[j] = max(ℓ[j − 1], ℓ[p[j]] + |I[j]|)
return ℓ[n]
```

W. I. S.: Iterative Table Fill-In

FindScheduleLength(I, p)

```
ℓ[0] = 0
for j = 1 to n
    do ℓ[j] = max(ℓ[j − 1], ℓ[p[j]] + |I[j]|)
return ℓ[n]
```

Running time: $O(n)$

W. I. S.: Iterative Table Fill-In

FindScheduleLength(I, p)

```
ℓ[0] = 0
for j = 1 to n
    do ℓ[j] = max(ℓ[j − 1], ℓ[p[j]] + |I[j]|)
return ℓ[n]
```

Running time: $O(n)$

Advantage over memoization:

- No need for recursion.
- Algorithm is often simpler.

W. I. S.: Iterative Table Fill-In

FindScheduleLength(I, p)

```
ℓ[0] = 0
for j = 1 to n
    do ℓ[j] = max(ℓ[j − 1], ℓ[p[j]] + |I[j]|)
return ℓ[n]
```

Running time: $O(n)$

Advantage over memoization:

- No need for recursion.
- Algorithm is often simpler.

Disadvantage over memoization:

- Need to worry about the order in which the table entries are computed:
  - All entries needed to compute the current entry need to be computed first.
- Memoization computes table entries as needed.

W. I. S.: Computing the Set of Intervals

FindSchedule(I, p)

```
ℓ[0] = 0
S[0] = []
for j = 1 to n
    do if ℓ[j − 1] > ℓ[p[j]] + |I[j]|
          then ℓ[j] = ℓ[j − 1]
               S[j] = S[j − 1]
          else ℓ[j] = ℓ[p[j]] + |I[j]|
               S[j] = [I[j]] ++ S[p[j]]
return S[n]
```

W. I. S.: Computing the Set of Intervals

FindSchedule(I, p)

```
ℓ[0] = 0
S[0] = [ ]
for j = 1 to n
    do if ℓ[j − 1] > ℓ[p[j]] + |I[j]|
            then ℓ[j] = ℓ[j − 1]
                 S[j] = S[j − 1]
            else ℓ[j] = ℓ[p[j]] + |I[j]|
                 S[j] = [I[j]] ++ S[p[j]]
return S[n]
```

Running time: $O(n)$

W. I. S.: Computing the Set of Intervals

FindSchedule(I, p)

```
ℓ[0] = 0
S[0] = []
for j = 1 to n
    do if ℓ[j − 1] > ℓ[p[j]] + |I[j]|
            then ℓ[j] = ℓ[j − 1]
                 S[j] = S[j − 1]
            else ℓ[j] = ℓ[p[j]] + |I[j]|
                 S[j] = [I[j]] ++ S[p[j]]
return S[n]
```

Running time: $O(n)$

This computes the sequence of intervals ordered from last to first.

This list is of course easy to reverse in linear time.

W. I. S.: The Missing Details

What's missing?

- Sort the intervals by their ending times.
- Compute the predecessor array p.

W. I. S.: The Missing Details

What's missing?

- Sort the intervals by their ending times.
- Compute the predecessor array p.

Solution:

- Sorting is easily done in $O(n \lg n)$ time.
- To compute p[j], perform binary search with I[j]'s starting time on the sorted array of ending times.

W. I. S.: The Missing Details

What's missing?

- Sort the intervals by their ending times.
- Compute the predecessor array p.

Solution:

- Sorting is easily done in $O(n \lg n)$ time.
- To compute p[j], perform binary search with I[j]'s starting time on the sorted array of ending times.

Theorem: The weighted interval scheduling problem can be solved in $O(n \lg n)$ time.

The Dynamic Programming Technique

The technique:

- Develop a recurrence expressing the optimal solution for a given problem instance in terms of optimal solutions for smaller problem instances:
- Evaluate this recurrence
  - Recursively using memoization or
  - Using iterative table fill-in.

The Dynamic Programming Technique

The technique:

- Develop a recurrence expressing the optimal solution for a given problem instance in terms of optimal solutions for smaller problem instances:
- Evaluate this recurrence
  - Recursively using memoization or
  - Using iterative table fill-in.

For this to work, the problem must exhibit the optimal substructure property: The optimal solution to a problem instance must be composed of optimal solutions to smaller problem instances.

The Dynamic Programming Technique

The technique:

- Develop a recurrence expressing the optimal solution for a given problem instance in terms of optimal solutions for smaller problem instances:
- Evaluate this recurrence
  - Recursively using memoization or
  - Using iterative table fill-in.

For this to work, the problem must exhibit the optimal substructure property: The optimal solution to a problem instance must be composed of optimal solutions to smaller problem instances.

A speed-up over the naïve recursive algorithm is achieved if the problem exhibits overlapping subproblems: The same subproblem occurs over and over again in the recursive evaluation of the recurrence.

Developing a Dynamic Programming Algorithm

Step 1: Think top-down:

- Consider an optimal solution (without worrying about how to compute it).
- Identify how the optimal solution of any problem instance decomposes into optimal solutions to smaller problem instances.
- Write down a recurrence based on this analysis.

Step 2: Formulate the algorithm, which computes the solution bottom-up:

- Since an optimal solution depends on optimal solutions to smaller problem instances, we need to compute those first.

Sequence Alignment

Given the search term "Dalhusy Computer Science", Google suggests the correction "Dalhousie Computer Science".

Sequence Alignment

Given the search term "Dalhusy Computer Science", Google suggests the correction "Dalhousie Computer Science".

Can Google read your mind?

Sequence Alignment

Given the search term "Dalhusy Computer Science", Google suggests the correction "Dalhousie Computer Science".

Can Google read your mind? No!

They use a clever algorithm to match your mistyped query against the phrases they have in their database.

"Dalhousie" is the closest match to "Dalhusy" they find.

Sequence Alignment

Given the search term "Dalhusy Computer Science", Google suggests the correction "Dalhousie Computer Science".

Can Google read your mind? No!

They use a clever algorithm to match your mistyped query against the phrases they have in their database.

"Dalhousie" is the closest match to "Dalhusy" they find.

What's a good similarity criterion?

Sequence Alignment

Problem: Given two strings $X = x_1x_2 \cdots x_m$ and $Y = y_1y_2 \cdots y_n$, extend them to two strings $X' = x_1'x_2' \cdots x_t'$ and $Y' = y_1'y_2' \cdots y_t'$ of the same length by inserting gaps so that the following dissimilarity measure $D(X', Y')$ is minimized:

$$D(X', Y') = \sum_{i=1}^{t} d(x_i', y_i')$$

$$d(x, y) = \begin{cases} \delta & x = \textvisiblespace \text{ or } y = \textvisiblespace \text{ (gap penalty)} \\ \mu_{x,y} & \text{otherwise (mismatch penalty)} \end{cases}$$

Sequence Alignment

Problem: Given two strings $X = x_1x_2 \cdots x_m$ and $Y = y_1y_2 \cdots y_n$, extend them to two strings $X' = x'_1x'_2 \cdots x'_t$ and $Y' = y'_1y'_2 \cdots y'_t$ of the same length by inserting gaps so that the following dissimilarity measure $D(X', Y')$ is minimized:

$$D(X', Y') = \sum_{i=1}^{t} d(x'_i, y'_i)$$

$$d(x, y) = \begin{cases} \delta & x = \text{␣} \text{ or } y = \text{␣} \text{ (gap penalty)} \\ \mu_{x,y} & \text{otherwise (mismatch penalty)} \end{cases}$$

Example:

```
Dalh␣usy␣
Dalhousie
```

$$D(X', Y') = 2\delta + \mu_{iy}$$

Sequence Alignment

Problem: Given two strings $X = x_1x_2 \cdots x_m$ and $Y = y_1y_2 \cdots y_n$, extend them to two strings $X' = x_1'x_2' \cdots x_t'$ and $Y' = y_1'y_2' \cdots y_t'$ of the same length by inserting gaps so that the following dissimilarity measure $D(X', Y')$ is minimized:

$$D(X', Y') = \sum_{i=1}^{t} d(x_i', y_i')$$

$$d(x, y) = \begin{cases} \delta & x = \text{␣} \text{ or } y = \text{␣} \text{ (gap penalty)} \\ \mu_{x,y} & \text{otherwise (mismatch penalty)} \end{cases}$$

Example:

```
Dalh␣usy␣
Dalhousie
```

$$D(X', Y') = 2\delta + \mu_{iy}$$

Another (more important?) application:
DNA sequence alignment to measure the similarity between different DNA samples.

Sequence Alignment: Problem Analysis

Assume $(x'_1 x'_2 \cdots x'_t, y'_1 y'_2 \cdots, y'_t)$ is an optimal alignment for $(x_1 x_2 \cdots x_m, y_1 y_2 \cdots y_n)$.

What choices do we have for the final pair (x'_t, y'_t)?

Sequence Alignment: Problem Analysis

Assume $(x'_1x'_2\cdots x'_t, y'_1y'_2\cdots, y'_t)$ is an optimal alignment for $(x_1x_2\cdots x_m, y_1y_2\cdots y_n)$.

What choices do we have for the final pair (x'_t, y'_t)?

- $x'_t = x_m$ and $y'_t = y_n$

Sequence Alignment: Problem Analysis

Assume $(x'_1 x'_2 \cdots x'_t, y'_1 y'_2 \cdots, y'_t)$ is an optimal alignment for $(x_1 x_2 \cdots x_m, y_1 y_2 \cdots y_n)$.

What choices do we have for the final pair (x'_t, y'_t)?

- $x'_t = x_m$ and $y'_t = y_n$
- $x'_t = x_m$ and $y'_t = \text{␣}$

Sequence Alignment: Problem Analysis

Assume $(x'_1 x'_2 \cdots x'_t, y'_1 y'_2 \cdots, y'_t)$ is an optimal alignment for $(x_1 x_2 \cdots x_m, y_1 y_2 \cdots y_n)$.

What choices do we have for the final pair (x'_t, y'_t)?

- $x'_t = x_m$ and $y'_t = y_n$
- $x'_t = x_m$ and $y'_t = \sqcup$
- $x'_t = \sqcup$ and $y'_t = y_n$

Sequence Alignment: Problem Analysis

Assume $(x'_1x'_2 \cdots x'_t, y'_1y'_2 \cdots, y'_t)$ is an optimal alignment for $(x_1x_2 \cdots x_m, y_1y_2 \cdots y_n)$.

What choices do we have for the final pair (x'_t, y'_t)?

- $x'_t = x_m$ and $y'_t = y_n$

  $(x'_1x'_2 \cdots x'_{t-1}, y'_1y'_2 \cdots y'_{t-1})$ must be an optimal alignment for $(x_1x_2 \cdots x_{m-1}, y_1y_2 \cdots y_{n-1})$.

Sequence Alignment: Problem Analysis

Assume $(x'_1 x'_2 \cdots x'_t, y'_1 y'_2 \cdots, y'_t)$ is an optimal alignment for $(x_1 x_2 \cdots x_m, y_1 y_2 \cdots y_n)$.

What choices do we have for the final pair (x'_t, y'_t)?

- $x'_t = x_m$ and $y'_t = y_n$

  $(x'_1 x'_2 \cdots x'_{t-1}, y'_1 y'_2 \cdots y'_{t-1})$ must be an optimal alignment for $(x_1 x_2 \cdots x_{m-1}, y_1 y_2 \cdots y_{n-1})$.

  Assume there's a better alignment $(x''_1 x''_2 \cdots x''_s, y''_1 y''_2 \cdots y''_s)$ with dissimilarity

$$\sum_{i=1}^{s} d(x''_i, y''_i) < \sum_{i=1}^{t-1} d(x'_i, y'_i).$$

Sequence Alignment: Problem Analysis

Assume $(x_1'x_2'\cdots x_t', y_1'y_2'\cdots, y_t')$ is an optimal alignment for $(x_1x_2\cdots x_m, y_1y_2\cdots y_n)$.

What choices do we have for the final pair (x_t', y_t')?

- $x_t' = x_m$ and $y_t' = y_n$

  $(x_1'x_2'\cdots x_{t-1}', y_1'y_2'\cdots y_{t-1}')$ must be an optimal alignment for $(x_1x_2\cdots x_{m-1}, y_1y_2\cdots y_{n-1})$.

  Assume there's a better alignment $(x_1''x_2''\cdots x_s'', y_1''y_2''\cdots y_s'')$ with dissimilarity

  $$\sum_{i=1}^{s} d(x_i'', y_i'') < \sum_{i=1}^{t-1} d(x_i', y_i').$$

  Then $(x_1''x_2''\cdots x_s''x_t', y_1''y_2''\cdots y_s''y_t')$ is an aligment for $(x_1x_2\cdots x_m, y_1y_2\cdots y_n)$ with dissimilarity

  $$\sum_{i=1}^{s} d(x_i'', y_i'') + d(x_t', y_t') < \sum_{i=1}^{t-1} d(x_i', y_i') + d(x_t', y_t') = \sum_{i=1}^{t} d(x_i', y_i'),$$

  a contradiction.

Sequence Alignment: Problem Analysis

Assume $(x_1'x_2'\cdots x_t', y_1'y_2'\cdots, y_t')$ is an optimal alignment for $(x_1x_2\cdots x_m, y_1y_2\cdots y_n)$.

What choices do we have for the final pair (x_t', y_t')?

- $x_t' = x_m$ and $y_t' = y_n$

  $(x_1'x_2'\cdots x_{t-1}', y_1'y_2'\cdots y_{t-1}')$ must be an optimal alignment for $(x_1x_2\cdots x_{m-1}, y_1y_2\cdots y_{n-1})$.

- $x_t' = x_m$ and $y_t' = \text{␣}$

  $(x_1'x_2'\cdots x_{t-1}', y_1'y_2'\cdots y_{t-1}')$ must be an optimal alignment for $(x_1x_2\cdots x_{m-1}, y_1y_2\cdots y_n)$.

Sequence Alignment: Problem Analysis

Assume $(x'_1x'_2\cdots x'_t, y'_1y'_2\cdots, y'_t)$ is an optimal alignment for $(x_1x_2\cdots x_m, y_1y_2\cdots y_n)$.

What choices do we have for the final pair (x'_t, y'_t)?

- $x'_t = x_m$ and $y'_t = y_n$

  $(x'_1x'_2\cdots x'_{t-1}, y'_1y'_2\cdots y'_{t-1})$ must be an optimal alignment for $(x_1x_2\cdots x_{m-1}, y_1y_2\cdots y_{n-1})$.

- $x'_t = x_m$ and $y'_t = \text{␣}$

  $(x'_1x'_2\cdots x'_{t-1}, y'_1y'_2\cdots y'_{t-1})$ must be an optimal alignment for $(x_1x_2\cdots x_{m-1}, y_1y_2\cdots y_n)$.

- $x'_t = \text{␣}$ and $y'_t = y_n$

  $(x'_1x'_2\cdots x'_{t-1}, y'_1y'_2\cdots y'_{t-1})$ must be an optimal alignment for $(x_1x_2\cdots x_m, y_1y_2\cdots y_{n-1})$.

Sequence Alignment: The Recurrence

Let $D(i, j)$ be the dissimilarity of the strings $x_1 x_2 \cdots x_i$ and $y_1 y_2 \cdots y_j$.

Sequence Alignment: The Recurrence

Let $D(i, j)$ be the dissimilarity of the strings $x_1 x_2 \cdots x_i$ and $y_1 y_2 \cdots y_j$.

We are interested in $D(m, n)$.

Sequence Alignment: The Recurrence

Let $D(i, j)$ be the dissimilarity of the strings $x_1 x_2 \cdots x_i$ and $y_1 y_2 \cdots y_j$.

We are interested in $D(m, n)$.

Recurrence:

$$D(i, j) = \begin{cases} \delta \cdot j & i = 0 \\ \delta \cdot i & j = 0 \\ \min(D(i-1, j-1) + \mu_{x_i, y_j}, D(i, j-1) + \delta, D(i-1, j) + \delta) & \text{otherwise} \end{cases}$$

Sequence Alignment: The Algorithm

SequenceAlignment(X, Y, μ, δ)

```
D[0, 0] = 0
A[0, 0] = [ ]
for i = 1 to m
    do D[i, 0] = D[i − 1, 0] + δ
       A[i, 0] = [(X[i], ␣)] ++ A[i − 1, 0]
for j = 1 to n
    do D[0, j] = D[0, j − 1] + δ
       A[0, j] = [(␣, Y[j])] ++ A[0, j − 1]
for i = 1 to m
    do for j = 1 to n
           do D[i, j] = D[i − 1, j − 1] + μ[X[i], Y[j]]
              A[i, j] = [(X[i], Y[j])] ++ A[i − 1, j − 1]
              if D[i, j] > D[i − 1, j] + δ
                 then D[i, j] = D[i − 1, j] + δ
                      A[i, j] = [(X[i], ␣)] ++ A[i − 1, j]
              if D[i, j] > D[i, j − 1] + δ
                 then D[i, j] = D[i, j − 1] + δ
                      A[i, j] = [(␣, Y[j])] ++ A[i, j − 1]
return A[m, n]
```

Sequence Alignment: The Algorithm

SequenceAlignment(X, Y, μ, δ)

```
D[0, 0] = 0
A[0, 0] = [ ]
for i = 1 to m
    do D[i, 0] = D[i − 1, 0] + δ
       A[i, 0] = [(X[i], ␣)] ++ A[i − 1, 0]
for j = 1 to n
    do D[0, j] = D[0, j − 1] + δ
       A[0, j] = [(␣, Y[j])] ++ A[0, j − 1]
for i = 1 to m
    do for j = 1 to n
           do D[i, j] = D[i − 1, j − 1] + μ[X[i], Y[j]]
              A[i, j] = [(X[i], Y[j])] ++ A[i − 1, j − 1]
              if D[i, j] > D[i − 1, j] + δ
                 then D[i, j] = D[i − 1, j] + δ
                      A[i, j] = [(X[i], ␣)] ++ A[i − 1, j]
              if D[i, j] > D[i, j − 1] + δ
                 then D[i, j] = D[i, j − 1] + δ
                      A[i, j] = [(␣, Y[j])] ++ A[i, j − 1]
return A[m, n]
```

Running time: $O(mn)$

Sequence Alignment: The Algorithm

```
SequenceAlignment(X, Y, μ, δ)

D[0, 0] = 0
A[0, 0] = [ ]
for i = 1 to m
    do D[i, 0] = D[i − 1, 0] + δ
       A[i, 0] = [(X[i], ␣)] ++ A[i − 1, 0]
for j = 1 to n
    do D[0, j] = D[0, j − 1] + δ
       A[0, j] = [(␣, Y[j])] ++ A[0, j − 1]
for i = 1 to m
    do for j = 1 to n
           do D[i, j] = D[i − 1, j − 1] + μ[X[i], Y[j]]
              A[i, j] = [(X[i], Y[j])] ++ A[i − 1, j − 1]
              if D[i, j] > D[i − 1, j] + δ
                 then D[i, j] = D[i − 1, j] + δ
                      A[i, j] = [(X[i], ␣)] ++ A[i − 1, j]
              if D[i, j] > D[i, j − 1] + δ
                 then D[i, j] = D[i, j − 1] + δ
                      A[i, j] = [(␣, Y[j])] ++ A[i, j − 1]
return A[m, n]
```

Running time: $O(mn)$

Again, the sequence alignment is reported back-to-front and can be reversed in $O(m + n)$ time.

Optimal Binary Search Trees

Balanced binary search trees (red-black trees, AVL trees, ...) guarantee $O(\lg n)$ time to find an element.

Optimal Binary Search Trees

Balanced binary search trees (red-black trees, AVL trees, ...) guarantee $O(\lg n)$ time to find an element.

Can we do better?

Optimal Binary Search Trees

Balanced binary search trees (red-black trees, AVL trees, . . .) guarantee $O(\lg n)$ time to find an element.

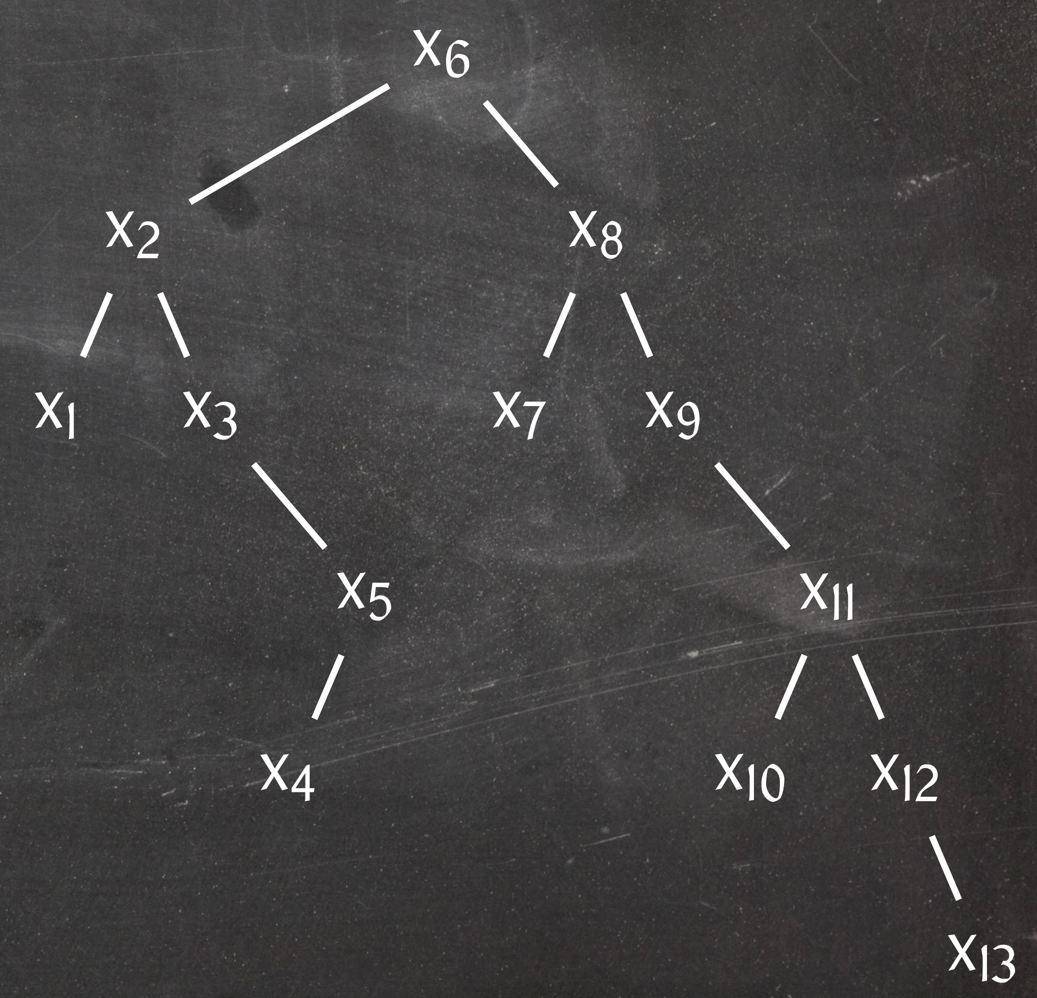
Can we do better? Not in the worst case.

Optimal Binary Search Trees

Balanced binary search trees (red-black trees, AVL trees, ...) guarantee $O(\lg n)$ time to find an element.

Can we do better? Not in the worst case.

Let $x_1 < x_2 < \cdots < x_n$ be the elements to be stored in the tree.



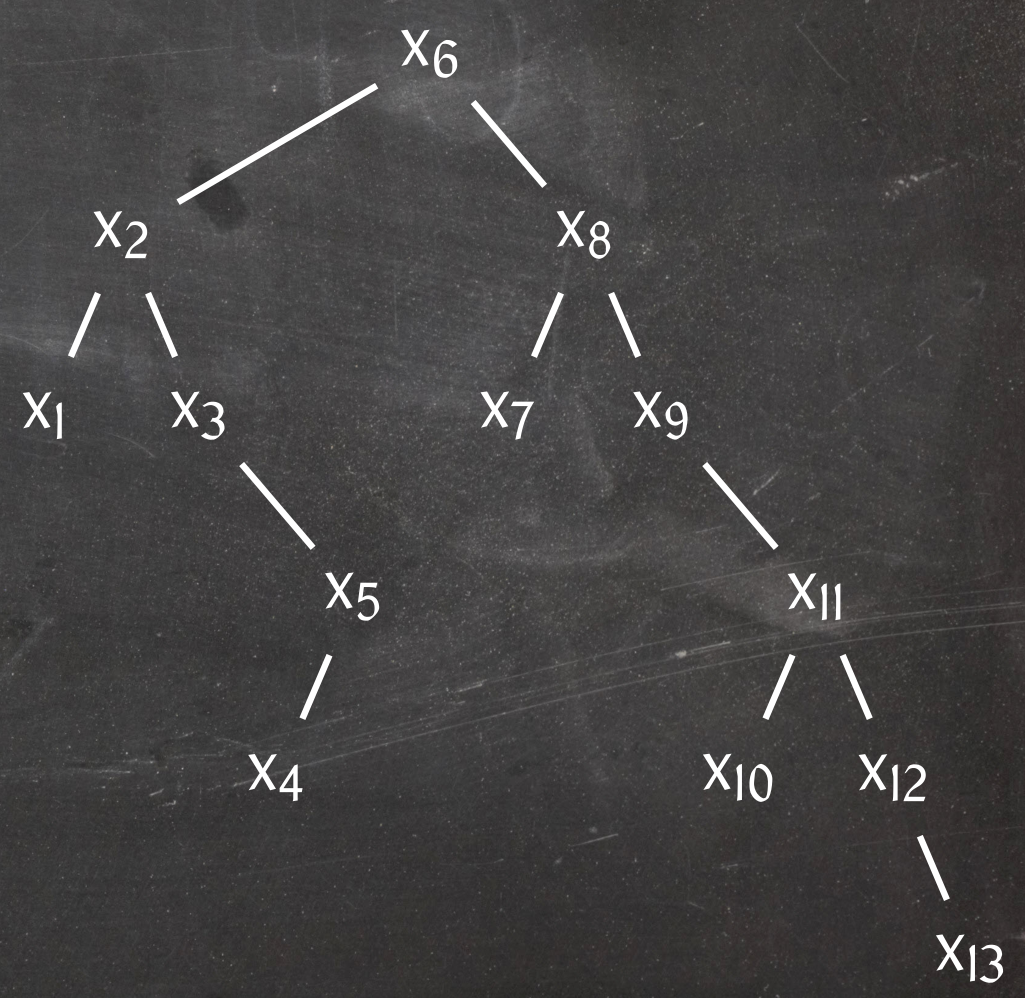
Optimal Binary Search Trees

Balanced binary search trees (red-black trees, AVL trees, ...) guarantee $O(\lg n)$ time to find an element.

Can we do better? Not in the worst case.

Let $x_1 < x_2 < \cdots < x_n$ be the elements to be stored in the tree.

Let $P = \{p_1, p_2, \ldots, p_n\}$ be the probabilities of searching for these elements.



Optimal Binary Search Trees

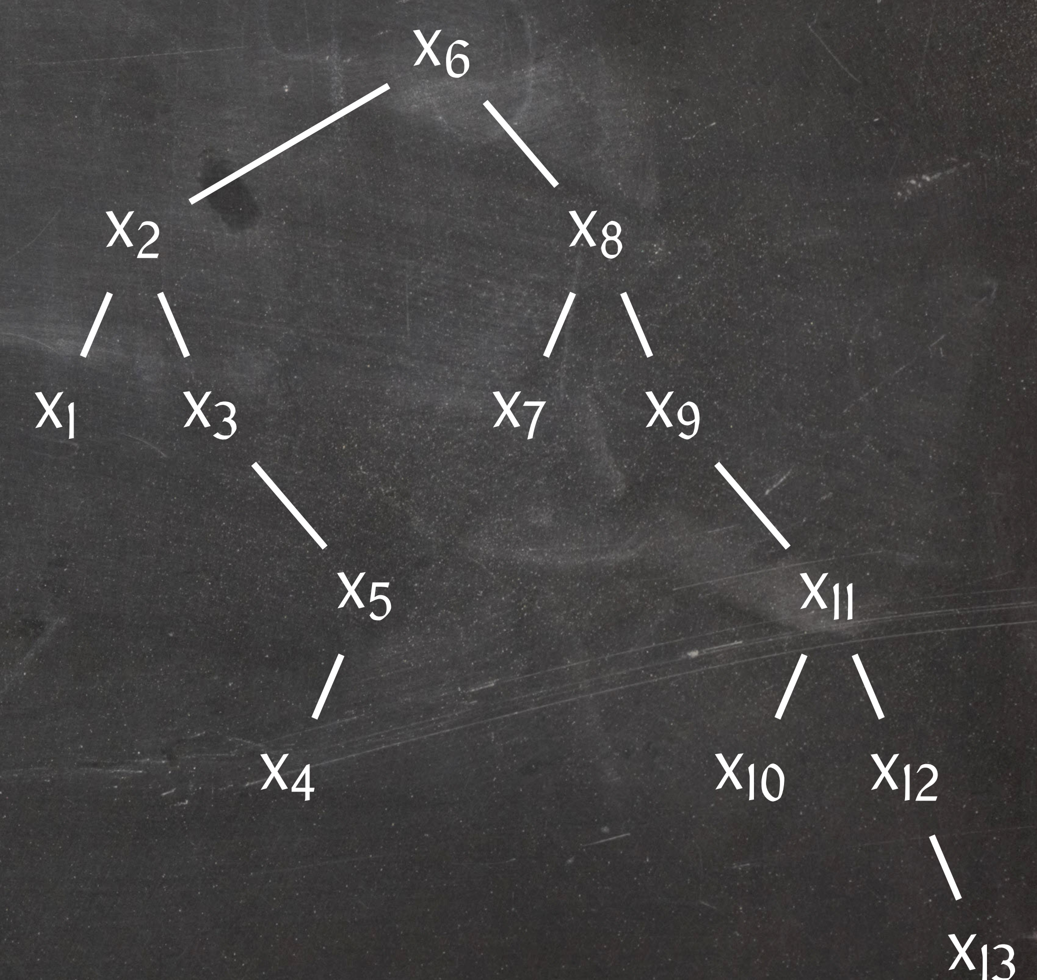
Balanced binary search trees (red-black trees, AVL trees, ...) guarantee $O(\lg n)$ time to find an element.

Can we do better? Not in the worst case.

Let $x_1 < x_2 < \cdots < x_n$ be the elements to be stored in the tree.

Let $P = \{p_1, p_2, \ldots, p_n\}$ be the probabilities of searching for these elements.

For a binary search tree T, let $d_T(x_i)$ denote the depth of element x_i in T.



Optimal Binary Search Trees

Balanced binary search trees (red-black trees, AVL trees, . . .) guarantee $O(\lg n)$ time to find an element.

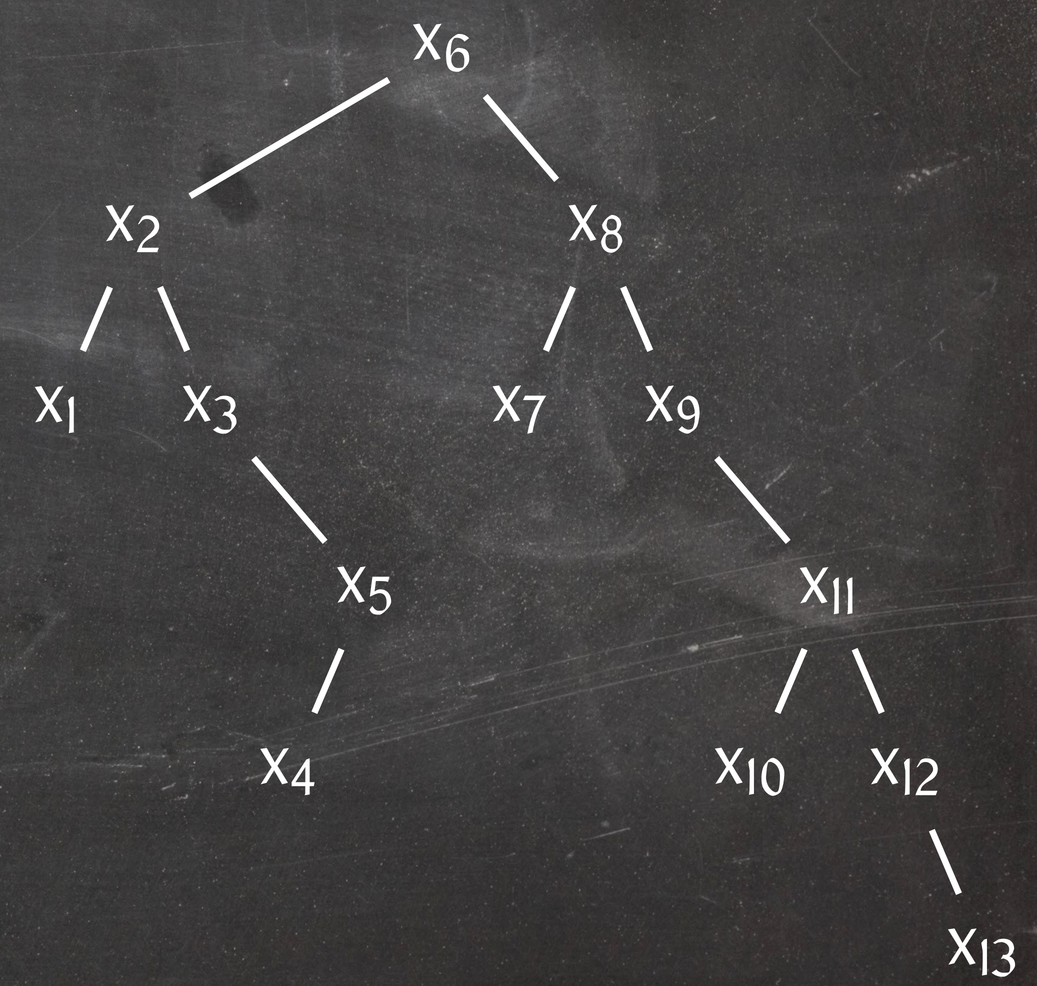
Can we do better? Not in the worst case.

Let $x_1 < x_2 < \cdots < x_n$ be the elements to be stored in the tree.

Let $P = \{p_1, p_2, \ldots, p_n\}$ be the probabilities of searching for these elements.

For a binary search tree T, let $d_T(x_i)$ denote the depth of element x_i in T.

The cost of searching for element x_i is in $O(d_T(x_i))$.



Optimal Binary Search Trees

Balanced binary search trees (red-black trees, AVL trees, ...) guarantee $O(\lg n)$ time to find an element.

Can we do better? Not in the worst case.

Let $x_1 < x_2 < \cdots < x_n$ be the elements to be stored in the tree.

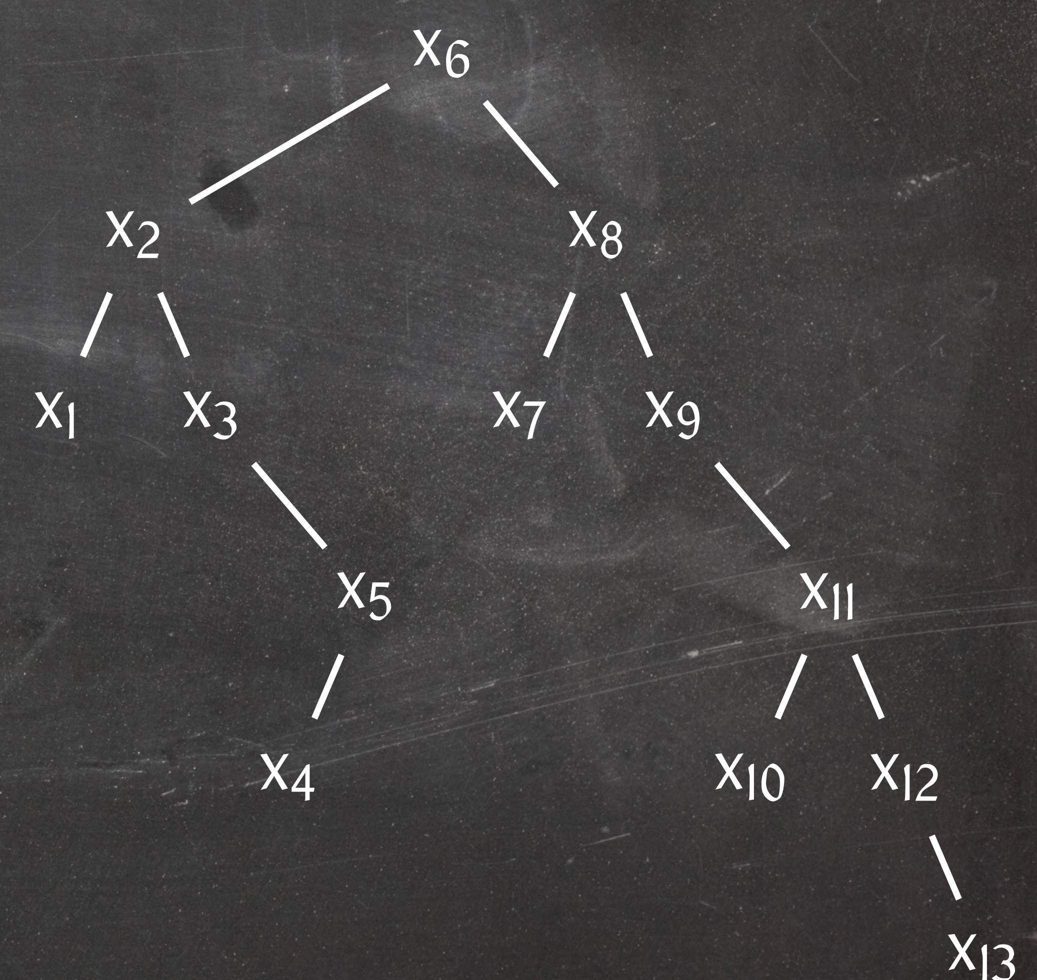
Let $P = \{p_1, p_2, \ldots, p_n\}$ be the probabilities of searching for these elements.

For a binary search tree T, let $d_T(x_i)$ denote the depth of element x_i in T.

The cost of searching for element x_i is in $O(d_T(x_i))$.

The expected cost of a random query is in $O(C_P(T))$, where

$$C_P(T) = \sum_{i=1}^{n} p_i d_T(x_i).$$



Optimal Binary Search Trees

Balanced binary search trees (red-black trees, AVL trees, ...) guarantee $O(\lg n)$ time to find an element.

Can we do better? Not in the worst case.

Let $x_1 < x_2 < \cdots < x_n$ be the elements to be stored in the tree.

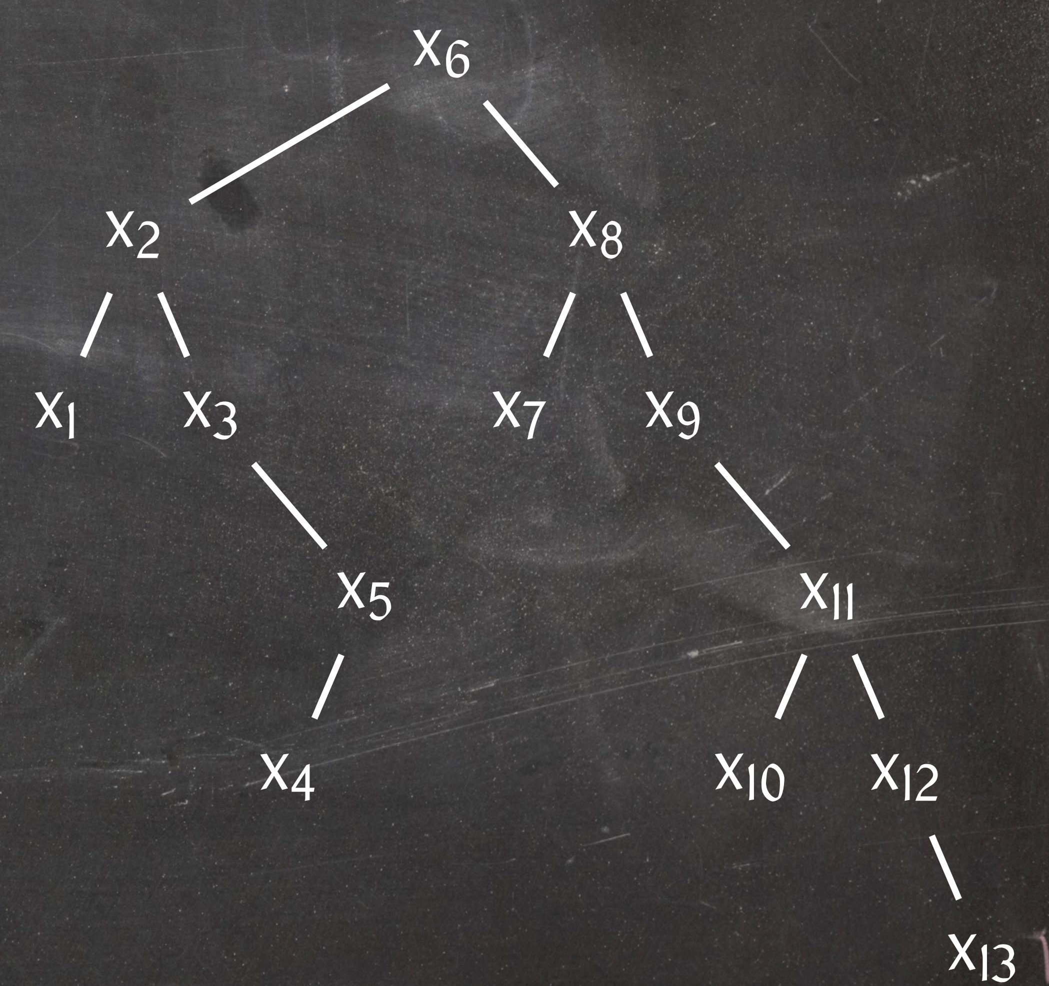
Let $P = \{p_1, p_2, \ldots, p_n\}$ be the probabilities of searching for these elements.

For a binary search tree T, let $d_T(x_i)$ denote the depth of element x_i in T.

The cost of searching for element x_i is in $O(d_T(x_i))$.

The expected cost of a random query is in $O(C_P(T))$, where

$$C_P(T) = \sum_{i=1}^{n} p_i d_T(x_i).$$



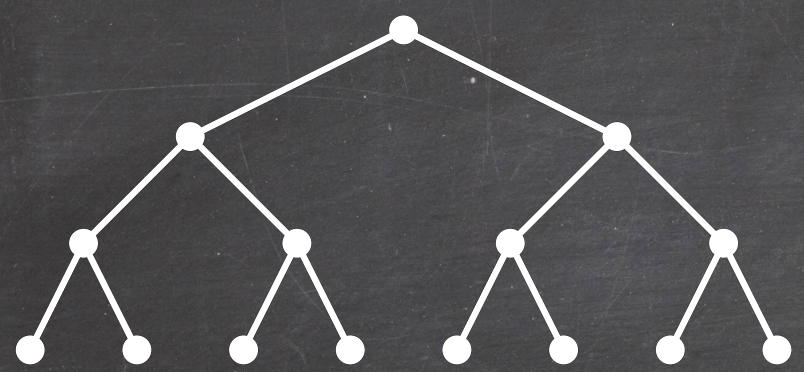
An optimal binary search tree is a binary search tree T that minimizes $C_P(T)$.

Balancing Is Not Necessarily Optimal

Assume $n = 2^k - 1$ and $p_i = 2^{-i}$ for all $1 \leq i \leq n - 1$ and $p_n = 2^{-n+1}$.

Balancing Is Not Necessarily Optimal

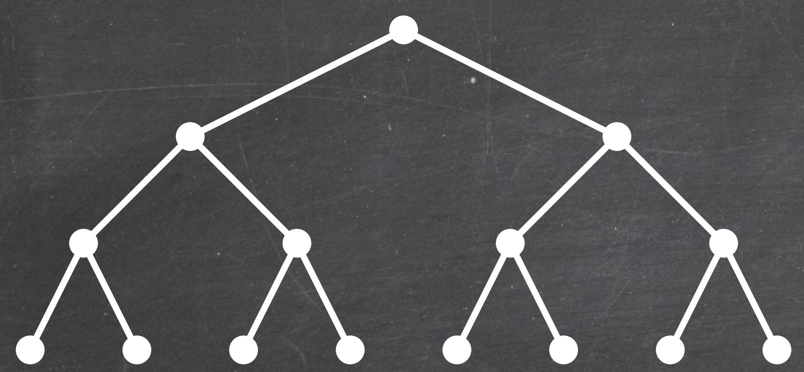
Assume $n = 2^k - 1$ and $p_i = 2^{-i}$ for all $1 \le i \le n - 1$ and $p_n = 2^{-n+1}$.



Balanced tree:

Balancing Is Not Necessarily Optimal

Assume $n = 2^k - 1$ and $p_i = 2^{-i}$ for all $1 \le i \le n - 1$ and $p_n = 2^{-n+1}$.



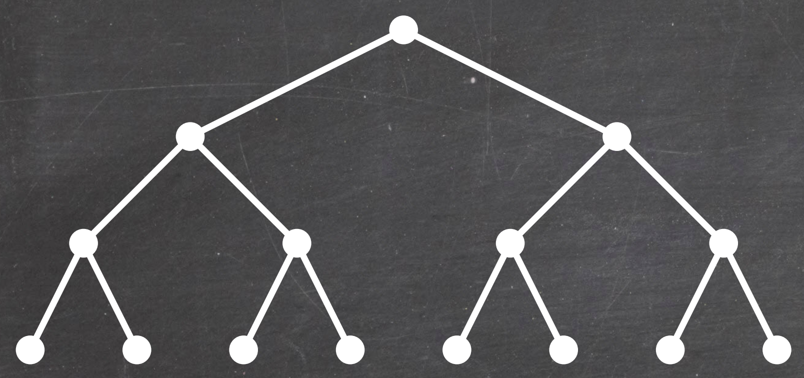
Balanced tree:

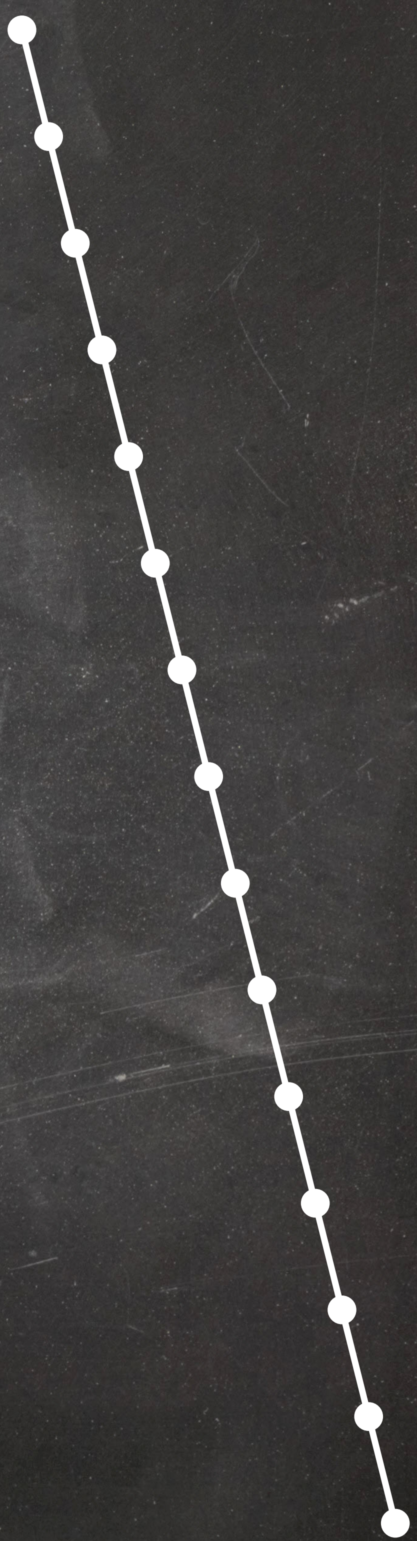
x_1 is at depth $\lg n$.

$$\Rightarrow \text{ Expected cost} \ge \frac{\lg n}{2}.$$

Balancing Is Not Necessarily Optimal

Assume $n = 2^k - 1$ and $p_i = 2^{-i}$ for all $1 \le i \le n-1$ and $p_n = 2^{-n+1}$.





Balanced tree:

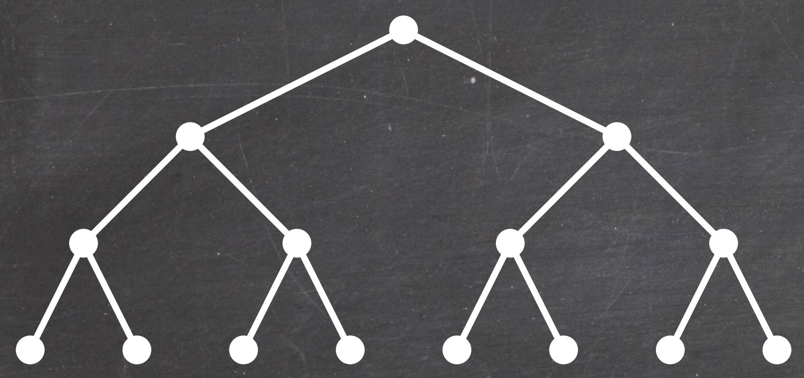
x_1 is at depth $\lg n$.

$\Rightarrow$ Expected cost $\geq \dfrac{\lg n}{2}$.

Long path:

Balancing Is Not Necessarily Optimal

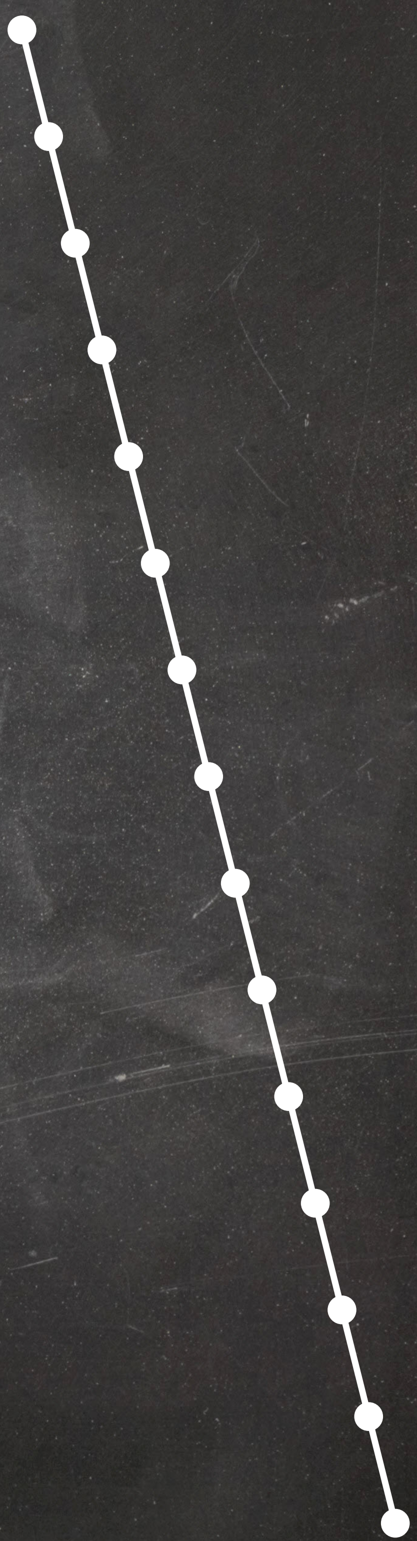
Assume $n = 2^k - 1$ and $p_i = 2^{-i}$ for all $1 \le i \le n-1$ and $p_n = 2^{-n+1}$.



Balanced tree:

x_1 is at depth $\lg n$.

$\Rightarrow$ Expected cost $\ge \dfrac{\lg n}{2}$.

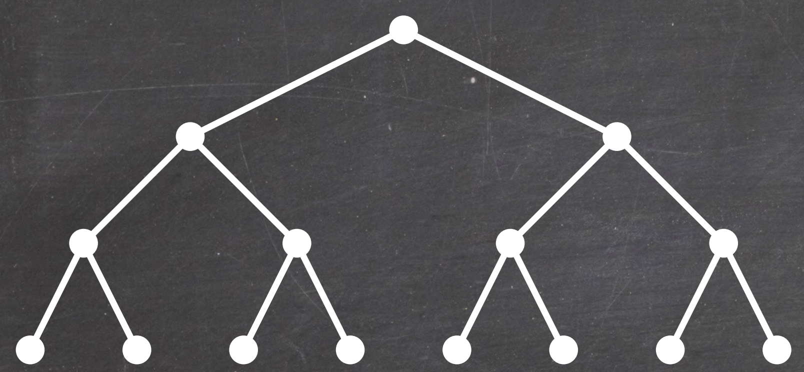


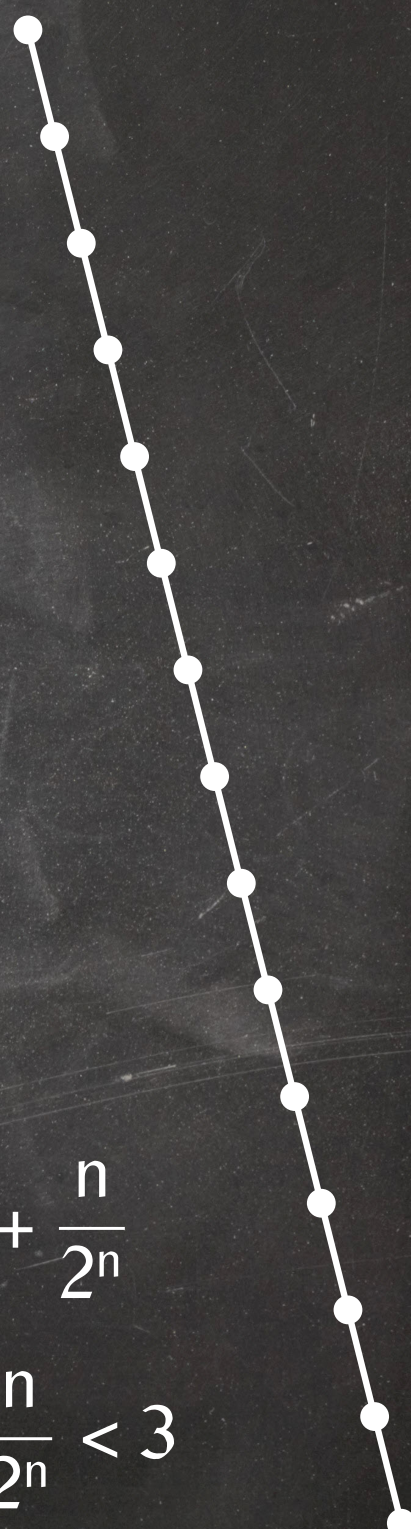
Long path:

Depth of x_i is i.

Balancing Is Not Necessarily Optimal

Assume $n = 2^k - 1$ and $p_i = 2^{-i}$ for all $1 \le i \le n-1$ and $p_n = 2^{-n+1}$.





Balanced tree:

x_1 is at depth $\lg n$.

$\Rightarrow$ Expected cost $\ge \dfrac{\lg n}{2}$.

Long path:

Depth of x_i is i.

$\Rightarrow$ Expected cost

$$= \sum_{i=1}^{n} \frac{i}{2^i} + \frac{n}{2^n} < \sum_{i=1}^{\infty} \frac{i}{2^i} + \frac{n}{2^n}$$

$$= \frac{1/2}{(1 - 1/2)^2} + \frac{n}{2^n} = 2 + \frac{n}{2^n} < 3$$

Optimal Binary Search Trees: Problem Analysis

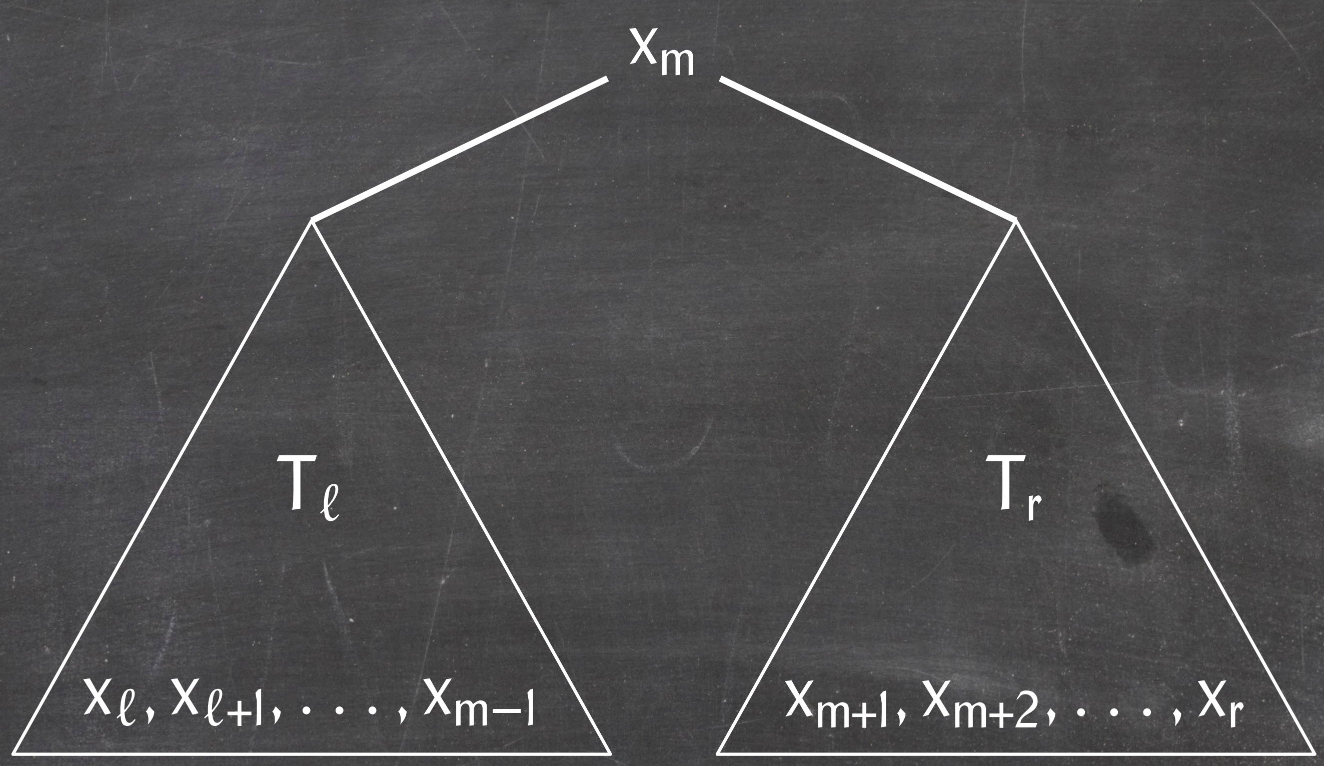
The structure of a binary search tree:

Assume we want to store elements $x_\ell, x_{\ell+1}, \ldots, x_r$.

Optimal Binary Search Trees: Problem Analysis

The structure of a binary search tree:

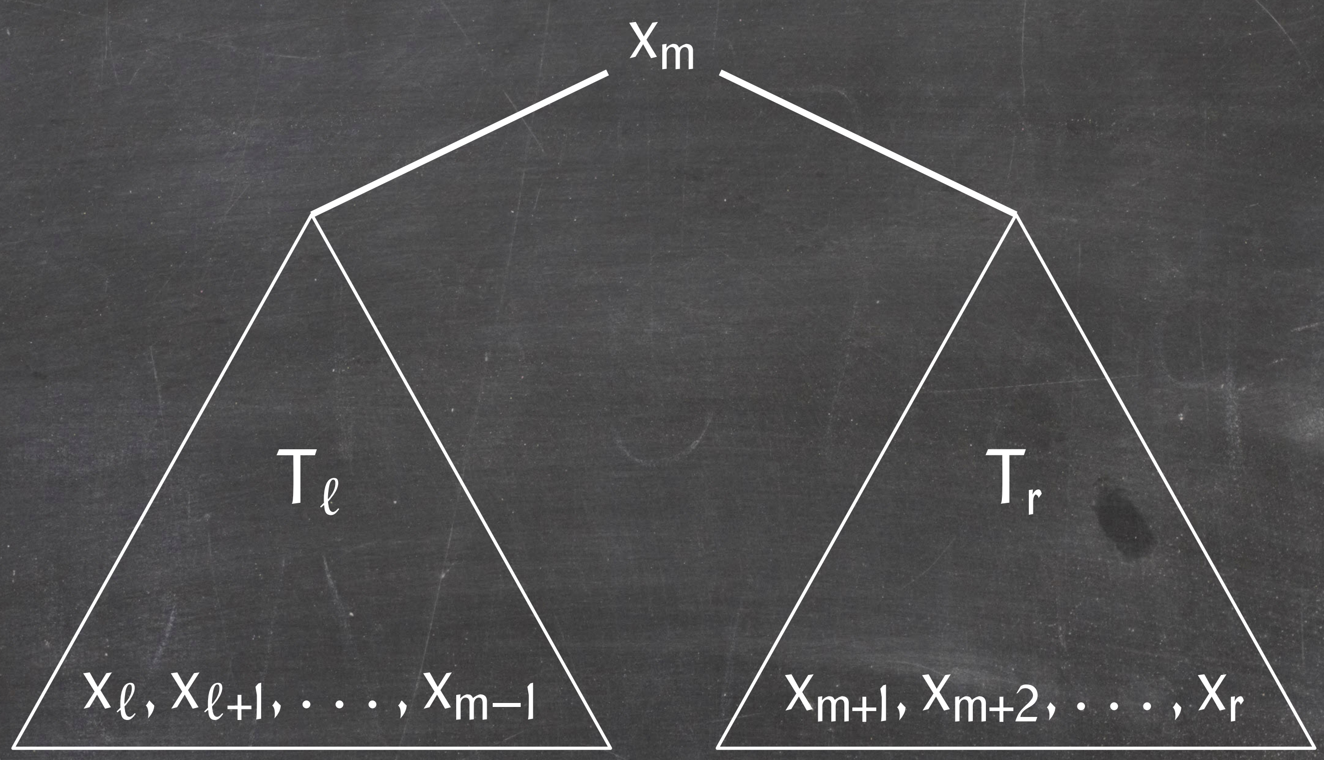
Assume we want to store elements $x_\ell, x_{\ell+1}, \ldots, x_r$.



Optimal Binary Search Trees: Problem Analysis

The structure of a binary search tree:

Assume we want to store elements $x_\ell, x_{\ell+1}, \ldots, x_r$.



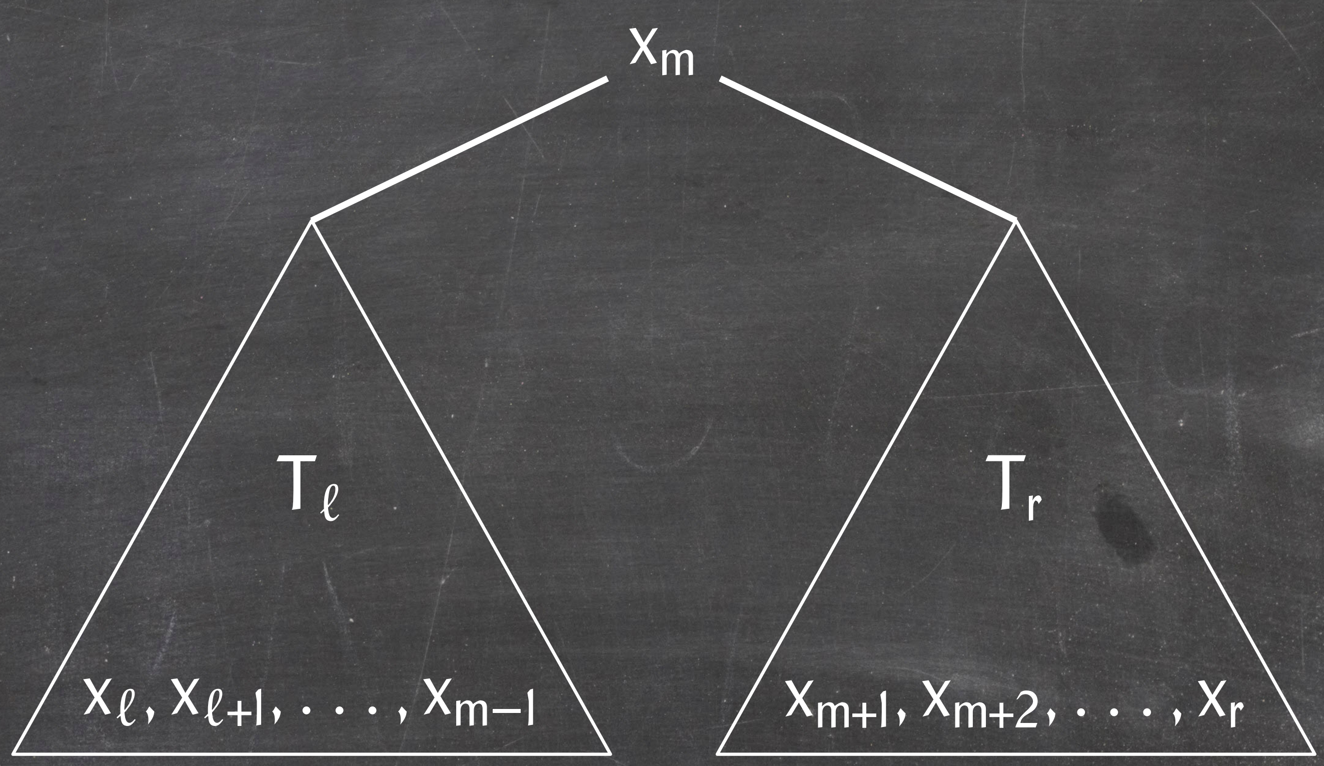
Let $p_{i,j} = \sum_{h=i}^{j} p_h$.

$$C_P(T) = p_{\ell,r} + C_P(T_\ell) + C_P(T_r)$$

Optimal Binary Search Trees: Problem Analysis

The structure of a binary search tree:

Assume we want to store elements $x_\ell, x_{\ell+1}, \ldots, x_r$.



Let $p_{i,j} = \sum_{h=i}^{j} p_h$.

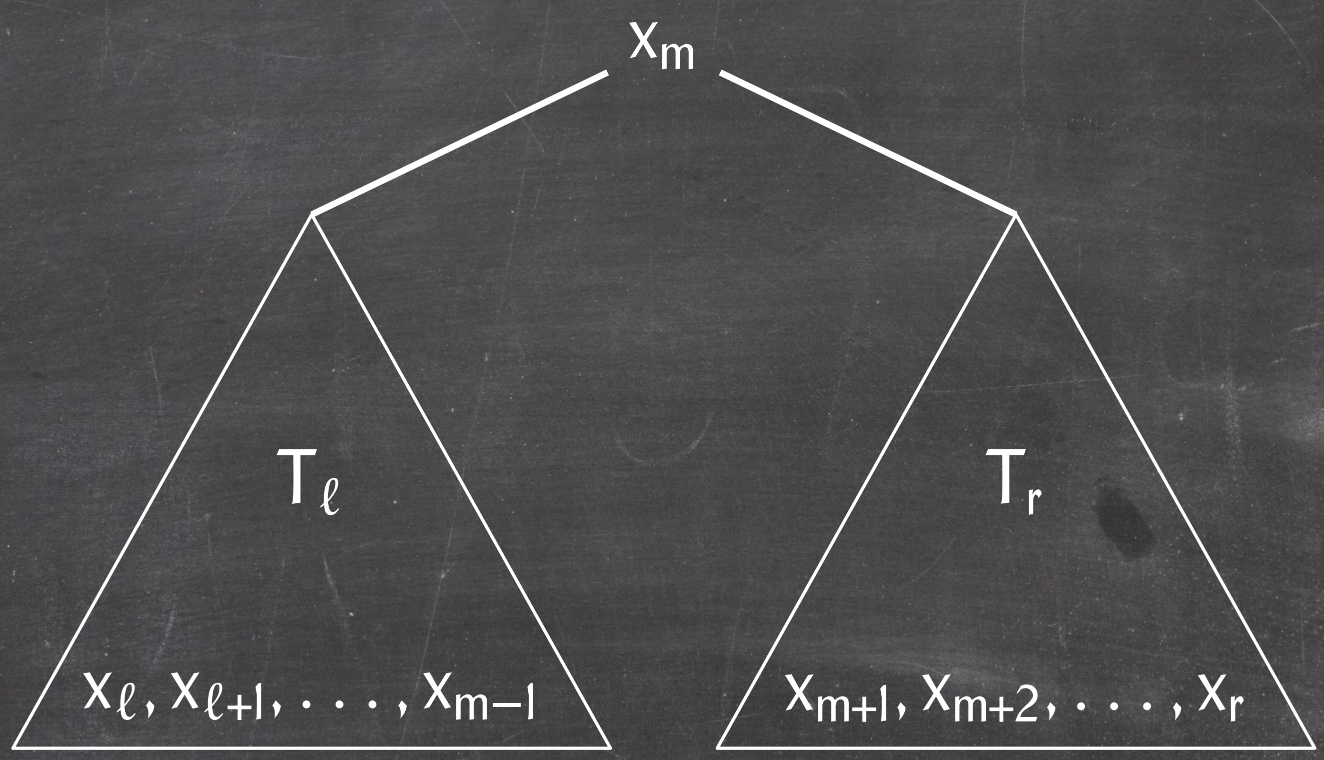
$$C_P(T) = p_{\ell,r} + C_P(T_\ell) + C_P(T_r)$$

$\Rightarrow$ T_ℓ and T_r are optimal search trees for $x_\ell, x_{\ell+1}, \ldots, x_{m-1}$ and $x_{m+1}, x_{m+2}, \ldots, x_r$, respectively.

Optimal Binary Search Trees: Problem Analysis

The structure of a binary search tree:

Assume we want to store elements $x_\ell, x_{\ell+1}, \ldots, x_r$.



Let $p_{i,j} = \sum_{h=i}^{j} p_h$.

$$C_P(T) = p_{\ell,r} + C_P(T_\ell) + C_P(T_r)$$

$\Rightarrow$ T_ℓ and T_r are optimal search trees for $x_\ell, x_{\ell+1}, \ldots, x_{m-1}$ and $x_{m+1}, x_{m+2}, \ldots, x_r$, respectively.

We need to figure out which element to store at the root!

Optimal Binary Search Trees: The Recurrence

Let $C(\ell, r)$ be the cost of an optimal binary search tree for $x_\ell, x_{\ell+1}, \ldots, x_r$.

We are interested in $C(1, n)$.

Optimal Binary Search Trees: The Recurrence

Let $C(\ell, r)$ be the cost of an optimal binary search tree for $x_\ell, x_{\ell+1}, \ldots, x_r$.

We are interested in $C(1, n)$.

$$C(\ell, r) = \begin{cases} 0 & r < \ell \\ p_{\ell,r} + \min_{\ell \le m \le r}(C_{\ell,m-1} + C_{m+1,r}) & \text{otherwise} \end{cases}$$

Optimal Binary Search Trees: The Algorithm

OptimalBinarySearchTree(X, P)

```
for i = 1 to n
    do P′[i, i] = P[i]
        for j = i + 1 to n
            do P′[i, j] = P′[i, j − 1] + P[j]
for i = 1 to n + 1
    do C[i, i − 1] = 0
        T[i, i − 1] = ∅
for ℓ = 0 to n − 1
    do for i = 1 to n − ℓ
            do C[i, i + ℓ] = ∞
                for j = i to i + ℓ
                    do if C[i, i + ℓ] > C[i, j − 1] + C[j + 1, i + ℓ]
                            then C[i, i + ℓ] = C[i, j − 1] + C[j + 1, i + ℓ]
                                 T[i, i + ℓ] = new node storing X[j]
                                 T[i, i + ℓ].left = T[i, j − 1]
                                 T[i, i + ℓ].right = T[j + 1, i + ℓ]
                C[i, i + ℓ] = C[i, i + ℓ] + P′[i, i + ℓ]
return T[1, n]
```

Optimal Binary Search Trees: The Algorithm

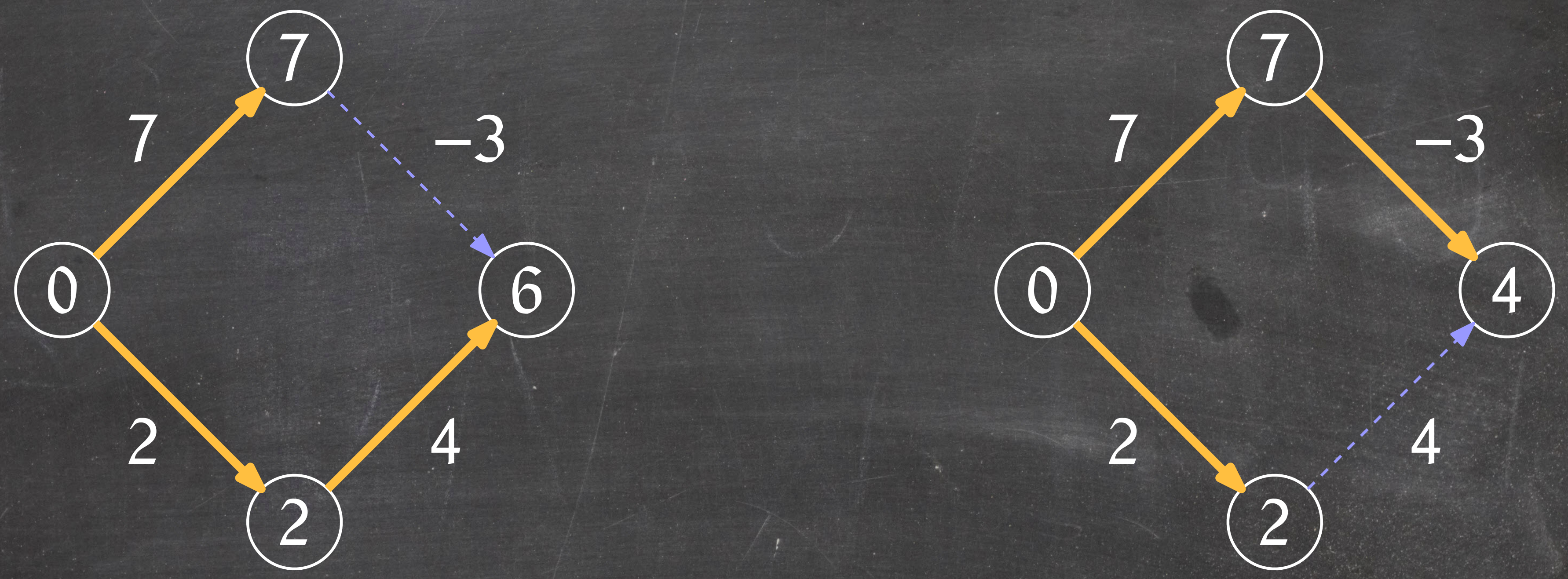
OptimalBinarySearchTree(X, P)

```
for i = 1 to n
    do P′[i, i] = P[i]
        for j = i + 1 to n
            do P′[i, j] = P′[i, j − 1] + P[j]
for i = 1 to n + 1
    do C[i, i − 1] = 0
        T[i, i − 1] = ∅
for ℓ = 0 to n − 1
    do for i = 1 to n − ℓ
            do C[i, i + ℓ] = ∞
                for j = i to i + ℓ
                    do if C[i, i + ℓ] > C[i, j − 1] + C[j + 1, i + ℓ]
                            then C[i, i + ℓ] = C[i, j − 1] + C[j + 1, i + ℓ]
                                 T[i, i + ℓ] = new node storing X[j]
                                 T[i, i + ℓ].left = T[i, j − 1]
                                 T[i, i + ℓ].right = T[j + 1, i + ℓ]
                C[i, i + ℓ] = C[i, i + ℓ] + P′[i, i + ℓ]
return T[1, n]
```

Lemma: An optimal binary search tree for n elements can be computed in $O(n^3)$ time.

Single-Source Shortest Paths

Dijkstra's algorithm may fail in the presence of negative-weight edges:

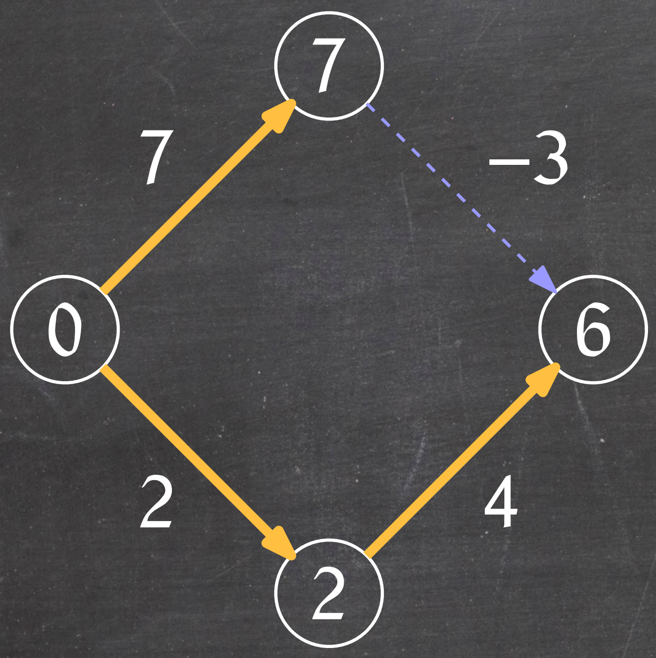


Dijkstra

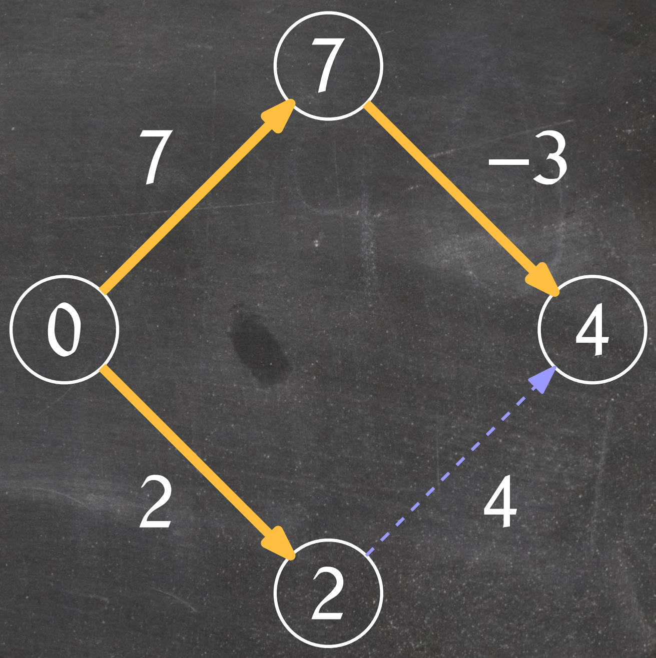
Correct

Single-Source Shortest Paths

Dijkstra's algorithm may fail in the presence of negative-weight edges:



Dijkstra

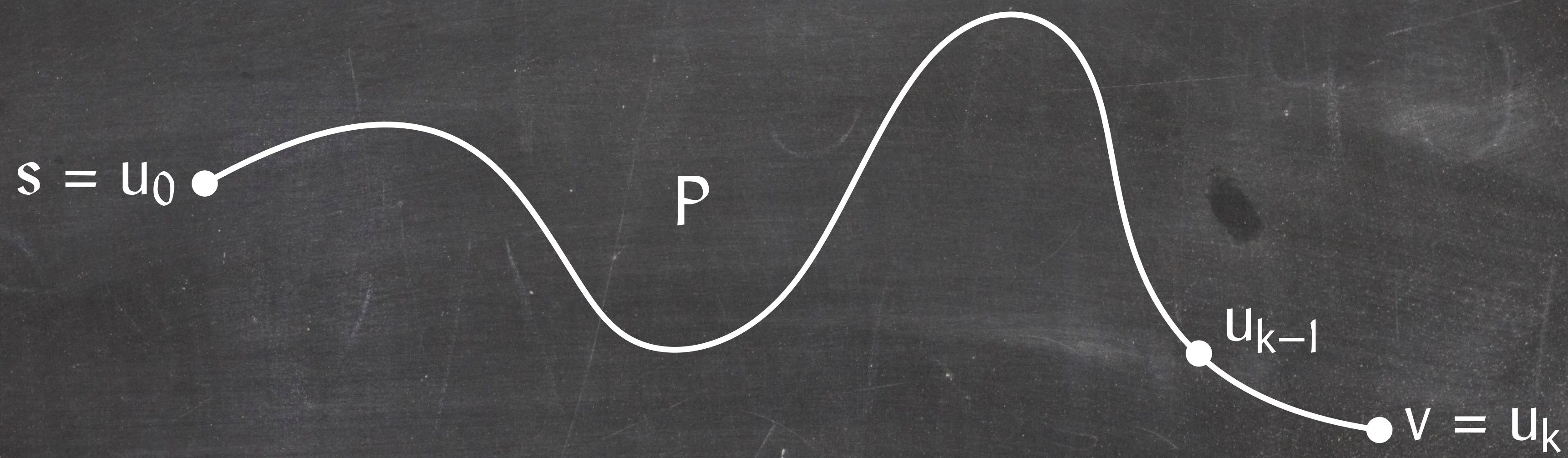


Correct

We need an algorithm that can deal with negative-length edges.

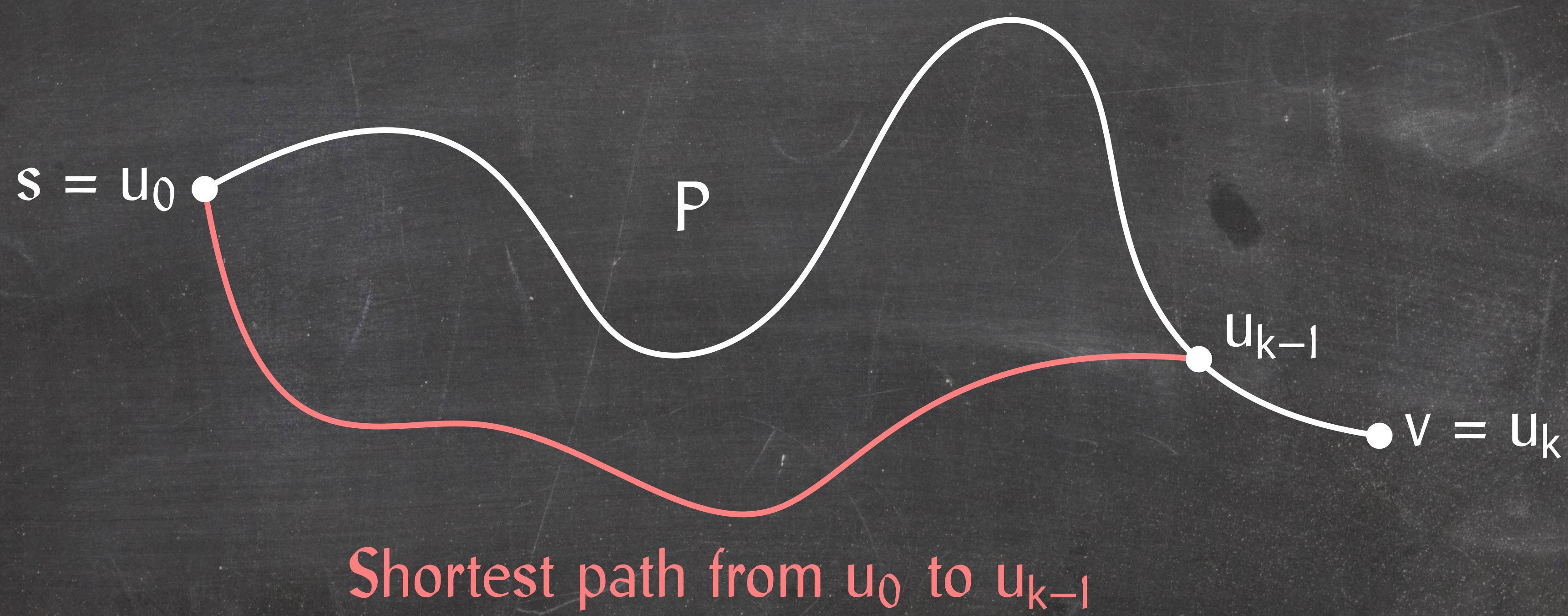
Single-Source Shortest Paths: Problem Analysis

Lemma: If $P = \langle u_0, v_1, \ldots, u_k \rangle$ is a shortest path from $u_0 = s$ to $u_k = v$, then $P' = (u_0, u_1, \ldots, u_{k-1})$ is a shortest path from u_0 to u_{k-1}.



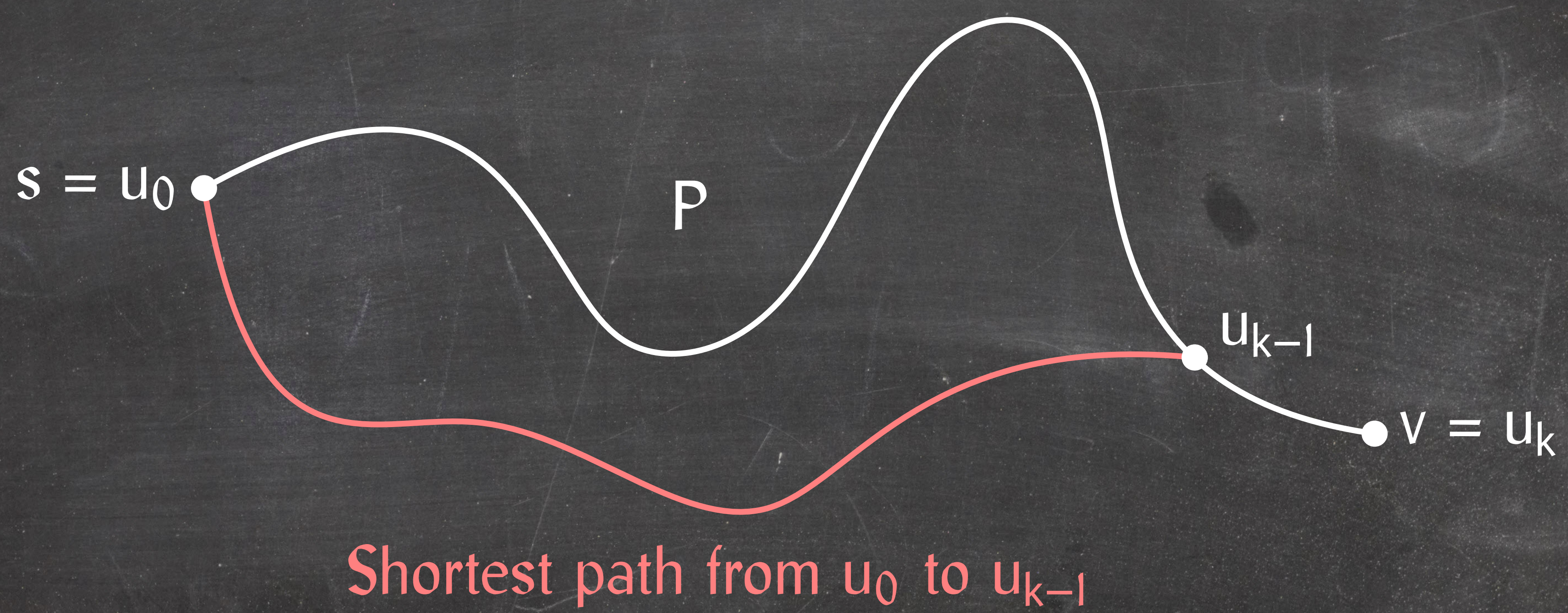
Single-Source Shortest Paths: Problem Analysis

Lemma: If $P = \langle u_0, v_1, \ldots, u_k \rangle$ is a shortest path from $u_0 = s$ to $u_k = v$, then $P' = (u_0, u_1, \ldots, u_{k-1})$ is a shortest path from u_0 to u_{k-1}.



Single-Source Shortest Paths: Problem Analysis

Lemma: If $P = \langle u_0, v_1, \ldots, u_k \rangle$ is a shortest path from $u_0 = s$ to $u_k = v$, then $P' = (u_0, u_1, \ldots, u_{k-1})$ is a shortest path from u_0 to u_{k-1}.



Observation: P' has one less edge than P.

Single-Source Shortest Paths: The Recurrence

Let $d_i(s, v)$ be the length of the shortest path $P_i(s, v)$ from s to v that has at most i edges.

Single-Source Shortest Paths: The Recurrence

Let $d_i(s, v)$ be the length of the shortest path $P_i(s, v)$ from s to v that has at most i edges.

$d_i(s, v) = \infty$ if there is no path with at most i edges from s to v.

Single-Source Shortest Paths: The Recurrence

Let $d_i(s, v)$ be the length of the shortest path $P_i(s, v)$ from s to v that has at most i edges.

$d_i(s, v) = \infty$ if there is no path with at most i edges from s to v.

$d(s, v) = d_{n-1}(s, v)$

Single-Source Shortest Paths: The Recurrence

Let $d_i(s, v)$ be the length of the shortest path $P_i(s, v)$ from s to v that has at most i edges.

$d_i(s, v) = \infty$ if there is no path with at most i edges from s to v.

$d(s, v) = d_{n-1}(s, v)$

Recurrence:

If $i = 0$, then there exists a path from s to v with at most i edges only if $v = s$:

$$d_0(s, v) = \begin{cases} 0 & v = s \\ \infty & \text{otherwise} \end{cases}$$

Single-Source Shortest Paths: The Recurrence

Let $d_i(s, v)$ be the length of the shortest path $P_i(s, v)$ from s to v that has at most i edges.

$d_i(s, v) = \infty$ if there is no path with at most i edges from s to v.

$d(s, v) = d_{n-1}(s, v)$

Recurrence:

If $i = 0$, then there exists a path from s to v with at most i edges only if $v = s$:

$$d_0(s, v) = \begin{cases} 0 & v = s \\ \infty & \text{otherwise} \end{cases}$$

If $i > 0$, then

Single-Source Shortest Paths: The Recurrence

Let $d_i(s, v)$ be the length of the shortest path $P_i(s, v)$ from s to v that has at most i edges.

$d_i(s, v) = \infty$ if there is no path with at most i edges from s to v.

$d(s, v) = d_{n-1}(s, v)$

Recurrence:

If $i = 0$, then there exists a path from s to v with at most i edges only if $v = s$:

$$d_0(s, v) = \begin{cases} 0 & v = s \\ \infty & \text{otherwise} \end{cases}$$

If $i > 0$, then

- $P_i(s, v)$ has at most $i - 1$ edges or

Single-Source Shortest Paths: The Recurrence

Let $d_i(s, v)$ be the length of the shortest path $P_i(s, v)$ from s to v that has at most i edges.

$d_i(s, v) = \infty$ if there is no path with at most i edges from s to v.

$d(s, v) = d_{n-1}(s, v)$

Recurrence:

If $i = 0$, then there exists a path from s to v with at most i edges only if $v = s$:

$$d_0(s, v) = \begin{cases} 0 & v = s \\ \infty & \text{otherwise} \end{cases}$$

If $i > 0$, then

- $P_i(s, v)$ has at most $i - 1$ edges or

- $P_i(s, v)$ has i edges.

Single-Source Shortest Paths: The Recurrence

Let $d_i(s, v)$ be the length of the shortest path $P_i(s, v)$ from s to v that has at most i edges.

$d_i(s, v) = \infty$ if there is no path with at most i edges from s to v.

$d(s, v) = d_{n-1}(s, v)$

Recurrence:

If $i = 0$, then there exists a path from s to v with at most i edges only if $v = s$:

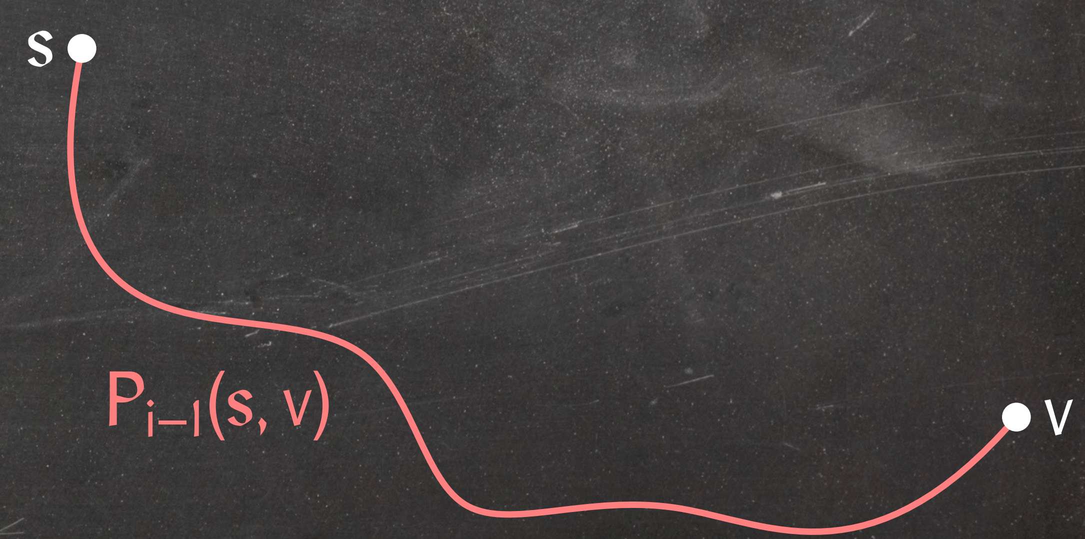
$$d_0(s, v) = \begin{cases} 0 & v = s \\ \infty & \text{otherwise} \end{cases}$$

If $i > 0$, then

- $P_i(s, v)$ has at most $i - 1$ edges or

$\Rightarrow P_i(s, v) = P_{i-1}(s, v)$

- $P_i(s, v)$ has i edges.



Single-Source Shortest Paths: The Recurrence

Let $d_i(s, v)$ be the length of the shortest path $P_i(s, v)$ from s to v that has at most i edges.

$d_i(s, v) = \infty$ if there is no path with at most i edges from s to v.

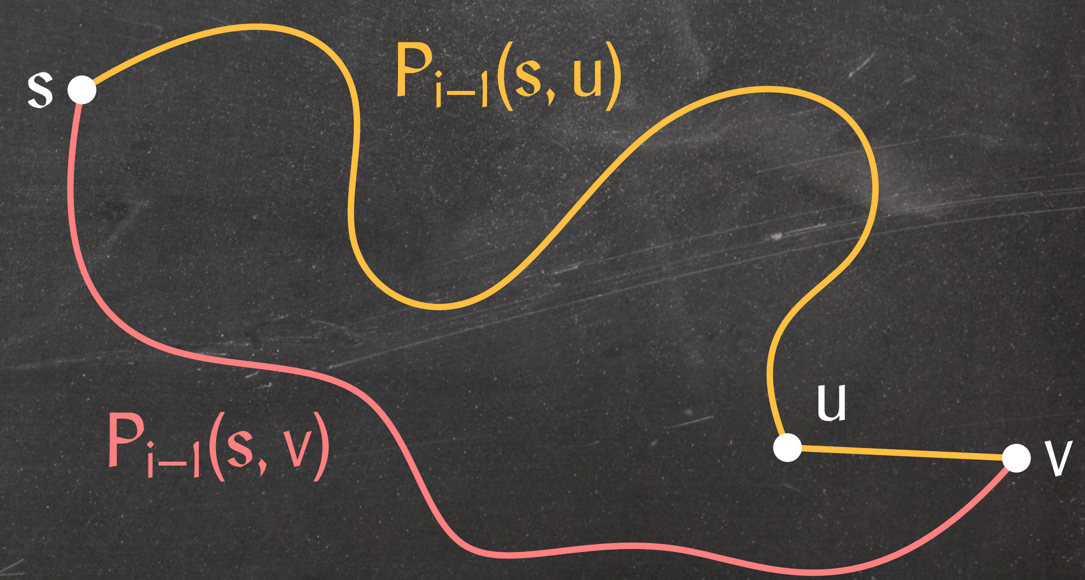
$d(s, v) = d_{n-1}(s, v)$

Recurrence:

If $i = 0$, then there exists a path from s to v with at most i edges only if $v = s$:

$$d_0(s, v) = \begin{cases} 0 & v = s \\ \infty & \text{otherwise} \end{cases}$$

If $i > 0$, then



- $P_i(s, v)$ has at most $i - 1$ edges or

$\Rightarrow$ $P_i(s, v) = P_{i-1}(s, v)$

- $P_i(s, v)$ has i edges.

$\Rightarrow$ $P_i(s, v) = P_{i-1}(s, u) \circ \langle(u, v)\rangle$ for some in-neighbour u of v.

Single-Source Shortest Paths: The Recurrence

Let $d_i(s, v)$ be the length of the shortest path $P_i(s, v)$ from s to v that has at most i edges.

$d_i(s, v) = \infty$ if there is no path with at most i edges from s to v.

$d(s, v) = d_{n-1}(s, v)$

Recurrence:

If $i = 0$, then there exists a path from s to v with at most i edges only if $v = s$:

$$d_0(s, v) = \begin{cases} 0 & v = s \\ \infty & \text{otherwise} \end{cases}$$

If $i > 0$, then

$$d_i(s, v) = \min(d_{i-1}(s, v), \min\{d_{i-1}(s, u) + w(u, v) \mid (u, v) \in E\})$$

Single-Source Shortest Paths: The Bellman-Ford Algorithm

BellmanFord(G, s)

```
for every vertex v ∈ G
   do d[v] = ∞
      P[v] = ∅
d[s] = 0
P[s] = [s]
for i = 1 to n − 1
   do for every vertex v ∈ G
         do for every in-edge e of v
               do if d[e.tail] + e.weight < d[v]
                     then d[v] = d[e.tail] + e.weight
                          P[v] = [v] ++ P[e.tail]
      return (d, P)
```

Single-Source Shortest Paths: The Bellman-Ford Algorithm

BellmanFord(G, s)

```
for every vertex v ∈ G
    do d[v] = ∞
       P[v] = ∅
d[s] = 0
P[s] = [s]
for i = 1 to n − 1
    do for every vertex v ∈ G
           do for every in-edge e of v
                  do if d[e.tail] + e.weight < d[v]
                         then d[v] = d[e.tail] + e.weight
                              P[v] = [v] ++ P[e.tail]
       return (d, P)
```

Lemma: The single-source shortest paths problem can be solved in $O(nm)$ time on any weighted graph, provided there are no negative cycles.

All-Pairs Shortest Paths

Goal: Compute the distance $d(u, v)$ (and the corresponding shortest path), for every pair of vertices $u, v \in G$.

All-Pairs Shortest Paths

Goal: Compute the distance $d(u, v)$ (and the corresponding shortest path), for every pair of vertices $u, v \in G$.

First idea: Run single-source shortest paths from every vertex $u \in G$.

All-Pairs Shortest Paths

Goal: Compute the distance $d(u, v)$ (and the corresponding shortest path), for every pair of vertices $u, v \in G$.

First idea: Run single-source shortest paths from every vertex $u \in G$.

Complexity:

- $O(n^2 m)$ using Bellman-Ford
- $O(n^2 \lg n + nm)$ for non-negative edge weights using Dijkstra

All-Pairs Shortest Paths

Goal: Compute the distance $d(u, v)$ (and the corresponding shortest path), for every pair of vertices $u, v \in G$.

First idea: Run single-source shortest paths from every vertex $u \in G$.

Complexity:

- $O(n^2 m)$ using Bellman-Ford
- $O(n^2 \lg n + nm)$ for non-negative edge weights using Dijkstra

Improved algorithms:

- Floyd-Warshall: $O(n^3)$

All-Pairs Shortest Paths

Goal: Compute the distance $d(u, v)$ (and the corresponding shortest path), for every pair of vertices $u, v \in G$.

First idea: Run single-source shortest paths from every vertex $u \in G$.

Complexity:

- $O(n^2 m)$ using Bellman-Ford
- $O(n^2 \lg n + nm)$ for non-negative edge weights using Dijkstra

Improved algorithms:

- Floyd-Warshall: $O(n^3)$
- Johnson: $O(n^2 \lg n + nm)$ (really cool!)
  - Run Bellman-Ford from an arbitrary vertex s in $O(nm)$ time.
  - Change edge weights so they are all non-negative but shortest paths don't change!
  - Run Dijkstra n times.

All-Pairs Shortest Paths: The Recurrence

Number the vertices $1, 2, \ldots, n$.

Let $d_i(u, v)$ be the length of the shortest path $P_i(u, v)$ that visits only vertices in $\{1, 2, \ldots, i\} \cup \{u, v\}$.

All-Pairs Shortest Paths: The Recurrence

Number the vertices $1, 2, \dots, n$.

Let $d_i(u, v)$ be the length of the shortest path $P_i(u, v)$ that visits only vertices in $\{1, 2, \dots, i\} \cup \{u, v\}$.

$d(u, v) = d_n(u, v)$

All-Pairs Shortest Paths: The Recurrence

Number the vertices $1, 2, \ldots, n$.

Let $d_i(u, v)$ be the length of the shortest path $P_i(u, v)$ that visits only vertices in $\{1, 2, \ldots, i\} \cup \{u, v\}$.

$d(u, v) = d_n(u, v)$

If $i = 0$, $P_0(u, v)$ cannot visit any vertices other than u and v:

$$d_0(u, v) = \begin{cases} w(u, v) & (u, v) \in E \\ \infty & \text{otherwise} \end{cases}$$

All-Pairs Shortest Paths: The Recurrence

Number the vertices $1, 2, \ldots, n$.

Let $d_i(u, v)$ be the length of the shortest path $P_i(u, v)$ that visits only vertices in $\{1, 2, \ldots, i\} \cup \{u, v\}$.

$d(u, v) = d_n(u, v)$

If $i = 0$, $P_0(u, v)$ cannot visit any vertices other than u and v:

$$d_0(u, v) = \begin{cases} w(u, v) & (u, v) \in E \\ \infty & \text{otherwise} \end{cases}$$

If $i > 0$, then $P_i(u, v)$ includes vertex i or it doesn't.

All-Pairs Shortest Paths: The Recurrence

Number the vertices $1, 2, \ldots, n$.

Let $d_i(u, v)$ be the length of the shortest path $P_i(u, v)$ that visits only vertices in $\{1, 2, \ldots, i\} \cup \{u, v\}$.

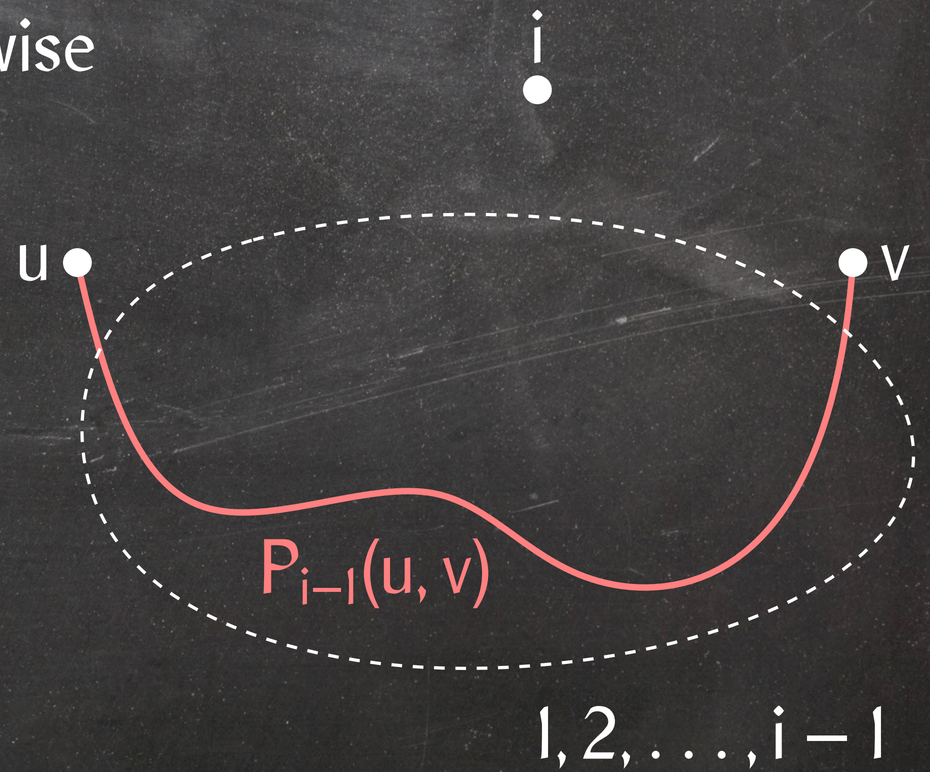
$d(u, v) = d_n(u, v)$

If $i = 0$, $P_0(u, v)$ cannot visit any vertices other than u and v:

$$d_0(u, v) = \begin{cases} w(u, v) & (u, v) \in E \\ \infty & \text{otherwise} \end{cases}$$

If $i > 0$, then $P_i(u, v)$ includes vertex i or it doesn't.

- If $i \notin P_i(u, v)$, then $P_i(u, v) = P_{i-1}(u, v)$.



All-Pairs Shortest Paths: The Recurrence

Number the vertices $1, 2, \ldots, n$.

Let $d_i(u, v)$ be the length of the shortest path $P_i(u, v)$ that visits only vertices in $\{1, 2, \ldots, i\} \cup \{u, v\}$.

$d(u, v) = d_n(u, v)$

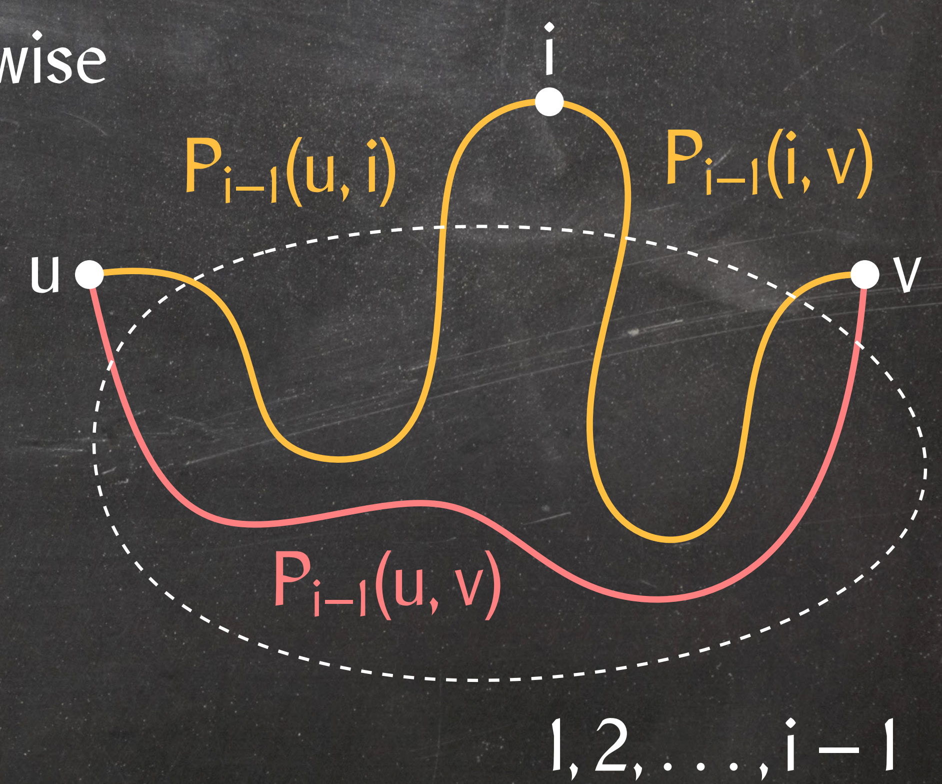
If $i = 0$, $P_0(u, v)$ cannot visit any vertices other than u and v:

$$d_0(u, v) = \begin{cases} w(u, v) & (u, v) \in E \\ \infty & \text{otherwise} \end{cases}$$

If $i > 0$, then $P_i(u, v)$ includes vertex i or it doesn't.

If $i \notin P_i(u, v)$, then $P_i(u, v) = P_{i-1}(u, v)$.

If $i \in P_i(u, v)$, then $P_i(u, v) = P_{i-1}(u, i) \circ P_{i-1}(i, v)$.



All-Pairs Shortest Paths: The Recurrence

Number the vertices $1, 2, \ldots, n$.

Let $d_i(u, v)$ be the length of the shortest path $P_i(u, v)$ that visits only vertices in $\{1, 2, \ldots, i\} \cup \{u, v\}$.

$d(u, v) = d_n(u, v)$

If $i = 0$, $P_0(u, v)$ cannot visit any vertices other than u and v:

$$d_0(u, v) = \begin{cases} w(u, v) & (u, v) \in E \\ \infty & \text{otherwise} \end{cases}$$

If $i > 0$, then $P_i(u, v)$ includes vertex i or it doesn't.

If $i \notin P_i(u, v)$, then $P_i(u, v) = P_{i-1}(u, v)$.

If $i \in P_i(u, v)$, then $P_i(u, v) = P_{i-1}(u, i) \circ P_{i-1}(i, v)$.

$$d_i(u, v) = \min(d_{i-1}(u, v), d_{i-1}(u, i) + d_{i-1}(i, v))$$

All-Pairs Shortest Paths: The Floyd-Warshall Algorithm

FloydWarshall(G)

```
for every pair of vertices u, v ∈ G
    do d[u, v] = ∞
       p[u, v] = Nothing
for every vertex v ∈ G
    do d[v, v] = 0
       p[v, v] = v
for every edge e ∈ G
    do d[e.tail, e.head] = e.weight
       p[e.tail, e.head] = e.tail
for i = 1 to n
    do for every pair of vertices u, v ∈ G such that i ∉ {u, v}
           do if d[u, v] > d[u, i] + d[i, v]
                 then d[u, v] = d[u, i] + d[i, v]
                      p[u, v] = p[i, v]
return (d, p)
```

All-Pairs Shortest Paths: The Floyd-Warshall Algorithm

FloydWarshall(G)

```
for every pair of vertices u, v ∈ G
    do d[u, v] = ∞
       p[u, v] = Nothing
for every vertex v ∈ G
    do d[v, v] = 0
       p[v, v] = v
for every edge e ∈ G
    do d[e.tail, e.head] = e.weight
       p[e.tail, e.head] = e.tail
for i = 1 to n
    do for every pair of vertices u, v ∈ G such that i ∉ {u, v}
           do if d[u, v] > d[u, i] + d[i, v]
                 then d[u, v] = d[u, i] + d[i, v]
                      p[u, v] = p[i, v]
return (d, p)
```

ReportPath(p, u, v)

```
if p[u, v] = Nothing
    then return Nothing
P = [v]
while v ≠ u
    do v = p[u, v]
       P.prepend(v)
return P
```

All-Pairs Shortest Paths: The Floyd-Warshall Algorithm

FloydWarshall(G)

```
for every pair of vertices u, v ∈ G
    do d[u, v] = ∞
       p[u, v] = Nothing
for every vertex v ∈ G
    do d[v, v] = 0
       p[v, v] = v
for every edge e ∈ G
    do d[e.tail, e.head] = e.weight
       p[e.tail, e.head] = e.tail
for i = 1 to n
    do for every pair of vertices u, v ∈ G such that i ∉ {u, v}
            do if d[u, v] > d[u, i] + d[i, v]
                    then d[u, v] = d[u, i] + d[i, v]
                         p[u, v] = p[i, v]
return (d, p)
```

ReportPath(p, u, v)

```
if p[u, v] = Nothing
    then return Nothing
P = [v]
while v ≠ u
    do v = p[u, v]
       P.prepend(v)
return P
```

Lemma: The all-pairs shortest paths problem can be solved in $O(n^3)$ time, provided there are no negative cycles.

Summary

Both greedy algorithms and dynamic programming are applicable when the problem has optimal substructure:

> The optimal solution for a given input instance contains within it optimal solutions to smaller input instances.

Summary

Both greedy algorithms and dynamic programming are applicable when the problem has optimal substructure:

> The optimal solution for a given input instance contains within it optimal solutions to smaller input instances.

Greedy algorithms are applicable when an optimal solution can be obtained by making a locally optimal choice and then solving the resulting subproblem.

Summary

Both greedy algorithms and dynamic programming are applicable when the problem has optimal substructure:

> The optimal solution for a given input instance contains within it optimal solutions to smaller input instances.

Greedy algorithms are applicable when an optimal solution can be obtained by making a locally optimal choice and then solving the resulting subproblem.

Dynamic programming exhaustively explores all possible choices and chooses the one that gives the best solution.

Summary

Both greedy algorithms and dynamic programming are applicable when the problem has optimal substructure:

> The optimal solution for a given input instance contains within it optimal solutions to smaller input instances.

Greedy algorithms are applicable when an optimal solution can be obtained by making a locally optimal choice and then solving the resulting subproblem.

Dynamic programming exhaustively explores all possible choices and chooses the one that gives the best solution.

Dynamic programming yields a faster solution than the naïve recursive algorithm when there are lots of overlapping subproblems.

Summary

The design of a dynamic programming algorithm proceeds in two phases:

1. Analyze the structure of an optimal solution to develop a recurrence for the cost of an optimal solution.

2. Develop an algorithm that uses the recurrence to compute an optimal solution
   - Recursively using memoization or
   - Iteratively by populating a table with the costs of the solutions to all possible subproblems.

   Both types of algorithms compute optimal solutions bottom-up.