Analysis of Algorithms

Dynamic Programming

Andres Mendez-Vazquez

February 14, 2018

Outline

1. Dynamic Programming
   - Bellman Equation
   - Elements of Dynamic Programming
   - Rod Cutting
2. Elements of Dynamic Programming
   - Optimal Substructure
   - Overlapping Subproblems
   - Reconstruction of Subproblems
   - Common Subproblems
3. Examples
   - Longest Increasing Subsequence
   - Matrix Multiplication
   - Longest Common Subsequence
4. Exercises

History

Dynamic Programming

The dynamic programming was developed in 1940's by Richard Bellman at RAND Corporation to solve problems by taking the best decisions one after another.

History

Dynamic Programming

The dynamic programming was developed in 1940's by Richard Bellman at RAND Corporation to solve problems by taking the best decisions one after another.

You can think as

1. Sending a recursive function to do different jobs.
2. Then, at the top of the recursion decide which job is the best one.

History

Dynamic Programming

The dynamic programming was developed in 1940's by Richard Bellman at RAND Corporation to solve problems by taking the best decisions one after another.

You can think as

1. Sending a recursive function to do different jobs.
2. Then, at the top of the recursion decide which job is the best one.

Actually the name comes from two notions

- **Dynamic** was chosen by Bellman to capture the temporal part of the problem.
- **Programming** referred to finding the optimal program in military logistic.

History

Dynamic Programming

The dynamic programming was developed in 1940's by Richard Bellman at RAND Corporation to solve problems by taking the best decisions one after another.

You can think as

1. Sending a recursive function to do different jobs.
2. Then, at the top of the recursion decide which job is the best one.

Actually the name comes from two notions

- **Dynamic** was chosen by Bellman to capture the temporal part of the problem.
- **Programming** referred to finding the optimal program in military logistic.

History

Dynamic Programming

The dynamic programming was developed in 1940's by Richard Bellman at RAND Corporation to solve problems by taking the best decisions one after another.

You can think as

1. Sending a recursive function to do different jobs.
2. Then, at the top of the recursion decide which job is the best one.

Actually the name comes from two notions

- **Dynamic** was chosen by Bellman to capture the temporal part of the problem.
- **Programming** referred to finding the optimal program in military logistic.

Outline

Bellman Equation

Definition

$$V(x_0) = \max_{a_0}[F(x_0) + \beta V(x_1)]$$

$$\text{s.t. } a_0 \in \Gamma(x_0), x_1 = T(x_0, a_0)$$

- Where $\Gamma(x_0)$ is a set of actions depend on the current state.
- $T(x_0, a_0)$ is a transition function.
- $F(x_0)$ payoff.

Bellman Equation

Definition

$$V(x_0) = \max_{a_0}[F(x_0) + \beta V(x_1)]$$

$$\text{s.t. } a_0 \in \Gamma(x_0), x_1 = T(x_0, a_0)$$

- Where $\Gamma(x_0)$ is a set of actions depend on the current state.
- $T(x_0, a_0)$ is a transition function.
- $F(x_0)$ payoff.

Bellman Equation

Definition

$$V(x_0) = \max_{a_0}[F(x_0) + \beta V(x_1)]$$

$$\text{s.t. } a_0 \in \Gamma(x_0), x_1 = T(x_0, a_0)$$

- Where $\Gamma(x_0)$ is a set of actions depend on the current state.
- $T(x_0, a_0)$ is a transition function.
- $F(x_0)$ payoff.

Bellman Equation

Definition

$$V(x_0) = \max_{a_0}[F(x_0) + \beta V(x_1)]$$

$$\text{s.t. } a_0 \in \Gamma(x_0), x_1 = T(x_0, a_0)$$

- Where $\Gamma(x_0)$ is a set of actions depend on the current state.
- $T(x_0, a_0)$ is a transition function.
- $F(x_0)$ payoff.

Looks Terrifying!!!

However

It is quite simple!!!

Outline

Elements of Dynamic Programming

Define the Optimal Structure

Characterize the structure of an optimal solution.

Define the Recursion

Recursively define the value of an optimal solution.

Compute the Solution

Compute the value of an optimal solution, typically bottom-up.

IMPORTANT!!!

We use an extra memory to stop the recursion!!!

Elements of Dynamic Programming

Define the Optimal Structure

Characterize the structure of an optimal solution.

Define the Recursion

Recursively define the value of an optimal solution.

Compute the Solution

Compute the value of an optimal solution, typically bottom-up.

IMPORTANT!!!

We use an extra memory to stop the recursion!!!

Elements of Dynamic Programming

Define the Optimal Structure

Characterize the structure of an optimal solution.

Define the Recursion

Recursively define the value of an optimal solution.

Compute the Solution

Compute the value of an optimal solution, typically bottom-up.

IMPORTANT!!!

We use an extra memory to stop the recursion!!!

Elements of Dynamic Programming

Define the Optimal Structure

Characterize the structure of an optimal solution.

Define the Recursion

Recursively define the value of an optimal solution.

Compute the Solution

Compute the value of an optimal solution, typically bottom-up.

IMPORTANT!!!

We use an extra memory to stop the recursion!!!

Elements of Dynamic Programming

Finally Rebuild the Optimal Solution

Construct an optimal solution from computed information.

Outline

Rod cutting

Problem

Given a rod of length n inches and a table of prices p_i for $i = 1, 2, ..., n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

Rod Cutting table

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Rod cutting

Problem

Given a rod of length n inches and a table of prices p_i for $i = 1, 2, ..., n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

Rod Cutting table

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Characterize the structure of an optimal solution

Example

For example for a rod of size 10, we could cut the rod in 3 parts, $10=4+3+3$.

Thus

Then, we can assume that an optimal solution cuts the rod in k pieces, $1 \leq k \leq n$ i.e. $k-1$ cuts.

Then

What?

Characterize the structure of an optimal solution

Example

For example for a rod of size 10, we could cut the rod in 3 parts, 10=4+3+3.

Thus

Then, we can assume that an optimal solution cuts the rod in k pieces, $1 \leq k \leq n$ i.e. $k-1$ cuts.

Then

What?

Characterize the structure of an optimal solution

Example

For example for a rod of size 10, we could cut the rod in 3 parts, 10=4+3+3.

Thus

Then, we can assume that an optimal solution cuts the rod in k pieces, $1 \leq k \leq n$ i.e. $k-1$ cuts.

Then

What?

Thus

The length of each piece can be numbered as

i_j with $1 \leq j \leq k$

The total size of the rod is then

Thus, the max revenue

Thus

The length of each piece can be numbered as

$$i_j \text{ with } 1 \leq j \leq k$$

The total size of the rod is then

$$n = i_1 + i_2 + ... + i_k$$

Thus, the max revenue

Thus

The length of each piece can be numbered as

$$i_j \text{ with } 1 \leq j \leq k$$

The total size of the rod is then

$$n = i_1 + i_2 + ... + i_k$$

Thus, the max revenue

$$r_n = p_{i_1} + p_{i_2} + ... + p_{i_k}$$

Example

For length $n = 4$ by brute force approach

1. price equal to 9
2. price equal to 1+8
3. price equal to 8+1
4. price equal to 1+1+5
5. price equal to 1+5+1
6. price equal to 5+1+1
7. price equal to 1+1+1+1
8. price equal to 5+5 Optimal!!!

Example

For length $n = 4$ by brute force approach

1. price equal to 9
2. price equal to 1+8
3. price equal to 8+1
4. price equal to 1+1+5
5. price equal to 1+5+1
6. price equal to 5+1+1
7. price equal to 1+1+1+1
8. price equal to 5+5 Optimal!!!

Example

For length $n = 4$ by brute force approach

1. price equal to 9
2. price equal to 1+8
3. price equal to 8+1
4. price equal to 1+1+5
5. price equal to 1+5+1
6. price equal to 5+1+1
7. price equal to 1+1+1+1
8. price equal to 5+5 Optimal!!!

Example

For length $n = 4$ by brute force approach

1. price equal to 9
2. price equal to 1+8
3. price equal to 8+1
4. price equal to 1+1+5
5. price equal to 1+5+1
6. price equal to 5+1+1
7. price equal to 1+1+1+1
8. price equal to 5+5 Optimal!!!

Example

For length $n = 4$ by brute force approach

1. price equal to 9
2. price equal to 1+8
3. price equal to 8+1
4. price equal to 1+1+5
5. price equal to 1+5+1
6. price equal to 5+1+1
7. price equal to 1+1+1+1
8. price equal to 5+5 Optimal!!!

Example

For length $n = 4$ by brute force approach

1. price equal to 9
2. price equal to 1+8
3. price equal to 8+1
4. price equal to 1+1+5
5. price equal to 1+5+1
6. price equal to 5+1+1
7. price equal to 1+1+1+1
8. price equal to 5+5 Optimal!!!

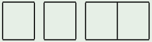
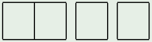
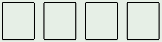
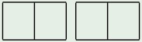
Example

For length $n = 4$ by brute force approach

1. price equal to 9
2. price equal to 1+8
3. price equal to 8+1
4. price equal to 1+1+5
5. price equal to 1+5+1
6. price equal to 5+1+1
7. price equal to 1+1+1+1
8. price equal to 5+5 Optimal!!!

Example

For length $n = 4$ by brute force approach

1.  price equal to 9
2.  price equal to 1+8
3.  price equal to 8+1
4.  price equal to 1+1+5
5.  price equal to 1+5+1
6.  price equal to 5+1+1
7.  price equal to 1+1+1+1
8.  **price equal to 5+5 Optimal!!!**

How can you obtain the recursion?

What about taking a decision each time?

In how to cut the rod!

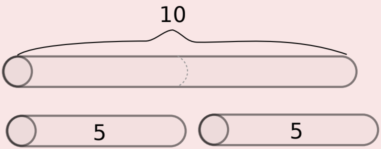
For example

How can you obtain the recursion?

What about taking a decision each time?

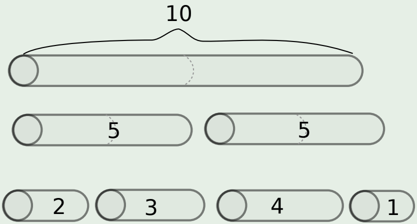
In how to cut the rod!

For example



It looks like what?

One more cut

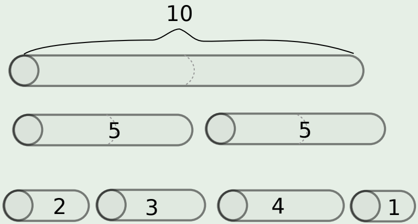


Yes

Recursion

It looks like what?

One more cut



Yes

Recursion

Thus, What can we do next?

We need to take decisions

One cut at each step.

Thus, What can we do next?

We need to take decisions

One cut at each step.

For example

1. No cut $n \Longrightarrow p_n$
2. $n = i_1 + i_{n-1} \Longrightarrow r_n = r_1 + r_{n-1}$
3. $n = i_2 + i_{n-2} \Longrightarrow r_n = r_2 + r_{n-2}$
4. $\ldots$

Thus, What can we do next?

We need to take decisions

One cut at each step.

For example

1. No cut $n \Longrightarrow p_n$
2. $n = i_1 + i_{n-1} \Longrightarrow r_n = r_1 + r_{n-1}$
3. $n = i_2 + i_{n-2} \Longrightarrow r_n = r_2 + r_{n-2}$
4. ...

In general

$n = i_j + i_{n-j} \Longrightarrow r = r_j + r_{n-1}$ for $j = 1, 2, ..., n-1$

Thus, What can we do next?

We need to take decisions

One cut at each step.

For example

1. No cut $n \Longrightarrow p_n$
2. $n = i_1 + i_{n-1} \Longrightarrow r_n = r_1 + r_{n-1}$
3. $n = i_2 + i_{n-2} \Longrightarrow r_n = r_2 + r_{n-2}$
4. ...

In general

$n = i_j + i_{n-j} \Longrightarrow r = r_j + r_{n-1}$ for $j = 1, 2, ..., n-1$

Thus, What can we do next?

We need to take decisions

One cut at each step.

For example

1. No cut $n \Longrightarrow p_n$
2. $n = i_1 + i_{n-1} \Longrightarrow r_n = r_1 + r_{n-1}$
3. $n = i_2 + i_{n-2} \Longrightarrow r_n = r_2 + r_{n-2}$
4. $\ldots$

In general

$n = i_j + i_{n-j} \Longrightarrow r = r_j + r_{n-1}$ for $j = 1, 2, ..., n-1$

Thus, we take a final decision!!!

Thus

Which One?

The Largest One

Thus, we take a final decision!!!

Thus

Which One?

The Largest One

$$r_n = \max\left\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, ..., r_{n-1} + r_1\right\}$$

Some stuff about the optimal solution

Did you notice the following?

Once you get an optimal solution!!! The Most Revenue!!!

Some stuff about the optimal solution

Did you notice the following?

Once you get an optimal solution!!! The Most Revenue!!!

The sub-solutions are optimal

Why?

Some stuff about the optimal solution

Did you notice the following?

Once you get an optimal solution!!! The Most Revenue!!!

The sub-solutions are optimal

Why?

Use contradiction

1. Imagine that a sub-solution has a better solution...
2. Then, you can substitute it in the original sub-solution.
3. Thus, you get something better than the original one.

Some stuff about the optimal solution

Did you notice the following?

Once you get an optimal solution!!! The Most Revenue!!!

The sub-solutions are optimal

Why?

Use contradiction

1. Imagine that a sub-solution has a better solution...
2. Then, you can substitute it in the original sub-solution.
3. Thus, you get something better than the original one.

Some stuff about the optimal solution

Did you notice the following?

Once you get an optimal solution!!! The Most Revenue!!!

The sub-solutions are optimal

Why?

Use contradiction

1. Imagine that a sub-solution has a better solution...
2. Then, you can substitute it in the original sub-solution.
3. Thus, you get something better than the original one.

Formally: Cut and Paste

Given

$n = i_1 + i_2 + ... + i_k$

Imagine, we split the problem in two parts

$A_1 = \{i_1, i_2, ..., i_l\}$ and $A_2 = \{i_{l+1}, i_2, ..., i_k\}$

Properties

Now imagine that exist a $A_1' = \left\{i_1', i_2', ..., i_l'\right\}$ such that:

$$r_n' = p_{i_1'} + p_{i_2'} + ... + p_{i_l'} > r_n = p_{i_1} + p_{i_2} + ... + p_{i_l}$$

Formally: Cut and Paste

Given

$n = i_1 + i_2 + ... + i_k$

Imagine, we split the problem in two parts

$A_1 = \{i_{1,}, i_2, ..., i_l\}$ and $A_2 = \{i_{l+1,}, i_2, ..., i_k\}$

Properties

Now imagine that exist a $A_1^{'} = \left\{i_1^{'}, i_2^{'}, ..., i_l^{'}\right\}$ such that:

$$r_n^{'} = p_{i_1^{'}} + p_{i_2^{'}} + ... + p_{i_l^{'}} > r_n = p_{i_1} + p_{i_2} + ... + p_{i_l}$$

Formally: Cut and Paste

Given

$n = i_1 + i_2 + ... + i_k$

Imagine, we split the problem in two parts

$A_1 = \{i_{1,}, i_2, ..., i_l\}$ and $A_2 = \{i_{l+1,}, i_2, ..., i_k\}$

Properties

Now imagine that exist a $A_1^{'} = \left\{i_1^{'}, i_2^{'}, ..., i_l\right\}$ such that:

$$r_n^{'} = p_{i_1^{'}} + p_{i_2^{'}} + ... + p_{i_{l'}} > r_n = p_{i_1} + p_{i_2} + ... + p_{i_l}$$

Then

Then, we have a set of cuts

$A_1' \cup A_2$ with better revenue than the original cut-set!!!

Clearly

Contradiction!!!

Then

Then, we have a set of cuts

$A_1' \cup A_2$ with better revenue than the original cut-set!!!

Clearly

Contradiction!!!

Rewrite the equation to simplify recursion

Did you notice that?

We can add a dummy variable $r_0 = 0$

Rewrite the equation to simplify recursion

Did you notice that?

We can add a dummy variable $r_0 = 0$

In addition, we have that

$$r_i = p_i \text{ for } i = 1, 2, ..., n$$

Rewrite the equation to simplify recursion

Did you notice that?

We can add a dummy variable $r_0 = 0$

In addition, we have that

$$r_i = p_i \text{ for } i = 1, 2, ..., n$$

We can then apply this...

1. $p_n = p_n + r_0$
2. $r_1 + r_{n-1} = p_1 + r_{n-1}$
3. $r_2 + r_{n-2} = p_2 + r_{n-2}$
4. ...

Rewrite the equation to simplify recursion

Did you notice that?

We can add a dummy variable $r_0 = 0$

In addition, we have that

$$r_i = p_i \text{ for } i = 1, 2, ..., n$$

We can then apply this...

1. $p_n = p_n + r_0$
2. $r_1 + r_{n-1} = p_1 + r_{n-1}$
3. $r_2 + r_{n-2} = p_2 + r_{n-2}$
4. ...

Rewrite the equation to simplify recursion

Did you notice that?

We can add a dummy variable $r_0 = 0$

In addition, we have that

$$r_i = p_i \text{ for } i = 1, 2, ..., n$$

We can then apply this...

1. $p_n = p_n + r_0$
2. $r_1 + r_{n-1} = p_1 + r_{n-1}$
3. $r_2 + r_{n-2} = p_2 + r_{n-2}$
4. ...

Rewrite the equation to simplify recursion

Did you notice that?

We can add a dummy variable $r_0 = 0$

In addition, we have that

$$r_i = p_i \text{ for } i = 1, 2, ..., n$$

We can then apply this...

1. $p_n = p_n + r_0$
2. $r_1 + r_{n-1} = p_1 + r_{n-1}$
3. $r_2 + r_{n-2} = p_2 + r_{n-2}$
4. ...

Then

We have that

$$r_n = \max_{1 \leq i \leq n} \left(p_i + r_{n-i}\right)$$

Then

We have that

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

So we need to convert this into something more programmable

You can define Cut-Rod$(p, n - i)$ where

- p is an array with the table values.
- $n - i$ is the size of the rod when going into the recursion.

Then

We have that

$$r_n = \max_{1 \leq i \leq n} \left(p_i + r_{n-i} \right)$$

So we need to convert this into something more programmable

You can define Cut-Rod$(p, n-i)$ where

- p is an array with the table values.
- $n-i$ is the size of the rod when going into the recursion.

Then

We have that

$$r_n = \max_{1 \leq i \leq n} \left(p_i + r_{n-i} \right)$$

So we need to convert this into something more programmable

You can define Cut-Rod$(p, n-i)$ where

- p is an array with the table values.
- $n-i$ is the size of the rod when going into the recursion.

Then

We have that

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

So we need to convert this into something more programmable

You can define Cut-Rod$(p, n-i)$ where

- p is an array with the table values.
- $n-i$ is the size of the rod when going into the recursion.

Finally

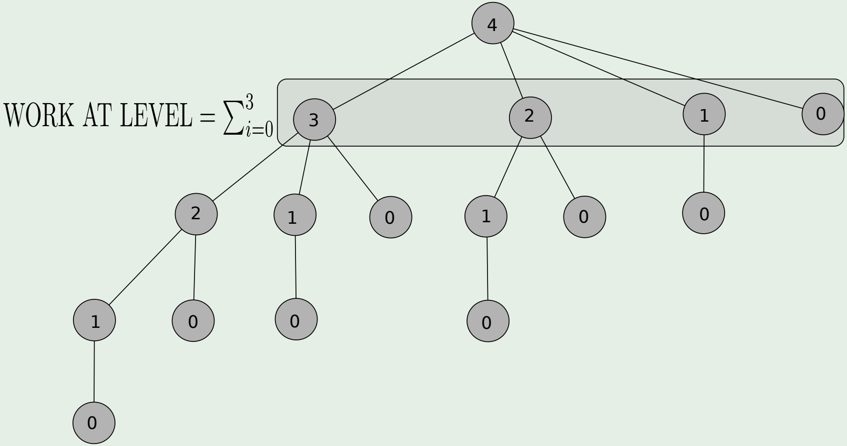
Code

Cut-Rod(p, n)

1. if $n == 0$
2. return 0
3. $q = -\infty$
4. for $i = 1$ to n
5. $q = \max\{q, p[i] + \text{Cut-Rod}(p, n - i)\}$
6. return q

How the recursion tree for this code looks like?

First, Did you notice this?



Recursion

We have finally

$$T(n)=\begin{cases}1 & \text{if } n=0\\ 1+\sum_{j=0}^{n-1}T(j) & \text{if } n>0\end{cases} \tag{1}$$

- 1 for calling into the root of the tree.
- $T(j)$ counts the number of call (Recursive included)

Recursion

We have finally

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 + \sum_{j=0}^{n-1} T(j) & \text{if } n > 0 \end{cases} \quad (1)$$

- 1 for calling into the root of the tree.
- $T(j)$ counts the number of call (Recursive included)

How many possible decisions are being considered when cutting?

Decision	cut at 1	cut at 2	$\cdots$	cut at n-1
Which One?	0 or 1	0 or 1	$\cdots$	0 or 1

Recursion

We have finally

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 + \sum_{j=0}^{n-1} T(j) & \text{if } n > 0 \end{cases} \quad (1)$$

- 1 for calling into the root of the tree.
- $T(j)$ counts the number of call (Recursive included)

How many possible decisions are being considered when cutting?

Decision	cut at 1	cut at 2	$\cdots$	cut at n-1
Which One?	0 or 1	0 or 1	$\cdots$	0 or 1

Recursion

We have finally

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 + \sum_{j=0}^{n-1} T(j) & \text{if } n > 0 \end{cases} \quad (1)$$

- 1 for calling into the root of the tree.
- $T(j)$ counts the number of call (Recursive included)

How many possible decisions are being considered when cutting?

Decision	cut at 1	cut at 2	$\cdots$	cut at n-1
Which One?	0 or 1	0 or 1	$\cdots$	0 or 1

What the tree is telling us?

The number of possible paths is equal to the number of leaves

- We have 2^{n-1} paths, which is equal to the number of leaves

Then

- The recursion consider explicitly all possible decisions

It is possible to prove by induction that

$$T(n) = 2^n \tag{2}$$

What the tree is telling us?

The number of possible paths is equal to the number of leaves

- We have 2^{n-1} paths, which is equal to the number of leaves

Then

- The recursion consider explicitly all possible decisions

It is possible to prove by induction that

$$T(n) = 2^{n} \tag{2}$$

What the tree is telling us?

The number of possible paths is equal to the number of leaves

- We have 2^{n-1} paths, which is equal to the number of leaves

Then

- The recursion consider explicitly all possible decisions

It is possible to prove by induction that

$$T(n) = 2^n \tag{2}$$

How we solve this?

We need something better

Dynamic programming approach!!!

How we solve this?

We need something better

Dynamic programming approach!!!

How?

- This is done by computing each sub-problem only once and storing its solution in some way.
- This is known as **time-memory trade-off**, and the savings may be dramatic.

How we solve this?

We need something better

Dynamic programming approach!!!

How?

- This is done by computing each sub-problem only once and storing its solution in some way.
- This is known as **time-memory trade-off**, and the savings may be dramatic.

How and Why

- Dynamic programming solution runs in polynomial time when the number of distinct subproblems involved is polynomial in the input size and they can be solved in polynomial time.

How we solve this?

We need something better

Dynamic programming approach!!!

How?

- This is done by computing each sub-problem only once and storing its solution in some way.
- This is known as **time-memory trade-off**, and the savings may be dramatic.

How and Why

- Dynamic programming solution runs in polynomial time when the number of distinct subproblems involved is polynomial in the input size and they can be solved in polynomial time.

First Approach: Top-down with Memoization

Basics in this approach

1. We write the procedure recursively in a natural manner.
2. However, we save the result of each subproblem (Usually in an array or hash table)

First Approach: Top-down with Memoization

Basics in this approach

1. We write the procedure recursively in a natural manner.
2. However, we save the result of each subproblem (Usually in an array or hash table)

Then

Each time the procedure tries to solve a subproblem it first checks to see whether it has previously solved this subproblem.

First Approach: Top-down with Memoization

Basics in this approach

1. We write the procedure recursively in a natural manner.
2. However, we save the result of each subproblem (Usually in an array or hash table)

Then

Each time the procedure tries to solve a subproblem it first checks to see whether it has previously solved this subproblem.

We can say the following

- We say that the recursive procedure has been Memoized.
- it "remembers" what results it has computed previously.

First Approach: Top-down with Memoization

Basics in this approach

1. We write the procedure recursively in a natural manner.
2. However, we save the result of each subproblem (Usually in an array or hash table)

Then

Each time the procedure tries to solve a subproblem it first checks to see whether it has previously solved this subproblem.

We can say the following

- We say that the recursive procedure has been Memoized.
- it "remembers" what results it has computed previously.

First Approach: Top-down with Memoization

Basics in this approach

1. We write the procedure recursively in a natural manner.
2. However, we save the result of each subproblem (Usually in an array or hash table)

Then

Each time the procedure tries to solve a subproblem it first checks to see whether it has previously solved this subproblem.

We can say the following

- We say that the recursive procedure has been Memoized.
- it "remembers" what results it has computed previously.

We require an Auxiliary Function to Accomplish this

Code

Memoized-Cut-Rod(p, n)

1. Let $r[0..n]$ be a new array
2. for $i = 0$ to n
3. $\quad r[i] = -\infty$
4. return Memoized-Cut-Rod-Aux(p, n, r)

We require an Auxiliary Function to Accomplish this

Code

Memoized-Cut-Rod(p, n)

1. Let $r\left[0..n\right]$ be a new array
2. for $i = 0$ to n
3. $\qquad r\left[i\right] = -\infty$
4. return Memoized-Cut-Rod-Aux(p, n, r)

We require an Auxiliary Function to Accomplish this

Code

Memoized-Cut-Rod(p, n)

1. Let $r[0..n]$ be a new array
2. for $i = 0$ to n
3. $\quad\quad r[i] = -\infty$
4. return Memoized-Cut-Rod-Aux(p, n, r)

We require an Auxiliary Function to Accomplish this

Code

Memoized-Cut-Rod(p, n)

1. Let $r\left[0..n\right]$ be a new array
2. for $i = 0$ to n
3. $\qquad r\left[i\right] = -\infty$
4. return Memoized-Cut-Rod-Aux(p, n, r)

Memoized-Cut-Rod-Aux(p, n, r)

Code

Memoized-Cut-Rod-Aux(p, n, r)

1. if $r[n] \geq 0$
2. return $r[n]$
3. if $n == 0$
4. $q = 0$
5. else $q = -\infty$
6. for $i = 1$ to n
7. $q = \max\{q, p[i] + \text{Memoized-Cut-Rod-Aux}(p, n-i, r)\}$
8. $r[n] = q$
9. return q

Memoized-Cut-Rod-Aux(p, n, r)

Code

Memoized-Cut-Rod-Aux(p, n, r)

1. if $r[n] \geq 0$
2. return $r[n]$
3. if $n == 0$
4. $q = 0$
5. else $q = -\infty$
6. for $i = 1$ to n
7. $q = \max\{q, p[i] + \text{Memoized-Cut-Rod-Aux}(p, n-i, r)\}$
8. $r[n] = q$
9. return q

Memoized-Cut-Rod-Aux(p, n, r)

Code

Memoized-Cut-Rod-Aux(p, n, r)

if $r[n] \geq 0$
    return $r[n]$
if $n == 0$
    $q = 0$
else $q = -\infty$
    for $i = 1$ to n
        $q = \max\{q, p[i] + \text{Memoized-Cut-Rod-Aux}(p, n-i, r)\}$
$r[n] = q$
return q

Memoized-Cut-Rod-Aux(p, n, r)

Code

Memoized-Cut-Rod-Aux(p, n, r)

if $r[n] \geq 0$
    return $r[n]$
if $n == 0$
    $q = 0$
else $q = -\infty$
    for $i = 1$ to n
        $q = \max\{q, p[i] + \text{Memoized-Cut-Rod-Aux}(p, n-i, r)\}$
$r[n] = q$
return q

Memoized-Cut-Rod-Aux(p, n, r)

Code

Memoized-Cut-Rod-Aux(p, n, r)

if $r[n] \geq 0$

    return $r[n]$

if $n == 0$

    $q = 0$

else $q = -\infty$

    for $i = 1$ to n

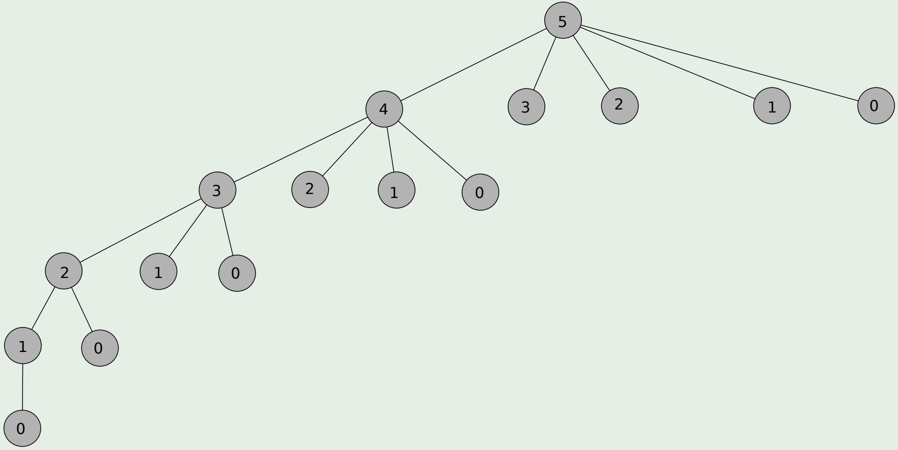
        $q = \max\{q, p[i] + \text{Memoized-Cut-Rod-Aux}(p, n-i, r)\}$

$r[n] = q$

return q

The Recursion Tree of Memoized-Cut-Rod

Tree for $n = 5$



Thus

We have that

- It solves each subproblem just once.
- It solves subproblems for sizes $i = 0, 1, \ldots, n$

Thus

We have that

- It solves each subproblem just once.
- It solves subproblems for sizes $i = 0, 1, ..., n$

Thus

- To solve a problem of size i the for loop in line 6 of Memoized-Cut-Rod-Aux iterates i times.

Thus

We have that

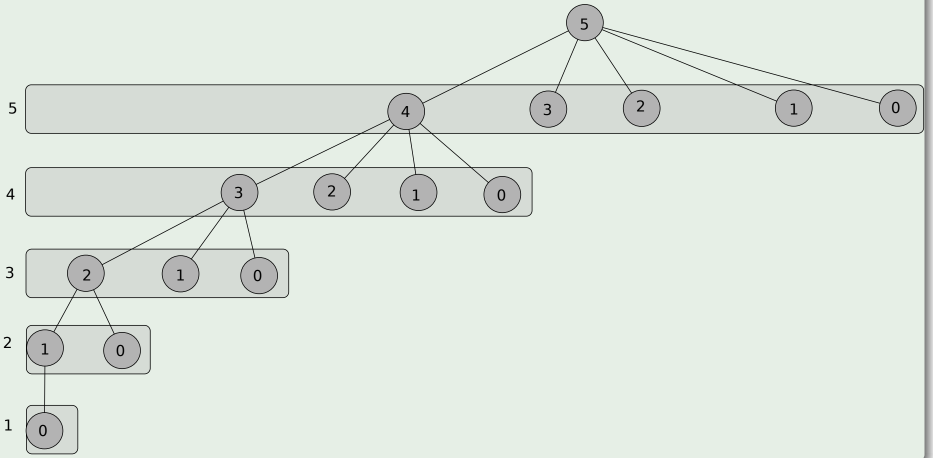
- It solves each subproblem just once.
- It solves subproblems for sizes $i = 0, 1, ..., n$

Thus

- To solve a problem of size i the for loop in line 6 of Memoized-Cut-Rod-Aux iterates i times.

Then look at this..

Something Notable



Complexity

Add the works

We have then

$$1 + 2 + 3 + ... + n = \frac{n(n+1)}{2} \quad (3)$$

Then, we have

$\Theta(n^2)$.

Complexity

Add the works

We have then

$$1 + 2 + 3 + ... + n = \frac{n(n+1)}{2} \tag{3}$$

Then, we have

$\Theta\left(n^2\right)$.

What about the Bottom-Up approach?

Simpler Solution

How?

The natural order of solving

A problem of size i is smaller than a subproblem of size j, if $i < j$.

It is simpler to solve problems in this orden

$j = 0, 1, 2, ..., n$ in order of increasing size.

What about the Bottom-Up approach?

Simpler Solution

How?

The natural order of solving

A problem of size i is smaller than a subproblem of size j, **if** $i < j$.

It is simpler to solve problems in this orden

$j = 0, 1, 2, \ldots, n$ in order of increasing size.

What about the Bottom-Up approach?

Simpler Solution

How?

The natural order of solving

A problem of size i is smaller than a subproblem of size j, **if** $i < j$.

It is simpler to solve problems in this orden

$j = 0, 1, 2, ..., n$ in order of increasing size.

Bottom-Up-Cut-Rod(p, n)

Code

Bottom-Up-Cut-Rod(p, n)

1. Let $r[0..n]$ be a new array
2. $r[0] = 0$
3. for $j = 1$ to n
4. $\quad q = -\infty$
5. $\quad$ for $i = 1$ to j
6. $\quad\quad q = \max\{q, p[i] + r[j-i]\}$
7. $\quad r[j] = q$
8. return $r[n]$

Bottom-Up-Cut-Rod(p, n)

Code

Bottom-Up-Cut-Rod(p, n)

1. Let $r\left[0..n\right]$ be a new array
2. $r\left[0\right] = 0$
3. for $j = 1$ to n
4. $\quad q = -\infty$
5. $\quad$ for $i = 1$ to j
6. $\qquad q = \max\left\{q, p\left[i\right] + r\left[j - i\right]\right\}$
7. $\quad r\left[j\right] = q$
8. return $r\left[n\right]$

Bottom-Up-Cut-Rod(p, n)

Code

Bottom-Up-Cut-Rod(p, n)

1. Let $r[0..n]$ be a new array
2. $r[0] = 0$
3. for $j = 1$ to n
4. $\qquad q = -\infty$
5. $\qquad$ for $i = 1$ to j
6. $\qquad\qquad q = \max\{q, p[i] + r[j-i]\}$
7. $\qquad r[j] = q$
8. return $r[n]$

Bottom-Up-Cut-Rod(p, n)

Code

Bottom-Up-Cut-Rod(p, n)

1. Let $r[0..n]$ be a new array
2. $r[0] = 0$
3. for $j = 1$ to n
4. $\quad q = -\infty$
5. $\quad$ for $i = 1$ to j
6. $\quad\quad q = \max\{q, p[i] + r[j-i]\}$
7. $\quad r[j] = q$
8. return $r[n]$

Bottom-Up-Cut-Rod(p, n)

Code

Bottom-Up-Cut-Rod(p, n)

1. Let $r[0..n]$ be a new array
2. $r[0] = 0$
3. for $j = 1$ to n
4. $\quad q = -\infty$
5. $\quad$ for $i = 1$ to j
6. $\quad\quad q = \max\{q, p[i] + r[j - i]\}$
7. $\quad r[j] = q$
8. return $r[n]$

Bottom-Up-Cut-Rod(p, n)

Code

Bottom-Up-Cut-Rod(p, n)

1. Let $r[0..n]$ be a new array
2. $r[0] = 0$
3. for $j = 1$ to n
4. $\quad q = -\infty$
5. $\quad$ for $i = 1$ to j
6. $\quad\quad q = \max\{q, p[i] + r[j - i]\}$
7. $\quad r[j] = q$
8. return $r[n]$

How to See Everything: Subproblem Graphs (DAG)

In dynamic programing

It is necessary to understand how subproblems depend on each other.

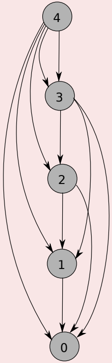
This information can be found in the subproblem graph which is a DAG

How to See Everything: Subproblem Graphs (DAG)

In dynamic programing

It is necessary to understand how subproblems depend on each other.

This information can be found in the subproblem graph which is a DAG



Reconstructing the Solution

How, we can do that?

Any Ideas?

We need to...

So...

Reconstructing the Solution

How, we can do that?

Any Ideas?

We need to...

Store each **choice** of the solution some way

So...

Reconstructing the Solution

How, we can do that?

Any Ideas?

We need to...

Store each **choice** of the solution some way

So...

We can reconstruct the solution path

Final Code

Code

Extended-Bottom-Up-Cut-Rod(p, n)

1. Let $r[0..n]$ and $s[0..n]$ be new arrays
2. $r[0] = 0$
3. for $j = 1$ to n
4. $\quad q = -\infty$
5. $\quad$ for $i = 1$ to j
6. $\quad\quad$ if $q < p[i] + r[j-i]$
7. $\quad\quad\quad q = p[i] + r[j-i]$
8. $\quad\quad\quad s[j] = i$
9. $\quad r[j] = q$
10. return r and s

Final Code

Code

Extended-Bottom-Up-Cut-Rod(p, n)

1. Let $r[0..n]$ and $s[0..n]$ be new arrays
2. $r[0] = 0$
3. for $j = 1$ to n
4. $q = -\infty$
5.   for $i = 1$ to j
6.   if $q < p[i] + r[j-i]$
7. $q = p[i] + r[j-i]$
8. $s[j] = i$
9. $r[j] = q$
10. return r and s

Final Code

Code

Extended-Bottom-Up-Cut-Rod(p, n)

Let $r[0..n]$ and $s[0..n]$ be new arrays
$r[0] = 0$
for $j = 1$ to n
    $q = -\infty$
    for $i = 1$ to j
        if $q < p[i] + r[j-i]$
            $q = p[i] + r[j-i]$
            $s[j] = i$
    $r[j] = q$
return r and s

Final Code

Code

Extended-Bottom-Up-Cut-Rod(p, n)

Let $r[0..n]$ and $s[0..n]$ be new arrays
$r[0] = 0$
for $j = 1$ to n
    $q = -\infty$
    for $i = 1$ to j
        if $q < p[i] + r[j - i]$
            $q = p[i] + r[j - i]$
            $s[j] = i$
    $r[j] = q$
return r and s

Final Code

Code

```
Extended-Bottom-Up-Cut-Rod(p, n)
Let r[0..n] and s[0..n] be new arrays
r[0] = 0
for j = 1 to n
        q = −∞
        for i = 1 to j
                if q < p[i] + r[j − i]
                        q = p[i] + r[j − i]
                        s[j] = i
        r[j] = q
return r and s
```

Final Code

Code

Extended-Bottom-Up-Cut-Rod(p, n)

```
Let r[0..n] and s[0..n] be new arrays
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        if q < p[i] + r[j - i]
            q = p[i] + r[j - i]
            s[j] = i
    r[j] = q
return r and s
```

Final Code

Code

Extended-Bottom-Up-Cut-Rod(p, n)

Let $r[0..n]$ and $s[0..n]$ be new arrays
$r[0] = 0$
for $j = 1$ to n
    $q = -\infty$
    for $i = 1$ to j
        if $q < p[i] + r[j-i]$
            $q = p[i] + r[j-i]$
            $s[j] = i$
    $r[j] = q$
return r and s

Printing Code

Code

Print-Cut-Rod-Solution(p, n)

1. (r, s) =Extended-Bottom-Up-Cut-Rod(p, n)
2. while $n > 0$
3. print $s[n]$
4. $n = n - s[n]$

Printing Code

Code

Print-Cut-Rod-Solution(p, n)

1. (r, s) =Extended-Bottom-Up-Cut-Rod(p, n)
2. while $n > 0$
3.   print $s[n]$
4. $n = n - s[n]$

Printing Code

Code

Print-Cut-Rod-Solution(p, n)

1. (r, s) =Extended-Bottom-Up-Cut-Rod(p, n)
2. while $n > 0$
3. print $s[n]$
4. $n = n - s[n]$

Example

From the previous problem

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Thus

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

Example

From the previous problem

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Thus

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

Outline

Optimal Substructure

In dynamic programming

A first step toward the solution is characterizing the problem and finding the optimal substructure.

We have the following steps

First

The problem consists in making choices.

Second

Given each problem, you are given a choice that leads to a solution.

Third

Each solution allows us to determine which subproblems need to be solved, and how to best characterize the resulting space of subproblems.

We have the following steps

First

The problem consists in making choices.

Second

Given each problem, you are given a choice that leads to a solution.

Third

Each solution allows us to determine which subproblems need to be solved, and how to best characterize the resulting space of subproblems.

We have the following steps

First

The problem consists in making choices.

Second

Given each problem, you are given a choice that leads to a solution.

Third

Each solution allows us to determine which subproblems need to be solved, and how to best characterize the resulting space of subproblems.

We have the following steps

Fourth

Use cut-and-paste to prove by contradiction that the optimal subproblem structure exists.

Now using the following problems

Unweighted shortest path

Find a path from u to v consisting of the fewest edges.

Unweighted longest simple path

Find a simple path from u to v consisting of the most edges.

Now using the following problems

Unweighted shortest path

Find a path from u to v consisting of the fewest edges.

Unweighted longest simple path

Find a simple path from u to v consisting of the most edges.

We can explain subtleties about the Optimal Substructure

Unweighted shortest path

It has an optimal substructure

Why?

First, given an optimal shortest path t between p and q.

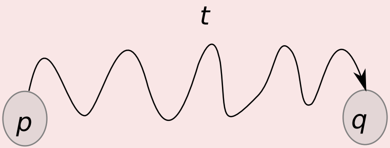
We can explain subtleties about the Optimal Substructure

Unweighted shortest path

It has an optimal substructure

Why?

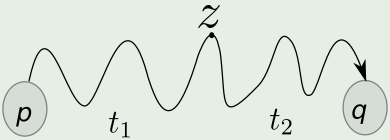
First, given an optimal shortest path t between p and q.



How do we prove this?

First

Assume an intermediate point z such that there are two paths t_1 and t_2, $t = t_1 \cup t_2$



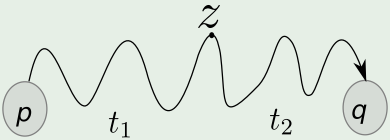
By contradiction

Thus, by contradiction, assume that there is a shorter path between z and q, t_2^1. Then, $|t_1 \cup t_2^1| < t \perp$ Quod Erat Demonstrandum (QED).

How do we prove this?

First

Assume an intermediate point z such that there are two paths t_1 and t_2, $t = t_1 \cup t_2$



By contradiction

Thus, by contradiction, assume that there is a shorter path between z and q, t_2^1. Then, $\left|t_1 \cup t_2^1\right| < t \perp$ Quod Erat Demonstrandum (QED).

However

Some problems do not have the optimal substructure

The longest unweighted path

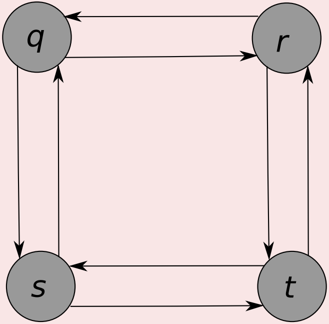
Example

However

Some problems do not have the optimal substructure

The longest unweighted path

Example



Examples

First: Possible path between q and t

$$q \longrightarrow r \longrightarrow t$$

But

$q \longrightarrow r$ is not the longest simple path from q and r nor the path $r \longrightarrow t$

Example of largest simple path for $q \longrightarrow r$

$q \longrightarrow s \longrightarrow t \longrightarrow r$

Examples

First: Possible path between q and t

$$q \longrightarrow r \longrightarrow t$$

But

$q \longrightarrow r$ is not the longest simple path from q and r nor the path $r \longrightarrow t$

Example of largest simple path for $q \longrightarrow r$

$q \longrightarrow s \longrightarrow t \longrightarrow r$

Examples

First: Possible path between q and t

$$q \longrightarrow r \longrightarrow t$$

But

$q \longrightarrow r$ is not the longest simple path from q and r nor the path $r \longrightarrow t$

Example of largest simple path for $q \longrightarrow r$

$q \longrightarrow s \longrightarrow t \longrightarrow r$

What the problem shows

We have that

- It not only does the problem lack optimal substructure.
  - We cannot necessarily assemble a "legal" solution to the problem from solutions to subproblems.

What the problem shows

We have that

- It not only does the problem lack optimal substructure.
  - We cannot necessarily assemble a “legal” solution to the problem from solutions to subproblems.

It is more

- No efficient dynamic programming algorithm for this problem has ever been found.
  - In fact, this problem is NP-complete.

What the problem shows

We have that

- It not only does the problem lack optimal substructure.
  - We cannot necessarily assemble a "legal" solution to the problem from solutions to subproblems.

It is more

- No efficient dynamic programming algorithm for this problem has ever been found.
  - In fact, this problem is NP-complete.

What the problem shows

We have that

- It not only does the problem lack optimal substructure.
  - We cannot necessarily assemble a "legal" solution to the problem from solutions to subproblems.

It is more

- No efficient dynamic programming algorithm for this problem has ever been found.
  - In fact, this problem is NP-complete.

Then, How can we use the DAG?

Get the Space Problem

- Use the elements of the space.
- Build a Graph using all the decisions that can be made.
- If you have a DAG!!! You have a optimal substructure!!!

Then, How can we use the DAG?

Get the Space Problem

- Use the elements of the space.
- Build a Graph using all the decisions that can be made.
- If you have a DAG!!! You have a optimal substructure!!!

Then, How can we use the DAG?

Get the Space Problem

- Use the elements of the space.
- Build a Graph using all the decisions that can be made.
- If you have a DAG!!! You have a optimal substructure!!!

What is the difference?

In the Unweighted Shortest Path the problems are independent

We mean that the solution to one sub-problem does not affect the solution of another subproblem.

In the Unweighted Longest Path

Remember vertices q and r in the second case!!!

Question

Then, Why the USP are independent?

What is the difference?

In the Unweighted Shortest Path the problems are independent

We mean that the solution to one sub-problem does not affect the solution of another subproblem.

In the Unweighted Longest Path

Remember vertices q and r in the second case!!!

Question

Then, Why the USP are independent?

What is the difference?

In the Unweighted Shortest Path the problems are independent

We mean that the solution to one sub-problem does not affect the solution of another subproblem.

In the Unweighted Longest Path

Remember vertices q and r in the second case!!!

Question

Then, Why the USP are independent?

Outline

Overlapping Subproblems

Why

This happens because the recursive solution revisits the same subproblem multiple times.

This is the main advantage of dynamic programming

It takes advantage of this by solving and storing the solution.

Properties

A dynamic-programming solution runs in polynomial time when the number of distinct subproblems involved is polynomial in the input size and they can be solved in polynomial time.

Overlapping Subproblems

Why

This happens because the recursive solution revisits the same subproblem multiple times.

This is the main advantage of dynamic programming

It takes advantage of this by solving and storing the solution.

Properties

A dynamic-programming solution runs in polynomial time when the number of distinct subproblems involved is polynomial in the input size and they can be solved in polynomial time.

Overlapping Subproblems

Why

This happens because the recursive solution revisits the same subproblem multiple times.

This is the main advantage of dynamic programming

It takes advantage of this by solving and storing the solution.

Properties

A dynamic-programming solution runs in polynomial time when the number of distinct subproblems involved is polynomial in the input size and they can be solved in polynomial time.

Overlapping Subproblems

We have two ways of solving the problem

- Top-down with Memoization.
- Bottom-up.

Outline

Reconstruction of Subproblems

To reconstruct

We use a table to store the choices such that we can reconstruct those of the sub-problem.

Outline

Common Subproblems

Something Notable

Finding the right subproblem takes creativity and experimentation.

However

There are a few standard choices that arise repeatedly in dynamic programming.

Common Subproblems

Something Notable

Finding the right subproblem takes creativity and experimentation.

However

There are a few standard choices that arise repeatedly in dynamic programming.

Number of Subproblems is Linear

We have the following input

The input is $x_1, x_2, ..., x_n$.

Subproblems

$x_1, x_2, ..., x_i$

Example

$x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6 \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$

Therefore

The number of subproblems is therefore linear.

Number of Subproblems is Linear

We have the following input

The input is $x_1, x_2, ..., x_n$.

Subproblems

$x_1, x_2, ..., x_i$

Example

$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$

Therefore

The number of subproblems is therefore linear.

Number of Subproblems is Linear

We have the following input

The input is $x_1, x_2, ..., x_n$.

Subproblems

$x_1, x_2, ..., x_i$

Example

$$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

Therefore

The number of subproblems is therefore linear.

Number of Subproblems is Linear

We have the following input

The input is $x_1, x_2, ..., x_n$.

Subproblems

$x_1, x_2, ..., x_i$

Example

$$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

Therefore

The number of subproblems is therefore linear.

Number of Subproblems is $O(nm)$

Input

The input is $x_1, x_2, ..., x_n$ and $y_1, y_2, ..., y_m$.

Subproblems

$x_1, x_2, ..., x_i$ and $y_1, y_2, ..., y_j$.

Example

$$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

$$\boxed{y_1 \quad y_2 \quad y_3 \quad y_4 \quad y_5} \quad y_6 \quad y_7 \quad y_8$$

Therefore

The number of subproblems is $O(mn)$.

Number of Subproblems is $O(nm)$

Input

The input is $x_1, x_2, ..., x_n$ and $y_1, y_2, ..., y_m$.

Subproblems

$x_1, x_2, ..., x_i$ and $y_1, y_2, ..., y_j$.

Example

$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$

$\boxed{y_1 \quad y_2 \quad y_3 \quad y_4 \quad y_5} \quad y_6 \quad y_7 \quad y_8$

Therefore

The number of subproblems is $O(mn)$.

Number of Subproblems is $O(nm)$

Input

The input is $x_1, x_2, ..., x_n$ and $y_1, y_2, ..., y_m$.

Subproblems

$x_1, x_2, ..., x_i$ and $y_1, y_2, ..., y_j$.

Example

$$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

$$\boxed{y_1 \quad y_2 \quad y_3 \quad y_4 \quad y_5} \quad y_6 \quad y_7 \quad y_8$$

Therefore

The number of subproblems is $O(mn)$.

Number of Subproblems is $O(nm)$

Input

The input is $x_1, x_2, ..., x_n$ and $y_1, y_2, ..., y_m$.

Subproblems

$x_1, x_2, ..., x_i$ and $y_1, y_2, ..., y_j$.

Example

$$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

$$\boxed{y_1 \quad y_2 \quad y_3 \quad y_4 \quad y_5} \quad y_6 \quad y_7 \quad y_8$$

Therefore

The number of subproblems is $O(mn)$.

Number of Subproblems is $O\left(n^2\right)$

Input

The input is $x_1, x_2, ..., x_n$.

Subproblems

$x_i, x_{i+1}, \ldots, x_j$

Example

$$x_1 \quad x_2 \quad \boxed{x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

Therefore

The number of subproblems is $O\left(n^2\right)$.

Number of Subproblems is $O\left(n^2\right)$

Input

The input is $x_1, x_2, ..., x_n$.

Subproblems

$x_i, x_{i+1}, ..., x_j$

Example

$$x_1 \quad x_2 \quad \boxed{x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

Therefore

The number of subproblems is $O\left(n^2\right)$.

Number of Subproblems is $O\left(n^2\right)$

Input

The input is $x_1, x_2, ..., x_n$.

Subproblems

$x_i, x_{i+1}, ..., x_j$

Example

$$x_1 \quad x_2 \quad \boxed{x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

Therefore

The number of subproblems is $O\left(n^2\right)$.

Number of Subproblems is $O(n^2)$

Input

The input is $x_1, x_2, ..., x_n$.

Subproblems

$x_i, x_{i+1}, ..., x_j$

Example

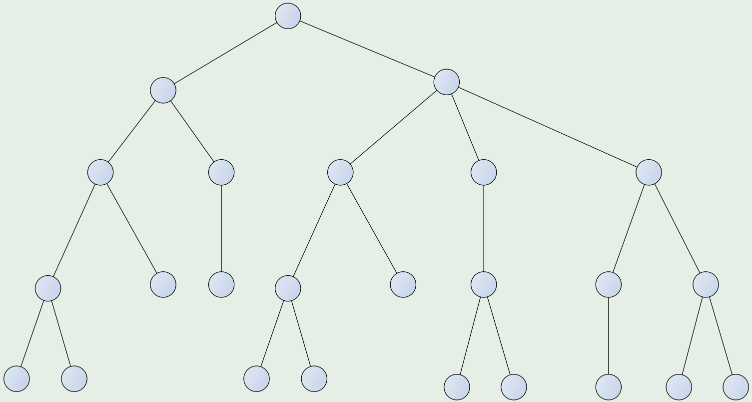
$$x_1 \quad x_2 \quad \boxed{x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

Therefore

The number of subproblems is $O(n^2)$.

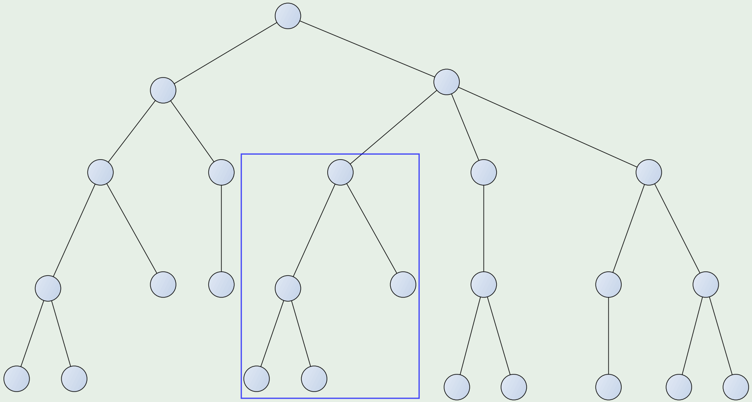
Input is a rooted subtree

Input



Input is a rooted subtree

Subproblem



Question

How Many Subproblems do you have?

Any Idea?

Outline

Definition

Input

A sequence $a_1, a_2, ..., a_n$

A subsequence

It is any subset of these numbers taken in order $a_{i_1}, a_{i_2}, ..., a_{i_k}$ where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$.

Thus

An increasing subsequence is one in which the numbers are getting strictly larger.

Definition

Input

A sequence $a_1, a_2, ..., a_n$

A subsequence

It is any subset of these numbers taken in order $a_{i_1}, a_{i_2}, ..., a_{i_k}$ where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$.

Thus

An increasing subsequence is one in which the numbers are getting strictly larger.

Definition

Input

A sequence $a_1, a_2, ..., a_n$

A subsequence

It is any subset of these numbers taken in order $a_{i_1}, a_{i_2}, ..., a_{i_k}$ where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$.

Thus

An increasing subsequence is one in which the numbers are getting strictly larger.

Definition

Output

The task is to find the increasing subsequence of greatest length.

Example

Definition

Output

The task is to find the increasing subsequence of greatest length.

Example

5 2 8 6 3 6 9 7

The Graph of increasing subsequences

To better understand the solution space, we can create the graph of all permissible transitions

- First, establish a node i for each element a_i, and add directed edges (i, j) whenever possible.
- i.e. Whenever $i < j$ and $a_i < a_j$.

The Graph of increasing subsequences

To better understand the solution space, we can create the graph of all permissible transitions

- First, establish a node i for each element a_i, and add directed edges (i, j) whenever possible.
- i.e. Whenever $i < j$ and $a_i < a_j$.

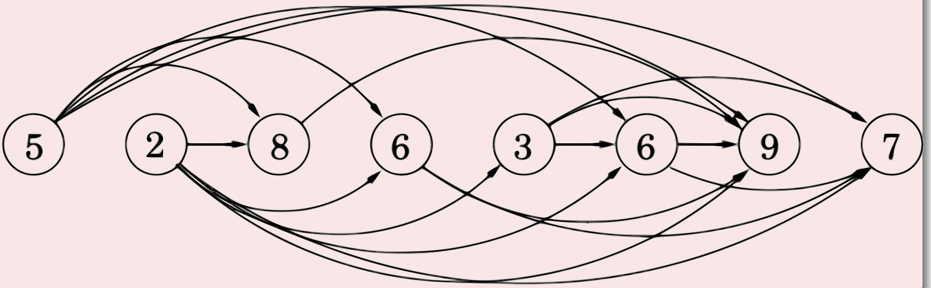
The Graph

The Graph of increasing subsequences

To better understand the solution space, we can create the graph of all permissible transitions

- First, establish a node i for each element a_i, and add directed edges (i, j) whenever possible.
- i.e. Whenever $i < j$ and $a_i < a_j$.

The Graph



Notice the following

We have

The graph is a DAG

Notice the following

We have

The graph is a DAG

Thus

- There is a one-to-one correspondence between increasing subsequences and paths in this DAG.
- Thus, find the longest path in the DAG.

Notice the following

We have

The graph is a DAG

Thus

- There is a one-to-one correspondence between increasing subsequences and paths in this DAG.
- Thus, find the longest path in the DAG.

Formulation

Something Notable

If we choose a number a_j to be in the longest increasing subsequence

We ask if the there is an edge to another

Is $(i, j) \in E$?

Thus, we need to choose all of them!!!

This can be done with a for loop

Formulation

Something Notable

If we choose a number a_j to be in the longest increasing subsequence

We ask if the there is an edge to another

Is $(i, j) \in E$?

Thus, we need to choose all of them!!!

This can be done with a for loop

Formulation

Something Notable

If we choose a number a_j to be in the longest increasing subsequence

We ask if the there is an edge to another

Is $(i, j) \in E$?

Thus, we need to choose all of them!!!

This can be done with a for loop

Thus

We start at a certain j

Then, we look at the previous i with $1 \leq i \leq j-1$

Here is the recursion for $\forall A[i] < A[j]$

$$L[j] = \begin{cases} 1 & \text{if there is no edge } (i,j) \in E \\ 1 + \max\{L[i_1], L[i_2], ..., L[i_h]\} & \text{For } (i_k, j) \in E, 1 \leq k \leq h \end{cases} \tag{4}$$

Thus

We start at a certain j

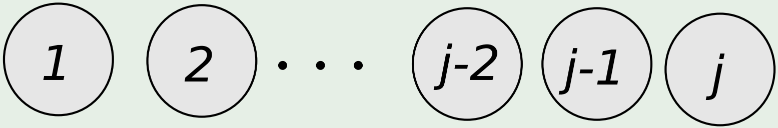
Then, we look at the previous i with $1 \leq i \leq j-1$

Here is the recursion for $\forall A[i] < A[j]$

$$L[j] = \begin{cases} 1 & \text{if there is no edge } (i,j) \in E \\ 1 + \max\left\{L[i_1], L[i_2], ..., L[i_h]\right\} & \text{For } (i_k, j) \in E, 1 \leq k \leq h \end{cases} \tag{4}$$

What is the meaning of this?

When is there an edge between i_k and j?



Clearly, this needs to be implemented in a machine

We have then that

A is an array that contains numbers indexed from 1 to n

Then, we have that

Instead of using $(i_k, j) \in E$ we use $A[i_k] < A[j]$

Instead of max

We use a loop and something like $q < temp$ for it

Clearly, this needs to be implemented in a machine

We have then that

A is an array that contains numbers indexed from 1 to n

Then, we have that

Instead of using $(i_k, j) \in E$ we use $A[i_k] < A[j]$

Instead of max

We use a loop and something like $q < temp$ for it

Clearly, this needs to be implemented in a machine

We have then that

A is an array that contains numbers indexed from 1 to n

Then, we have that

Instead of using $(i_k, j) \in E$ we use $A[i_k] < A[j]$

Instead of max

We use a loop and something like $q < temp$ for it

Recursive Function

The final recursive code

Recursive-Longest-Subsequence(A, n)

1. $q = 1$

Recursive Function

The final recursive code

Recursive-Longest-Subsequence(A, n)

1. $q = 1$
2. // Assume n as part of your solution
3. // Thus $A[i] < A[n]$ here $j == n$
4. for $i = 1$ to $n - 1$
5. $\quad t =$Recursive-Longest-Subsequence(A, i)
6. $\quad$ if $A[i] < A[n]$ and $q < 1 + t$
7. $\quad\quad q = 1 + t$
8. return q

Recursive Function

The final recursive code

Recursive-Longest-Subsequence(A, n)

1. $q = 1$
2. // Assume n as part of your solution
3. // Thus $A[i] < A[n]$ here $j == n$
4. for $i = 1$ to $n - 1$
5. $\quad$ $t =$Recursive-Longest-Subsequence(A, i)
6. $\quad$ if $A[i] < A[n]$ and $q < 1 + t$
7. $\quad\quad$ $q = 1 + t$
8. return q

Recursive Function

The final recursive code

Recursive-Longest-Subsequence(A, n)

1. $q = 1$
2. // Assume n as part of your solution
3. // Thus $A[i] < A[n]$ here $j == n$
4. for $i = 1$ to $n - 1$
5. $\quad t =$Recursive-Longest-Subsequence(A, i)
6. $\quad$ if $A[i] < A[n]$ and $q < 1 + t$
7. $\quad\quad q = 1 + t$
8. return q

Recursive Function

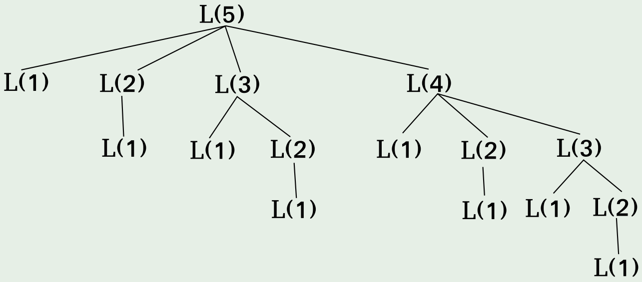
The final recursive code

Recursive-Longest-Subsequence(A, n)

1. $q = 1$
2. // Assume n as part of your solution
3. // Thus $A[i] < A[n]$ here $j == n$
4. for $i = 1$ to $n - 1$
5. $\quad t =$Recursive-Longest-Subsequence(A, i)
6. $\quad$ if $A[i] < A[n]$ and $q < 1 + t$
7. $\quad\quad q = 1 + t$
8. return q

What about the Complexity?

Recursion Tree - Can somebody Guess the Complexity?



How we save in recursive calls

First

Let $L[1..n]$ an array to store the values the longest subsequence

Bottom-Up Solution

Code

Bottom-Up-Longest-Subsequence(A, n)

```
Let L[1..n]
max = 0
for i = 1 to n
        L[i] = 1
for j = 2 to n
        for i = 1 to j − 1
                if A[i] < A[j] and L[j] < L[i] + 1
                        L[j] = L[i] + 1
for i = 1 to n
        if max < L[i]
                max = L[i]
return max
```

Step 1

- An array to store the values the longest subsequence.

Bottom-Up Solution

Code

Bottom-Up-Longest-Subsequence(A, n)

```
Let L[1..n]
max = 0
for i = 1 to n
        L[i] = 1
for j = 2 to n
        for i = 1 to j − 1
                if A[i] < A[j] and L[j] < L[i] + 1
                        L[j] = L[i] + 1
for i = 1 to n
        if max < L[i]
                max = L[i]
return max
```

Step 2

- A measure about the longest subsequence.

Bottom-Up Solution

Code

Bottom-Up-Longest-Subsequence(A, n)

Let $L[1..n]$
$max = 0$
for $i = 1$ to n
  $L[i] = 1$
for $j = 2$ to n
  for $i = 1$ to $j - 1$
    if $A[i] < A[j]$ and $L[j] < L[i] + 1$
      $L[j] = L[i] + 1$
for $i = 1$ to n
  if $max < L[i]$
    $max = L[i]$
return max

Step 3

- Initialize everything to 1 (Itself).

Bottom-Up Solution

Code

Bottom-Up-Longest-Subsequence(A, n)

Let $L\left[1..n\right]$
$max = 0$
for $i = 1$ to n
    $L\left[i\right] = 1$
for $j = 2$ to n
    for $i = 1$ to $j - 1$
        if $A[i] < A[j]$ and $L\left[j\right] < L\left[i\right] + 1$
            $L\left[j\right] = L\left[i\right] + 1$
for $i = 1$ to n
    if $max < L\left[i\right]$
        $max = L\left[i\right]$
return max

Step 4

- We know that the subproblem with size 1 has a solution, thus you need to start at 2.

Bottom-Up Solution

Code

Bottom-Up-Longest-Subsequence(A, n)

1. Let $L\left[1..n\right]$
2. $max = 0$
3. for $i = 1$ to n
4. $\quad L\left[i\right] = 1$
5. for $j = 2$ to n
6. $\quad$ for $i = 1$ to $j - 1$
7. $\quad\quad$ if $A[i] < A[j]$ and $L\left[j\right] < L\left[i\right] + 1$
8. $\quad\quad\quad L\left[j\right] = L\left[i\right] + 1$
9. for $i = 1$ to n
10. $\quad$ if $max < L\left[i\right]$
11. $\quad\quad max = L\left[i\right]$
12. return max

Step 5

- Get solutions to the subproblems less or equal than $j - 1$

Bottom-Up Solution

Code

Bottom-Up-Longest-Subsequence(A, n)

```
Let L[1..n]
max = 0
for i = 1 to n
        L[i] = 1
for j = 2 to n
        for i = 1 to j - 1
                if A[i] < A[j] and L[j] < L[i] + 1
                        L[j] = L[i] + 1
for i = 1 to n
        if max < L[i]
                max = L[i]
return max
```

Step 6

- Take a decision

Bottom-Up Solution

Code

Bottom-Up-Longest-Subsequence(A, n)

```
Let L[1..n]
max = 0
for i = 1 to n
        L[i] = 1
for j = 2 to n
        for i = 1 to j - 1
                if A[i] < A[j] and L[j] < L[i] + 1
                        L[j] = L[i] + 1
for i = 1 to n
        if max < L[i]
                max = L[i]
return max
```

Step 7

- If the decision is true then increase the counter for solution starting at j

Bottom-Up Solution

Code

Bottom-Up-Longest-Subsequence(A, n)

```
Let L[1..n]
max = 0
for i = 1 to n
    L[i] = 1
for j = 2 to n
    for i = 1 to j − 1
        if A[i] < A[j] and L[j] < L[i] + 1
            L[j] = L[i] + 1
for i = 1 to n
    if max < L[i]
        max = L[i]
return max
```

Step 8

- Find the Maximum Value

What about backtracking the Solution

We can do the following

You can have an array $S[1..n]$ initialized to the sequence $1, 2, ..., n$

Thus, each time

$A[i] < A[j]$ and $L[j] < L[i] + 1$ is true, we set $S[j] = i$.

Then

After returning the L and S we can get the index of the max to backtrack the answer.

What about backtracking the Solution

We can do the following

You can have an array $S[1..n]$ initialized to the sequence $1, 2, ..., n$

Thus, each time

$A[i] < A[j]$ and $L[j] < L[i] + 1$ is true, we set $S[j] = i$.

Then

After returning the L and S we can get the index of the max to backtrack the answer.

What about backtracking the Solution

We can do the following

You can have an array $S[1..n]$ initialized to the sequence $1, 2, ..., n$

Thus, each time

$A[i] < A[j]$ and $L[j] < L[i] + 1$ is true, we set $S[j] = i$.

Then

After returning the L and S we can get the index of the max to backtrack the answer.

Outline

Definition of The Problem

Input

A sequence of Matrices $\langle A_1, A_2, ..., A_n \rangle$

Output

We want a fully parenthesized product, where the final result is a single matrix or the product of two fully parenthesized matrix products.

Why

Take in consideration the following algorithm

Definition of The Problem

Input

A sequence of Matrices $\langle A_1, A_2, ..., A_n \rangle$

Output

We want a fully parenthesized product, where the final result is a single matrix or the product of two fully parenthesized matrix products.

Why

Take in consideration the following algorithm

Definition of The Problem

Input

A sequence of Matrices $\langle A_1, A_2, ..., A_n \rangle$

Output

We want a fully parenthesized product, where the final result is a single matrix or the product of two fully parenthesized matrix products.

Why

Take in consideration the following algorithm

Why? Look at this pseudocode

MATRIX-MULTIPLY(A,B)

```
if A.columns ≠ B.rows
        error "incompatible dimensions"
else let C be a new A.rows × B.columns matrix
            for i = 1 to A.rows
                for for j = 1 to B.columns
                    c_ij = 0
                    for k = 1 to A.columns
                        c_ij = c_ij + a_ik · b_kj
return C
```

Why? Look at this pseudocode

MATRIX-MULTIPLY(A,B)

if $A.columns \neq B.rows$
    error "incompatible dimensions"
else let C be a new $A.rows \times B.columns$ matrix
    for $i = 1$ to $A.rows$
        for for $j = 1$ to $B.columns$
            $c_{ij} = 0$
            for $k = 1$ to $A.columns$
                $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
return C

Then

If A is $N \times M$ and B is a $M \times P$ then the cost is $N \cdot M \cdot P$.

Why? Look at this pseudocode

MATRIX-MULTIPLY(A,B)

if $A.columns \neq B.rows$
    error "incompatible dimensions"
else let C be a new $A.rows \times B.columns$ matrix
        for $i = 1$ to $A.rows$
            for for $j = 1$ to $B.columns$
                $c_{ij} = 0$
                for $k = 1$ to $A.columns$
                    $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
return C

Then

If A is $N \times M$ and B is a $M \times P$ then the cost is $N \cdot M \cdot P$.

Why? Look at this pseudocode

MATRIX-MULTIPLY(A,B)

if $A.columns \neq B.rows$

    error "incompatible dimensions"

else let C be a new $A.rows \times B.columns$ matrix

        for $i = 1$ to $A.rows$

            for for $j = 1$ to $B.columns$

                $c_{ij} = 0$

                for $k = 1$ to $A.columns$

                    $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$

return C

Then

If A is $N \times M$ and B is a $M \times P$ then the cost is $N \cdot M \cdot P$.

Why? Look at this pseudocode

MATRIX-MULTIPLY(A,B)

```
if A.columns ≠ B.rows
        error "incompatible dimensions"
else let C be a new A.rows × B.columns matrix
        for i = 1 to A.rows
                for for j = 1 to B.columns
                        c_ij = 0
                        for k = 1 to A.columns
                                c_ij = c_ij + a_ik · b_kj
return C
```

Then

If A is $N \times M$ and B is a $M \times P$ then the cost is $N \cdot M \cdot P$.

Why? Look at this pseudocode

MATRIX-MULTIPLY(A,B)

if $A.columns \neq B.rows$
    error "incompatible dimensions"
else let C be a new $A.rows \times B.columns$ matrix
    for $i = 1$ to $A.rows$
        for for $j = 1$ to $B.columns$
            $c_{ij} = 0$
            for $k = 1$ to $A.columns$
                $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
return C

Then

If A is $N \times M$ and B is a $M \times P$ then the cost is $N \cdot M \cdot P$.

Example of Matrix Multiplications

Given the following matrices

- A, B, C with 10×100, 100×5 and 5×50
- Cost in scalar operations of (AB) is $10 \cdot 100 \cdot 5 = 5000$
- Cost in scalar operations of (BC) is $100 \cdot 5 \cdot 50 = 25000$

Example of Matrix Multiplications

Given the following matrices

- A, B, C with 10×100, 100×5 and 5×50
- Cost in scalar operations of (AB) is $10 \cdot 100 \cdot 5 = 5000$
- Cost in scalar operations of (BC) is $100 \cdot 5 \cdot 50 = 25000$

Then

Cost in scalar operations of $(AB)C$ is $5000 + 10 \cdot 5 \cdot 50 = 7500$

Cost in scalar operations of A(BC) is $25000 + 10 \cdot 100 \cdot 50 = 75000$

Example of Matrix Multiplications

Given the following matrices

- A, B, C with 10×100, 100×5 and 5×50
- Cost in scalar operations of (AB) is $10 \cdot 100 \cdot 5 = 5000$
- Cost in scalar operations of (BC) is $100 \cdot 5 \cdot 50 = 25000$

Then

Cost in scalar operations of $(AB)C$ is $5000 + 10 \cdot 5 \cdot 50 = 7500$
Cost in scalar operations of A(BC) is $25000 + 10 \cdot 100 \cdot 50 = 75000$

Example of Matrix Multiplications

Given the following matrices

- A, B, C with 10×100, 100×5 and 5×50
- Cost in scalar operations of (AB) is $10 \cdot 100 \cdot 5 = 5000$
- Cost in scalar operations of (BC) is $100 \cdot 5 \cdot 50 = 25000$

Then

Cost in scalar operations of $(AB)C$ is $5000 + 10 \cdot 5 \cdot 50 = 7500$

Cost in scalar operations of A(BC) is $25000 + 10 \cdot 100 \cdot 50 = 75000$

Example of Matrix Multiplications

Given the following matrices

- A, B, C with 10×100, 100×5 and 5×50
- Cost in scalar operations of (AB) is $10 \cdot 100 \cdot 5 = 5000$
- Cost in scalar operations of (BC) is $100 \cdot 5 \cdot 50 = 25000$

Then

Cost in scalar operations of $(AB)C$ is $5000 + 10 \cdot 5 \cdot 50 = 7500$
Cost in scalar operations of A(BC) is $25000 + 10 \cdot 100 \cdot 50 = 75000$

Matrix-Chain Multiplication

Problem

Given a chain $\langle A_1, A_2, ..., A_n \rangle$ of n matrices, where A_i has dimension $p_{i-1} \times p_i$. We want to fully parenthesize the product $A_1 A_2 ... A_n$ to minimize the number of scalar multiplications

Solving by brute force

Count all the possible parenthesizations

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$

Which is the sequence of Catalan Numbers which grows

$$\Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$$

Solving by brute force

Count all the possible parenthesizations

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$

Which is the sequence of Catalan Numbers which grows

$$\Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$$

Did you notice the following?

If we have the following sequence $A_{k-1}\left(A_k A_{k+1}\right)$

We have that A_{k-1} has dimension $p_{k-2} \times p_{k-1}$, A_k has dimension $p_{k-1} \times p_k$ and A_{k+1} has dimension $p_k \times p_{k+1}$.

Did you notice the following?

If we have the following sequence $A_{k-1}\left(A_k A_{k+1}\right)$

We have that A_{k-1} has dimension $p_{k-2} \times p_{k-1}$, A_k has dimension $p_{k-1} \times p_k$ and A_{k+1} has dimension $p_k \times p_{k+1}$.

The final matrix has dimensions

It has dimension $p_{k-2} \times p_{k+1}$.

Did you notice the following?

If we have the following sequence $A_{k-1}\left(A_k A_{k+1}\right)$

We have that A_{k-1} has dimension $p_{k-2} \times p_{k-1}$, A_k has dimension $p_{k-1} \times p_k$ and A_{k+1} has dimension $p_k \times p_{k+1}$.

The final matrix has dimensions

It has dimension $p_{k-2} \times p_{k+1}$.

Properties

With cost of multiplication:

1. For the first parenthesis $p_{k-1}p_kp_{k+1}$ with final dimension $p_{k-1} \times p_{k+1}$.
2. For A_{k-1} against what is inside parenthesis $p_{k-2}p_{k-1}p_{k+1}$ with final dimensions $p_{k-2} \times p_{k+1}$.
3. Total cost is then $p_{k-2}p_{k-1}p_{k+1} + p_{k-1}p_kp_{k+1}$

Did you notice the following?

If we have the following sequence $A_{k-1}\left(A_k A_{k+1}\right)$

We have that A_{k-1} has dimension $p_{k-2} \times p_{k-1}$, A_k has dimension $p_{k-1} \times p_k$ and A_{k+1} has dimension $p_k \times p_{k+1}$.

The final matrix has dimensions

It has dimension $p_{k-2} \times p_{k+1}$.

Properties

With cost of multiplication:

1. For the first parenthesis $p_{k-1}p_kp_{k+1}$ with final dimension $p_{k-1} \times p_{k+1}$.
2. For A_{k-1} against what is inside parenthesis $p_{k-2}p_{k-1}p_{k+1}$ with final dimensions $p_{k-2} \times p_{k+1}$.
3. Total cost is then $p_{k-2}p_{k-1}p_{k+1} + p_{k-1}p_kp_{k+1}$

Did you notice the following?

If we have the following sequence $A_{k-1}\left(A_kA_{k+1}\right)$

We have that A_{k-1} has dimension $p_{k-2} \times p_{k-1}$, A_k has dimension $p_{k-1} \times p_k$ and A_{k+1} has dimension $p_k \times p_{k+1}$.

The final matrix has dimensions

It has dimension $p_{k-2} \times p_{k+1}$.

Properties

With cost of multiplication:

1. For the first parenthesis $p_{k-1}p_kp_{k+1}$ with final dimension $p_{k-1} \times p_{k+1}$.
2. For A_{k-1} against what is inside parenthesis $p_{k-2}p_{k-1}p_{k+1}$ with final dimensions $p_{k-2} \times p_{k+1}$.
3. Total cost is then $p_{k-2}p_{k-1}p_{k+1} + p_{k-1}p_kp_{k+1}$

In addition

Look at the following multiplication

$$(A_i \cdots A_k)\,(A_{k+1} \cdots A_j)$$

We have the following

1. $(A_i \cdots A_k)$ is a matrix with dimensions $p_{i-1} \times p_k$

In addition

Look at the following multiplication

$$(A_i \cdots A_k)\,(A_{k+1} \cdots A_j)$$

We have the following

1. $(A_i \cdots A_k)$ is a matrix with dimensions $p_{i-1} \times p_k$
2. $(A_{k+1} \cdots A_j)$ is a matrix with dimensions $p_k \times p_j$

In addition

Look at the following multiplication

$$(A_i \cdots A_k)\,(A_{k+1} \cdots A_j)$$

We have the following

1. $(A_i \cdots A_k)$ is a matrix with dimensions $p_{i-1} \times p_k$
2. $(A_{k+1} \cdots A_j)$ is a matrix with dimensions $p_k \times p_j$

The total cost of this multiplication is

$m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$ **(In addition, you want to minimize the cost)**

Then use the Cut-and-Paste to probe optimal substructure

Given $i < j$

Suppose the optimal paranthesization of

$$A_i, A_{i+1}, ..., A_j$$

USE CONTRADICTION!

Now, the Recursion can be wrote!!!

Given that m[i,j] is the minimum number of scalar multiplications

$$m[i,j] = \begin{cases} 0 & \text{if } i == j \\ \min\limits_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

The Recursive Solution

Recursive Algorithm

1. Recursive-Matrix-Chain(p, i, j)
2. if $i == j$
3. return 0
4. $m[i, j] = \infty$
5. for $k = i$ to $j - 1$
6. $q =$ Recursive-Matrix-Chain(p, i, k) + ...
7. Recursive-Matrix-Chain$(p, k+1, j)$ +...
8. $p_{i-1}p_k p_j$
9. if $q < m[i, j]$
10. $m[i, j] = q$
11. return $m[i, j]$

The Recursive Solution

Recursive Algorithm

1. Recursive-Matrix-Chain(p, i, j)
2. if $i == j$
3. return 0
4. $m[i, j] = \infty$
5. for $k = i$ to $j - 1$
6. $q =$ Recursive-Matrix-Chain$(p, i, k) + \ldots$
7. Recursive-Matrix-Chain$(p, k + 1, j) + \ldots$
8. $p_{i-1} p_k p_j$
9. if $q < m[i, j]$
10. $m[i, j] = q$
11. return $m[i, j]$

The Recursive Solution

Recursive Algorithm

Recursive-Matrix-Chain(p, i, j)

if $i == j$

    return 0

$m[i, j] = \infty$

for $k = i$ to $j - 1$

    $q =$ Recursive-Matrix-Chain(p, i, k) + ...

        Recursive-Matrix-Chain$(p, k + 1, j)$ +...

        $p_{i-1}p_kp_j$

    if $q < m[i, j]$

        $m[i, j] = q$

return $m[i, j]$

The Recursive Solution

Recursive Algorithm

Recursive-Matrix-Chain(p, i, j)

if $i == j$

    return 0

$m[i, j] = \infty$

for $k = i$ to $j - 1$

    $q =$ Recursive-Matrix-Chain(p, i, k) + ...

        Recursive-Matrix-Chain$(p, k + 1, j)$ +...

        $p_{i-1} p_k p_j$

    if $q < m[i, j]$

        $m[i, j] = q$

return $m[i, j]$

The Recursive Solution

Recursive Algorithm

Recursive-Matrix-Chain(p, i, j)

if $i == j$

    return 0

$m[i, j] = \infty$

for $k = i$ to $j - 1$

    $q =$ Recursive-Matrix-Chain(p, i, k) + ...

        Recursive-Matrix-Chain$(p, k+1, j)$ +...

        $p_{i-1} p_k p_j$

    if $q < m[i, j]$

        $m[i, j] = q$

return $m[i, j]$

The Recursive Solution

Recursive Algorithm

Recursive-Matrix-Chain(p, i, j)

if $i == j$

  return 0

$m[i, j] = \infty$

for $k = i$ to $j - 1$

  $q =$ Recursive-Matrix-Chain(p, i, k) + ...

    Recursive-Matrix-Chain$(p, k + 1, j)$ +...

    $p_{i-1} p_k p_j$

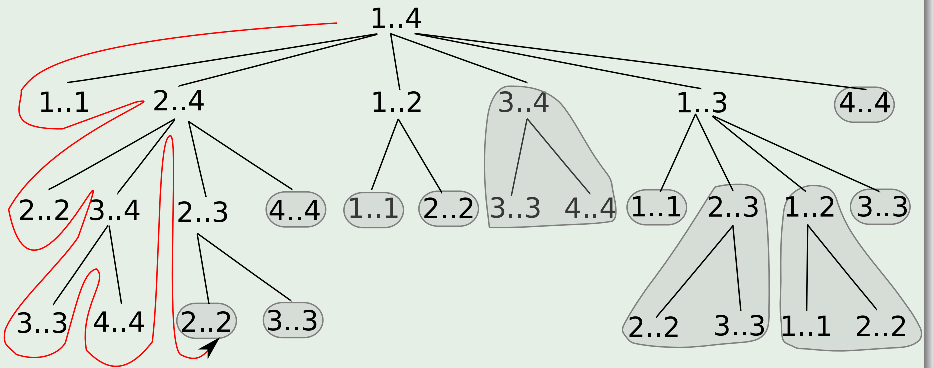
  if $q < m[i, j]$

    $m[i, j] = q$

return $m[i, j]$

Again!!! Overlapping substructure

Red Line Represents the Recursion Path



This is a nightmare

We have the following recursion

$$\begin{aligned} T(1) &\geq 1, \\ T(n) &\geq 1 + \sum_{k=1}^{n-1} \left[T(n-k) + T(k) + 1\right] \text{ for } n > 1. \end{aligned}$$

First

Did you notice?

$T(i)$ appears once as $T(k)$ and once as $T(n-k)$ for $i = 1, 2, ..., n-1$.

We have then

$$T(n) \geq 1 + 2\sum_{i=1}^{n-1} [T(i)] + n - 1.$$

First

Did you notice?

$T\left(i\right)$ appears once as $T\left(k\right)$ and once as $T\left(n-k\right)$ for $i = 1, 2, ..., n-1$.

We have then

$$T(n) \geq 1 + 2\sum_{i=1}^{n-1}\left[T(i)\right] + n - 1.$$

Then

We decide to guess $T(n) = \Omega(2^n)$

- We shall guess the following $T(n) \geq 2^{n-1}$ for all $n \geq 1$
- First for $n = 1$ $T(1) \geq 1 = 2^0$

Then

We decide to guess $T(n) = \Omega(2^n)$

- We shall guess the following $T(n) \geq 2^{n-1}$ for all $n \geq 1$
- First for $n = 1$ $T(1) \geq 1 = 2^0$

Then

Now, for $n \geq 2$

$$
\begin{aligned}
T(n) &\geq 2\sum_{i=1}^{n-1}\left[T(i)\right]+n\\
&= 2\sum_{i=1}^{n-1}2^{i-1}+n\\
&= 2\sum_{i=0}^{n-2}2^{i}+n\\
&= 2\left(\frac{2^{n-1}-1}{2-1}\right)+n\\
&= 2\left(2^{n-1}-1\right)+n\\
&= 2^{n}-2+n\\
&\geq 2^{n}\\
&\geq 2^{n-1}
\end{aligned}
$$

Then

Now, for $n \geq 2$

$$
\begin{aligned}
T(n) &\geq 2\sum_{i=1}^{n-1}\left[T(i)\right] + n \\
&= 2\sum_{i=1}^{n-1} 2^{i-1} + n \\
&= 2\sum_{i=0}^{n-2} 2^{i} + n \\
&= 2\left(\frac{2^{n-1}-1}{2-1}\right) + n \\
&= 2\left(2^{n-1}-1\right) + n \\
&= 2^{n} - 2 + n \\
&\geq 2^{n} \\
&\geq 2^{n-1}
\end{aligned}
$$

Then

Now, for $n \geq 2$

$$
\begin{aligned}
T(n) &\geq 2\sum_{i=1}^{n-1}\left[T(i)\right] + n \\
&= 2\sum_{i=1}^{n-1} 2^{i-1} + n \\
&= 2\sum_{i=0}^{n-2} 2^{i} + n \\
&= 2\left(\frac{2^{n-1}-1}{2-1}\right) + n \\
&= 2\left(2^{n-1}-1\right) + n \\
&= 2^{n} - 2 + n \\
&\geq 2^{n} \\
&\geq 2^{n-1}
\end{aligned}
$$

Then

Now, for $n \geq 2$

$$
\begin{aligned}
T(n) &\geq 2\sum_{i=1}^{n-1}\left[T(i)\right] + n \\
&= 2\sum_{i=1}^{n-1} 2^{i-1} + n \\
&= 2\sum_{i=0}^{n-2} 2^{i} + n \\
&= 2\left(\frac{2^{n-1}-1}{2-1}\right) + n \\
&= 2\left(2^{n-1}-1\right) + n \\
&= 2^{n} - 2 + n \\
&\geq 2^{n} \\
&\geq 2^{n-1}
\end{aligned}
$$

Then

Now, for $n \geq 2$

$$
\begin{aligned}
T(n) &\geq 2\sum_{i=1}^{n-1}\left[T(i)\right] + n\\
&= 2\sum_{i=1}^{n-1} 2^{i-1} + n\\
&= 2\sum_{i=0}^{n-2} 2^{i} + n\\
&= 2\left(\frac{2^{n-1}-1}{2-1}\right) + n\\
&= 2\left(2^{n-1}-1\right) + n\\
&= 2^{n} - 2 + n\\
&\geq 2^{n}\\
&\geq 2^{n-1}
\end{aligned}
$$

Then

Now, for $n \geq 2$

$$
\begin{aligned}
T(n) &\geq 2\sum_{i=1}^{n-1}\left[T(i)\right] + n\\
&= 2\sum_{i=1}^{n-1} 2^{i-1} + n\\
&= 2\sum_{i=0}^{n-2} 2^{i} + n\\
&= 2\left(\frac{2^{n-1}-1}{2-1}\right) + n\\
&= 2\left(2^{n-1}-1\right) + n\\
&= 2^{n} - 2 + n\\
&\geq 2^{n}\\
&\geq 2^{n-1}
\end{aligned}
$$

Then

Now, for $n \geq 2$

$$
\begin{aligned}
T(n) &\geq 2\sum_{i=1}^{n-1}\left[T(i)\right] + n \\
&= 2\sum_{i=1}^{n-1} 2^{i-1} + n \\
&= 2\sum_{i=0}^{n-2} 2^{i} + n \\
&= 2\left(\frac{2^{n-1}-1}{2-1}\right) + n \\
&= 2\left(2^{n-1}-1\right) + n \\
&= 2^{n} - 2 + n \\
&\geq 2^{n} \\
&\geq 2^{n-1}
\end{aligned}
$$

Then

Now, for $n \geq 2$

$$\begin{aligned}
T(n) &\geq 2\sum_{i=1}^{n-1}\left[T(i)\right] + n\\
&= 2\sum_{i=1}^{n-1} 2^{i-1} + n\\
&= 2\sum_{i=0}^{n-2} 2^{i} + n\\
&= 2\left(\frac{2^{n-1}-1}{2-1}\right) + n\\
&= 2\left(2^{n-1}-1\right) + n\\
&= 2^n - 2 + n\\
&\geq 2^n\\
&\geq 2^{n-1}
\end{aligned}$$

Thus

We want to avoid to calculate the same value many times

Use bottom up approach and store values at each step.

We get two arrays or tables

The first one, m

It is used to hold the information about the cost of multiplying the matrices

The second one, s

It is used to hold the place where the parenthesis is selected to minimize the cost

We get two arrays or tables

The first one, m

It is used to hold the information about the cost of multiplying the matrices

The second one, s

It is used to hold the place where the parenthesis is selected to minimize the cost

How do we simulate the recursion Bottom-Up?

We do the following...

We use the following strategy:

- Solve the chain of matrices with small size (The smallest is 2 matrices... after all 1 matrix has cost 0)

Thus, we need

A loop from 2 to n for solving small sequences to larger ones.

In addition

An inner loop from 1 to $n-l+1$ (We do not want to get out of the sequence of matrices) for solving the smaller problems for the outer loop

How do we simulate the recursion Bottom-Up?

We do the following...

We use the following strategy:

- Solve the chain of matrices with small size (The smallest is 2 matrices... after all 1 matrix has cost 0)

Thus, we need

A loop from 2 to n for solving small sequences to larger ones.

In addition

An inner loop from 1 to $n-l+1$ (We do not want to get out of the sequence of matrices) for solving the smaller problems for the outer loop

How do we simulate the recursion Bottom-Up?

We do the following...

We use the following strategy:

- Solve the chain of matrices with small size (The smallest is 2 matrices... after all 1 matrix has cost 0)

Thus, we need

A loop from 2 to n for solving small sequences to larger ones.

In addition

An inner loop from 1 to $n - l + 1$ (We do not want to get out of the sequence of matrices) for solving the smaller problems for the outer loop

Then...

A value

j that is holding the ending index of the subsequence being taken in consideration.

Then a third loop

To go from i to $j-1$ to take the necessary decisions

Then...

A value

j that is holding the ending index of the subsequence being taken in consideration.

Then a third loop

To go from i to $j - 1$ to take the necessary decisions

Bottom-Up Algorithm

MATRIX-CHAIN-ORDER(p)

1. **n = p.length-1**
2. **let** $m[1..n, 1..n]$ **and** $s[1..n-1, 2..n]$ **be new tables**
3. **for** $i = 1$ **to n**
4. $\quad m[i, i] = 0$
5. **for** $l = 2$ **to** n
6. $\quad$ **for** $i = 1$ **to** $n - l + 1$
7. $\quad\quad j = i + l - 1$
8. $\quad\quad m[i, j] = \infty$
9. $\quad\quad$ **for** $k = i$ **to** $j - 1$
10. $\quad\quad\quad q = m[i, k] + m[k+1, j] + p_{i-1}p_kp_j$
11. $\quad\quad\quad$ **if** $q < m[i, j]$
12. $\quad\quad\quad\quad m[i, j] = q$
13. $\quad\quad\quad\quad s[i, j] = k$
14. **return** m **and** s

Bottom-Up Algorithm

MATRIX-CHAIN-ORDER(p)

n = p.length-1
let $m[1..n, 1..n]$ **and** $s[1..n-1, 2..n]$ **be new tables**
for $i = 1$ **to n**
    $m[i, i] = 0$
for $l = 2$ **to** n
    for $i = 1$ **to** $n - l + 1$
        $j = i + l - 1$
        $m[i, j] = \infty$
        for $k = i$ **to** $j - 1$
            $q = m[i, k] + m[k + 1, j] + p_{i-1}p_k p_j$
            if $q < m[i, j]$
                $m[i, j] = q$
                $s[i, j] = k$
return m **and** s

Bottom-Up Algorithm

MATRIX-CHAIN-ORDER(p)

1. **n = p.length-1**
2. **let** $m\left[1..n, 1..n\right]$ **and** $s\left[1..n-1, 2..n\right]$ **be new tables**
3. **for** $i = 1$ **to n**
4. $\quad m\left[i, i\right] = 0$
5. **for** $l = 2$ **to** n
6. $\quad$ **for** $i = 1$ **to** $n - l + 1$
7. $\quad\quad j = i + l - 1$
8. $\quad\quad m\left[i, j\right] = \infty$
9. $\quad\quad$ **for** $k = i$ **to** $j - 1$
10. $\quad\quad\quad q = m\left[i, k\right] + m\left[k+1, j\right] + p_{i-1}p_kp_j$
11. $\quad\quad\quad$ **if** $q < m\left[i, j\right]$
12. $\quad\quad\quad\quad m\left[i, j\right] = q$
13. $\quad\quad\quad\quad s\left[i, j\right] = k$
14. **return** m **and** s

Bottom-Up Algorithm

MATRIX-CHAIN-ORDER(p)

n = p.length-1
let $m[1..n, 1..n]$ **and** $s[1..n-1, 2..n]$ **be new tables**
for $i = 1$ **to n**
    $m[i,i] = 0$
for $l = 2$ **to** n
    for $i = 1$ **to** $n - l + 1$
        $j = i + l - 1$
        $m[i,j] = \infty$
        for $k = i$ **to** $j - 1$
            $q = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$
            if $q < m[i,j]$
                $m[i,j] = q$
                $s[i,j] = k$
return m **and** s

Bottom-Up Algorithm

MATRIX-CHAIN-ORDER(p)

```
n = p.length-1
let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n
    for i = 1 to n - l + 1
        j = i + l - 1
        m[i, j] = ∞
        for k = i to j - 1
            q = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

Bottom-Up Algorithm

MATRIX-CHAIN-ORDER(p)

```
n = p.length-1
let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables
for i = 1 to n
        m[i, i] = 0
for l = 2 to n
        for i = 1 to n - l + 1
                j = i + l - 1
                m[i, j] = ∞
                for k = i to j - 1
                        q = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j
                        if q < m[i, j]
                                m[i, j] = q
                                s[i, j] = k
return m and s
```

Bottom-Up Algorithm

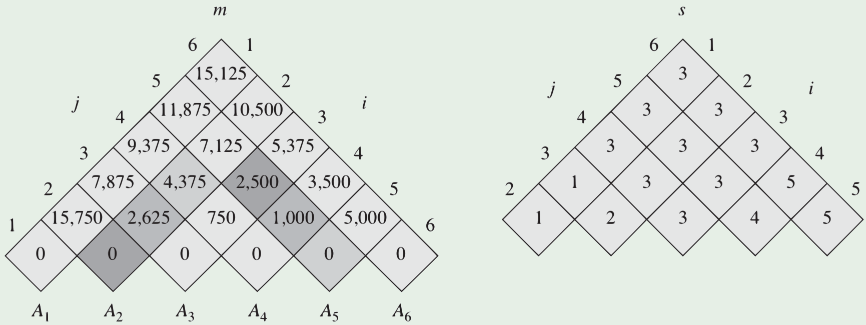
MATRIX-CHAIN-ORDER(p)

n = p.length-1
let $m[1..n, 1..n]$ **and** $s[1..n-1, 2..n]$ **be new tables**
for $i = 1$ **to n**
    $m[i, i] = 0$
for $l = 2$ **to** n
    for $i = 1$ **to** $n - l + 1$
        $j = i + l - 1$
        $m[i, j] = \infty$
        for $k = i$ **to** $j - 1$
            $q = m[i, k] + m[k+1, j] + p_{i-1}p_kp_j$
            if $q < m[i, j]$
                $m[i, j] = q$
                $s[i, j] = k$
return m **and** s

Example

Example

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimensions	35×30	30×15	15×5	5×10	10×20	20×25



Complexity

By looking at the algorithm we have

$l \leftarrow n-1$

$i \leftarrow n-l-1$

$j \leftarrow i+l-1$

Then

$O(n^3)$

Complexity

By looking at the algorithm we have

$l \leftarrow n - 1$

$i \leftarrow n - l - 1$

$j \leftarrow i + l - 1$

Then

$O(n^3)$

Reconstruct the Output

PRINT-OPTIMAL-PARENS(s, i, j)

```
if i == j
        print "A_i"
else print "("
        PRINT-OPTIMAL-PARENS(s, i, s[i, j])
        PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
        print ")"
```

Final solution for the example

$((A_1(A_2A_3))((A_4A_5)A_6))$

Reconstruct the Output

PRINT-OPTIMAL-PARENS(s, i, j)

if $i == j$
    print "A_i"
else print "("
    PRINT-OPTIMAL-PARENS$(s, i, s[i, j])$
    PRINT-OPTIMAL-PARENS$(s, s[i, j] + 1, j)$
    print ")"

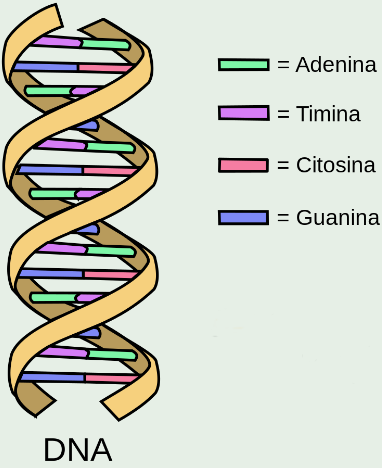
Final solution for the example

$((A_1(A_2A_3))((A_4A_5)A_6)$

Outline

In Biology

Biological applications often need to compare the DNA of two (or more) different organisms.



Why?

Because given these strands

- S_1= ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
- S_1= GTCGTTCGGAATGCCGTTGCTCTGTAAA

Why?

Because given these strands

- $S_1 =$ ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
- $S_1 =$ GTCGTTCGGAATGCCGTTGCTCTGTAAA

We want

To determine how "similar" the two strands are, as some measure of how closely related the two organisms are.

Why?

Because given these strands

- S_1= ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
- S_1= GTCGTTCGGAATGCCGTTGCTCTGTAAA

We want

To determine how "similar" the two strands are, as some measure of how closely related the two organisms are.

Ways of Measuring Similarity

For example

We can say that two DNA strands are similar if one is a substring of the other.

However

This does not happen in the previous example...

A better measure

Imagine that you are given another strand S_3 in which the bases on it appears in S_1 and S_2 (Common Basis)

Ways of Measuring Similarity

For example

We can say that two DNA strands are similar if one is a substring of the other.

However

This does not happen in the previous example...

A better measure

Imagine that you are given another strand S_3 in which the bases on it appears in S_1 and S_2 (Common Basis)

Ways of Measuring Similarity

For example

We can say that two DNA strands are similar if one is a substring of the other.

However

This does not happen in the previous example...

A better measure

Imagine that you are given another strand S_3 in which the bases on it appears in S_1 and S_2 (Common Basis)

The Longer Strand

The Longer S_3

The more similar the organism, represented by S_1 and S_2, are.

Thus

We need to find S_3 the Longest Common Subsequence

The Longer Strand

The Longer S_3

The more similar the organism, represented by S_1 and S_2, are.

Thus

We need to find S_3 the Longest Common Subsequence

Longest Common Subsequence

Definition

Given a sequence $X = \langle x_1, x_2, ..., x_m \rangle$, a sequence $Z = \langle z_1, z_2, ..., z_k \rangle$ is a subsequence of X if there exist a strictly increasing sequence $\langle i_1, i_2, ..., i_k \rangle$ of indices of X such that $x_i = z_j$.

Therefore

Given two sequences X and Y, we say that Z is a common subsequence of X and Y, if Z is a subsequence of both X and Y.

Longest Common Subsequence

Definition

Given a sequence $X = \langle x_1, x_2, ..., x_m \rangle$, a sequence $Z = \langle z_1, z_2, ..., z_k \rangle$ is a subsequence of X if there exist a strictly increasing sequence $\langle i_1, i_2, ..., i_k \rangle$ of indices of X such that $x_i = z_j$.

Therefore

Given two sequences X and Y, we say that Z is a common subsequence of X and Y, if Z is a subsequence of both X and Y.

Characterizing the LCS

Theorem 15.1 (Optimal substructure of an LCS)

Let $X = \langle x_1, x_2, ..., x_m \rangle$ and $Y = \langle y_1, y_2, ..., y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, ..., z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Characterizing the LCS

Theorem 15.1 (Optimal substructure of an LCS)

Let $X = \langle x_1, x_2, ..., x_m \rangle$ and $Y = \langle y_1, y_2, ..., y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, ..., z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Characterizing the LCS

Theorem 15.1 (Optimal substructure of an LCS)

Let $X = \langle x_1, x_2, ..., x_m \rangle$ and $Y = \langle y_1, y_2, ..., y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, ..., z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Characterizing the LCS

Theorem 15.1 (Optimal substructure of an LCS)

Let $X = \langle x_1, x_2, ..., x_m \rangle$ and $Y = \langle y_1, y_2, ..., y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, ..., z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Overlapping Property

To find an LCS for X and Y, we may need to find

- LCS of X_{n-1} and Y_{n-1}
- LCS of X and Y_{n-1}
- LCS of Y and X_{m-1}

Overlapping Property

To find an LCS for X and Y, we may need to find

- LCS of X_{n-1} and Y_{n-1}
- LCS of X and Y_{n-1}
- LCS of Y and X_{m-1}

Overlapping Property

To find an LCS for X and Y, we may need to find

- LCS of X_{n-1} and Y_{n-1}
- LCS of X and Y_{n-1}
- LCS of Y and X_{m-1}

Thus

For the first case

$$Recursion(i, j) = Recursion(i - 1, j - 1) + 1$$

Second case

$$Recursion(i, j) = Recursion(i, j - 1)$$

However, you have the too

$$Recursion(i, j) = Recursion(i - 1, j)$$

Thus

For the first case

$$Recursion(i,j) = Recursion(i-1,j-1)+1$$

Second case

$$Recursion(i,j) = Recursion(i,j-1)$$

However, you have the too

$$Recursion(i,j) = Recursion(i-1,j)$$

Thus

For the first case

$Recursion(i,j) = Recursion(i-1, j-1) + 1$

Second case

$Recursion(i,j) = Recursion(i, j-1)$

However, you have the too

$Recursion(i,j) = Recursion(i-1, j)$

Then, we can collapse second and third case

In the following way

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

The Final Recurrence

Let $c[i,j]$ the length of the common subsequence of X_i, Y_j

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > \text{ and } x_i \neq y_j \end{cases}$$

Thus, we can do the following

It is possible

To develop an exponential algorithm.

Thus, we can do the following

It is possible

To develop an exponential algorithm.

However

- Let us to develop an algorithm that takes $O(mn)$

Thus, we can do the following

It is possible

To develop an exponential algorithm.

However

- Let us to develop an algorithm that takes $O\left(mn\right)$

First, we need to take in account

- $X = \langle x_1, x_2, x_3, ..., x_m \rangle$
- $Y = \langle y_1, y_2, y_3, ..., y_n \rangle$

Thus, we can do the following

It is possible

To develop an exponential algorithm.

However

- Let us to develop an algorithm that takes $O(mn)$

First, we need to take in account

- $X = \langle x_1, x_2, x_3, ..., x_m \rangle$
- $Y = \langle y_1, y_2, y_3, ..., y_n \rangle$

We do the following

Use extra memory

- You can store the result of $c[i,j]$ values in a table $c[0..m.0..n]$
  - In order to use it, the entries are computed in row-major order.

We do the following

Use extra memory

- You can store the result of $c[i,j]$ values in a table $c[0..m.0..n]$
  - In order to use it, the entries are computed in row-major order.

Row-Major Order

The procedure fills in the first row of c from left to right, then the second row, and so on.

We do the following

Use extra memory

- You can store the result of $c[i,j]$ values in a table $c[0..m.0..n]$
  - In order to use it, the entries are computed in row-major order.

Row-Major Order

The procedure fills in the first row of c from left to right, then the second row, and so on.

Why?

Clearly, we are using the bottom-up approach, so we get the results for the smallest problem first!!!

We do the following

Use extra memory

- You can store the result of $c[i,j]$ values in a table $c[0..m.0..n]$
  - In order to use it, the entries are computed in row-major order.

Row-Major Order

The procedure fills in the first row of c from left to right, then the second row, and so on.

Why?

Clearly, we are using the bottom-up approach, so we get the results for the smallest problem first!!!

We also have a table to store the decisions

Ok, What type of symbols are in that table?

	y_i	a	v	c	r	e
x_i	0	0	0	0	0	0
a	0	$\nwarrow$				
b	0					
c	0			$\nwarrow$		
d	0					
e	0					$\nwarrow$

Thus, for the different cases

$x_m = y_n$

- Simply use the symbol " $\nwarrow$ ".
- After all we are consuming the same symbol

$c[i-1, j] \geq c[i, j-1]$

- Simply use the symbol " $\uparrow$ ".
- After all you are moving up in the rows

$c[i-1, j] < c[i, j-1]$

- Simply use the symbol " $\leftarrow$ ".
- After all you are moving left in the columns

Thus, for the different cases

$x_m = y_n$

- Simply use the symbol " $\nwarrow$ ".
- After all we are consuming the same symbol

$c[i-1,j] \geq c[i,j-1]$

- Simply use the symbol " $\uparrow$ ".
- After all you are moving up in the rows

$c[i-1,j] < c[i,j-1]$

- Simply use the symbol " $\leftarrow$ ".
- After all you are moving left in the columns

Thus, for the different cases

$x_m = y_n$

- Simply use the symbol "$\nwarrow$".
- After all we are consuming the same symbol

$c[i-1,j] \geq c[i,j-1]$

- Simply use the symbol "$\uparrow$".
- After all you are moving up in the rows

$c[i-1,j] < c[i,j-1]$

- Simply use the symbol "$\leftarrow$".
- After all you are moving left in the columns

How, we fill $c\left[0..m.0..n\right]$

Something Notable

We need to increase the columns and the rows.

Thus

- for $i = 1$ to m
-         for $j = 1$ to n

In addition, $c\left[0..m, 0\right]$ and $c\left[0, 0..n\right]$

If one of your subproblems is empty:

- We know that the common elements are 0.

How, we fill $c\left[0..m.0..n\right]$

Something Notable

We need to increase the columns and the rows.

Thus

- for $i = 1$ to m
- for $j = 1$ to n

In addition, $c\left[0..m,0\right]$ and $c\left[0,0..n\right]$

If one of your subproblems is empty:

- We know that the common elements are 0.

How, we fill $c\,[0..m.0..n]$

Something Notable

We need to increase the columns and the rows.

Thus

- for $i = 1$ to m
-         for $j = 1$ to n

In addition, $c\,[0..m, 0]$ and $c\,[0, 0..n]$

If one of your subproblems is empty:

- We know that the common elements are 0.

Final Algorithm - Complexity $O(mn)$

LCS-Length(X, Y)

1. $m = X.length$
2. $n = Y.length$
3. **let** $b[1..m, 1..n]$ **and** $c[0..m, 0..n]$ **be new tables**
4. **for** $i = 1$ **to** m
5. $\quad c[i, 0] = 0$
6. **for** $j = 0$ **to** n
7. $\quad c[0, j] = 0$
8. **for** $i = 1$ **to** m
9. $\quad$ **for** $j = 1$ **to** n
10. $\quad\quad$ **if** $x_i == y_j$
11. $\quad\quad\quad c[i, j] = c[i-1, j-1] + 1$
12. $\quad\quad\quad b[i, j] = " \nwarrow "$
13. $\quad\quad$ **elseif** $c[i-1, j] \geq c[i, j-1]$
14. $\quad\quad\quad c[i, j] = c[i-1, j]$
15. $\quad\quad\quad b[i, j] = " \uparrow "$
16. $\quad\quad$ **else** $c[i, j] = c[i, j-1]$
17. $\quad\quad\quad b[i, j] = " \leftarrow "$
18. **return** c **and** b

Final Algorithm - Complexity $O(mn)$

LCS-Length(X, Y)

1. $m = X.length$
2. $n = Y.length$
3. **let** $b\left[1..m, 1..n\right]$ **and** $c\left[0..m, 0..n\right]$ **be new tables**
4. **for** $i = 1$ **to** m
5. $\qquad c\left[i, 0\right] = 0$
6. **for** $j = 0$ **to** n
7. $\qquad c\left[0, j\right] = 0$
8. **for** $i = 1$ **to** m
9. $\qquad$ **for** $j = 1$ **to** n
10. $\qquad\qquad$ **if** $x_i == y_j$
11. $\qquad\qquad\qquad c\left[i, j\right] = c\left[i-1, j-1\right] + 1$
12. $\qquad\qquad\qquad b\left[i, j\right] = " \nwarrow "$
13. $\qquad\qquad$ **elseif** $c[i-1, j] \geq c[i, j-1]$
14. $\qquad\qquad\qquad c\left[i, j\right] = c\left[i-1, j\right]$
15. $\qquad\qquad\qquad b\left[i, j\right] = " \uparrow "$
16. $\qquad\qquad$ **else** $c\left[i, j\right] = c\left[i, j-1\right]$
17. $\qquad\qquad\qquad b\left[i, j\right] = " \leftarrow "$
18. **return** c **and** b

Final Algorithm - Complexity $O(mn)$

LCS-Length(X, Y)

$m = X.length$
$n = Y.length$
let $b[1..m, 1..n]$ **and** $c[0..m, 0..n]$ **be new tables**
for $i = 1$ **to** m
    $c[i, 0] = 0$
for $j = 0$ **to** n
    $c[0, j] = 0$
for $i = 1$ **to** m
    for $j = 1$ **to** n
        if $x_i == y_j$
            $c[i, j] = c[i-1, j-1] + 1$
            $b[i, j] = "\nwarrow"$
        elseif $c[i-1, j] \geq c[i, j-1]$
            $c[i, j] = c[i-1, j]$
            $b[i, j] = "\uparrow"$
        else $c[i, j] = c[i, j-1]$
            $b[i, j] = "\leftarrow"$
return c **and** b

Final Algorithm - Complexity $O(mn)$

LCS-Length(X, Y)

- $m = X.length$
- $n = Y.length$
- **let** $b\left[1..m, 1..n\right]$ **and** $c\left[0..m, 0..n\right]$ **be new tables**
- **for** $i = 1$ **to** m
  - $c\left[i, 0\right] = 0$
- **for** $j = 0$ **to** n
  - $c\left[0, j\right] = 0$
- **for** $i = 1$ **to** m
  - **for** $j = 1$ **to** n
    - **if** $x_i == y_j$
      - $c\left[i, j\right] = c\left[i-1, j-1\right] + 1$
      - $b\left[i, j\right] = " \nwarrow "$
    - **elseif** $c[i-1, j] \geq c[i, j-1]$
      - $c\left[i, j\right] = c\left[i-1, j\right]$
      - $b\left[i, j\right] = " \uparrow "$
    - **else** $c\left[i, j\right] = c\left[i, j-1\right]$
      - $b\left[i, j\right] = " \leftarrow "$
- **return** c **and** b

Final Algorithm - Complexity $O(mn)$

LCS-Length(X, Y)

1. $m = X.length$
2. $n = Y.length$
3. **let** $b\left[1..m, 1..n\right]$ **and** $c\left[0..m, 0..n\right]$ **be new tables**
4. **for** $i = 1$ **to** m
5. $c\left[i, 0\right] = 0$
6. **for** $j = 0$ **to** n
7. $c\left[0, j\right] = 0$
8. **for** $i = 1$ **to** m
9.   **for** $j = 1$ **to** n
10.   **if** $x_i == y_j$
11. $c\left[i, j\right] = c\left[i-1, j-1\right] + 1$
12. $b\left[i, j\right] = " \nwarrow "$
13.   **elseif** $c[i-1, j] \geq c[i, j-1]$
14. $c\left[i, j\right] = c\left[i-1, j\right]$
15. $b\left[i, j\right] = " \uparrow "$
16.   **else** $c\left[i, j\right] = c\left[i, j-1\right]$
17. $b\left[i, j\right] = " \leftarrow "$
18. **return** c **and** b

Final Algorithm - Complexity $O(mn)$

LCS-Length(X, Y)

$m = X.length$

$n = Y.length$

let $b\left[1..m, 1..n\right]$ **and** $c\left[0..m, 0..n\right]$ **be new tables**

for $i = 1$ **to** m

    $c\left[i, 0\right] = 0$

for $j = 0$ **to** n

    $c\left[0, j\right] = 0$

for $i = 1$ **to** m

    for $j = 1$ **to** n

        if $x_i == y_j$

            $c\left[i, j\right] = c\left[i-1, j-1\right] + 1$

            $b\left[i, j\right] = " \nwarrow "$

        elseif $c[i-1, j] \geq c[i, j-1]$

            $c\left[i, j\right] = c\left[i-1, j\right]$

            $b\left[i, j\right] = " \uparrow "$

        else $c\left[i, j\right] = c\left[i, j-1\right]$

            $b\left[i, j\right] = " \leftarrow "$

return c **and** b

Final Algorithm - Complexity $O(mn)$

LCS-Length(X, Y)

$m = X.length$

$n = Y.length$

let $b\left[1..m, 1..n\right]$ **and** $c\left[0..m, 0..n\right]$ **be new tables**

for $i = 1$ **to** m

    $c\left[i, 0\right] = 0$

for $j = 0$ **to** n

    $c\left[0, j\right] = 0$

for $i = 1$ **to** m

    for $j = 1$ **to** n

        if $x_i == y_j$

            $c\left[i, j\right] = c\left[i-1, j-1\right] + 1$

            $b\left[i, j\right] = " \nwarrow "$

        elseif $c[i-1, j] \geq c[i, j-1]$

            $c\left[i, j\right] = c\left[i-1, j\right]$

            $b\left[i, j\right] = " \uparrow "$

        else $c\left[i, j\right] = c\left[i, j-1\right]$

            $b\left[i, j\right] = " \leftarrow "$

return c **and** b

Final Algorithm - Complexity $O(mn)$

LCS-Length(X, Y)

$m = X.length$

$n = Y.length$

let $b\left[1..m, 1..n\right]$ **and** $c\left[0..m, 0..n\right]$ **be new tables**

for $i = 1$ **to** m

    $c\left[i, 0\right] = 0$

for $j = 0$ **to** n

    $c\left[0, j\right] = 0$

for $i = 1$ **to** m

    for $j = 1$ **to** n

        if $x_i == y_j$

            $c\left[i, j\right] = c\left[i-1, j-1\right] + 1$

            $b\left[i, j\right] = " \nwarrow "$

        elseif $c[i-1, j] \geq c[i, j-1]$

            $c\left[i, j\right] = c\left[i-1, j\right]$

            $b\left[i, j\right] = " \uparrow "$

        else $c\left[i, j\right] = c\left[i, j-1\right]$

            $b\left[i, j\right] = " \leftarrow "$

return c **and** b

Final Algorithm - Complexity $O(mn)$

LCS-Length(X, Y)

$m = X.length$

$n = Y.length$

let $b\,[1..m, 1..n]$ **and** $c\,[0..m, 0..n]$ **be new tables**

for $i = 1$ **to** m

    $c\,[i, 0] = 0$

for $j = 0$ **to** n

    $c\,[0, j] = 0$

for $i = 1$ **to** m

    for $j = 1$ **to** n

        if $x_i == y_j$

            $c\,[i, j] = c\,[i-1, j-1] + 1$

            $b\,[i, j] = "\nwarrow"$

        elseif $c[i-1, j] \geq c[i, j-1]$

            $c\,[i, j] = c\,[i-1, j]$

            $b\,[i, j] = "\uparrow"$

        else $c\,[i, j] = c\,[i, j-1]$

            $b\,[i, j] = "\leftarrow"$

return c **and** b

Example

The matrices after running the algorithm

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Constructing the LCS

PRINT-LCS(b, X, i, j)

if $i == 0$ or $j == 0$

    return

if $b[i, j] ==$ "$\nwarrow$"

    PRINT-LCS$(b, X, i-1, j-1)$

    print x_i

elseif $b[i, j] ==$ "$\uparrow$"

    PRINT-LCS$(b, X, i-1, j)$

else PRINT-LCS$(b, X, i, j-1)$

Constructing the LCS

PRINT-LCS(b, X, i, j)

```
if i == 0 or j == 0
        return
if b[i, j] == "↖"
        PRINT-LCS(b, X, i − 1, j − 1)
        print x_i
elseif b[i, j] == "↑"
        PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

Complexity

$O(m + n)$

Constructing the LCS

PRINT-LCS(b, X, i, j)

if $i == 0$ or $j == 0$
    return
if $b[i, j] ==$ "$\nwarrow$"
    PRINT-LCS$(b, X, i-1, j-1)$
    print x_i
elseif $b[i, j] ==$ "$\uparrow$"
    PRINT-LCS$(b, X, i-1, j)$
else PRINT-LCS$(b, X, i, j-1)$

Complexity

$O(m+n)$

Constructing the LCS

PRINT-LCS(b, X, i, j)

if $i == 0$ or $j == 0$

    return

if $b[i,j] ==$ "$\nwarrow$"

    PRINT-LCS$(b, X, i-1, j-1)$

    print x_i

elseif $b[i,j] ==$ "$\uparrow$"

    PRINT-LCS$(b, X, i-1, j)$

else PRINT-LCS$(b, X, i, j-1)$

Complexity

$O(m+n)$

Exercises

From Cormen's book solve

- 15.3-3
- 15.3-5
- 15.2-3
- 15.2-4
- 15.2-5
- 15.4-2
- 15.4-4
- 15.4-5