Analysis of Algorithms

Dynamic Programming

Andres Mendez-Vazquez

February 14, 2018

Notes

Outline

Notes

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.

Notes

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.

Notes

Looks Terrifying!!!

However

It is quite simple!!!

Notes

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!!!

Notes

Elements of Dynamic Programming

Finally Rebuild the Optimal Solution

Construct an optimal solution from computed information.

Notes

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

Notes

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?

Notes

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

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

Notes

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!!!**

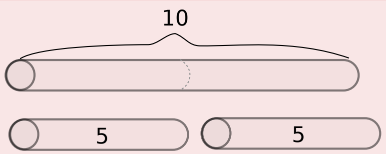
Notes

How can you obtain the recursion?

What about taking a decision each time?

In how to cut the rod!

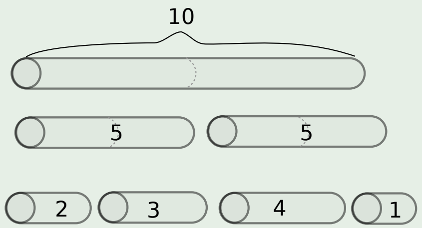
For example



Notes

It looks like what?

One more cut



Yes

Recursion

Notes

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 $j = 1, 2, ..., n-1$

Notes

Thus, we take a final decision!!!

Thus

Which One?

The Largest One

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

Notes

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.

Notes

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:

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

Notes

Then

Then, we have a set of cuts

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

Clearly

Contradiction!!!

Notes

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. ...

Notes

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.

Notes

Finally

Code

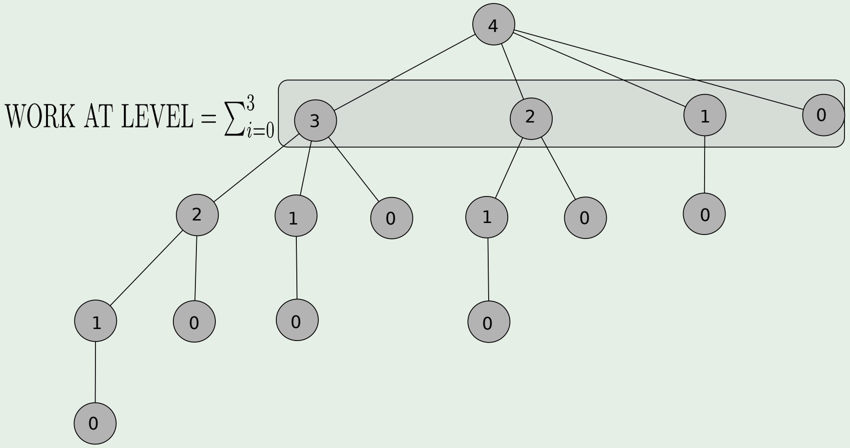
Cut-Rod(p, n)

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

Notes

How the recursion tree for this code looks like?

First, Did you notice this?



Notes

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

Notes

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

$$T(n) = 2^n \quad (2)$$

Notes

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.

Notes

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.

Notes

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)

Notes

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

Code

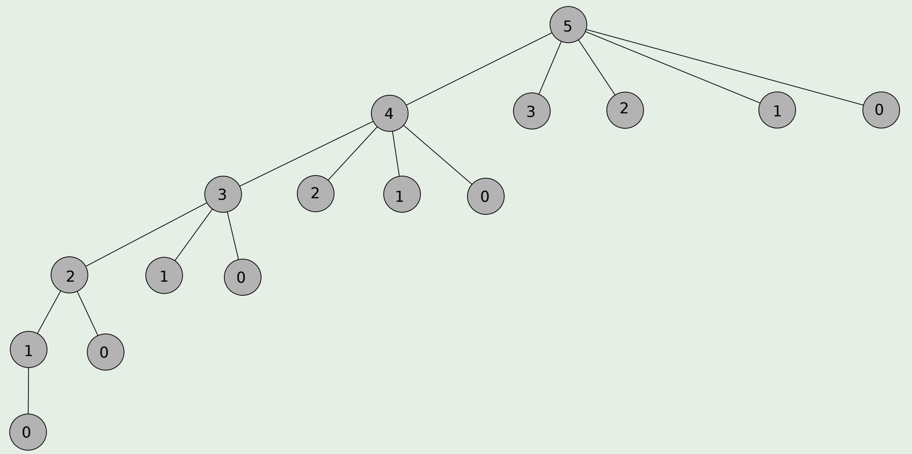
Memoized-Cut-Rod-Aux(p, n, r)

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

Notes

The Recursion Tree of Memoized-Cut-Rod

Tree for $n = 5$



Notes

Thus

We have that

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

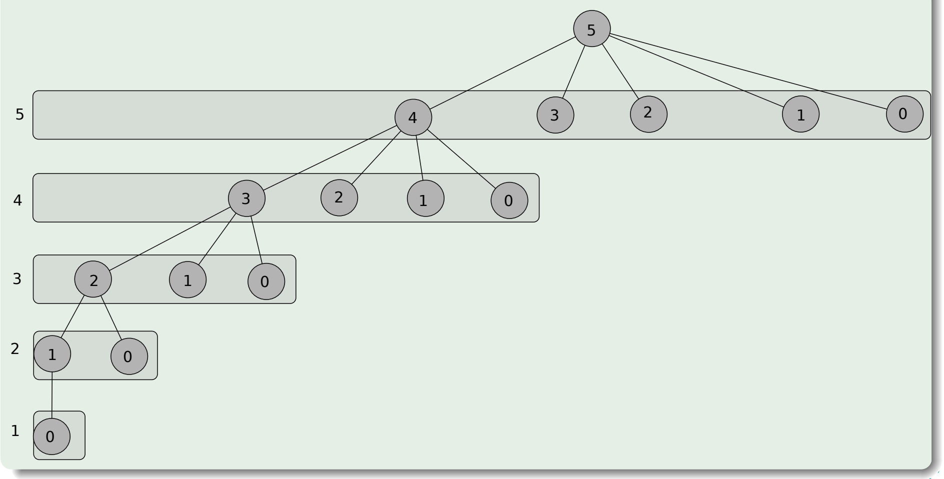
Thus

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

Notes

Then look at this..

Something Notable



Notes

Complexity

Add the works

We have then

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

Then, we have

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

Notes

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.

Notes

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 $r[n]$

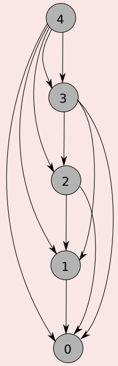
Notes

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



Notes

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

Notes

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

Notes

Printing Code

Code

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

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

Notes

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

Notes

Optimal Substructure

In dynamic programming

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

Notes

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.

Notes

We have the following steps

Fourth

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

Notes

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.

Notes

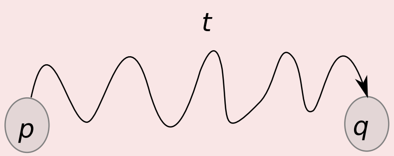
We can explain subtleties about the Optimal Substructure

Unweighted shortest path

It has an optimal substructure

Why?

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

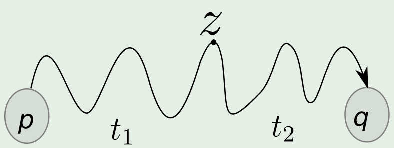


Notes

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).

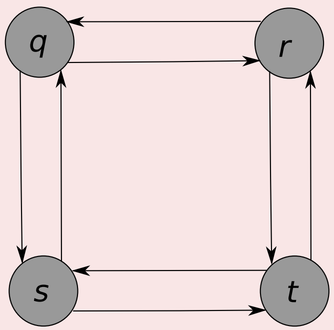
Notes

However

Some problems do not have the optimal substructure

The longest unweighted path

Example



Notes

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 $q \longrightarrow r$

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

Notes

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.

Notes

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!!!

Notes

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?

Notes

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.

Notes

Overlapping Subproblems

We have two ways of solving the problem

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

Notes

Reconstruction of Subproblems

To reconstruct

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

Notes

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.

Notes

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.

Notes

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 $O(mn)$.

Notes

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

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

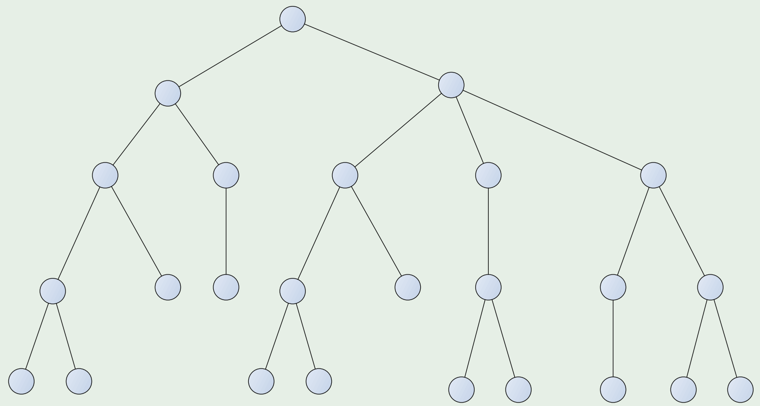
Subproblems
$x_i, x_{i+1}, ..., x_j$

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 $O(n^2)$.

Notes

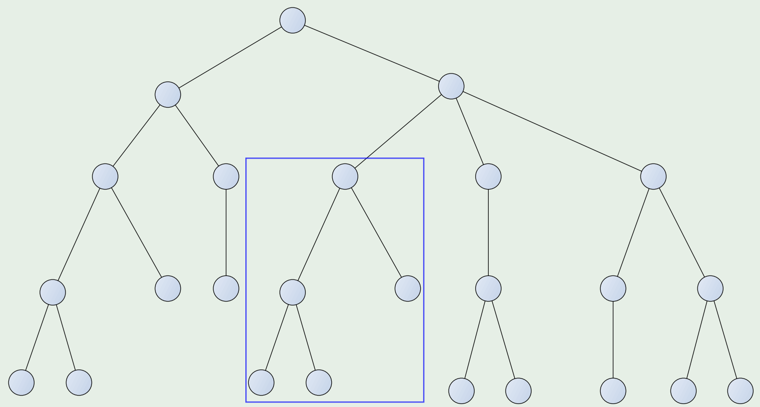
Input is a rooted subtree

Input


Notes

Input is a rooted subtree

Subproblem



Notes

Question

How Many Subproblems do you have?

Any Idea?

Notes

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.

Notes

Definition

Output

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

Example

5 2 8 6 3 6 9 7

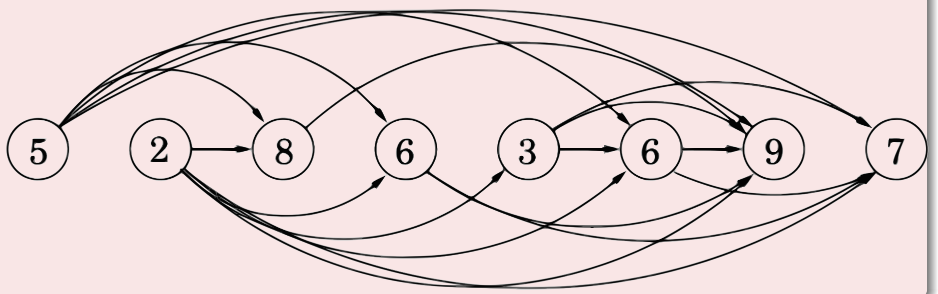
Notes

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



Notes

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.

Notes

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

Notes

Thus

We start at a certain j

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

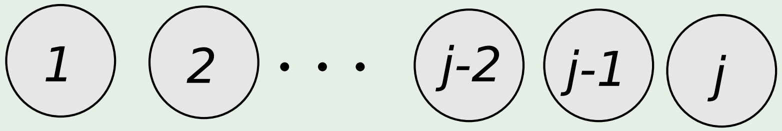
Here is the recursion for $\forall A[i] < A[j]$

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

Notes

What is the meaning of this?

When is there an edge between i_k and j?



Notes

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

Notes

Recursive Function

The final recursive code

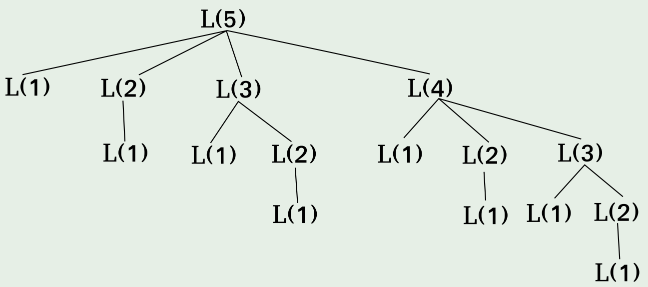
Recursive-Longest-Subsequence(A, n)

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. $\qquad q = 1 + t$
8. return q

Notes

What about the Complexity?

Recursion Tree - Can somebody Guess the Complexity?



Notes

How we save in recursive calls

First

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

Notes

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.

Step 2

- A measure about the longest subsequence.

Step 3

- Initialize everything to 1 (Itself).

Step 4

- We know that the

Notes

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.

Notes

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

Notes

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 $N \cdot M \cdot P$.

Notes

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 $25000 + 10 \cdot 100 \cdot 50 = 75000$

Notes

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

Notes

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

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

Notes

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 $p_{k-2}p_{k-1}p_{k+1} + p_{k-1}p_kp_{k+1}$

Notes

In addition

Look at the following multiplication

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

We have the following

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

The total cost of this multiplication is

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

Notes

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!

Notes

Now, the Recursion can be wrote!!!

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

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

Notes

The Recursive Solution

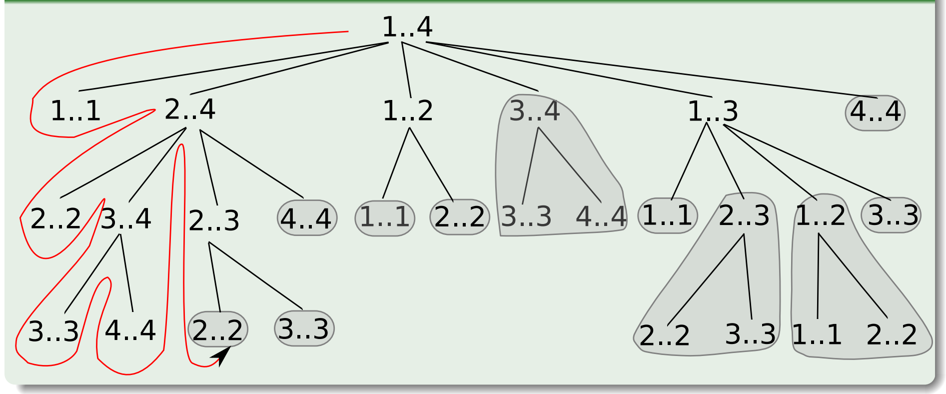
Recursive Algorithm

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 $m[i, j]$

Notes

Again!!! Overlapping substructure

Red Line Represents the Recursion Path



Notes

This is a nightmare

We have the following recursion

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

Notes

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

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

Notes

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 $n = 1$ $T(1) \geq 1 = 2^0$

Notes

Then

Now, for $n \geq 2$

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

Notes

Thus

We want to avoid to calculate the same value many times

Use bottom up approach and store values at each step.

Notes

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

Notes

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

Notes

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

Notes

Bottom-Up Algorithm

MATRIX-CHAIN-ORDER(p)

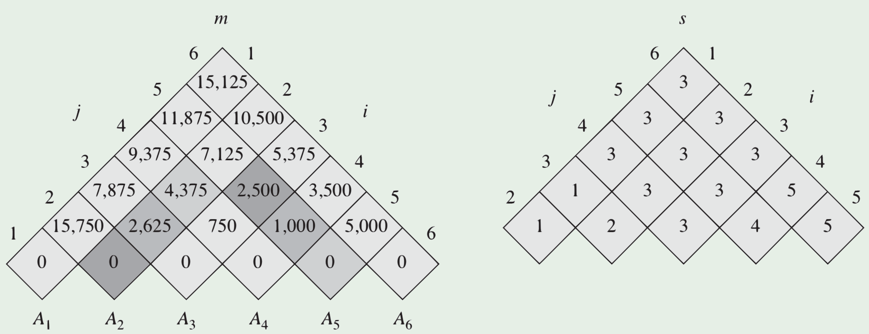
n = p.length-1
let $m[1..n, 1..n]$ **and** $s[1..n-1, 2..n]$ **be new tables**
for $i = 1$ **to n**
    $m[i,i] = 0$
for $l = 2$ **to** n
    for $i = 1$ **to** $n - l + 1$
        $j = i + l - 1$
        $m[i,j] = \infty$
        for $k = i$ **to** $j - 1$
            $q = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$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

Notes

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



Notes

Complexity

By looking at the algorithm we have

$l \leftarrow n-1$
$i \leftarrow n-l-1$
$j \leftarrow i+l-1$

Then

$O(n^3)$

Notes

Reconstruct the Output

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

1. if $i == j$
2. print "A_i"
3. else print "("
4. PRINT-OPTIMAL-PARENS($s, i, s[i, j]$)
5. PRINT-OPTIMAL-PARENS($s, s[i, j] + 1, j$)
6. print ")"

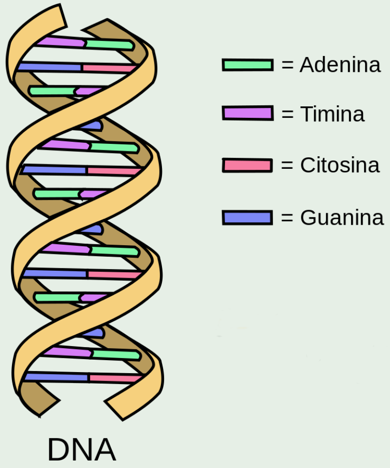
Final solution for the example

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

Notes

In Biology

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



Notes

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.

Notes

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)

Notes

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

Notes

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.

Notes

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}.

Notes

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}

Notes

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

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

Notes

Then, we can collapse second and third case

In the following way

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

Notes

The Final Recurrence

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

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

Notes

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

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

Notes

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!!!

Notes

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					$\nwarrow$

Notes

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

Notes

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.

Notes

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] = "↖"
                elseif c[i − 1, j] ≥ c[i, j − 1]
                        c[i, j] = c[i − 1, j]
                        b[i, j] = "↑"
                else c[i, j] = c[i, j − 1]
                        b[i, j] = "←"
return c and b
```


Notes

Example

The matrices after running the algorithm

i		j 0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2	B	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
3	C	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
4	B	0	$\nwarrow$ 1	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\leftarrow$ 3
5	D	0	$\uparrow$ 1	$\nwarrow$ 2	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3
6	A	0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\uparrow$ 3	$\nwarrow$ 4
7	B	0	$\nwarrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\nwarrow$ 4	$\uparrow$ 4

Notes

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

$O(m+n)$

Notes

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

Notes

Notes