Design & Analysis of Algorithms
CSE 246

Dynamic Programming (Part 1)

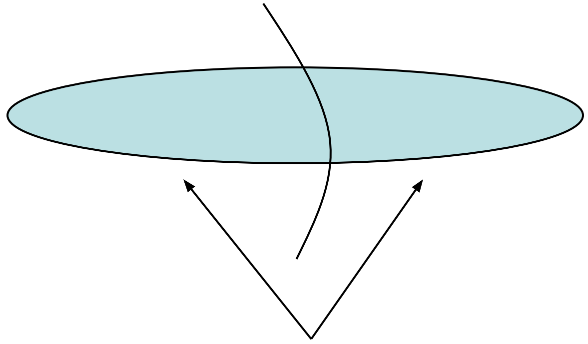
Dynamic Programming

- An algorithm design technique (like divide and conquer)
- Divide and conquer
  - Partition the problem into independent subproblems
  - Solve the subproblems recursively
  - Combine the solutions to solve the original problem

DP - Two key ingredients

- Two key ingredients for an optimization problem to be suitable for a dynamic-programming solution:

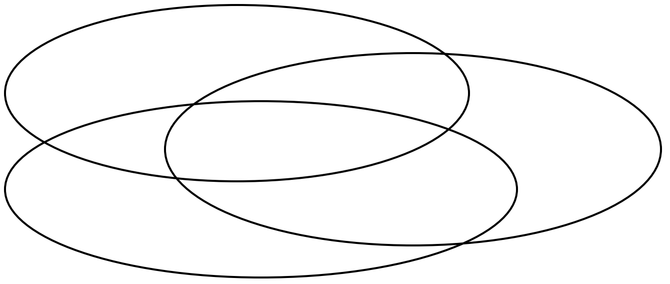
1. optimal substructures



Each substructure is optimal.

(Principle of optimality)

2. overlapping subproblems



Subproblems are dependent.

(otherwise, a divide-and-conquer approach is the choice.)

Three basic components

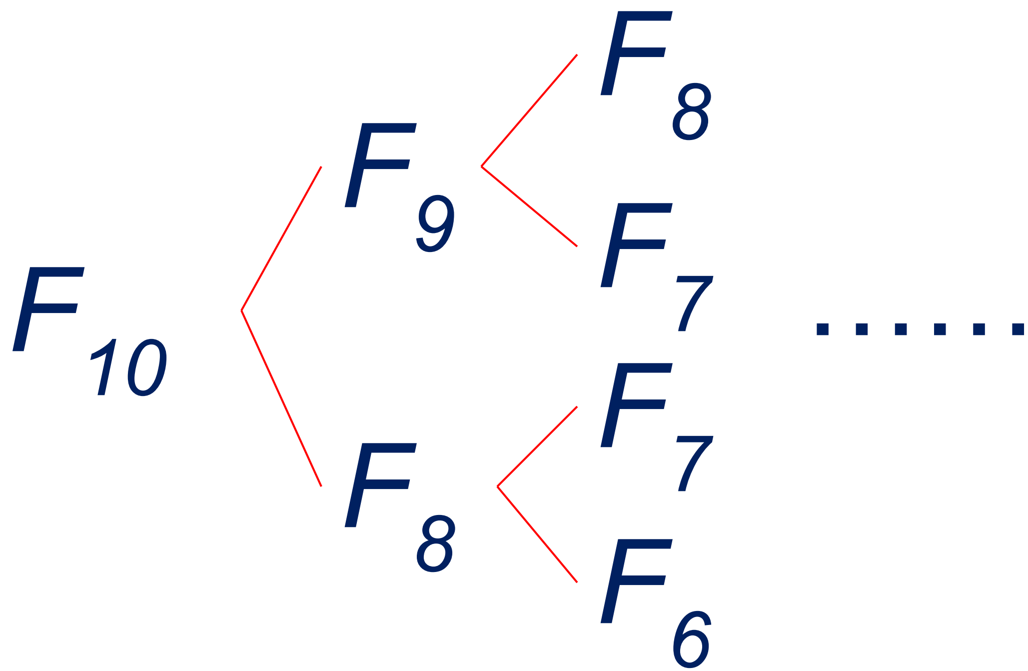
- The development of a dynamic-programming algorithm has three basic components:
  - The recurrence relation (for defining the value of an optimal solution);
  - The tabular computation (for computing the value of an optimal solution);
  - The traceback (for delivering an optimal solution).

Fibonacci numbers

The *Fibonacci numbers* are defined by the following recurrence:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2} \quad \text{for } i>1.$$

How to compute F_{10}?



Dynamic Programming

- Applicable when subproblems are not independent
  - Subproblems share subsubproblems

E.g.: Fibonacci numbers:

- Recurrence: $F(n) = F(n-1) + F(n-2)$
- Boundary conditions: $F(1) = 0$, $F(2) = 1$
- Compute: $F(5) = 3$, $F(3) = 1$, $F(4) = 2$

  - A divide and conquer approach would repeatedly solve the common subproblems
  - Dynamic programming solves every subproblem just once and stores the answer in a table

Tabular computation

- The tabular computation can avoid recompuation.

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}
0	1	1	2	3	5	8	13	21	34	55

Result

Ways of solving

- Top Down Approach

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}
0	1	1	2	3	5	8	13	21	34	55

- Bottom Up Approach

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}
0	1	1	2	3	5	8	13	21	34	55

Dynamic Programming Algorithm

1. Characterize the structure of an optimal solution
2. **Recursively** define the value of an optimal solution
3. Compute the value of an optimal solution in a bottom-up fashion
4. Construct an optimal solution from computed information

Results are stored in a DP Table

Coin Change Problem

- Suppose you are given ***n*** types of coin - C_1, C_2, … , C_n coin, and another number ***K.***
- Is it possible to make K using above types of coin?
  - Number of each coin is infinite
  - Number of each coin is finite
- Find minimum number of coin that is required to make ***K***?
  - Number of each coin is infinite
  - Number of each coin is finite

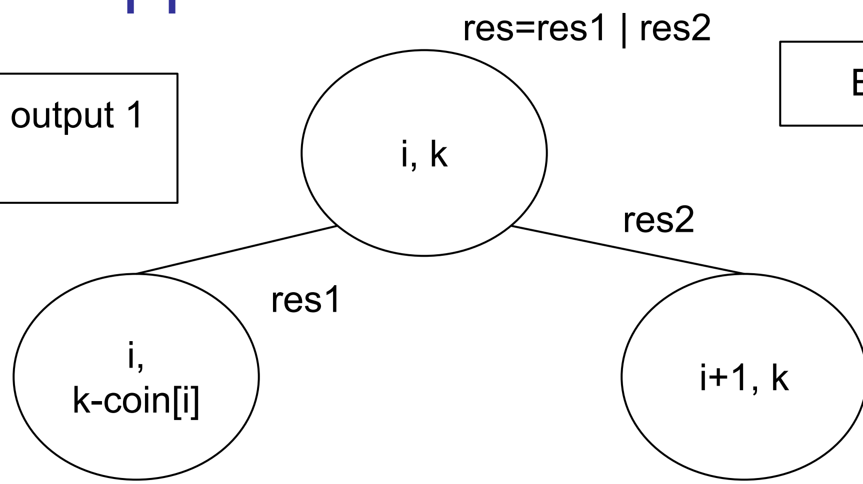
Coin Change Problem: Variation 1

- Variation 1
  - You are given **n** number of different types of coins, coins[1], coins[2], coins[3],....., coins[n]
  - You can use any coin infinite number of times
  - You have to say, can you make the amount K ?

Coin Change Problem: Variation 1

- ## Recursive Approach

If k can be made output 1
else 0



Bottom Up Approach

Base cases
- amount can be made (k==0)
- all coin been used (i>n)

Complexity
- O(n * k)

Coin Change Problem: Variation 2

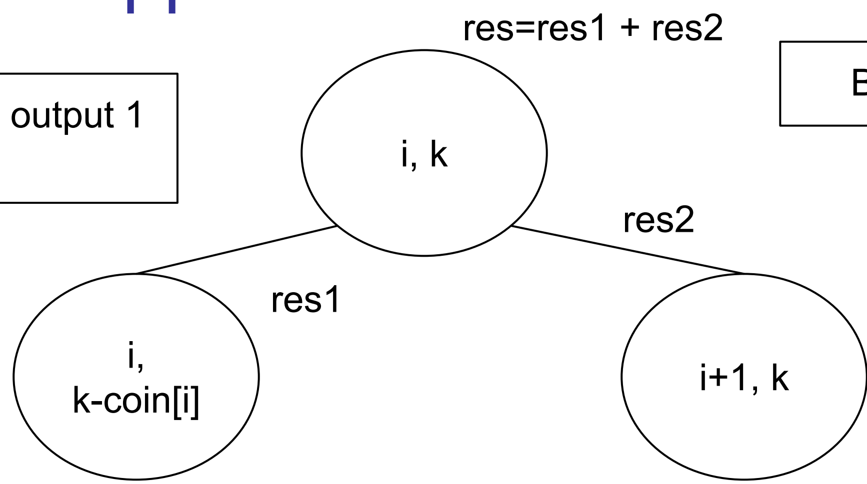
- Variation 2
  - You are given **n** number of different types of coins, coins[1], coins[2], coins[3],....., coins[n]
  - You can use any coin infinite number of times
  - You have to say, the number of ways to make the amount K ?

Coin Change Problem: Variation 2

- ## Recursive Approach

If k can be made output 1
else 0

Bottom Up Approach



Base cases
- amount can be made (k==0)
- all coin been used (i>n)

Complexity
- O(n * k)

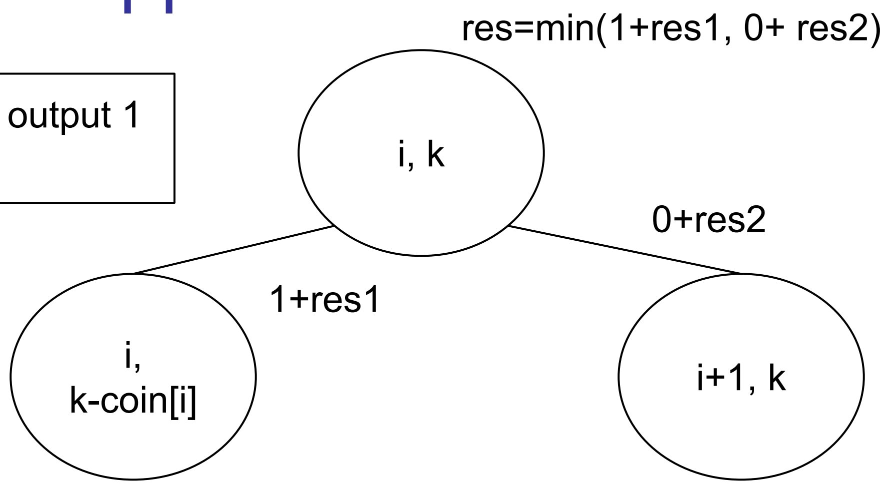
Coin Change Problem: Variation 3

- Variation 3
  - You are given **n** number of different types of coins, coins[1], coins[2], coins[3],....., coins[n]
  - You can use any coin infinite number of times
  - You have to say, the minimum number of coins to make the amount K ?

Coin Change Problem: Variation 3

- ## Recursive Approach

If k can be made output 1
else 0



Bottom Up Approach

Base cases
- amount can be made (k==0)
- all coin been used (i>n)

Complexity
- O(n * k)

Variation 3: Iterative Approach

```
P(i, k) = min {
      1+P(i,k+C_i),
      P(i+1,k)
}
starting: P(0,0)
Base: i>=n or k==K
```

```
P(i, k) = min {
      1+P(i,k-C_i),
      P(i-1,k)
}
starting: P(n-1,K)
Base: i<0 or k==0
```

```
For (i = 1; i<=n; i++) {
      For(k=1; k<=K; k++) {
            r1 = 1+P[i][k-C[i]]
            r2 = P[i-1][k]
            P[i][k] = min(r1, r2)
      }
}
```

base cases:

i/k	0	1	2	….	k
0	0	INF	INF	INF	INF
1	0				
..	0				
i	0				

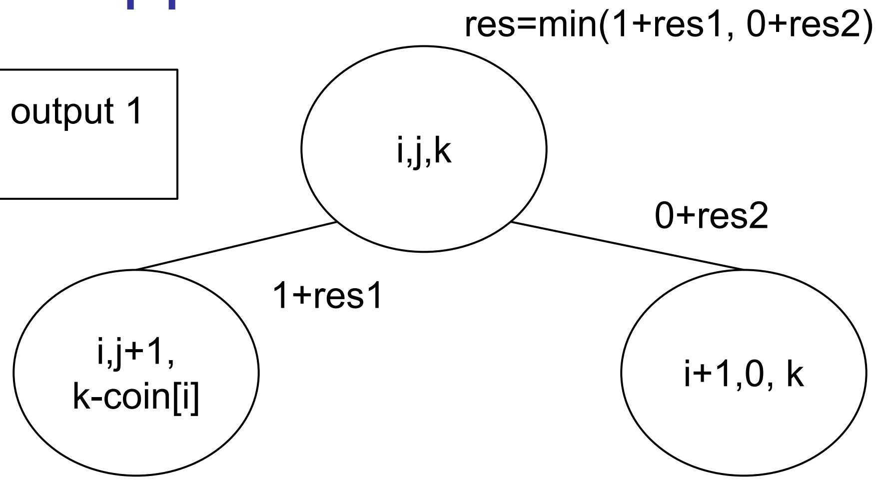
Coin Change Problem: Variation 4

- Variation 4
  - You are given **n** number of different types of coins, coins[1], coins[2], coins[3],....., coins[n]
  - Each coin[i] has an amount[i] which denotes the maximum number of times, that coin can be used
  - You have to say, the minimum number of coins to make the amount K ?
  - Mainly a constraint over Variation 3

Coin Change Problem: Variation 4

- ## Recursive Approach

If k can be made output 1
else 0



Bottom Up Approach

Base cases
- amount can be made (k==0)
- all coin been used (i>n)

Complexity
- O (n * m * k)

m = maximum number of coins given for a particular unit among all input

Variation 4: Iterative Approach

P(i,k) = min {
    j+P(i+1,k+C_i*j), // limited j
    P(i+1,k)
}
starting: P(0,0)
Base: i>=n or k==K

P(i, k) = min {
    j+P(i-1,k-C_i*j), // limited j
    P(i-1,k)
}
starting: P(n-1,K)
Base: i<0 or k==0

```
For (i = 1; i<=n; i++) {
    For(k=1; k<=K; k++) {
        For(j=1; j<=L[i];j++) {
            r1 = j+P[i-1][k-C[i]*j]
        }
        r2 = P[i-1][k]
        P[i][k] = min(r1, r2)
    }
}
```

base cases:

i/k	0	1	2	….	k
0	0	INF	INF	INF	INF
1	0				
..	0				
i	0				

Coin Change Problem: Path Print

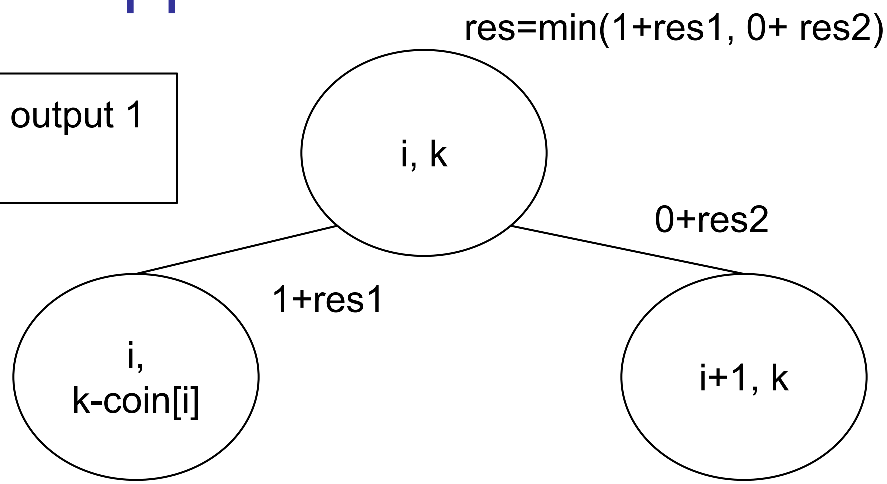
- A modification of variation 3
  - You are given **n** number of different types of coins,

    coins[1], coins[2], coins[3],....., coins[n]
  - You can use any coin infinite number of times
  - You have to say, the minimum number of coins to make the amount K ?
  - Also you need to print the coins how you took

Coin Change Problem: Path Print

- Recursive Approach

If k can be made output 1
else 0



next_state[i][k].first
next_state[i][k].second

Base cases
- amount can be made (k==0)
- all coin been used (i>n)

Complexity
- O(n * k)

Bottom Up Approach

The 0-1 Knapsack Problem

The Knapsack Problem

- **The 0-1 knapsack problem**
  - A thief robbing a store finds n items: the i-th item is worth v_i dollars and weights w_i pounds (v_i, w_i integers)
  - The thief can only carry W pounds in his knapsack
  - Items must be taken entirely or left behind
  - Which items should the thief take to maximize the value of his load?
- **The fractional knapsack problem**
  - Similar to above
  - The thief can take fractions of items

The 0-1 Knapsack Problem

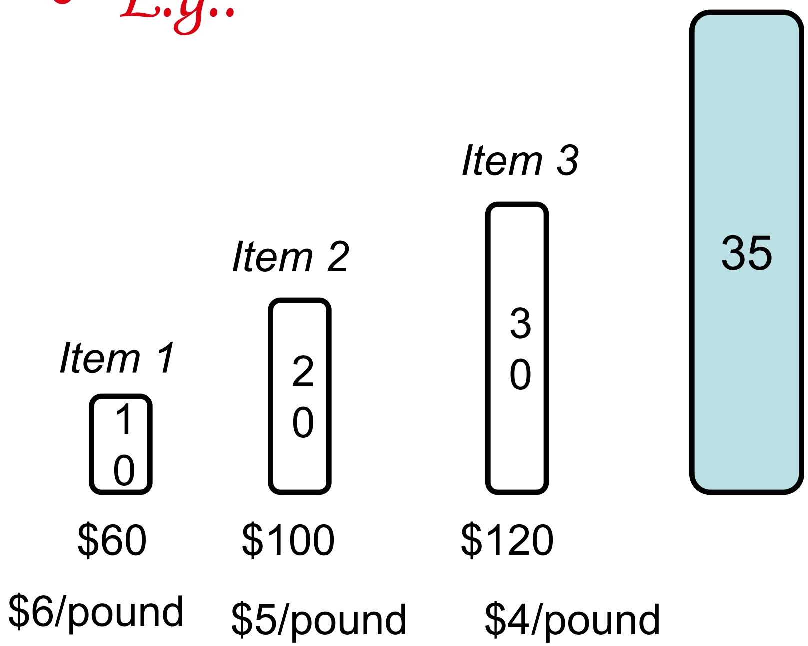
- Thief has a knapsack of capacity W
- There are n items: for i-th item value v_i and weight w_i
- Goal:
  - find x_i such that for all $x_i = \{0, 1\}$, $i = 1, 2, .., n$

    $\sum w_i x_i \leq W$ and

    $\sum x_i v_i$ is maximum

0-1 Knapsack - Greedy Strategy

- *E.g.:*



Greedy choice
- item 3 [Gives max money satisfying weight constraint] - Gain: $120

Optimal Choice
- item 1, 2 - Gain: $160

- Greedy Choice Does not give optimal solution

Two directional valid approaches

$P(i,w) = \text{Max}\ \{$
$\quad B_i + P(i+1, w+w_i)$
$\quad P(i+1, w)$
$\}$

Starting call: P(0,0)
Break: (i>=n or w == W)

Exactly same type of recursion, just considering it from two different perspectives/directions

This provides an idea of how we can write iterative version of this problem

$P(i,w) = \text{Max}\ \{$
$\quad B_i + P(i-1, w-w_i)$
$\quad P(i-1, w)$
$\}$

Starting call: P(n-1,W)
Break: (i<0 or w == 0)

for i^{th} calculation, we need to have all the calculation completed for $(i-1)^{th}$

P[i][w] = The maximum amount of positive benefit possible using coins upto i not violating knapsack weight w.

- The previous results have to be already calculated.

0-1 Knapsack - Dynamic Programming

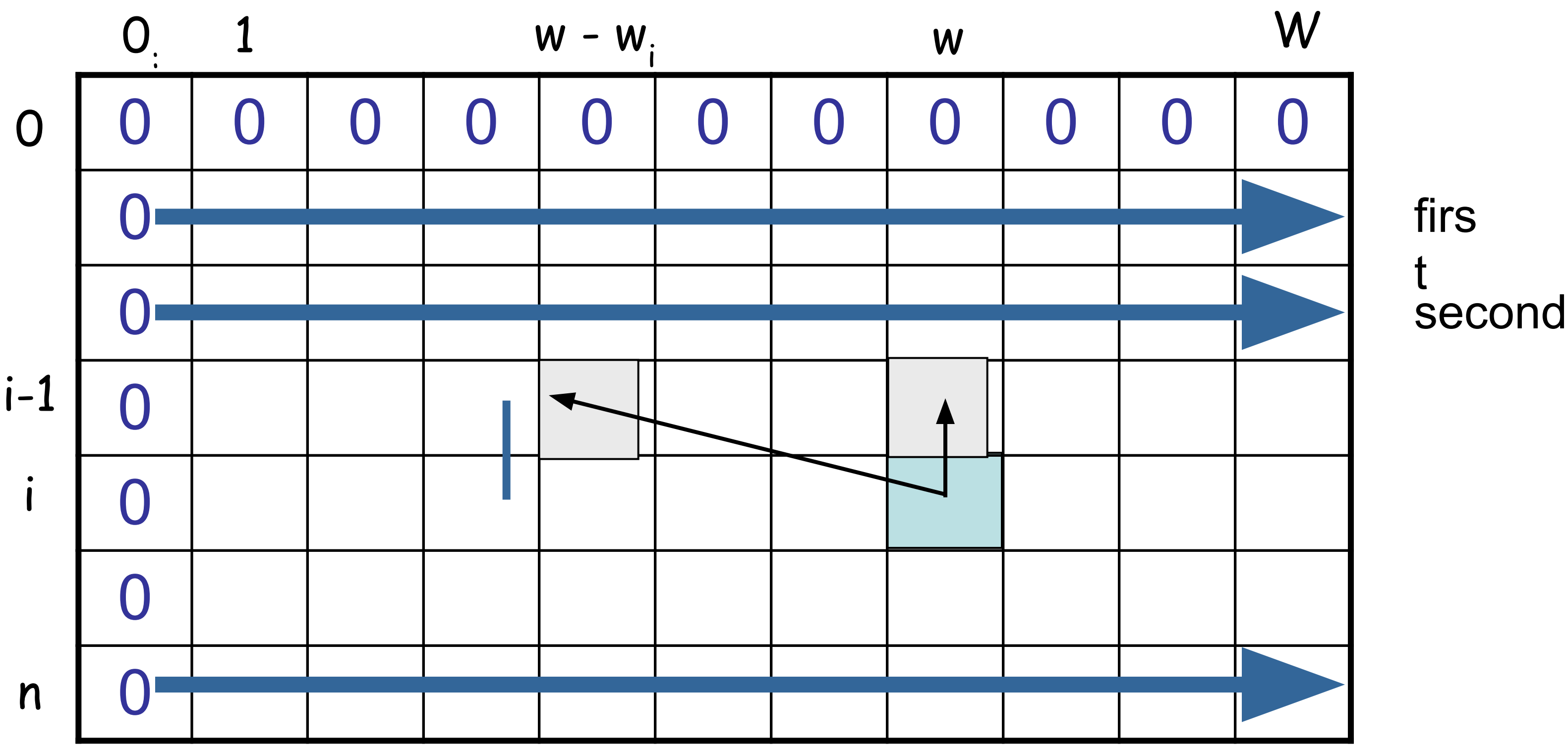
- $P(i, w)$ – the maximum profit that can be obtained from items 1 to i, if the knapsack has size w
- Case 1: thief takes item i

  $P(i, w) = v_i + P(i - 1, w-w_i)$
- Case 2: thief does not take item i

  $P(i, w) = P(i - 1, w)$

0-1 Knapsack - Dynamic Programming

$$P(i, w) = \max\{\underbrace{v_i + P(i-1, w-w_i)}_{\text{Item } i \text{ was taken}}, \underbrace{P(i-1, w)}_{\text{Item } i \text{ was not taken}}\}$$



Example:

W = 5

$$P(i, w) = \max\{v_i + P(i-1, w-w_i), P(i-1, w)\}$$

Item	Weight	Value
1	2	12
2	1	10
3	3	20
4	2	15

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

P(1, 1) = P(0, 1) = 0

P(1, 2) = max{12+0, 0} = 12

P(1, 3) = max{12+0, 0} = 12

P(1, 4) = max{12+0, 0} = 12

P(1, 5) = max{12+0, 0} = 12

P(2, 1)= max{10+0, 0} = 10

P(2, 2)= max{10+0, 12} = 12

P(2, 3)= max{10+12, 12} = 22

P(2, 4)= max{10+12, 12} = 22

P(2, 5)= max{10+12, 12} = 22

P(3, 1)= P(2,1) = 10

P(3, 2)= P(2,2) = 12

P(3, 3)= max{20+0, 22}=22

P(3, 4)= max{20+10,22}= 30

P(3, 5)= max{20+12,22}= 32

P(4, 1)= P(3,1) = 10

P(4, 2)= max{15+0, 12} = 15

P(4, 3)= max{15+10, 22}=25

P(4, 4)= max{15+12, 30}=30

P(4, 5)= max{15+22, 32}=37

Reconstructing the Optimal Solution

W = 5

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

Item	Weight	Value
1	2	12
2	1	10
3	3	20
4	2	15

- Start at $P(n, W)$
- When you go left-up ⇒ item i has been taken
- When you go straight up ⇒ item i has not been taken

Path Print

$P(i,w) = \text{Max}\ \{$

$\quad B_i + P(i-1,\ w-w_i)$

$\quad P(i-1,\ w)$

$\}$

Starting call: $P(n-1,W)$

Break: $(i<0 \text{ or } w == 0)$

$P[i][w] = \{$

- $P[i-1][w-w_i] == P[i][w] - b[i]$
- $P[i-1][w] == P[i][w]$

$\}$

- Can go to the cell where the value condition matches

W = 5

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

Item	Weight	Value
1	2	12
2	1	10
3	3	20
4	2	15

Overlapping Subproblems

$P(i, w) = \max \{v_i + P(i - 1, w - w_i), P(i - 1, w) \}$

	0	1				w					W
0	0	0	0	0	0	0	0	0	0	0	0
	0										
	0										
i-1	0										
i	0										
	0										
n	0										

E.g.: all the subproblems shown in grey may depend on $P(i-1, w)$

Longest Common Subsequence (LCS)

Longest Common Subsequence (LCS)

- Application: comparison of two DNA strings
- Ex: X= {A B C B D A B }, Y= {B D C A B A}
- Longest Common Subsequence:
- X = A **B** **C** **B** D **A** B
- Y = **B** D **C** A **B** **A**
- Brute force algorithm would compare each subsequence of X with the symbols in Y

Longest Common Subsequence

- Given two sequences

  $X = \langle x_1, x_2, \ldots, x_m \rangle$

  $Y = \langle y_1, y_2, \ldots, y_n \rangle$

  find a maximum length common subsequence (LCS) of X and Y

- *E.g.:*

  $X = \langle A, B, C, B, D, A, B \rangle$

- Subsequences of X:
  - A subset of elements in the sequence taken in order

  $\langle A, B, D \rangle$, $\langle B, C, D, B \rangle$, etc.

Example

S1=	A	B	C	B	D	A	B
S2=	B	D	C	A	B	A	

- ⟨B, C, B, A⟩ and ⟨B, D, A, B⟩ are longest common subsequences of X and Y (length = 4)
- ⟨B, C, A⟩, however is not a LCS of X and Y

Brute-Force Solution

- For every subsequence of X, check whether it’s a subsequence of Y
- There are 2^m subsequences of X to check
- Each subsequence takes $\Theta(n)$ time to check
  - scan Y for first letter, from there scan for second, and so on
- Running time: $\Theta(n2^m)$

LCS Algorithm

- First we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.
- Define X_i, Y_j to be the prefixes of X and Y of length i and j respectively
- Define $c[i,j]$ to be the length of LCS of X_i and Y_j
- Then the length of LCS of X and Y will be $c[m,n]$

$$c[i,j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

LCS recursive solution

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- We start with $i = j = 0$ (empty substrings of x and y)
- Since X_0 and Y_0 are empty strings, their LCS is always empty (i.e. $c[0,0] = 0$)
- LCS of empty string and any other string is empty, so for every i and j: $c[0, j] = c[i,0] = 0$

LCS recursive solution

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- When we calculate *c[i,j]*, we consider two cases:
- **First case:** *x[i]=y[j]*:
  - one more symbol in strings X and Y matches, so the length of LCS X_i and Y_j equals to the length of LCS of smaller strings X_{i-1} and Y_{i-1} , plus 1

LCS recursive solution

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i]=y[j], \\ \max(c[i,j-1],c[i-1,j]) & \text{otherwise} \end{cases}$$

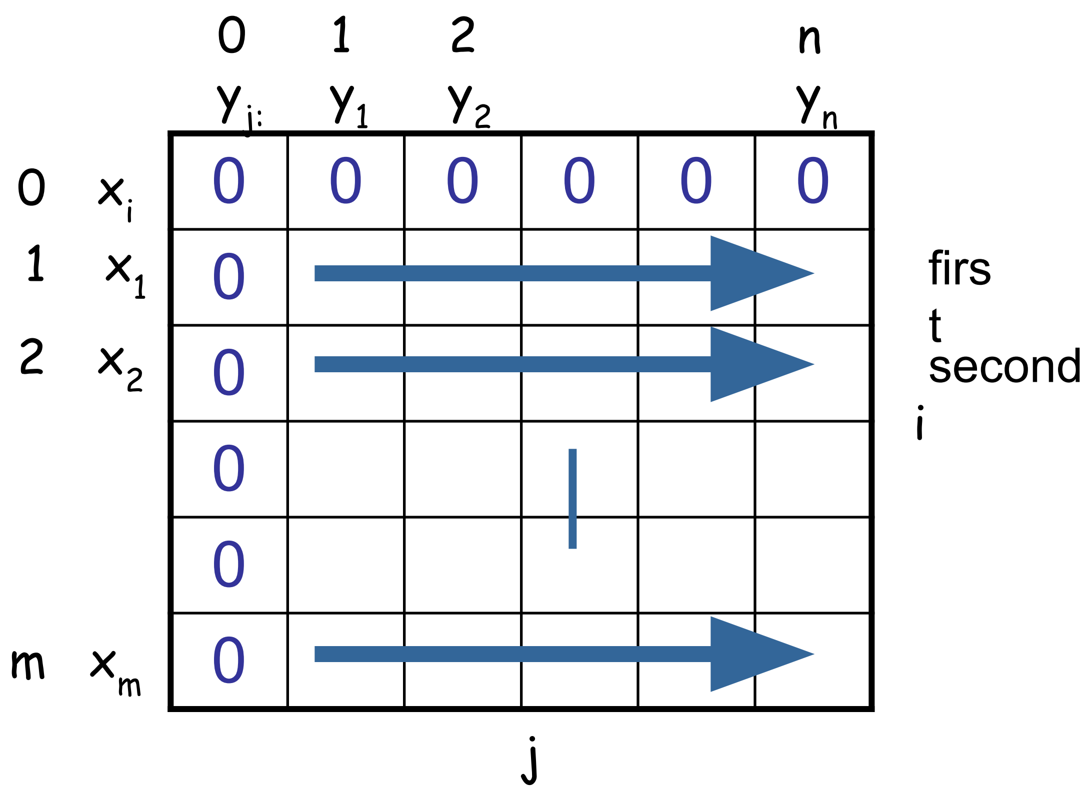
- **Second case:** *x[i] != y[j]*
  - As symbols don't match, our solution is not improved, and the length of LCS(X_i , Y_j) is the same as before (i.e. maximum of LCS(X_i, Y_{j-1}) and LCS(X_{i-1},Y_j)

Why not just take the length of LCS(X_{i-1}, Y_{j-1}) ?

3. Computing the Length of the LCS

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

Definition:
C[i, j] = Max matching possible between X[1,i] and Y[1, j]



Additional Information: Path Print

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

b & c:

		0	1	2	3		n
		y_j:	A	C	D		F
0	x_i	0	0	0	0	0	0
1	A	0					
2	B	0			c[i-1,j]		
3	C	0		↖ c[i,j-1]	↑		
		0					
m	D	0					

i

j

A matrix b[i, j]:

- For a subproblem [i, j] it tells us what choice was made to obtain the optimal value
- If $x_i = y_j$
  b[i, j] = " " ↖
- Else, if c[i - 1, j] ≥ c[i, j-1]
  b[i, j] = " ↑ "
  else
  b[i, j] = " ← "

LCS-LENGTH(X, Y, m, n)

```
for i ← 1 to m
    do c[i, 0] ← 0
for j ← 0 to n
    do c[0, j] ← 0
for i ← 1 to m
    do for j ← 1 to n
        do if x_i = y_j
            then c[i, j] ← c[i - 1, j - 1] + 1
                 b[i, j ] ← "   "
            else if c[i - 1, j] ≥ c[i, j - 1]
                then 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
```

The length of the LCS if one of the sequences is empty is zero

Case 1: $x_i = y_j$

Case 2: $x_i \neq y_j$

Running time: Θ (mn)

Example

$X = \langle A, B, C, B, D, A \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

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

If $x_i = y_j$

$b[i, j] = $ " ↖ "

Else if

c[i 1 1,

j] ≥ c[i, j-1]

$b[i, j] = $ " ↑ "

else

$b[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	↑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

4. Constructing a LCS

- Start at b[m, n] and follow the arrows
- When we encounter a "↖" in b[i, j] ⇒ $x_i = y_j$ is an element of the LCS

		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	↑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

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

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

Running time: $\Theta(m + n)$

Initial call: PRINT-LCS(b, X, length[X], length[Y])

Improving the Code

- If we only need the length of the LCS
  - LCS-LENGTH works only on two rows of c at a time
    - The row being computed and the previous row
  - We can reduce the asymptotic space requirements by storing only these two rows

LCS Algorithm Running Time

- LCS algorithm calculates the values of each entry of the array c[m,n]
- So what is the running time?

O(m*n)

since each c[i,j] is calculated in constant time, and there are m*n elements in the array

Longest Increasing Subsequence (LIS)

Longest increasing subsequence(LIS)

- The longest increasing subsequence is to find a longest increasing subsequence of a given sequence of distinct integers $a_1a_2\dots a_n$.

e.g. 9 2 5 3 7 11 8 10 13 6

2 3 7

5 7 10 13

are increasing subsequences.

We want to find a longest one.

9 7 11

3 5 11 13

are not increasing subsequences.

A naive approach for LIS

- Let **L[i]** be the length of a longest increasing subsequence ending at position *i*.

$$L[i] = 1 + \max_{j=0..i-1}\{L[j] \mid a_j < a_i\}$$

(use a dummy a_0 = minimum, and $L[0]=0$)

Index	0	1	2	3	4	5	6	7	8	9	10
Input	0	9	2	5	3	7	11	8	10	13	6
Length	0	1	1	2	2	3	4	4	5	6	3
Prev	-1	0	0	2	2	4	5	5	7	8	4
Path	1	1	1	1	1	2	2	2	2	2	2

The subsequence 2, 3, 7, 8, 10, 13 is a longest increasing subsequence.

This method runs in $O(n^2)$ time.

An $O(n \log n)$ method for LIS

- Lets given an array be A and an additional array be B.
- Main idea will be similar to insertion sort, where we will always try to keep the B sorted.
- Iterate over each element of A (A[i]), find the smallest value in B (B[j]) which is larger than current index of A. Replace B[j] with A[i]. The size of B will denote the length of LIS

Can omit inserting the repeating element if asked for strictly increasing subsequence (greedy choice).

An $O(n \log n)$ method for LIS

9	2	5	3	7	11	8	10	13	6

parent	itr	B	Links
-1	0	9	9
-1	1	2	2
1	2	2 5	(2 -> 5)
1	3	2 3	(2 -> 5), (2 -> 3)
3	4	2 3 7	(2 -> 5), (2 ->3->7)
4	5	2 3 7 11	(2 -> 5), (2->3->7->11)
4	6	2 3 7 8	(2 -> 5), (2->3->7->8)
6	7	2 3 7 8 10	(2 -> 5), (2->3->7->8->10)
7	8	2 3 7 8 10 13	(2 -> 5), (2->3->7->8->10->13)
3	9	2 3 6 8 10 13	(2 -> 5), (2->3->7->8->10->13), (2->3->6)

An $O(n \log n)$ method for LIS

A, B = { }

For(i = 0; i< n; i++) {

- j <- using binary search find index j where B[j] > A[i] and closest to it.
- Replace B[j] with A[i]
- if no such j is found, add A[i] to the end of B
- while inserting keeping track of the previous index makes the trails of LIS

}

Sum of Subset Problem

- Problem:
  - Suppose you are given N positive integer numbers A[1…N] and it is required to produce another number K using a subset of A[1..N] numbers. How can it be done using Dynamic programming approach?
- Example:

  N = 6, A[1..N] = {2, 5, 8, 12, 6, 14}, K = 19

  Result: 2 + 5 + 12 = 19

Maximum-sum interval

- Given a sequence of real numbers $a_1 a_2 \ldots a_n$, find a consecutive subsequence with the maximum sum.

9 −3 1 7 −15 2 3 −4 2 −7 6 −2 8 4 -9

For each position, we can compute the maximum-sum interval starting at that position in $O(n)$ time. Therefore, a naive algorithm runs in $O(n^2)$ time.

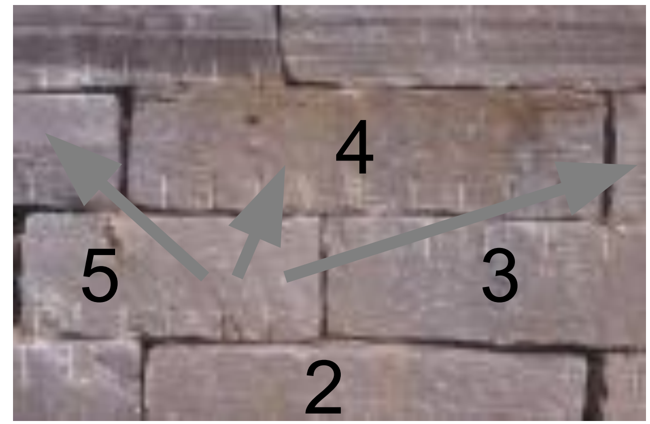
Try Yourself

Rock Climbing Problem

- A rock climber wants to get from the bottom of a rock to the top by the safest possible path.
- At every step, he reaches for handholds above him; some holds are safer than other.
- From every place, he can only reach a few nearest handholds.

Rock climbing (cont)

❖Suppose we have a wall instead of the rock.



At every step our climber can reach exactly three handholds: above, above and to the right and above and to the left.

There is a table of “danger ratings” provided. The “Danger” of a path is the sum of danger ratings of all handholds on the path.

Rock Climbing (cont)

- We represent the wall as a table.

- Every cell of the table contains the danger rating of the corresponding block.

2	8	9	5	8
4	4	6	2	3
5	7	5	6	1
3	2	5	4	8

The obvious greedy algorithm does not give an optimal solution.The rating of this path is 13.

The rating of an optimal path is 12.

However, we can solve this problem by a dynamic programming strategy in polynomial time.

Idea: once we know the rating of a path to every handhold on a layer, we can easily compute the ratings of the paths to the holds on the next layer.

For the top layer, that gives us an answer to the problem itself.

For every handhold, there is only one "path" rating. Once we have reached a hold, we don't need to know how we got there to move to the next level.

This is called an "optimal substructure" property. Once we know optimal solutions to subproblems, we can compute an optimal solution to the problem itself.

Recursive solution:

To find the best way to get to stone j in row i, check the cost of getting to the stones

- (i-1,j-1),
- (i-1,j) and
- (i-1,j+1), and take the cheapest.

Problem: each recursion level makes three calls for itself, making a total of 3^n calls – too much!

Solution - memorization

We query the value of A(i,j) over and over again.

Instead of computing it each time, we can compute it once, and remember the value.

A simple recurrence allows us to compute A(i,j) from values below.

Dynamic programming

- Step 1: Describe an array of values you want to compute.
- Step 2: Give a recurrence for computing later values from earlier (bottom-up).
- Step 3: Give a high-level program.
- Step 4: Show how to use values in the array to compute an optimal solution.

Rock climbing: step 1.

- *Step 1: Describe an array of values you want to compute.*

- For $1 \leq i \leq n$ and $1 \leq j \leq m$, define $A(i,j)$ to be the cumulative rating of the least dangerous path from the bottom to the hold (i,j).

- The rating of the best path to the top will be the minimal value in the last row of the array.

Rock climbing: step 2.

- *Step 2: Give a recurrence for computing later values from earlier (bottom-up).*

- Let $C(i,j)$ be the rating of the hold (i,j). There are three cases for $A(i,j)$:

- Left $(j=1)$: $C(i,j)+min\{A(i-1,j),A(i-1,j+1)\}$

- Right $(j=m)$: $C(i,j)+min\{A(i-1,j-1),A(i-1,j)\}$

- Middle: $C(i,j)+min\{A(i-1,j-1),A(i-1,j),A(i-1,j+1)\}$

- For the first row $(i=1)$, $A(i,j)=C(i,j)$.

Rock climbing: simpler step 2

- Add initialization row: $A(0,j)=0$. No danger to stand on the ground.
- Add two initialization columns: $A(i,0)=A(i,m+1)=\infty$. It is infinitely dangerous to try to hold on to the air where the wall ends.
- Now the recurrence becomes, for every i,j:

$$A(i,j) = C(i,j)+\min\{A(i-1,j-1),A(i-1,j),A(i-1,j+1)\}$$

Rock climbing: Iterative version

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

We started from bottom reached top, tried to find a path minimizing the danger

what if, we start from top reach bottom and minimize the danger

Solution will remain the same

Rock climbing: Iterative version

A[i,j] = Danger[i, j] + min(A[i-1, j], A[i-1, j], A[i-1, j+1])

A[i, j] = Minimum Danger value to reach here from the top row using valid transitions

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞						∞
2	∞						∞
3	∞						∞
4	∞						∞

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞						∞
2	∞						∞
3	∞						∞
4	∞						∞

Initialization: $A(i,0)=A(i,m+1)=\infty$, $A(0,j)=0$

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞						∞
3	∞						∞
4	∞						∞

The values in the first row are the same as C(i,j).

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7					∞
3	∞						∞
4	∞						∞

$A(2,1)=5+\min\{\infty,3,2\}=7$

.

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9				∞
3	∞						∞
4	∞						∞

$A(2,1)=5+\min\{\infty,3,2\}=7.$
$A(2,2)=7+\min\{3,2,5\}=9$

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7			∞
3	∞						∞
4	∞						∞

$A(2,1)=5+\min\{\infty,3,2\}=7.$
$A(2,2)=7+\min\{3,2,5\}=9$
$A(2,3)=5+\min\{2,5,4\}=7.$

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞						∞
4	∞						∞

The best cumulative rating on the second row is 5.

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞						∞

The best cumulative rating on the third row is 7.

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞	13	19	16	12	15	∞

The best cumulative rating on the last row is 12.

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞	13	19	16	12	15	∞

The best cumulative rating on the last row is 12.

So the rating of the best path to the top is 12.

Rock climbing example: step 4

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞	13	19	16	12	15	∞

To find the actual path we need to retrace backwards the decisions made during the calculation of A(i,j).

Rock climbing example: step 4

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	**5**	8

The last hold was (4,4).

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞	13	19	16	12	15	∞

To find the actual path we need to retrace backwards the decisions made during the calculation of A(i,j).

Rock climbing example: step 4

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	**2**	3
2	8	9	**5**	8

The hold before the last was (3,4), since min{13,7,8} was 7.

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞	13	19	16	12	15	∞

To find the actual path we need to retrace backwards the decisions made during the calculation of A(i,j).

Rock climbing example: step 4

C(i,j):

3	2	5	4	8
5	7	5	6	**1**
4	4	6	**2**	3
2	8	9	**5**	8

The hold before that was (2,5), since min{7,10,5} was 5.

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞	13	19	16	12	15	∞

To find the actual path we need to retrace backwards the decisions made during the calculation of A(i,j).

Rock climbing example: step 4

C(i,j):

3	2	5	**4**	8
5	7	5	6	**1**
4	4	6	**2**	3
2	8	9	**5**	8

Finally, the first hold was (1,4), since min{5,4,8} was 4.

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞	13	19	16	12	15	∞

To find the actual path we need to retrace backwards the decisions made during the calculation of A(i,j).

Rock climbing example: step 4

C(i,j):

3	2	5	**4**	8
5	7	5	6	**1**
4	4	6	**2**	3
2	8	9	**5**	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞	13	19	16	12	15	∞

We are done!

Printing out the solution recursively

```
PrintBest(A,i,j) // Printing the best path ending at (i,j)
  if (i==0) OR (j=0) OR (j=m+1)
    return;
  if (A[i-1,j-1]<=A[i-1,j]) AND (A[i-1,j-1]<=A[i-1,j+1])
    PrintBest(A,i-1,j-1);
  elseif (A[i-1,j]<=A[i-1,j-1]) AND (A[i-1,j]<=A[i-1,j+1])
    PrintBest(A,i-1,j);
  elseif (A[i-1,j+1]<=A[i-1,j-1]) AND (A[i-1,j+1]<=A[i-1,j])
    PrintBest(A,i-1,j+1);
  printf(i,j)
```

Matrix Chain Multiplication(MCM)

A = [10 x 30], B=[30 x 5], C=[5 x 40]

- Need to calculate (ABC) ? (AB)C or A(BC)
- # Operations = (AB)C = (10 * 5 * 30) + (10 * 40 * 5) = 1500 + 2000 = 3500
- # Operations = A(BC) = (30 * 5 * 40) + (10 * 30 * 40) = 6000 + 12000 = 18000
- so operation wise (AB)C <<< A(BC)

- So, the question is how to efficiently separate the calculation -> MCM teaches about this concept of partition

MCM

- Recursion
  - Given (A1, A2, A3, ……, AN) Matrixes
  - Divide [A1-Aj]+[Aj-AN]+Merge Results from these two partitions
  - Merge results = Row(A1) * Col(Aj) * Row(AN)
  - Find the j with best partition over trying All j within range

```
def Func(i,k) {
        res = INF
        for(j=i, j<=k;j++) {
                res=min(res,   Func(i,j) + Func(j+1,k) + merge(A[i],A[j],A[k])
        }
        return res
}
```

MCM: Iterative Approach

A[i, k] = A[i,j]+A[j+1, k] + Merge Between two segments

i = starting of segment
k = ending of segment
j = the breaking point of segment / length

MCM: Iterative Approach

- Calculate Base Case / for 1 length

A[i, k] = Minimum number of operations that will occur starting from i, ending at k

For any A[i,k] calculation, the needed values have to be calculated already

i/k	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

MCM: Iterative Approach

- ## Calculate for 2 length

A[i, k] = Minimum number of operations that will occur starting from i, ending at k

A[1,2] = R(matrix[1]) * C(matrix[1]) * C(matrix[2])
A[2,3] = R(matrix[2]) * C(matrix[3]) * C(matrix[3])

i/k	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

MCM: Iterative Approach

- ## Calculate for 3 length

A[i, k] = Minimum number of operations that will occur starting from i, ending at k

A[1,3] = MIN {
A[1,2]+A[3,3]+ R(matrix[1]) * C(matrix[2]) * C(matrix[3]),
A[1,1]+A[2,3]+ R(matrix[1]) * C(matrix[1]) * C(matrix[3])
}

i/k	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

MCM: Iterative Approach

- Calculate for 4 length

A[i, k] = Minimum number of operations that will occur starting from i, ending at k

A[1,4] = MIN {

A[1,1]+A[2,4]+ R(matrix[1]) * C(matrix[1]) * C(matrix[4]) ,
A[1,2]+A[3,4]+ R(matrix[1]) * C(matrix[2]) * C(matrix[4]),
A[1,3]+A[4,4]+ R(matrix[1]) * C(matrix[3]) * C(matrix[4])
}

i/k	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

MCM: Iterative Approach

```
For (i = 1; i<=n; i++) A[i,i] = 0
```

```
For (length = 2; length<=n; length++) {
	for(i = 1; i<=n-1; i++) {
		k = i+length-1
		res = INF
		for(int mid=1; mid<length; mid++) {
			j = i+mid-1
			res = min(res, A[i, j] + A[j+1, k] + R(M[i]) * C(M[j]) * C(M[k]) )
		}
		DP[i,k] = res
}
```

Important Points to Ask

- Are we trying with every options ?
- We can never make loops during calculation, A->B->A will never give solution
- Identify the states, the combination of possibilities, can never leave one