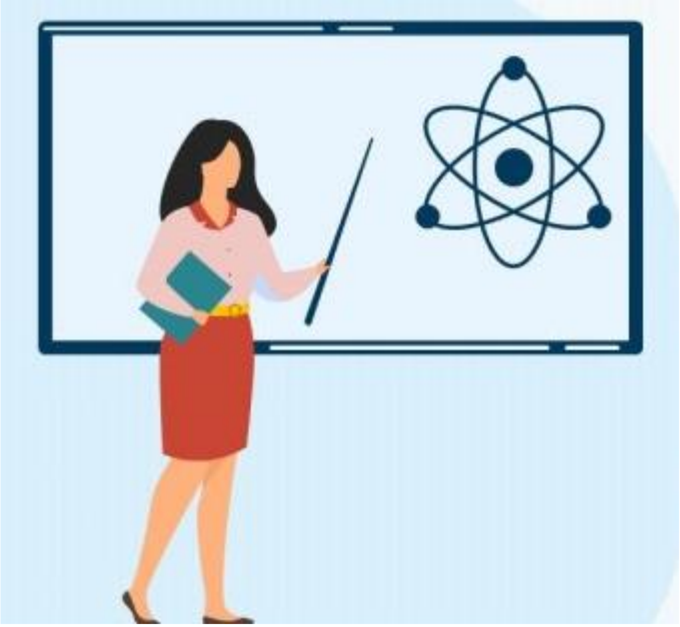
Dynamic Programming



Suman Pandey

Agenda

- What is dynamic Programing
  - Principle of Optimality
- Difference between Dynamic Programming and Greedy Method
- Ex. Fibonacci
  - Recursive
  - Dynamic Programming
    - Memoization
    - Tabulation
- Ex: 0/1 Knapsack Problem
  - Recursive
  - Dynamic Programming
    - Tabulation
- Ex: Matrix Chain Multiplication
  - Recursive
  - Dynamic Programming
    - Tabulation
- Ex: Longest Common Subsequence
  - Recursion
  - Dynamic Programming
    - Memoization
    - Tabulation

Dynamic Programming

- Dynamic Programming is used for solving **Optimization problem**
  - **Maximize** something
  - **Minimize** something
- **Optimization** Problem can be solved using
  - **Greedy**
  - **Dynamic Programming**
- It should contain **overlapping subproblems**
- DP follows **Principle of Optimality**
  - Which means, problem can be solved by taking a **sequence of decision**
  - **For ex:** Shall I include this particular item or not? Shall I include next item or not ? ……………… … so on an so forth.
  - Usually we take decisions from last object towards first object
- In Dynamic Programming, you should try all possible solutions and then pickup the best solution.
  - In **linear** approach considering all solution will take too much time. (exponential time complexity)
  - Dynamic programming **reduces** the time complexity
- There are two approach for Dynamic Programming
  - **Memoization (Top Down)**
  - **Tabulation (Bottom Up)**

Divide and Conquer Vs Dynamic Programming

- Similarity
  - Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems.
- Difference
  - divide-and-conquer algorithms partition the problem into **disjoint subproblems**, solve the subproblems recursively, and then combine their solutions to solve the original problem. So in Divide-and-conquer it will never happen that you solve the same problem again.
  - In contrast, dynamic programming applies when the **subproblems overlap**—that is, when subproblems share sub subproblems.
  - A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.

Greedy Vs Dynamic Algorithms

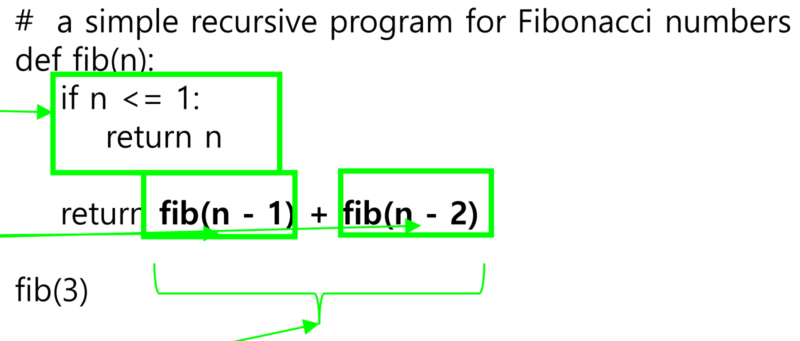
- Both Dynamic Programming and Greedy are used to solve Optimization Problems
- **Greedy**
  - Greedy deals with forming the solution step by step by choosing the local optimum at each step and finally reaching a global optimum.
  - Greedy does not deal with multiple possible solutions, its just builds the one solution that it believes to be correct
  - Greedy believes that choosing local optimum at each stage will lead to form the global optimum
- **DP**
  - DP does not deal with such uncertain assumptions
  - DP finds a solution to all sub problems and chooses the best ones to form the local optimum
  - DP guarantees the correct answer each and every time whereas Greedy is not.
  - DP is much slower than Greedy, Greedy deals with only one subproblem, however DP deals with all the sub problems.
  - DP works only when there is **overlapping subproblems**.
- How to choose between Greedy and DP – We will discuss after covering both the topics

Ex: Fibonacci

Fib(0) – 0
Fib(1) – 1
Fib(2) – 0 + 1 = 1
Fib(3) – 1 + 1 = 2
Fib(4) – 2 + 1 = 3
Fib(5) – 3 + 2 = 5
Fib(6) – 3 + 5 = 8
.
.
.
Fib(n) = Fib(n-1) + Fib(n-2)

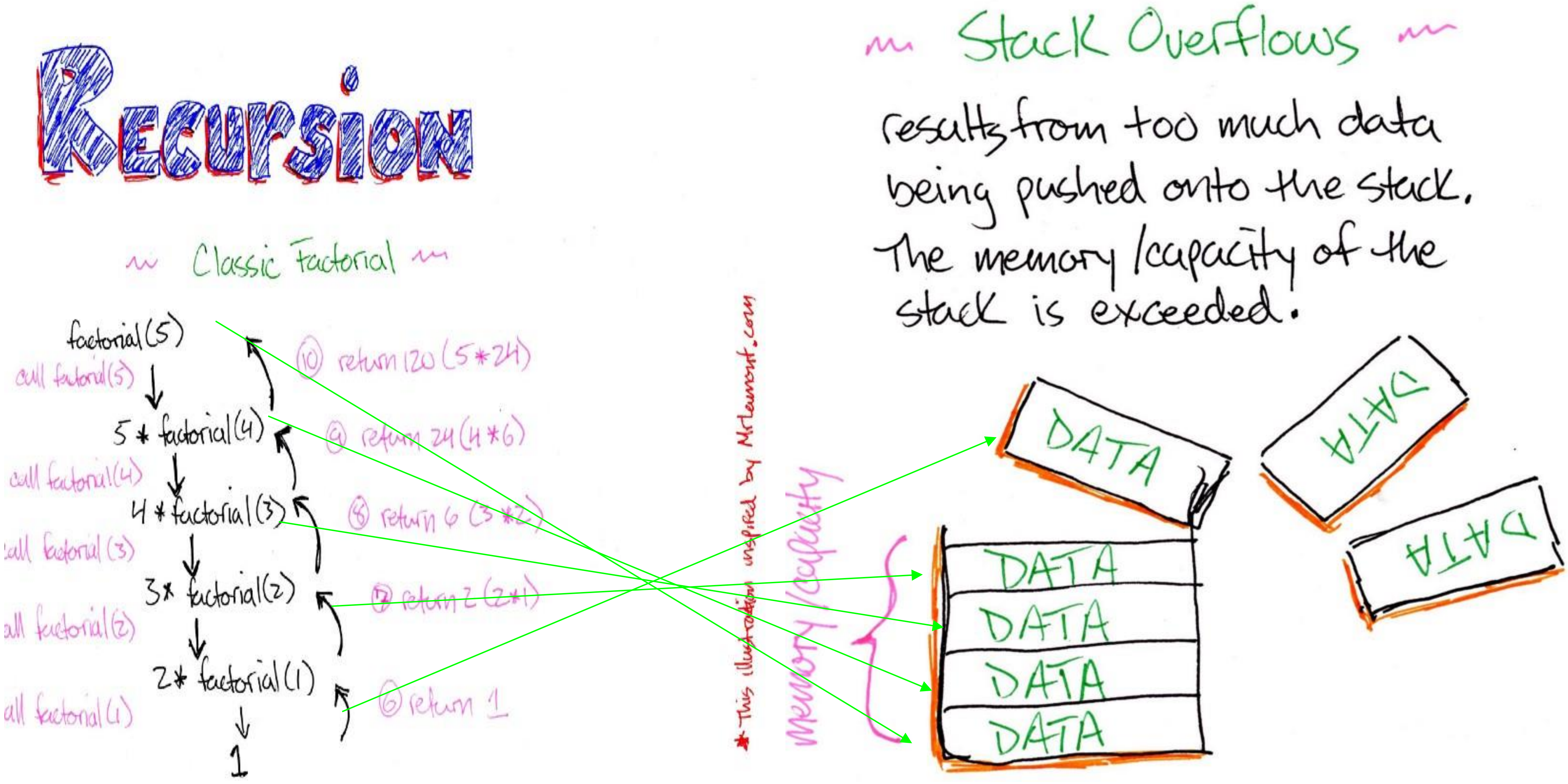
Recursive Functions

- This about the function int fib(n)
- Base case
  - Based on smallest valid input
- Decreasing function
  - Every time the recursive function will be called for smaller value than previous
- Choice Diagram
  - This is the main code
  - What to do with the returned results of the reduced sub problems ?
  - Ex:
    - We multiply call the recursive function (Factorial)
    - We add the results of several recursion (Fibonacci)
    - Find minimum or maximum (Optimization problems)
    - Find the partition (Quick Sort)
    - Merge results (Merge sort)



```
#  a simple recursive program for Fibonacci numbers
def fib(n):
    if n <= 1:
        return n

    return fib(n - 1) + fib(n - 2)

fib(3)
```

Problem with Recursive functions – Stack Overflow



Stack Overflows

results from too much data being pushed onto the stack. The memory /capacity of the stack is exceeded.

Ex: Fibonacci (**Overlapping Sub-problems**)

- Dynamic Programming is mainly used when solutions of the same subproblems are needed again and again.
  - solutions to subproblems are stored in a table (so that it is not required to be recomputed)
  - Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again.

```
#  a simple recursive program for Fibonacci numbers
def fib(n):
    if n <= 1:
        return n

    return fib(n - 1) + fib(n - 2)

fib(3)
```

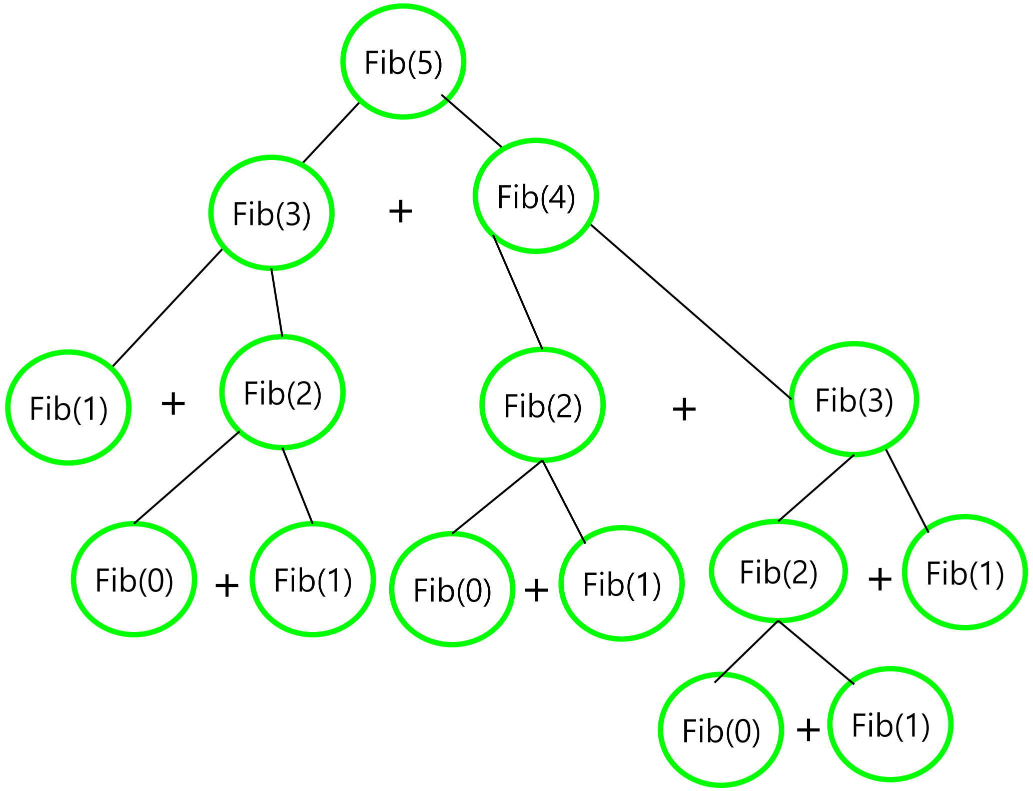
Overlapping sub problem

Fib(0) – 0
Fib(1) – 1
Fib(2) – 0 + 1 = 1
Fib(3) – 1 + 1 = 2
Fib(4) – 2 + 1 = 3
Fib(5) – 3 + 2 = 5
.
.
.
.
Fib(n) – fib(n-1) + fib(n-2)

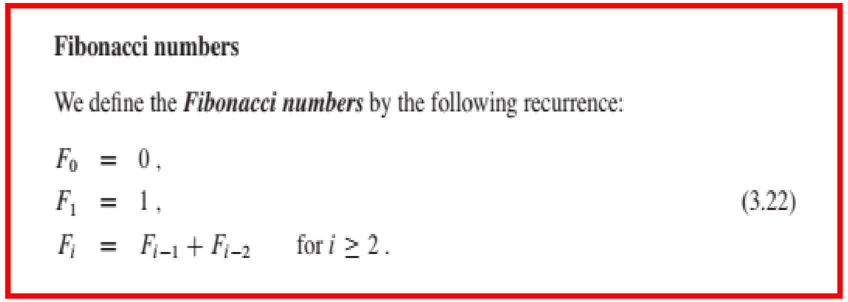
Ex: Fibonacci (**Overlapping Sub-problems**)

```
#  a simple recursive program for Fibonacci numbers
def fib(n):
    if n <= 1:
        return n  # for 0 and 1 direct result is returned
     return fib(n - 1) + fib(n - 2)
fib(3)
```

Recursion tree for Fib(5)



Fib(0) – 0
Fib(1) – 1
Fib(2) – 0 + 1 = 1
Fib(3) – 1 + 1 = 2
Fib(4) – 2 + 1 = 3
Fib(5) – 3 + 2 = 5
.
.
.
Fib(n) – fib(n-1) + fib(n-2)



Fibonacci numbers

We define the ***Fibonacci numbers*** by the following recurrence:

$$
\begin{aligned}
F_0 &= 0\,, \\
F_1 &= 1\,, \qquad\qquad\qquad\qquad (3.22)\\
F_i &= F_{i-1} + F_{i-2} \quad \text{for } i \geq 2\,.
\end{aligned}
$$

Fib(3) is called two times

Fib(2) is called three times

Instead of computing it again, we could reuse the old stored value. There are following two different ways to store the values so that these values can be reused:

a) Memoization (Top Down)

b) Tabulation (Bottom Up)

Ex: Fibonacci (**Memoization**)

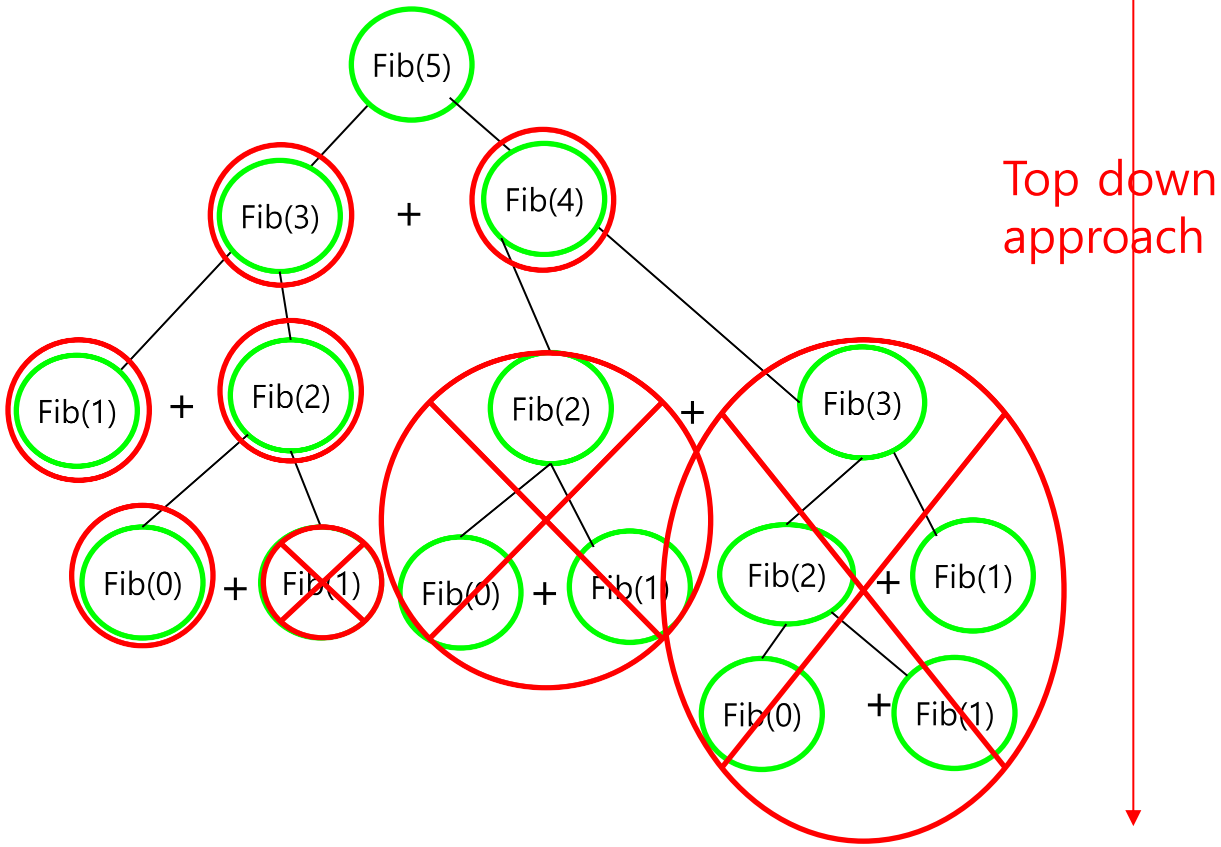
It’s a **top down** approach

We **initialize** a **lookup array** with all initial values as NIL.

Whenever we need the solution to a subproblem, we **first** look into the lookup table. If the precomputed value is there then we return that value.

otherwise, we calculate the value and put the result in the lookup table so that it can be reused later.

Recursion tree for Fib(5)



Lookup array initialization

-1	-1	-1	-1	-1	-1
0	1	2	3	4	5

Once we call Fib(0), Fib(1), Fib(2).... Fib(5) it shouldn’t be called again, so store the value in lookup table

Keep storing the function value as we encounter the function

0	1	1	2	4	5
0	1	2	3	4	5

This way we avoided repetitive function calls. Instead of 15, we only made 6 function calls.

Ex: Fibonacci (**Memoization**)

Simple recursive

```
#  a simple recursive program for Fibonacci numbers
def fib(n):
    if n <= 1:
        return n  # for 0 and 1 direct result is returned
     return fib(n - 1) + fib(n - 2)
fib(3)
```

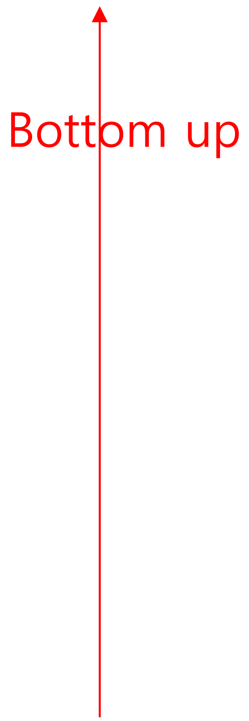
Memoization

```
# a program for Memorized version of nth Fibonacci number
# function to calculate nth Fibonacci number
def fib(n, lookup):

        # base case
        if n <= 1 :
                lookup[n] = n

        # if the value is not calculated previously then calculate it
        if lookup[n] is None:
                lookup[n] = fib(n-1 , lookup) + fib(n-2 , lookup)

        # return the value corresponding to that value of n
        return lookup[n]
# end of function

# Driver program to test the above function
def main():
        n = 34
        # Declaration of lookup table
        # Handles till n = 100
        lookup = [None] * 101
        print ("Fibonacci Number is ", fib(n, lookup))

if __name__=="__main__":
        main()
```

Ex: Fibonacci (**Tabulation**)

- Tabulation is **Iterative** approach, which means it wont use the recursion, rather use **loops**.
- It's a **bottom up** approach, and returns the last entry from the **table**.
- Ex: we first calculate fib(0) then fib(1) then fib(2) then fib(3), and so on. So literally, we are building the solutions of subproblems bottom-up.

This is **Tabulation approach** to solve the Dynamic Programming problems



```
# Python program Tabulated (bottom up) version
def fib(n):
    # array declaration with 0
    f = [0] * (n + 1)
    # base case assignment
    f[1] = 1

    # calculating the Fibonacci and storing the values
    for i in range(2 , n + 1):
        f[i] = f[i - 1] + f[i - 2]
    return f[n]

# Driver program to test the above function
def main():
    n = 9
    print ("Fibonacci number is " , fib(n))
if __name__=="__main__":
    main()
```

Fibonacci : **Time Complexity analysis**

Simple recursive

```
#  a simple recursive program for Fibonacci numbers
def fib(n):
    if n <= 1:
        return n  # for 0 and 1 direct result is returned
     return fib(n - 1) + fib(n - 2)
fib(3)
```

$T(n) = T(n-1) + T(n-2) + 1$

$O(2^n)$

tabulation

```
def fib(n):
                # array declaration with 0
                f = [0] * (n + 1)
                # base case assignment
                f[1] = 1

                # calculating the Fibonacci and storing the values
                for i in range(2 , n + 1):
                                f[i] = f[i - 1] + f[i - 2]
                return f[n]

if __name__=="__main__":
                n = 9
                print ("Fibonacci number is " , fib(n))
```

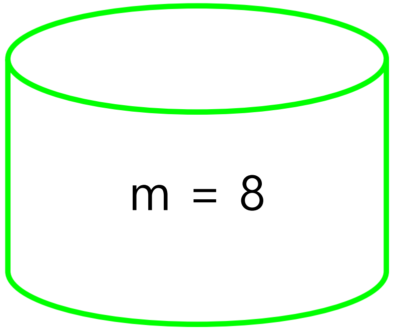
O(n)

Ex: 0/1 Knapsack Problem

There are 4 objects
n = 4

For each object you have some profit and weight
P = {1, 2, 5, 6}
W = {2, 3, 4, 5}

There is a bag of capacity 8
m = 8



Max $\sum_{o}^{i} p\,x$

Min $\sum_{o}^{i} w\,x \leq m$

Goal is to fill the bag such that, total profit is **maximized**
Output $x_{0/1}$ = { 0 , 1, 0, 1 } if the item is included 1 and not included 0

Problems that are similar to 0/1 Knapsack Problem

- Subset sum
- Equal sum partition
- Count of the subset sum
- Minimum subset sum difference
- Target Sum
- Number of subsets by given difference

Note: if you learn the 0/1 knapsack problem, then with minor changes you can solve all these 6 problems.

0/1 Knapsack Problem

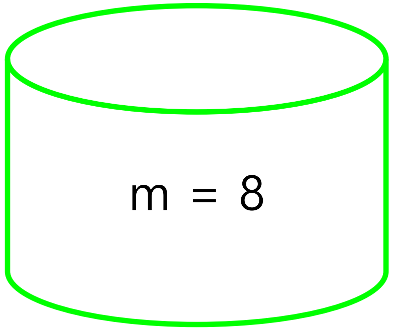
For each object you have some profit and weight

$n = 4$
$m = 8$
$P = \{1, 2, 5, 6\}$
$W = \{2, 3, 4, 5\}$

Input

$x_{0/1} = \{ 0 , 1, ., . \}$

Output



There could be many possible solution to this

$x_{0/1} = \{ 0 , 0, 0, 0 \}$ -> no object is included

$x_{0/1} = \{ 1 , 1, 1, 1 \}$ -> all objects are included

$x_{0/1} = \{ 1 , 0, 0, 0 \}$ -> only one object is included

$x_{0/1} = \{ 0 , 0, 0, 1 \}$ -> only one object is included

.

.

Try all of them and pick up one

Total how many solutions could be there ?

$= 2^4$

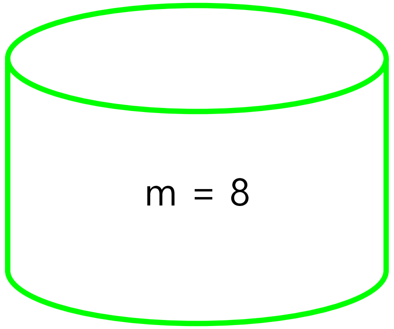
For n objects we will have 2^n combinations

DP characteristics in 0/1 Knapsack Problem

- First identify if the problem is a DP problem by asking these questions
  - **Optimization problem ?**
    - **Ans : Yes**
    - **Maximization** of profit
  - **Principle of Optimality ?**
    - **Ans: Yes**
    - **sequence of decision,** Include or do not include an item
- DP => Recursion => memoization => Tabulation
- It is the best to first solve a DP problem using recursion

There are 4 objects
$n = 4$

For each object you have some profit and weight
$P = \{1, 2, 5, 6\}$
$W = \{2, 3, 4, 5\}$



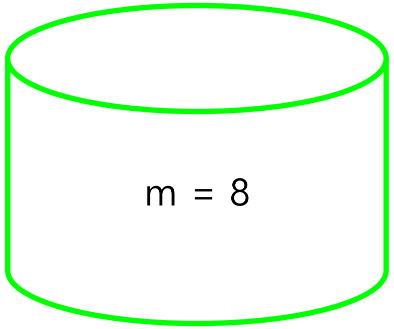
Max $\sum_{o}^{i} p\, x$

Variations of Knapsack

- Fractional Knapsack
- 0/1 knapsack
- Unbounded Knapsack

There are 4 objects
n = 4

For each object you have some profit and weight
P = {1, 2, 5, 6}
W = {2, 3, 4, 5}



Max $\sum_{o}^{i} p\,x$

Constraints $\sum_{o}^{i} w\,x \leq m$

Goal is to fill the bag such that, total profit is **maximized**

Fractional Knapsack Output – You can include fraction of an object too.
Ex: If your bag has 1 kg left, you can fill is with the fraction of any object and get the profit also in fraction – **Greedy approach**

0/1 Knapsack – You can either include the object entirely or you just don't include it.
Ex: if your bag has 1 kg left, and you don't have any item of 1 kg, then you just leave that space empty

Unbounded knapsack – You can add multiple occurrence of the same object.
Ex: you can have item 4 in the bag multiple times. If having them multiple times give high profit

0/1 Knapsack Problem

Tabulation

0/1 Knapsack (Tabulation)

1. What will be the size of the matrix ?

Matrix will be of **n X m** size (depends on the values that are changing in algorithm)

- int V[n+1] [m +1]
- This matrix will store the profit earned by different sub problems

For each object you have some profit and weight

P = {1, 2, 5, 6}

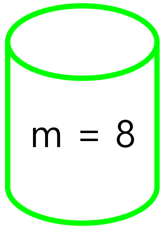
W = {2, 3, 4, 5}

Max $\sum_{o}^{i} p\,x$

There are 4 objects

n = 4

m = 8



V

p	w		0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
1	2	1	0								
2	3	2	0				▨				
5	4	3	0							▨	
6	5	4	0								▨

This index will contain the answer

2. What is Sub-problem here ?

- At Index V[2, 4]
- Subproblem is P = {1, 2 } W ={ 2, 3 } m =4

- At Index V[3, 7]
- Subproblem is P = {1, 2,5 } W ={ 2, 3, 4 } m = 7

- At Index V[4, 8]
- Subproblem is P = {1, 2,5, 6 } W ={ 2, 3, 4, 5 } m = 8

Fill the Profit Table for different combinations of object

n = 4
m = 8

p = {1, 2, 5, 6}
w = {2, 3, 4, 5}

Initialization phase

When capacity of bag is 0 (boundary condition, means m=0)

When no object is included

object

profit

capacity

p	w		0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
1	2	1	0								
2	3	2	0								
5	4	3	0								
6	5	4	0								

We will take each object at a time and calculate the profit if it is included

- At 1st row, we consider object 1
- At 2nd row, we consider object 2 and object above that – (2 and 1)
- .
- .
- At 4th row, we consider object 4 and object above that – (4,3,2,1)

This way all combinations will be covered.

Filling the Profit Table

p	w		0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
6	5	4	0								

- At 1st row, we consider object 1
  - What is the weight of object 1 -> 2
  - So it can be filled only when bag capacity is 2
  - What is the profit ? -> 1, so fill 1 in index [1,2]
  - At other columns capacity of the bag is increasing but at this point we are considering only one object, hence the profit will be 1 for all other columns also
- At the 2nd row, we consider object 2 and 1
  - Weight of the object 2 -> 3
  - So it can be filled only when bag capacity is 3
  - What is the profit ? -> 2, so fill 2 in index [2,3]
  - Left side of index [2,3] will be same as rows above.
  - Right side, at index [2,5] two objects 1 and 2 can be filled, so fill profit 2+1 at index [2,5]
  - All the further column will also be 3. and index [2,4] will be 2
- At the 3rd row, we consider object 3, 2, and 1
  - Weight of the object 3 -> 4
  - So it can be filled only when bag capacity is 4, that is index [3,4], fill 5 at [3,4]
  - Left side of index [3,4] all will be same as row above
  - Right side, at index [3,5] –> 4 (3), [3,6] -> 6 (3,1), [3,7] ->7(3,2), [3,8] -> 8(3,2)

Fill the Profit Table for different combinations of object

n = 4
m = 8

p = {1, 2, 5, 6}
w = {2, 3, 4, 5}

- At 4st row, we consider object 1
  - Weight of the object 4 -> 5
  - So it can be filled only when bag capacity is 5, that is index [4,5], fill 6 at [4,5]
  - Left side of index [4,5] all will be same as row above
    - At index [4,4] we can only include object 4 that has profit 5 so filled 5
    - At index [4,3] we can only include object 3 that has profit 2 so filled 2
    - ..
    - ..
    - ..
  - Right side
    - At index [4,6] –> 6 (include object 4)
    - At index [4,7] -> 7 (include object 4 and 1)
    - At index [4,8] -> 8 (include object 4 and 2)

Filling the Table

p	w		0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
6	5	4	0	0	1	2	5	6	6	7	8

We can Fill the Profit Table using a formula also

n = 4
m = 8

p = {1, 2, 5, 6}
w = {2, 3, 4, 5}

➢ At 4st row, we will fill with the **formula**

IF we call this table as **V**
i – row = 4st row
j – column = 1 ~ 8

Weight of ith object (5)

V[i,j] = max{ V[i-1, j], V[i-1, j – w[i]] + p[i] }

There is no such index as -4, so
At this point knapsack m < w[i]

~~V[4,1] =max { V [3, 1], V[3, 1 – 5] + 6 } - > max {V [3, 1], V[3, -4] + 6 } -> V [3, 1] -> 0~~

~~V[4,2] =max { V [3, 2], V[3, 2 – 5] + 6 } - > max {V [3, 2], V[3, -3] + 6 } -> V [3, 2] -> 1~~

~~V[4,3] =max { V [3, 3], V[3, 3 – 5] + 6 } - > max {V [3, 3], V[3, -2] + 6 } -> V [3, 3] -> 2~~

~~V[4,4] =max { V [3, 4], V[3, 4 – 5] + 6 } - > max {V [3, 4], V[3, -1] + 6 } -> V [3, 4] -> 5~~

V[4,5] =max { V [3, 5], V[3, 5 – 5] + 6 } - > max {V [3, 5], V[3, 0] + 6 } -> max{ 5 , 6 } -> 6

V[4,6] =max { V [3, 6], V[3, 6 – 5] + 6 } - > max {V [3, 6], V[3, 1] + 6 } -> max{ 6 , 6 } -> 6

V[4,7] =max { V [3, 7], V[3, 7 – 5] + 6 } - > max {V [3, 7], V[3, 2] + 6 } -> max{ 7 , 7 } -> 7

V[4,8] =max { V [3, 8], V[3, 8 – 5] + 6 } - > max {V [3, 8], V[3, 3] + 6 } -> max{ 7 , 8 } -> 8

Filling the Table

p	w		0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
6	5	4	0	0	1	2	5	6	6	7	8

We have already found that if avail able container capacity is 6, then w e can fill it best with previous three objects in this way **V[i-1, j]**

We have already found that if available c ontainer capacity is 5, then we can fill it b est with this profit - **6**. Then how about checking the rest of the weight available and how can we fill the rest of the weight best
Rest of the weight available **– j-w[i]**
How can we fill it best - **V[i-1, j – w[i]]**

If maximum comes from here **V[i-1, j – w[i]] + p[i]** , then we can say that we need to include this 4th object as well.

Python code (0/1 Knapsack Problem)

```
# a dynamic approach
# Returns the maximum value that can be stored by the bag
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W + 1)] for x in range(n + 1)]
    print (K)
    #Table in bottom up manner
    for i in range(n + 1):
        for w in range(W + 1):
            if i == 0 or w == 0:
                K[i][w] = 0
            elif wt[i-1] <= w:
                K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w])
            else:
                K[i][w] = K[i-1][w]
    print (K)
    return K[n][W]
#Main
val = [1,2,5,6]
wt = [2,3,4,5]
W = 8
n = len(val)
print(knapSack(W, wt, val, n))
```

Backtracking to find the items included in Optimum result

n = 4
m = 8

p = {1, 2, 5, 6}
w = {2, 3, 4, 5}

Filling the Table

p	w		0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
6	5	4	0	0	1	2	5	6	6	7	8

x_1, x_2, x_3, x_4

- We have to take sequence of decision to decide which Item to include to maximize profit
- To find out that we needed to build the data, now we have the data ready
- We also know maximum profit is 8
- We now have to derive how did you come to this maximum profit, which all items we included

- To find that Algorithmically
  - We start with last two, last column [4,8] -> profit is 8
    - Check if 8 is there in previous rows.
    - Not found in previous rows which means 4th this profit is calculated including 4th o bject

      x_1, x_2, x_3, x_4
      _ _ _ 1
    - Profit of 4th object is 6, remain profit -> 8 - 6 = 2
  - Remaining profit is 2
    - Check if 2 is there in previous row, row for 3rd object, to find out if 3rd object is incl uded to make this profit
    - 2 is found in 3rd row, but its also found in the row above, which means this profit is not because 3rd object

      x_1, x_2, x_3, x_4
      _ _ 0 1
  - Remaining profit is still 2
    - Check if 2 is there in 2nd row, to find out if 2nd object is included to make this profit
    - 2 is found in 2nd row, and its not found in the row above that, which 2nd object was included in the profit.

      x_1, x_2, x_3, x_4
      _ 1 0 1
    - Profit of the 2nd object is 2, remain profit 2 – 2 =0
  - Remaining profit is 0
    - Check if 0 is there in 1st row, to find out if 1st object is included to make this profit
    - 0 is found in 1st row, and its also found in row above, which means this profit is not because of 1st object, so don't include first object

      x_1, x_2, x_3, x_4
      0 1 0 1

0/1 Knapsack Problem

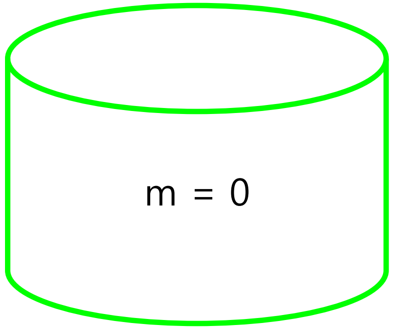
Recursion

0/1 Knapsack (Recursive)

- Lets first write the function
- Base case
  - Think of the smallest valid input
- How to decrease the inputs
- Choice Diagram

There are 4 objects
n = 4

For each object you have some profit and weight
P = {3, 6, 5, 6}
W = {0, 3, 4, 5}



Max $\sum_o^i p\, x$

int knapsack (int W[], int P[], int m, int n)

P = {1, 2, 5, 6} – n element – smallest valid **n** could be **0**
W = {2, 3, 4, 5}
m : - m weight – smallest valid **m** could be **0**

```
if ( n== 0 || m == 0)
        return 0
```

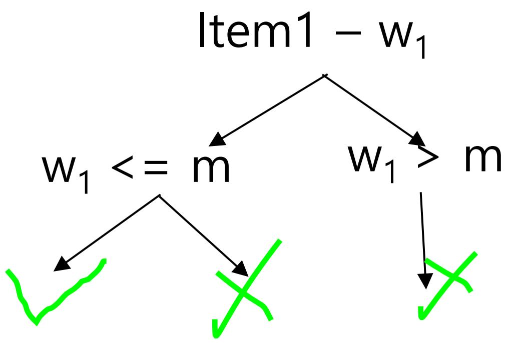
P = {1, 2, 5, 6}   We will check if we want to include this in our choice or not, and remove it from the list
W = {2, 3, 4, 5}   this is to reduce the input size

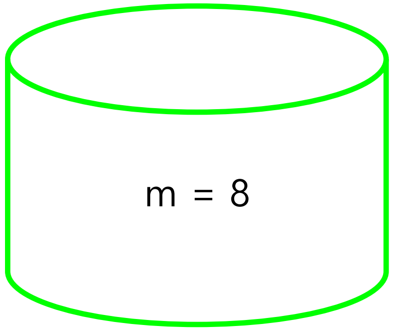
0/1 Knapsack (Recursive)

- Lets first write the function
- Base case
  - Think of the smallest valid input
- How to decrease the inputs
- **Choice Diagram**

There are 4 objects
n = 4

For each object you have some profit and weight
P = {1, 2, 5, 6}
W = {0, 3, 4, 5}




m = 8

Max $\sum_{o}^{i} p\,x$

int knapsack (int W[], int P[], int m, int n)

```
if ( n== 0 || m == 0)
        return 0
```

```
if ( w[n-1] <= m )
        # if we include
        # we will earn the price and reduce the total available capacity
        # now we will have to choose from the rest of n-1 elements
        #  p[n-1]  + knapsack ( w, p, m – w[n-1] , n-1 )
        # if we don't include
        # we will not earn any price and we wont reduce the available capacity
        # knapsack ( w, p, m,n-1)
        # we need to return maximum of above two options
        return max( p[n-1]  + knapsack ( w, p, m – w[n-1] , n-1 ), knapsack ( w, p, m,n-1) )
elseif ( w[n-1] > m )
        return knapsack ( w, p, m,n-1)
```

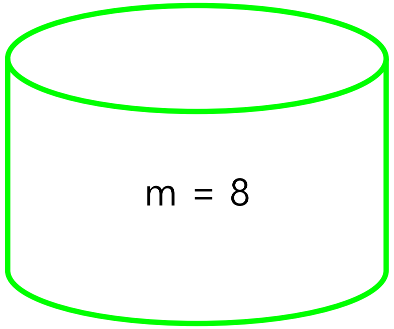
n
P = {1, 2, 5, 6}
W = {2, 3, 4, 5}

0/1 Knapsack (Recursive)

- Lets first write the function
- Base case
  - Think of the smallest valid input
- How to decrease the inputs
- Choice Diagram

There are 4 objects
n = 4

For each object you have some profit and weight
P = {1, 2, 5, 6}
W = {2, 3, 4, 5}



Max $\sum_{o}^{i} p\, x$

```
int knapsack (int w[], int P[], int m, int n)

        if ( n== 0 || m == 0)
                return 0

        if ( w[n-1] <= m )
                return max( p[n-1]  + knapsack ( w, p, m – w[n-1] , n-1 ), knapsack ( w, p, m,n-1) )
        elseif ( w[n-1] >m )
                return knapsack ( w, p, m,n-1)
```

Choice when we are including the item

Choice when we are not including the item

0/1 Knapsack Problem Converting A Recursive code to => Memoization

Hot to convert code from Recursive to Memoization ?

0/1 Knapsack (Memoization)

- What kind of table? How will you decide that ?
  - Ans: You need to see which values are changing.

    in this example n is changing (n-1) and m is changing m – p[n-1]

    So Matrix will be of **n X m** size
  - int V[n+1] [m +1]

For each object you have some profit and weight

P = {1, 2, 5, 6}

W = {2, 3, 4, 5}

m = 8

Max $\sum_{o}^{i} p\, x$

```
int V[100][100]
for i in range 100 :
  for j in range 100:
              V[i][j]=-1
```

#1. Change -> initialization of matrix

# this 100 and 100 can be based on your initial condition, m<100 , n<100

# instead of global, you can take this matrix as a static variable inside of the function as well

V

p	w		0	1	2	3	4	5	6	7	8
		0	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	2	1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	3	2	-1	-1	-1	-1	-1	-1	-1	-1	-1
5	4	3	-1	-1	-1	-1	-1	-1	-1	-1	-1
6	5	4	-1	-1	-1	-1	-1	-1	-1	-1	-1

```
int  knapsack (int w[], int P[], int m, int n)

        if ( n== 0 || m == 0)
          return 0
        if V[n][m] != -1 :
           return V[n][m]
        if ( w[n-1] <= m )
                    return  V[n][m]  =   max( p[n-1]  + knapsack ( w, p, m – w[n-1] , n-1 ), knapsack ( w, p, m,n-1) )
        elseif ( w[n-1] >m )
                    return  V[n][m]  =   knapsack ( w, p, m,n-1)
```

#2. Change -> if the value esist return it

#3. Store -> if the value does not exist store it

0/1 Knapsack Problem
Converting A Recursive code to
=> Tabulation

Hot to convert code from Recursive to Tabular ?

0/1 Knapsack (Recursion to Tabulation)

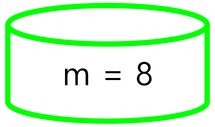
- What will be the size of table? How will you decide that ?
  - Ans: You need to see which values are changing.
    in this example n is changing (n-1) and m is changing m – p[n-1]
    So Matrix will be of **n X m** size
  - int V[n+1] [m +1]

- How to initialize this table ?
  - Base Condition -> initialization in tabulation
  - The base condition of recursive function will change to initialization in tabulation
  - If return of base condition is 0, fill all the row n = 0 and column m = 0 with 0

For each object you have some profit and weight
P = {1, 2, 5, 6}
W = {2, 3, 4, 5}

Max $\sum_{o}^{i} p\, x$

There are 4 objects
n = 4
m = 8


m = 8

V

p	w		0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
1	2	1	0								
2	3	2	0								
5	4	3	0								
6	5	4	0								

```
int  knapsack (int w[], int P[], int m, int n)
        if ( n== 0 || m == 0)
                return 0
        if ( w[n-1] <= m )
                return max( p[n-1]  + knapsack ( w, p, m – w[n-1] , n-1 ), knapsack ( w, p, m,n-1) )
        elseif ( w[n-1] > m  )
                return knapsack ( w, p, m,n-1)
```

0/1 Knapsack (Recursion to Tabulation)

Recursion

```
if ( n== 0 || m == 0)
        return 0
```

```
if ( w[n-1] <= m )
        return max( p[n-1]  + knapsack ( w, p, m – w[n-1] , n-1 ),
                    knapsack ( w, p, m,n-1) )
elseif ( w[n-1] >m )
        return knapsack ( w, p, m,n-1)
```

```
def knapSack(W, wt, val, n):
   K = [[0 for x in range(W + 1)] for x in range(n + 1)]
   for i in range(n + 1):
      for w in range(W + 1):
         if i == 0 or w == 0:
            K[i][w] = 0
         elif wt[i-1] <= w:
            K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w])
         else:
            K[i][w] = K[i-1][w]
   print (K)
   return K[n][W]
```

Tabulation V[n+1][m+1]

```
for i in range n+1          # for looping i and j are introduced
   for j in range m+1
     if ( i == 0 || j == 0 )
             V[i][j]=0
```

```
 if ( w[n-1] <= m )
    V[n][m] = max ( p[ n-1] + V [n-1] [ m – w[n-1]],
                    V[ n-1][m] )
Else
   V[n][m] = V[n-1][m]
```

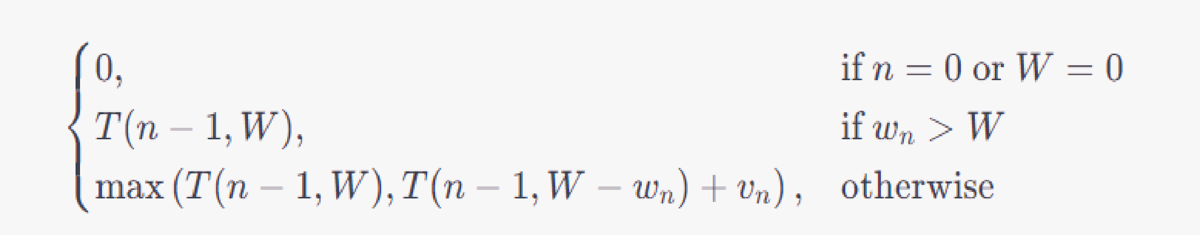
Change n to i and m to j in iterative version

```
for i in  range n+1
   for j in range m+1
           if ( w[n-1] <= m )
              V[i][j] = max ( p[ i-1] + V [i-1] [ j – w[i-1]],
                              V[ i-1][j] )
           else
             V[i][j] = V[i-1][j]
```

0/1 Knapsack : Time Complexity

Simple recursive

```
int  knapsack (int w[], int P[], int m, int n)

          if ( n== 0 || m == 0)
                    return 0

          if ( w[n-1] <= m )
                    return max(
                    p[n-1]  + knapsack ( w, p, m – w[n-1] , n-1 ),
                    knapsack ( w, p, m,n-1)
                    )
          elseif ( w[n-1] > m  )
                    return knapsack ( w, p, m,n-1)
```



$$\begin{cases} 0, & \text{if } n = 0 \text{ or } W = 0 \\ T(n-1, W), & \text{if } w_n > W \\ \max\left(T(n-1, W), T(n-1, W - w_n) + v_n\right), & \text{otherwise} \end{cases}$$

***O*(2^n)**

tabulation

```
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W + 1)] for x in range(n + 1)]
    for i in range(n + 1):
        for w in range(W + 1):
            if i == 0 or w == 0:
                K[i][w] = 0
            elif wt[i-1] <= w:
                K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w])
            else:
                K[i][w] = K[i-1][w]
    print (K)
    return K[n][W]
```

***O*(*nW*)**

Unbounded Knapsack Problem

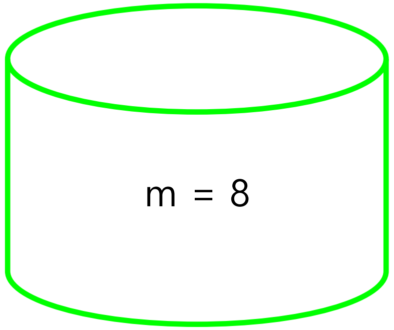
Max $\sum_{o}^{i} p\, x$

Constraints $\sum_{o}^{i} w\, x \le m$

For each object you have some profit and weight
P = {1, 2, 5, 6}
W = {2, 3, 4, 5}

0/1 Knapsack – You can either include the object entirely or you just don’t include it.
Ex: if your bag has 1 kg left, and you don’t have any item of 1 kg, then you just leave that space empty

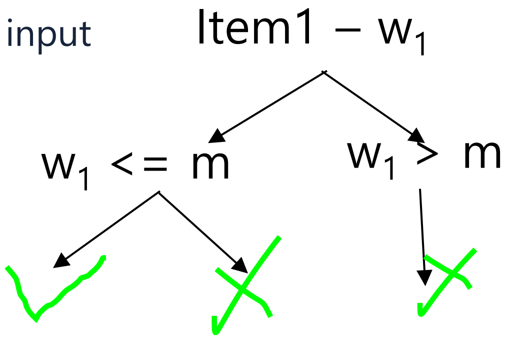
Unbounded knapsack – You can add multiple occurrence of the same object.
Ex: you can have item 4 in the bag multiple times. If having them multiple times give high profit



Unbounded Knapsack (Recursive)

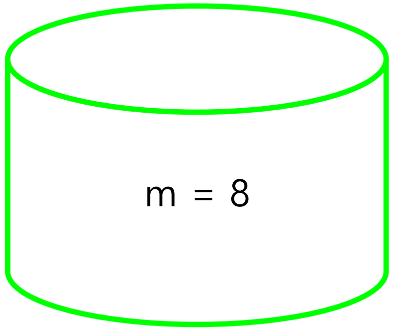
- Lets first write the function
- Base case
  - Think of the smallest valid input
- How to decrease the inputs
- **Choice Diagram**

But, you don't remove this item from consideration
You keep the possibility of taking this item again



There are 4 objects
n = 4

For each object you have some profit and weight
P = {1, 2, 5, 6}
W = {0, 3, 4, 5}



Max $\sum_o^i p\, x$

int knapsack (int W[], int P[], int m, int n)

```
if ( n== 0 || m == 0)
        return 0
```

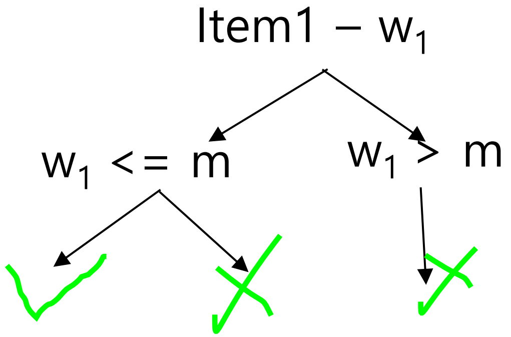
```
if ( w[n-1] <= m )
        # if we include
        # we will earn the price and reduce the total available capacity
        # However, unlike 0/1 knapsack, in unbounded knapsack, we can take this item again. So we don't reduce n.
        #  p[n-1]  + knapsack ( w, p, m – w[n-1] , n-1 )
        # if we don't include
        # we will not earn any price and we wont reduce the available capacity, however we will reduce n as we decided not to take it
        # knapsack ( w, p, m,n-1)
        # we need to return maximum of above two options
        return max( p[n-1]  + knapsack ( w, p, m – w[n-1] , n ), knapsack ( w, p, m,n-1) )
elseif ( w[n-1] > m )
        return knapsack ( w, p, m,n-1)
```

n
P = {1, 2, 5, 6}
W = {2, 3, 4, 5}

Comparing Unbounded Knapsack to 0/1 Knapsack Recursive Code

0/1 Knapsack

```
def knapSack01( l, val, W, n):
    if n == 0 or W == 0:
        return 0

    if (l[n-1] <= W):
        return max(
            val[n-1] + knapSack01( l, val, W-l[n-1], n-1),
            knapSack01(l, val, W,  n-1))
    else:
        return knapSack01( l, val, W, n-1)
```



Item1 – w_1

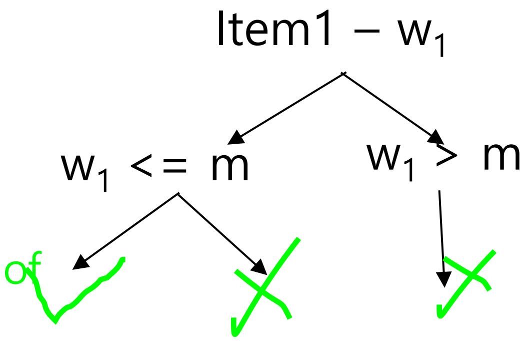
$w_1 <= m$

$w_1 > m$

✓ ✗ ✗

Unbounded Knapsack

```
def unbondedknapSack( l, val, W, n):
    if n == 0 or W == 0:
        return 0

    if (l[n-1] <= W):
        return max(
            val[n-1] + unbondedknapSack( l, val, W-l[n-1], n),
            unbondedknapSack(l, val, W,  n-1))
    else:
        return unbondedknapSack( l, val, W, n-1)
```



Item1 – w_1

$w_1 <= m$

$w_1 > m$

✓ ✗ ✗

You keep the possibility of taking this item again

Unbounded Knapsack (Recursion to Tabulation)

Recursion

```
def unbondedknapSack( l, val, W, n):
    if n == 0 or W == 0:
        return 0

    if (l[n-1] <= W):
        return max(
             val[n-1] + unbondedknapSack( l, val, W-l[n-1], n),
             unbondedknapSack(l, val, W,  n-1))
    else:
        return unbondedknapSack( l, val, W, n-1)
```

Tabulation V[n+1][m+1]

```
def unboundedKnapSackDP(wt, val, W, n):
    K = [[0 for x in range(W + 1)] for x in range(n + 1)]
    for i in range(n + 1):
        for w in range(W + 1):
            if i == 0 or w == 0:
                K[i][w] = 0
            elif wt[i-1] <= w:
                K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w])
            else:
                K[i][w] = K[i-1][w]

    return K[n][W]
```

- Once, you have written the recursive code. You can simply convert it into DP
- As W and n are changing they will define the table’s row and column
- Base Case can be converted to initialization
- Recursive function call will help you deduce the formula to fill the table.

Ex: Matrix Chain Multiplication

1. What is Matrix Multiplication
2. What is Matrix Chain Multiplication
3. Recursion
3. Formula using DP
4. How to use DP formula
5. Example Problem

A1	.	A2	.	A3	.	A4	Matrix Product
5 x 4		4 x 6		6 x 2		2 x 7	Dimensions of matrix

What is Matrix Multiplication

$$\overset{A}{\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}} * \overset{B}{\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix}} = \overset{C}{\begin{bmatrix} a_{11} * b_{11} + a_{12} * b_{21} + a_{13} * b_{31} & a_{11} * b_{12} + a_{12} * b_{22} + a_{13} * b_{32} \\ a_{21} * b_{11} + a_{22} * b_{21} + a_{23} * b_{31} & a_{21} * b_{12} + a_{22} * b_{22} + a_{23} * b_{32} \end{bmatrix}}$$

2 x 3 3 x 2 → 2 x 2 = 4 elements

No of columns of first matrix should be same as no of rows of second in order to multiply these matrix

We can do A * B But we cant do B * A (4 x 3 cant be multiplied to 5 x 4, however 5 x 4 can be multiplied by 4 x3)
So if A * B is possible doesn’t mean B * A is also possible

For getting each element we multiplied 3 times
2 rows of A to 2 columns of B

To get 4 element we multiplied 2 x 2 x 3 = 12 multiplication -> **Cost of multiplication**

What is Matrix Chain Multiplication

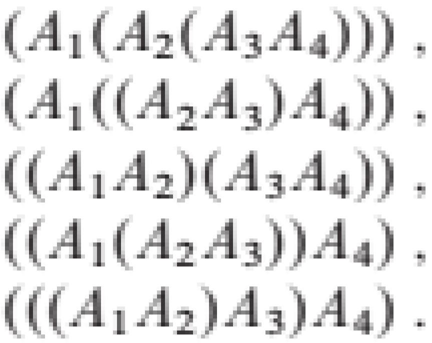
A_1	.	A_2	.	A_3	.	A_4	Matrix Product
5 x 4		4 x 6		6 x 2		2 x 7	Dimensions of matrix
d0 d1		d1 d2		d2 d3		d3 d4	

Can we multiply this Matrix
Ans: Yes: because columns of A_1 is same as rows of A_2 , Columns of A_2 is same as rows of A_3, Columns of A_3 is same as rows of A_4

How can we multiply ?
Ans : Multiply a pair at a time

Which pair should I select such that the total cost of multiplication can be minimized ?

This means , How we should parenthesize it, To reduce the total number of scalar multiplications?



$(A_1(A_2(A_3A_4)))$,
$(A_1((A_2A_3)A_4))$,
$((A_1A_2)(A_3A_4))$,
$((A_1(A_2A_3))A_4)$,
$(((A_1A_2)A_3)A_4)$.

This can be parenthesize in 5 ways

This is an **optimization Problem**

1. We must **minimize the cost**
2. We have to take a **sequence of decision**.

Cost of First option

You will be just given d0, d1, d2, d3 -> these 4 values for a matrix of size 3

A1	x	A2	x	A3
2 3		3 4		4 2
d0 d1		d1 d2		d2 d3

If you multiply in this way A1 x (A2 x A3)

How many multi for A1xA2 and A3 A1(0) A2xA3 (3 x 4 x 2)

Cost of multiplication MCM(1,1)=0 MCM(2,3) =24

Resulting matrix 2 3 3 2

How many multi resulting 2 x 3 x 2

Total cost = 3 x 4 x 2 + 2 x 3 x 2

Total cost of multiplication = MCM(1,1)+ MCM(2,3) + d0 xd1 xd3

i k k+1,j

Lets device the formula - **MCM(i,k) + MCM(k+1,j) + di-1 x dk x dj = 36**

Cost of Second option

You will be just given d0, d1, d2, d3 -> these 4 values for a matrix of size 3

A1 x	A2 x	A3
2 3	3 4	4 2
d0 d1	d1 d2	d2 d3

If you multiply in this way	(A1 x A2) x	A3
How many multi for A1xA2 and A3	(2 x 3 x 4)	A3(0)
Cost of multiplication	MCM(1,2)=24	MCM(3,3) =0
Resulting matrix	2 4	4 2
How many multi resulting	2 x 4 x 2	
Total cost =	2 x 3 x 4 + 2 x 4 x 2	

Total cost of multiplication = MCM(1,2)+ MCM(3,3) + d0 xd2 xd3
i k k+1,j

Lets device the formula - **MCM(i,k) + MCM(k+1,j) + di-1 x dk x dj = 40**

Input to Matrix Multiplication

- Input: p[] = {40, 20, 30, 10, 30}
- Output: 26000

- Input: p[] = {5, 4, 6, 2, 7}
- Output : 158

Explanation: There are 4 matrices of dimensions 40x20, 20x30, 30x10 and 10x30. Let the input 4 matrices be A, B, C and D. The minimum number of multiplications are obtained by putting parenthesis in following way (A(BC))D --> 20*30*10 + 40*20*10 + 40*10*30

Ex: Matrix Chain Multiplication (Recursion)

A1	.	A2	.	A3	.	A4	Matrix Product
5 x 4		4 x 6		6 x 2		2 x 7	Dimensions of matrix

Matrix Multiplication idea development

arr:	5	4	6	2	7
	0	1	2	3	4

A_1 -> 5 x 4
A_2 -> 4 x 6
A_3 -> 6 x 2
A_4 -> 2 x 7

A_1 -> arr[0] x arr[1]
A_2 -> arr[1] x arr[2]
A_3 -> arr[2] x arr[3]
A_4 -> arr[3] x arr[4]

A_i -> arr[i-1] x arr[i]

$$(A_1(A_2(A_3A_4)))\,,$$
$$(A_1((A_2A_3)A_4))\,,$$
$$((A_1A_2)(A_3A_4))\,,$$
$$((A_1(A_2A_3))A_4)\,,$$
$$(((A_1A_2)A_3)A_4)\,.$$

A_1 . A_2 . A_3 . A_4
5 x 4 4 x 6 6 x 2 2 x 7

- ➢ We have to decide at what point we will break this

Shall we put the bracket
A_1 (A_2 . A_3 . A_4
or
A_1 . A_2 (A_3 . A_4
or
A_1 . A_2 . A_3 (A_4

- ➢ If we choose to put the bracket at first index

A_1 (A_2 . A_3 . A_4

Now for the rest of the matrix we need to decide again

A_1 (A_2 (A_3 . A_4
or
A_1 (A_2 . A_3 (A_4

- ➢ If we choose to put the bracket at first index

A_1 . A_2 . A_3 (A_4

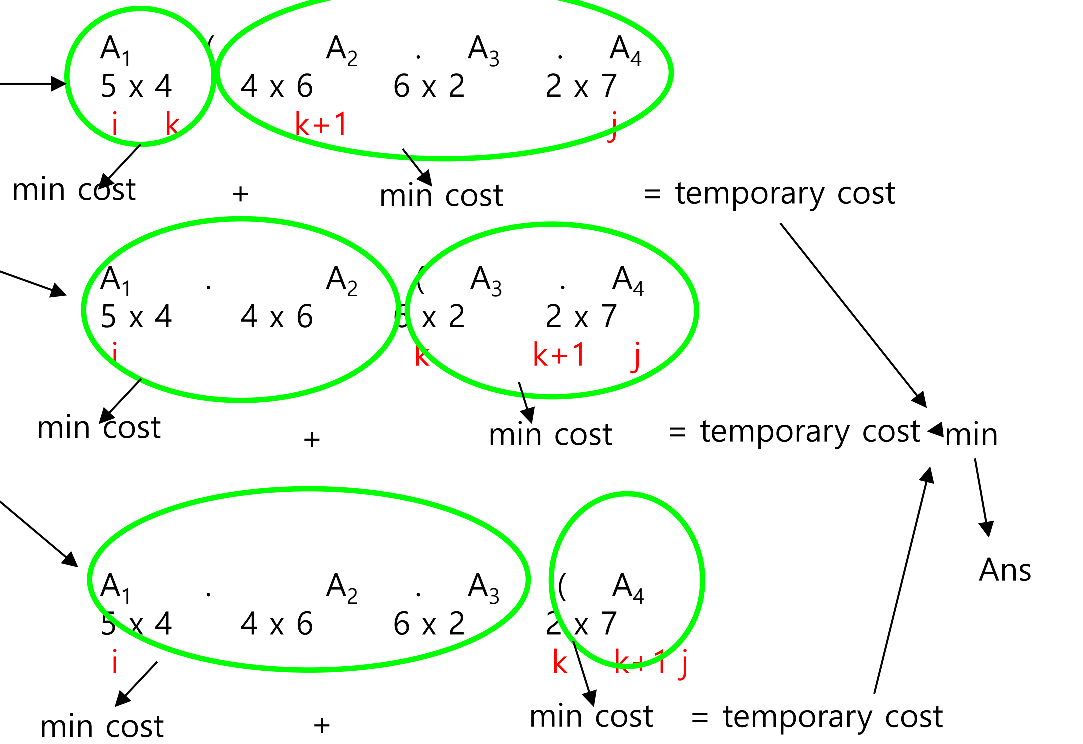
Now for the rest of the matrix we need to decide again

A_1 (A_2 . A_3 (A_4
or
A_1 . A_2 (A_3 (A_4

We are trying to iterate the matrix and dividing it into **two parts**
And again trying to find the best partition in the two divided parts.
At this point we can see the pattern for **recursive function**.

Arr	5	4	6	2	7
	i	k	K+1		j

- To divide this matrixes, we actually need 3 indexes i, k, j
- We will divide it from i ~ k and k+1 ~ j



A_1 (A_2 . A_3 . A_4
5 x 4 4 x 6 6 x 2 2 x 7
i k k+1 j
min cost + min cost = temporary cost

A_1 . A_2 (A_3 . A_4
5 x 4 4 x 6 6 x 2 2 x 7
i k k+1 j
min cost + min cost = temporary cost ← min

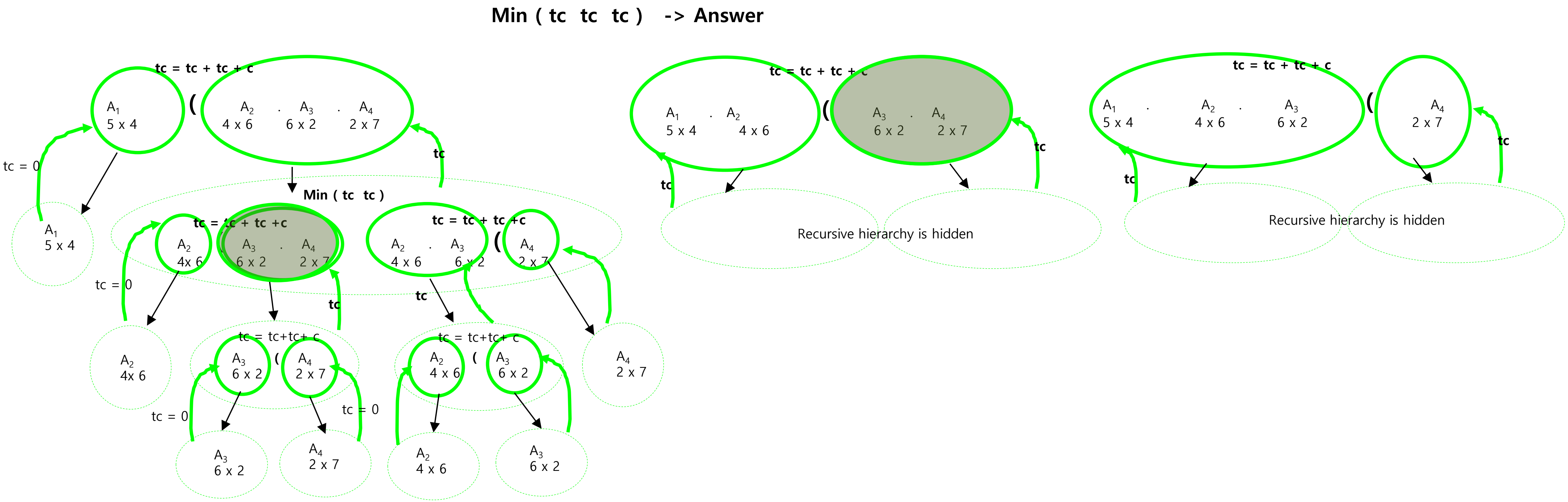
Ans

A_1 . A_2 . A_3 (A_4
5 x 4 4 x 6 6 x 2 2 x 7
i k k+1 j
min cost + min cost = temporary cost

At this point we understand few things
1. We need to break the array so we need i k j – three indexes
2. We will keep breaking the partitions recursively, until we get the smallest partition
3. From each partition we will get the temporary cost, and we nee d to use minimum function to find the best cost

Recursive call tree for MatricChainMultiplication

matrixChain(arr[] , i , j)

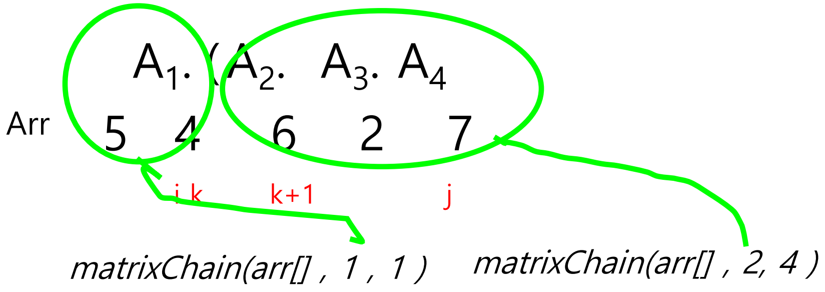


Matrix Multiplication idea development

arr:	5	4	6	2	7
	0	1	2	3	4

A_1 -> arr[0] x arr[1]
A_2 -> arr[1] x arr[2]
A_3 -> arr[2] x arr[3]
A_4 -> arr[3] x arr[4]

A_i -> arr[i-1] x arr[i]



matrixChain(arr[] , 1 , 1) *matrixChain(arr[] , 2, 4)*

At this point we understand few things

1. We need to break the array so we need **i k j** – three indexes
2. We will keep breaking the partitions recursively, until we get the smallest partition
3. From each partition we will get the temporary cost, and we need to use minimum function to find the best cost

- We need to break the array so we need **i k j** – three indexes

Can you keep i at 0
Ans : no arr[0-1] x arr[0] -> is not possible
So at the beginning **i will be at 1**

Can you keep j at last index
Ans: yes arr[4-1] x arr[4] -> is possible
So at the beginning **j will be at n-1**

- Lets declare the function now

```
int matrixChain(arr[] , i , j )
```

- Base Condition
  - Base condition you have to think in terms of input only, your inputs are arr, i and j
  - The minimum array size could be 0 or 1 element. That is possible if i >= j
  - Why i and j ? Because i and j is changing, arr[] is passed as it is.

```
        if ( i = j )
              return 0
```

  - i = j means given array size is 1

5 4
i=1 j=1

For given array [5,4] – cost is 0
i initialized with 1, j is initialized with n-1

- Think about how to move k -> from i to j
  - we have to make recursive call from i to k and k+1 to j

Arr	5	4	6	2	7
		i k	k+1		j

Matrix Multiplication idea development

arr	5	4	6	2	7
	0	1	2	3	4

A_1 -> arr[0] x arr[1]
A_2 -> arr[1] x arr[2]
A_3 -> arr[2] x arr[3]
A_4 -> arr[3] x arr[4]

A_i -> arr[i-1] x arr[i]

- function declaration

```
int matrixChain(arr[] , i , j )
```

- Base Condition

```
if ( i > = j )
      return 0
```

- **Choice Diagram**
  - For breaking the array into two parts

A_1 (A_2 . A_3 . A_4
or
A_1 . A_2 (A_3 . A_4
or
A_1 . A_2 . A_3 (A_4

  - To do that we will have to run a loop and partition at each k
  - And find the minimum cost among all these three partition.

we can run partitioning loop

```
for ( int k = i; k <= j -1 ; k++)
      inside this for loop you will make recursive call for both the partitions
      and add its cost
```

A_1 . A_2 (A_3 . A_4

*

matrixChain(arr, I , k) *matrixChain(arr, k+1, j)*

minimum cost + minimum cost

but that is not sufficient, you will also have to add one more cost

minimum cost + minimum cost + cost of $(A_1A_2)*(A_3A_4)$

- We need to think about k, k is used to break the array to make recursive call from i to k and k+1 to j
  - Think about how to move k -> from i to j
  - Can we start k at index i

  Ans: yes

Arr	5	4	6	2	7
		i k	k+1		j

i to k
we have one matrix
5 x 4

k+1 to j
we have 3 matrix
4 x 6
6 x 2
2 x 7

- Can we take k upto j
- Ans : no

Arr	5	4	6	2	7	
		i			k j	k+1

- Can we take k upto j-1
- Ans : yes

Arr	5	4	6	2	7
		i		k	k+1 j

i to k
we have three matrix
x
5 x 4
4 x 6
6 x 2

k+1 to j
we have at least matri
2 x 7

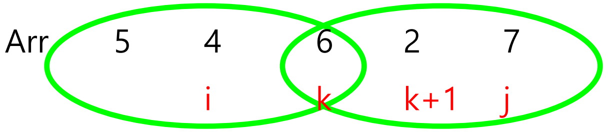
Matrix Multiplication idea development

arr	5	4	6	2	7
	0	1	2	3	4

A_1 -> arr[0] x arr[1]
A_2 -> arr[1] x arr[2]
A_3 -> arr[2] x arr[3]
A_4 -> arr[3] x arr[4]

A_i -> arr[i-1] x arr[i]

- temporary cost = minimum cost + minimum cost + **cost of $(A_1A_2)*(A_3A_4)$**


Arr 5 4 6 2 7
i k k+1 j

for (i to k)
5 x 4 4 x 6
min cost – 5 x 4 x 6
dimension of this new matrix – 5 x 6

*

for (k+1 to j)
6 x 2 2 x 7
min cost – 6 x 2 x 7
dimension of this new matrix 6 x 7

cost of multiplying these two will be 5 x 6 x 7

now lets find where is 5, 6 and 7
arr[i-1] x arr[k] x arr[j]

- Once we have temporary cost from all the iterations, then find the **minimum** among the temporary cost

Matrix Multiplication idea development

arr	5	4	6	2	7
	0	1	2	3	4

A_1 -> arr[0] x arr[1]
A_2 -> arr[1] x arr[2]
A_3 -> arr[2] x arr[3]
A_4 -> arr[3] x arr[4]

A_i -> arr[i-1] x arr[i]

Now lets write the entire code

```
int matrixChain( int arr[], int i , int j )
{
     if (i == j )
          return 0;

     int min = INT_MAX;

     for ( int k = i; k < = j -1 ; k++)
     {
      int temporary = matrixChain( arr, i , k) + matrixChain ( arr, k+1, j )
                        + arr[i+1] * arr[k] * arr[j] ;

       if ( temporary < min )
             min = temperory ;

     }

     return min

}
```

To solve this problem systematically

1. Think about inputs to the function
   Find i and j
2. Base condition
3. Find k loop scheme
4. Calculate answer from temporary ans

Matrix Chain Multiplication using Recursion

```
# A naive recursive implementation that
# simply follows the above optimal
# substructure property
import sys
# Matrix A[i] has dimension p[i-1] x p[i]
# for i = 1..n
def MatrixChainOrder(p, i, j):

    if i == j:
        return 0

    _min = sys.maxsize

    for k in range(i, j):

        count = (MatrixChainOrder(p, i, k)
             + MatrixChainOrder(p, k + 1, j)
                   + p[i-1] * p[k] * p[j])

        if count < _min:
            _min = count;


    # Return minimum count
    return _min;


# Driver program to test above function
arr = [5, 4, 6, 2, 7];
n = len(arr);

print("Minimum number of multiplications is ",
                MatrixChainOrder(arr, 1, n-1));
```

Input: p[] = {40, 20, 30, 10, 30} Output: 26000 There are 4 matrices of dimensions 40x20, 20x30, 30x10 and 10x30. Let the input 4 matrices be A, B, C and D. The minimum number of multiplications are obtained by putting parenthesis in following way (A(BC))D --> 20*30*10 + 40*20*10 + 40*10*30

Problems based on Matrix Chain Multiplication

```
Input: symbol[] = {T, F, T} operator[] = {^, &}
 Output: 2
 The given expression is "T ^ F & T", it
 evaluates true in two ways "((T ^ F) & T)" and
 "(T ^ (F & T))"
```

1. Matrix Chain Multiplication (MCM)
2. Printing MCM
3. Evaluate Expression to True/Boolean Paranthesization
4. Min/Max value of an Expression Ex: 2 * 3 + 5 -> 2 * (3 + 5)
5. Palindrome Partitioning Ex: aab -> [["a","a","b"],["aa","b"]]
6. Scamble String
7. Egg Dropping Problem

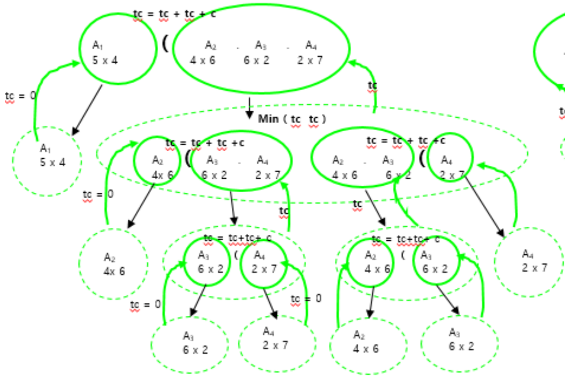
Note: How will you identify if the problem is MCM problem

Hint :1) you will be given an array or string as input

2) The solution will require you to break the array in two parts with i ~ k and k ~ j

3) from each partition you will get a temporary answer

4) you will have to manipulate (max, min) temporary answer to get your result



Ex: Matrix Chain Multiplication (Tabulation)

A1	.	A2	.	A3	.	A4	Matrix Product
5 x 4		4 x 6		6 x 2		2 x 7	Dimensions of matrix

What is Matrix Chain Multiplication

A_1 . A_2 . A_3 . A_4 Matrix Product
5 x 4 4 x 6 6 x 2 2 x 7 Dimensions of matrix

Dynamic Programming uses Tabulation method , these are the Tables m and s
Hint: Decide the size of the table based on changing values in recursion that are i and j
m - represents the cost of each matrix multiplication
S - represents the parenthesis (If the question is to only find the cost, we don't need to maintain S)
It's a **bottom up** approach, so we will fill up the table starting with the **smallest** problem possible
Hint: To initialize this table you can think of base case in recursion, based on base case for all i >= j assign 0, this also give you hint that this table needs to be filled diagonally

One element A1, A2, A3, A4 – m[1,1] m[2,2] m[3,3] m[4,4] - > Cost of multiplication is -> 0

Diagonal filling

Two elements A1 * A2 - m[1,2] Cost of multiplication is ->5 * 4 * 6 = 120

A2 * A3 - m[2,3] Cost of multiplication is ->4 * 6 * 2 = 48

A3 * A4 - m[3,4] Cost of multiplication is ->6 * 2 * 7 = 84

Diagonal filling

Three elements A1 * A2 * A3

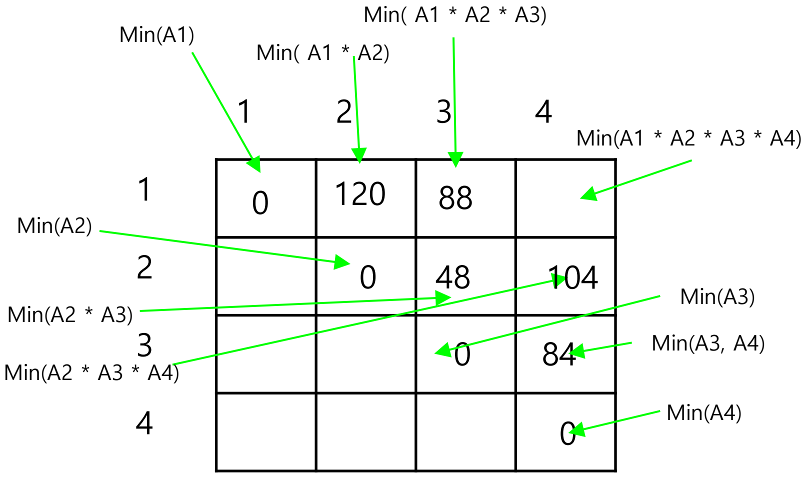
There are two ways to multiply this

A1 * (A2 * A3) or (A1 * A2) * A3
5 x 4 4 x 6 6 x 2 5 x 4 4 x 6 6 x 2

0 + 48 + 5 * 4 * 2 = 88 120 + 0 + 5 * 6 * 2 = 180
min (88, 180) = 88 parenthesis will be at A1

A2 * A3 * A4
A2 * (A3 * A4) or (A2 * A3) * A4
4 x 6 6 x 2 2 x 7 4 x 6 6 x 2 2 x 7
0 + 84 + 4 * 6 * 7 = 252 48 + 0 + 4 * 2 * 7 = 104
min (252, 104) = 104 parenthesis will be at A3

m



	1	2	3	4
1	0	120	88	
2		0	48	104
3			0	84
4				0

S	1	2	3	4
1		1	1	
2			2	3
3				3
4				

What is Matrix Chain Multiplication

A_1 . A_2 . A_3 Matrix Product

5 x 4 4 x 6 6 x 2 Dimensions of matrix

d0 d1 d1 d2 d2 d3

Lets try to make a formula for this multiplication

Lets find the minimum cost for A1 * A2 * A3 -> m[1, 3]

Three elements A1 * A2 * A3

two ways to multiply this

(A1 * (A2 * A3)) or ((A1 * A2) * A3)

5 x 4 4 x 6 6 x 2 5 x 4 4 x 6 6 x 2

d0 d1 d1 d2 d2 d3

m [1,1] + m[2,3] + 5 * 4 * 2 m [1,2] + m [3,3] + 5 * 6 * 2

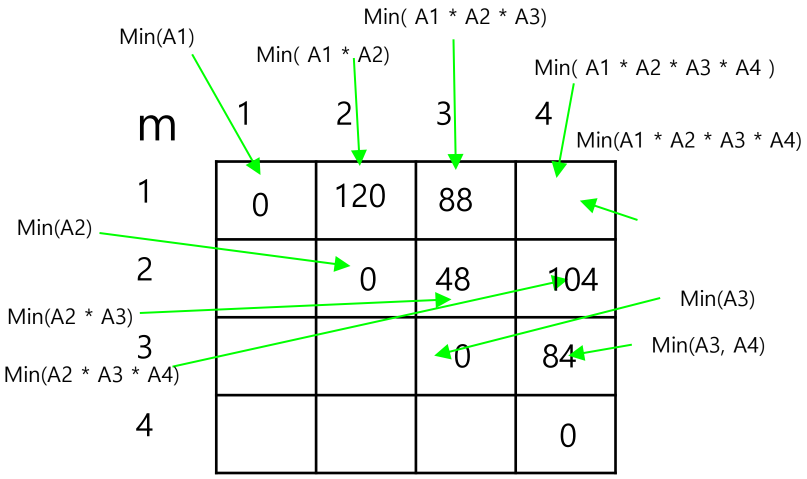
↑ ↑ ↑ ↑ ↑ ↑ ↑

i k k+1 j d_0 * d_1 * d_3 i k k+1 j d_0 * d_2 * d_3

m[i, k] + m[k+1, j] + d_{i-1} * d_k * d_j m[i,k] + m[k+1, j] + d_{i-1} * d_k *d_j

m[i , j] = min {m[i, k] + m[k+1, j] + d_{i-1} * d_k * d_j}

i<= k < j



m	1	2	3	4
1	0	120	88	
2		0	48	104
3			0	84
4				0

s	1	2	3	4
1		1	1	
2			2	3
3				3
4				

What is Matrix Chain Multiplication

A_1	.	A_2	.	A_3	.	A_4
5 x 4		4 x 6		6 x 2		2 x 7
d0 d1		d1 d2		d2 d3		d3 d4

Matrix Product Dimensions

Lets try to make a formula for 4 Element -> min (A1*A2*A3*A4) -> m[1,4]

$$m[i\,,j] = \min_{i \le k < j} \{m[i,k] + m[k+1,j] + d_{i-1} * d_k * d_j\}$$

$$m[1,4] = \min_{1 \le k < 4} \begin{cases} k=1,\ m[1,1] + m[2,4] + d_0 * d_1 * d_4\ , & \text{-> (A1) *(A2 * A3 * A4)} \\ k=2,\ m[1,2] + m[3,4] + d_0 * d_2 * d_4\ , & \text{-> (A1 * A2) * (A3 * A4)} \\ k=3,\ m[1,3] + m[4,4] + d_0 * d_3 * d_4 & \text{-> (A1 * A2 * A3) * (A4)} \end{cases}$$

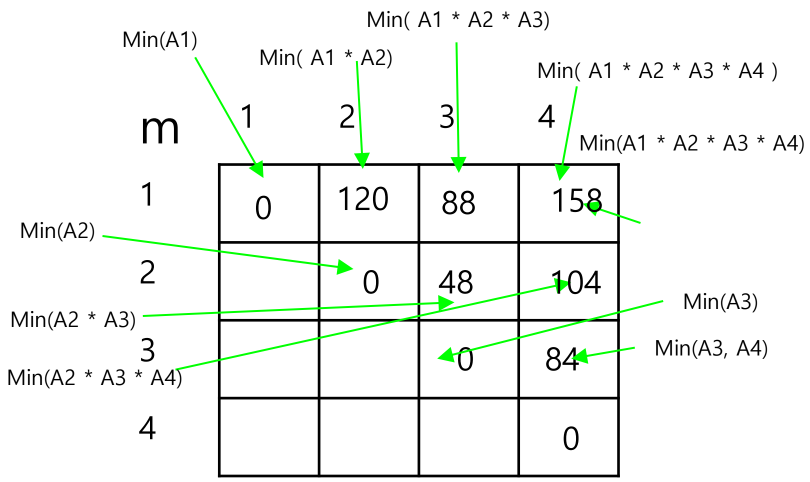
min (A2*A3*A4) -> m[2,4]
min (A1 *A2) -> m[1,2]
min (A3 *A4) -> m[3,4]
min (A1 *A2 * A3) -> m[1,3]

We have already calculated it, and stored it in the table we just need to reuse it

We can use the same formula to calculate these values

$$m[1,4] = \min_{1 \le k < 4} \begin{cases} k=1,\ 0 + 104 + 5 * 4 * 7\ , & \text{-> 244} \\ k=2,\ 120 + 84 + 5 * 6 * 7\ , & \text{-> 414} \\ k=3,\ 88 + 0 + 5 * 2 * 7 & \text{-> 158} \end{cases}$$

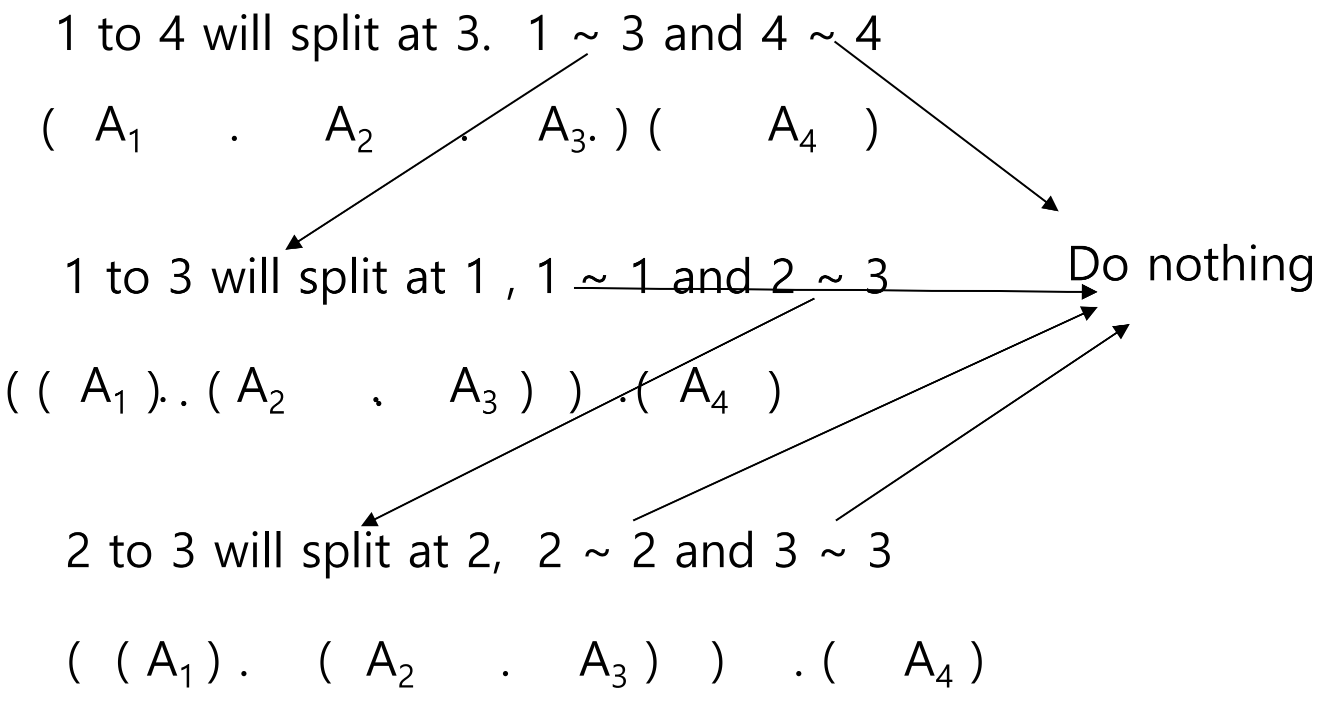
(A1 * A2 * A3) * A4



m	1	2	3	4
1	0	120	88	158
2		0	48	104
3			0	84
4				0

s	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

Backtracking to find the parenthesis position



1 to 4 will split at 3. 1 ~ 3 and 4 ~ 4

(A_1 . A_2 . A_3.) (A_4)

1 to 3 will split at 1 , 1 ~ 1 and 2 ~ 3

Do nothing

((A_1). . (A_2 . A_3)) . (A_4)

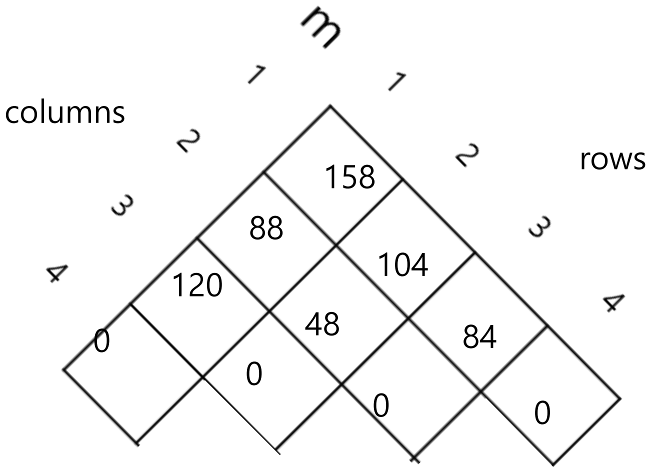
2 to 3 will split at 2, 2 ~ 2 and 3 ~ 3

((A_1) . (A_2 . A_3)) . (A_4)

s	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

You can also draw DP tables like this also

m	1	2	3	4
1	0	120	88	158
2		0	48	104
3			0	84
4				0



In your book, the table is drawn like this

Matrix Chain Multiplication using Tabulation

```
# Dynamic Programming Python implementation of Matrix Chain Multiplication.
import sys
def MatrixChainOrder(p, n):
    # For simplicity of the program, one extra row and one
    # extra column are allocated in m[][].  0th row and 0th
    # column of m[][] are not used

    m = [[0 for x in range(n)] for x in range(n)]


    for i in range(1, n):
        m[i][i] = 0

    # L is chain length.
    for L in range(2, n):
        for i in range(1, n-L + 1):
            j = i + L-1
            _min = sys.maxsize
            for k in range(i, j):
                count = m[i][k] + m[k + 1][j] + p[i-1]*p[k]*p[j]
                if count < _min:
                    m[i][j] = count
                    _min = count

    return m[1][n-1]
# Driver program to test above function
arr = [5, 4, 6, 2, 7]
size = len(arr)
print("Minimum number of multiplications is " + str(MatrixChainOrder(arr, size)))
```

Run the loop to fill the table
This loop will run Diagonally

Complexity – n^3

Input: p[] = {40, 20, 30, 10, 30} Output: 26000 There are 4 matrices of dimensions 40x20, 20x30, 30x10 and 10x30. Let the input 4 matrices be A, B, C and D. The minimum number of multiplications are obtained by putting parenthesis in following way (A(BC))D --> 20*30*10 + 40*20*10 + 40*10*30

```
#Recursive Code
def MatrixChainOrder(p, i, j):

    if i == j:
        return 0
    _min = sys.maxsize

    for k in range(i, j):
        count = (MatrixChainOrder(p, i, k) + MatrixChainOrder(p, k + 1, j)
                 + p[i-1] * p[k] * p[j])

        if count < _min:
            _min = count;

    # Return minimum count
    return _min;
```

Complexity – 2^n

Dynamic Programming

- Dynamic Programming is used for solving **Optimization problem**
  - **Maximize** something
  - **Minimize** something
- Two important properties of DP
  - **overlapping sub problems**.
  - **Optimal substructure**
- **Optimization** Problem can be solved using
  - **Greedy**
  - **Dynamic Programming**
- DP follows **Principle of Optimality**
  - Which means, problem can be solved by taking a **sequence of decision**
  - **For ex:** Shall I include this particular item or not? Shall I include next item or not ? ……………… … so on an so forth.
  - Usually we take decisions from last object towards first object
- In Dynamic Programming, you should try all possible solutions and then pickup the best solution.
  - In **linear** approach considering all solution will take too much time. (exponential time complexity)
  - Dynamic programming **reduces** the time complexity
- There are two approach for Dynamic Programming
  - **Memoization (Top Down)**
  - **Tabulation (Bottom Up)**

Dynamic Programming

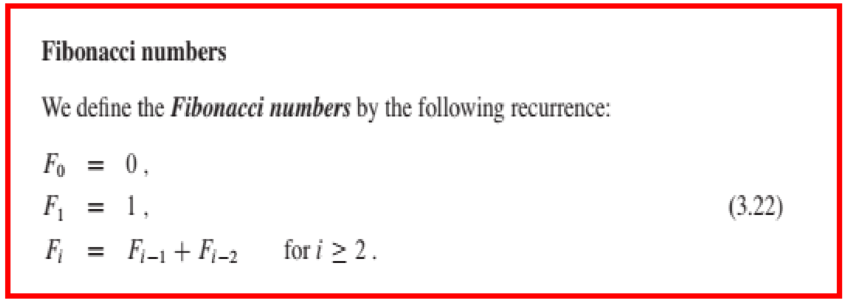
- Two important properties of DP
  - **Overlapping sub problems**.
  - **Optimal substructure**

problem exhibits ***optimal substructure*** if an optimal solution to the problem contains within it optimal solutions to subproblems.
Whenever a problem exhibits optimal substructure,
we have a good clue that dynamic programming might apply.

In dynamic programming, we build an optimal solution to the problem from optimal solutions to subproblems. Consequently, we must take care to ensure that the range of subproblems we consider includes those used in an optimal solution (**overlapping** sub problems).

Matrix chain multiplication

$$m[i\,,\,j] = \min_{i <= k < j} \{m[i, k] + m[k+1, j] + d_{i-1} * d_k * d_j\}$$



Fibonacci numbers

We define the ***Fibonacci numbers*** by the following recurrence:

$$\begin{aligned} F_0 &= 0\,, \\ F_1 &= 1\,, \qquad\qquad\qquad (3.22) \\ F_i &= F_{i-1} + F_{i-2} \quad \text{for } i \geq 2\,. \end{aligned}$$

0/1 knapsack

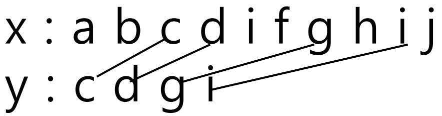
$$V[i,j] = \max\{ V[i-1, j], V[i-1, j - w[i]] + p[i] \}$$

Ex: Longest Common Subsequence

1. What is LCS
2. LCS using recursion
3. LCS using Memoization
4. LCA using DP

Longest Common Subsequence (LCS Problem)

Input :



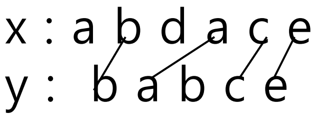
x : a b c d i f g h i j

y : c d g i

Output : 4

Characters are not together but are appearing in the same order in the String
Output is the length of the longest common subsequence

Input:



x : a b d a c e

y : b a b c e

Output: 4

One subsequence is b a c e
Second subsequence starting from a is a b c e

Input:

x : p a r k c h e h y u n

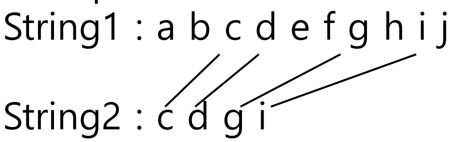
y : s u m a n p a n d e y

Output : 3

One subsequence is u n
Second subsequence starting from a is a n
Third subsequence starting from p is **p a n**
Fourth subsequence starting from e is e y

Ex: Longest Common Subsequence (Recursion)

Example 1:



String1 : a b c d e f g h i j

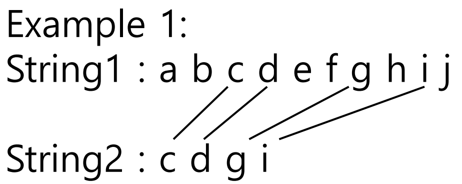
String2 : c d g i

Output : 4

LCS (Recursion)

Function declaration

- Function will return the maximum length of common subsequence, so return type will be int
- You are given two strings as input, and in each function call you will reduce the size of the input, hence arguments will be String x, String y, int n, int m


Example 1:
String1 : a b c d e f g h i j
String2 : c d g i

```
int  LCS ( String x, String y, int n , int m)
```

Base case

- Based on smallest valid input

What is your Input ?
x: a b c d e f g h i j -> n (10)
y: c d g i   -> m (4)

Can we take n = 0 and m = 0 ?
Ans: **yes**, we can have an empty string.

For smallest input what will be length of longest common subsequence ?
Ans: 0 , if you don't have a string you cant have common subsequence.

```
if ( n == 0 || m == 0 )
      return 0
```

Decreasing function

- Reducing the size of input

a b c d e f g h i ~~j~~
c d g ~~i~~

We keep reducing the size of the input string from last element.

Choice Diagram

LCS (Recursion)

- ➢ Base case

```
if ( n == 0 || m == 0 )
        return 0
```

- ➢ Decreasing function
  - ➢ Reducing the size of input   a b c d e f g h i ~~j~~,   c d g ~~i~~

- ➢ Choice Diagram

❖ **first choice,** when the last element of the string **matches**

x(n) : a b c d e f g h i
y (m): c d g i

if (x[n-1] == y [m-1])
we have found one match, and **increase the returning length by 1**
reduce the length of x and y both
make a recursive call with x length **n-1** and y length **m-1**

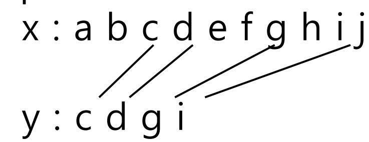
❖ **second choice,** when the last element of the string **does not matches**

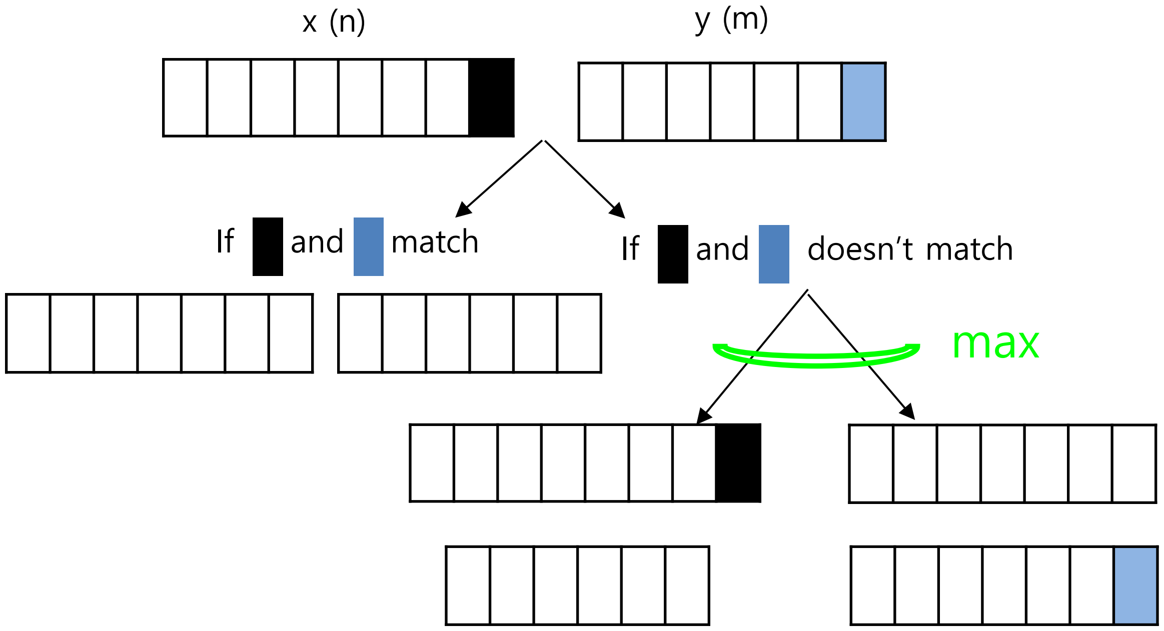
x(n) : a b c d e f g h i j
y (m): c d g i
In this case we have another two choices

❖ we can reduce n to **n-1** and keep **m** as it is
❖ we can reduce m to **m-1** and keep **n** as it is
} We will choose one that could return maximum

Note: in this case we do not increase the length of returning string, as the match is not found. Hence call the recursive function.

Example 1:





LCS (Recursion)

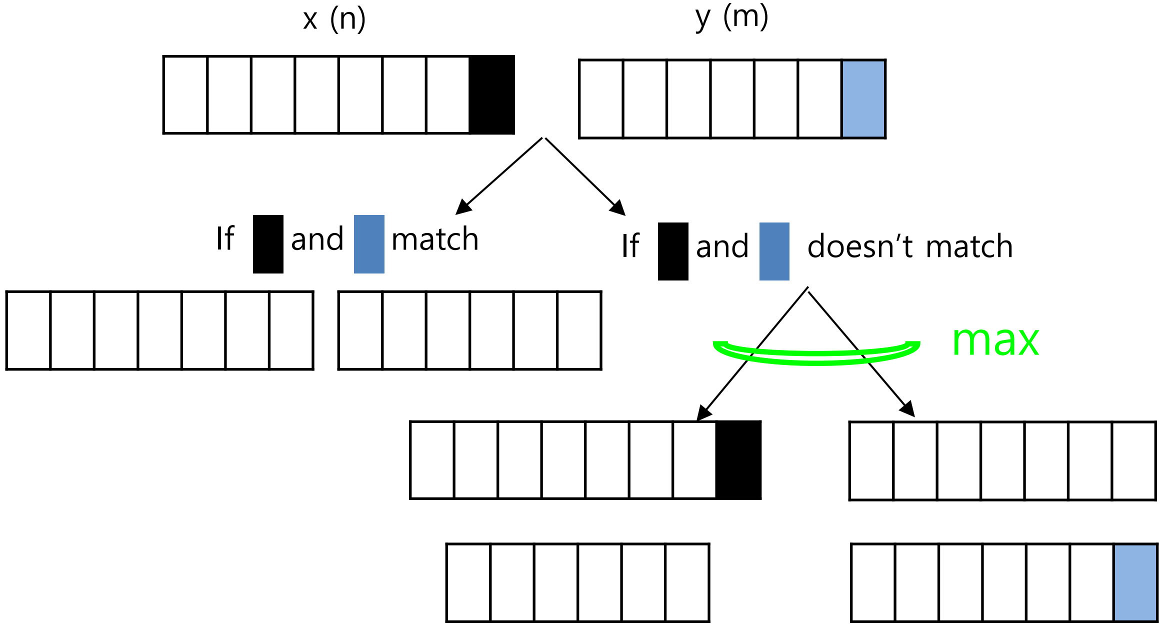
Now lets write the entire code

```
int LCS( String x, String y, int n , int m )
{
     // base case
     if ( n == 0 || m == 0 )
             return 0;

     // choice diagram
      if ( x[n-1] == y[m-1] )
         return 1 + LCS(x, y, n-1, m-1)
      else
         return max( LCS ( x, y, n , m-1) , LCS (x, y, n -1, m) )

}
```

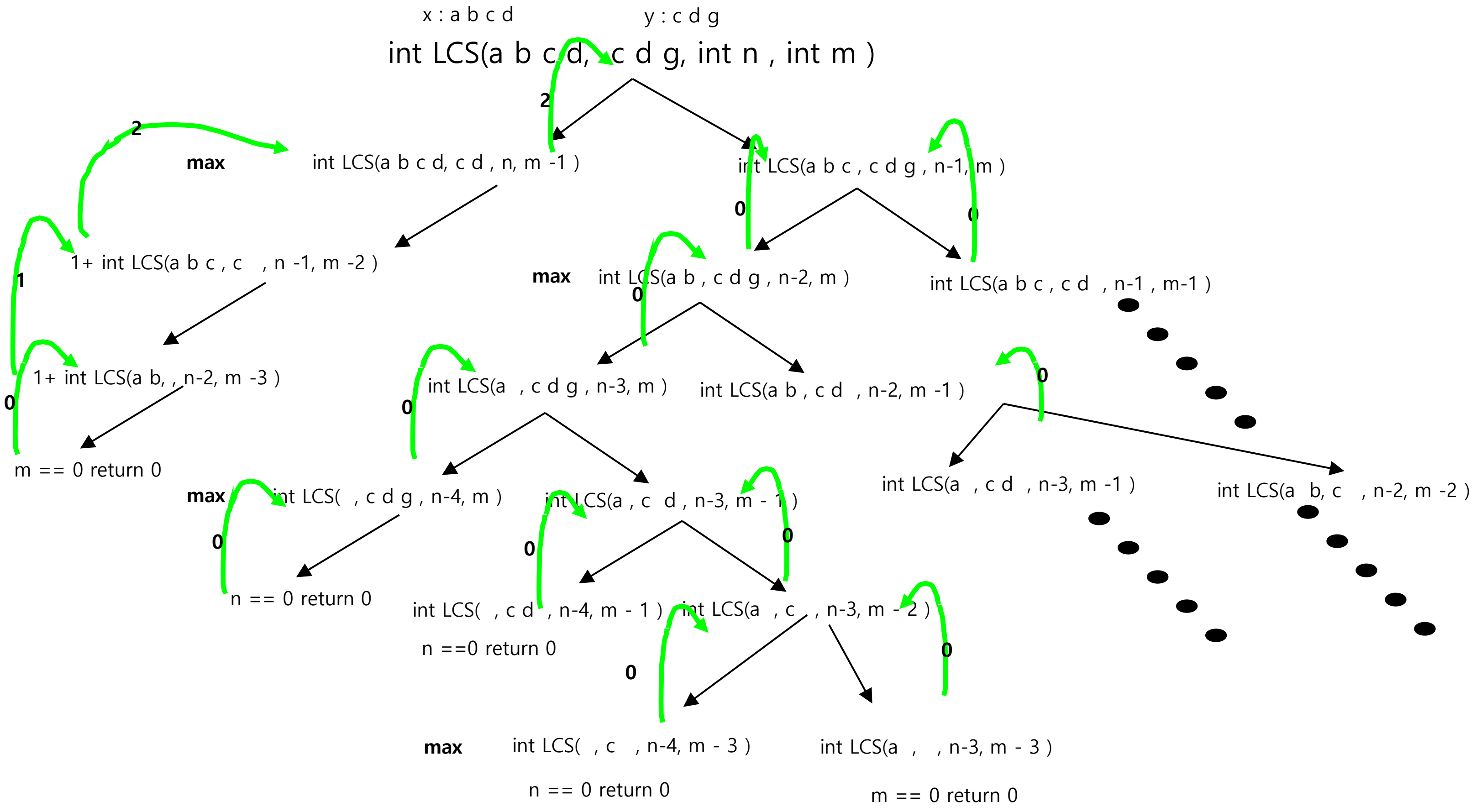
Example 1:

x : a b c d e f g h i j

y : c d g i



LCS (Recursion)



x : a b c d

y : c d g

int LCS(a b c d, c d g, int n , int m)

2

2

max

int LCS(a b c d, c d , n, m -1)

int LCS(a b c , c d g , n-1, m)

0

0

1+ int LCS(a b c , c , n -1, m -2)

1

max

int LCS(a b , c d g , n-2, m)

0

int LCS(a b c , c d , n-1 , m-1)

1+ int LCS(a b, , n-2, m -3)

0

0

int LCS(a , c d g , n-3, m)

int LCS(a b , c d , n-2, m -1)

0

m == 0 return 0

int LCS(a , c d , n-3, m -1)

int LCS(a b, c , n-2, m -2)

max

int LCS(, c d g , n-4, m)

0

int LCS(a , c d , n-3, m - 1)

0

0

n == 0 return 0

int LCS(, c d , n-4, m - 1)

int LCS(a , c , n-3, m - 2)

n ==0 return 0

0

0

max

int LCS(, c , n-4, m - 3)

int LCS(a , , n-3, m - 3)

n == 0 return 0

m == 0 return 0

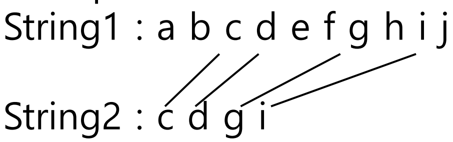
Now lets write the entire code

```
int LCS( String x, String y, int n , int m )
{
      // base case
      if ( n == 0 || m == 0 )
              return 0;

      // choice diagram
       if ( x[n-1] == y[m-1] )
          return 1 + LCS(x, y, n-1, m-1)
       else
          return max( LCS ( x, y, n , m-1) ,
LCS (x, y, n -1, m) )

}
```

Ex: Longest Common Subsequence (Memoization)

- Why do we need memorization ?
- How to know the size of Matrix ?
- How does it work?

Example 1:



String1 : a b c d e f g h i j

String2 : c d g i

Output : 4

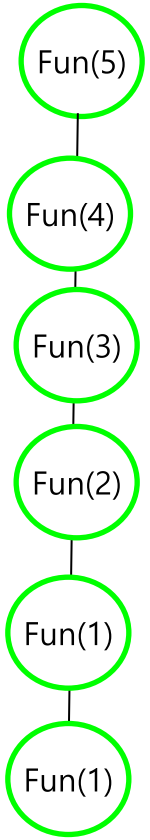
Why we need memoization

- ➢ Why do we need memoization or Tabulation ?

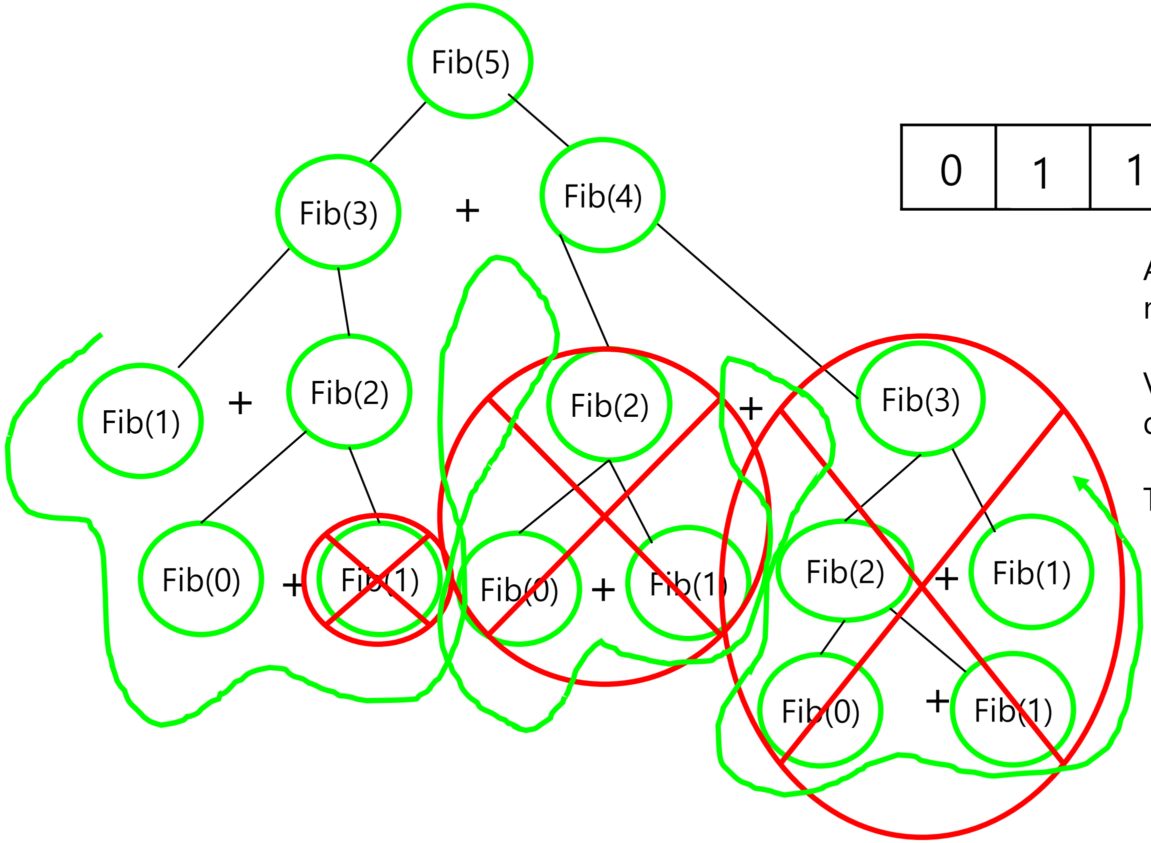
Ans: We use DP, only when we have overlapping sub problem. Which means we are calling the same recursive function (a recursive function with same list of arguments) again and again.

Recursion without Overlapping sub problem

Don’t use DP for these kind of problems



Recursion with Overlapping sub problem (Fibonacci)



0	1	1	2	4	5

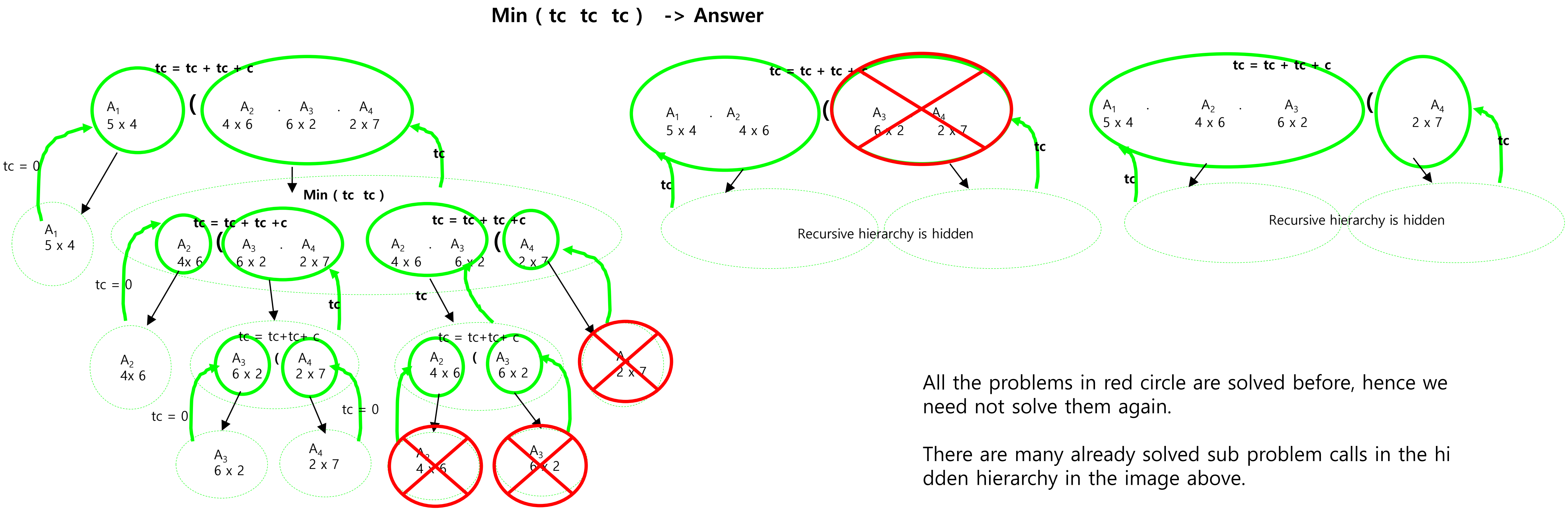
All the problems in red circle are solved before, hence we need not solve them again.

Values of smaller problems are stored in a table and reuse d whenever required

This approach reduces time complexity significantly.

Overlapping sub problem in MatricChainMultiplication

matrixChain(arr[] , i , j)



Min (tc tc tc) -> Answer
tc = tc + tc + c
A_1 5 x 4
(A_2 4 x 6 . A_3 6 x 2 . A_4 2 x 7
tc = 0
tc
Min (tc tc)
A_1 5 x 4
tc = tc + tc +c
A_2 4x 6
(A_3 6 x 2 . A_4 2 x 7
tc = 0
tc
tc = tc + tc +c
A_2 4 x 6 . A_3 6 x 2
(A_4 2 x 7
tc
A_2 4x 6
tc = tc+tc+ c
A_3 6 x 2 (A_4 2 x 7
tc = 0
tc = 0
A_3 6 x 2
A_4 2 x 7
tc = tc+tc+ c
A_2 4 x 6 (A_3 6 x 2
A_2 4 x 6
A_3 6 x 2
A_4 2 x 7
tc = tc + tc + c
A_1 5 x 4 . A_2 4 x 6
(A_3 6 x 2 A_4 2 x 7
tc
tc
Recursive hierarchy is hidden
tc = tc + tc + c
A_1 5 x 4 . A_2 4 x 6 . A_3 6 x 2
(A_4 2 x 7
tc
tc
Recursive hierarchy is hidden

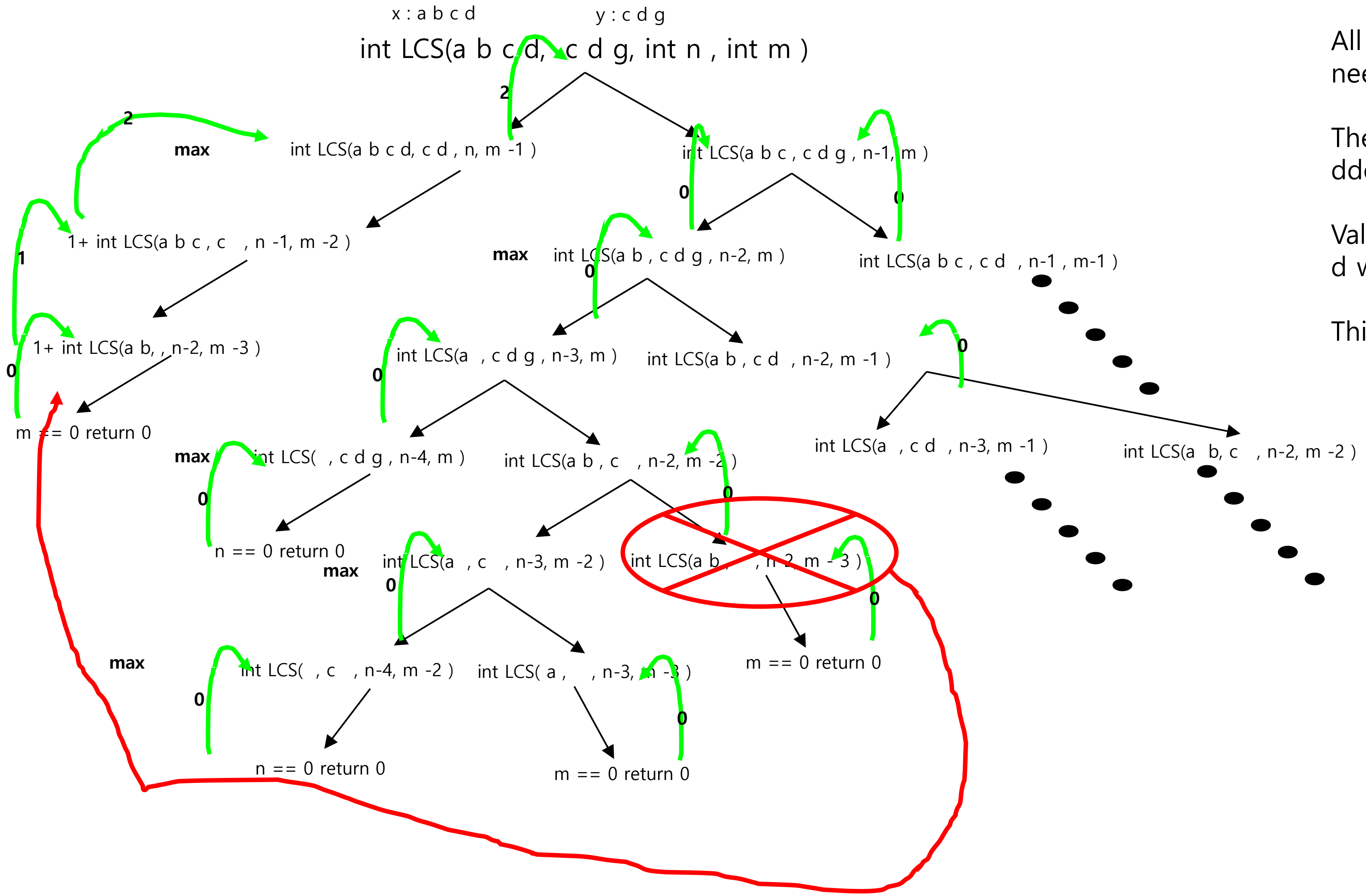
All the problems in red circle are solved before, hence we need not solve them again.

There are many already solved sub problem calls in the hi dden hierarchy in the image above.

Values of smaller problems are stored in a table and reuse d whenever required

This approach reduces time complexity significantly.

Overlapping sub problem in (LCS Problem)


x : a b c d
y : c d g
int LCS(a b c d, c d g, int n , int m)
2
max
int LCS(a b c d, c d , n, m -1)
int LCS(a b c , c d g , n-1, m)
2
1+ int LCS(a b c , c , n -1, m -2)
1
0
0
max
int LCS(a b , c d g , n-2, m)
0
int LCS(a b c , c d , n-1 , m-1)
1+ int LCS(a b, , n-2, m -3)
0
int LCS(a , c d g , n-3, m)
0
int LCS(a b , c d , n-2, m -1)
0
m == 0 return 0
max
int LCS(, c d g , n-4, m)
int LCS(a b , c , n-2, m -2)
int LCS(a , c d , n-3, m -1)
int LCS(a b, c , n-2, m -2)
0
0
n == 0 return 0
max
int LCS(a , c , n-3, m -2)
int LCS(a b , , n-2, m - 3)
0
0
max
int LCS(, c , n-4, m -2)
int LCS(a , , n-3, m -3)
m == 0 return 0
0
0
n == 0 return 0
m == 0 return 0

All the problems in red circle are solved before, hence we need not solve them again.

There are many already solved sub problem calls in the hidden hierarchy in the image above.

Values of smaller problems are stored in a table and reused whenever required

This approach reduces time complexity significantly.

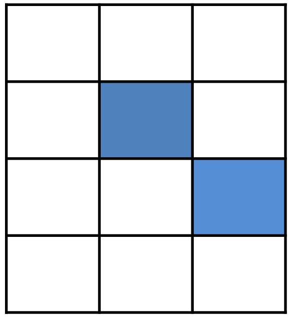
Memoization and Tabulation

- Knapsack / MCM / LCS can be solved with

1. Recursion

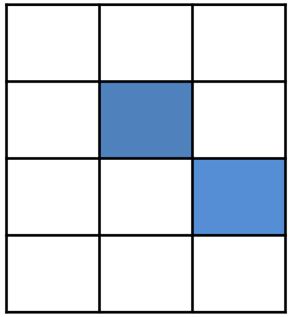
2. Memoization (Top Down approach)

Recursion +



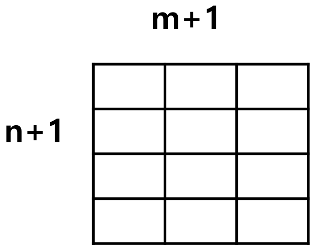
3. Tabulation (Bottom Up approach – problem solving starts with smallest problem first)

Only table recursion is converted into iteration



LCS (Memoization)

- What will be the size of the Table ?
  - We make the tables for the values that are changing in recursive call Such as **n and m .** The size of the table will be **n+1** and **m+1**



- How do we initialize the table in memoization ?
  - We initialize the table with -1
  - It helps us check if the smaller sub-function is already solved or not.

-1	-1	-1
-1	-1	-1
-1	-1	-1
-1	-1	-1

  - We solve the sub-problem(recursive call), and if the value in the table is -1 then we store the result in the table, if the value is not -1 then we do not need to solve that problem at all.

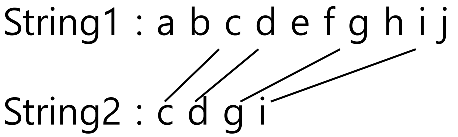
How to Modify the recursive code to memorization.

```
// Create table
int t[1001][1001];

int LCS( String x, String y, int n , int m )
{
    // base case
    if ( n == 0 || m == 0 )
          return 0;

    if (t[m][n] ! = -1 )
          return t[m][n] ;
    else {
    // choice diagram
     if ( x[n-1] == y[m-1] )
        return  t[m][n] =  1 + LCS(x, y, n-1, m-1)

     else

        return  t[m][n] = max( LCS ( x, y, n , m-1) , LCS (x, y, n -1, m) )
     }
}

Int main()
{
    // initialize t with -1
    memset ( t, -1, sizeof (t)) ;
    //read x and y
    LCS(x, y, x.length , y.length);
}
```

Ex: Longest Common Subsequence (Tabulation)

- Why do we need Tabulation ?
- How to know the size of Matrix ?
- How does it work?

Example 1:



String1 : a b c d e f g h i j

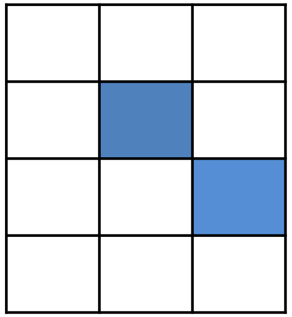
String2 : c d g i

Output : 4

Why is Tabulation better than Memoization

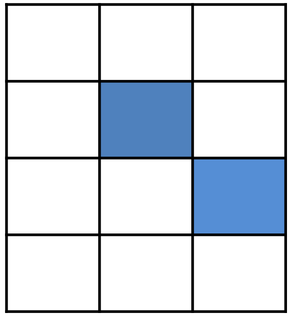
- Knapsack / MCM / LCS can be solved with
  1. Recursion
  2. Memoization (Top Down approach)

     Recursion +



  3. Tabulation (Bottom Up approach – problem solving starts with smallest problem first)

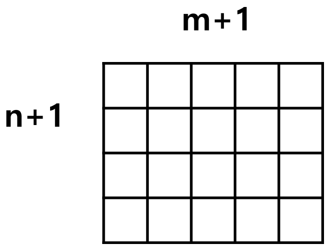
     Only table recursion is converted into iteration



Recursion can sometimes cause stack overflow, as recursion uses Stack memory. Using tabulation recursive calls can be totally omitted. -> this is the benefit of tabulation over memoization

LCS (Tabulation)

- What will be the size of the Table ?
  - We make the tables for the values that are changing in recursive version of LCS Such as **n and m .** The size of the table will be **n+1** and **m+1**



- How do we initialize the table in Tabulation ?
  - The base condition of the recursive version of LCS will change into the initialization of tabulation

m+1

n+1

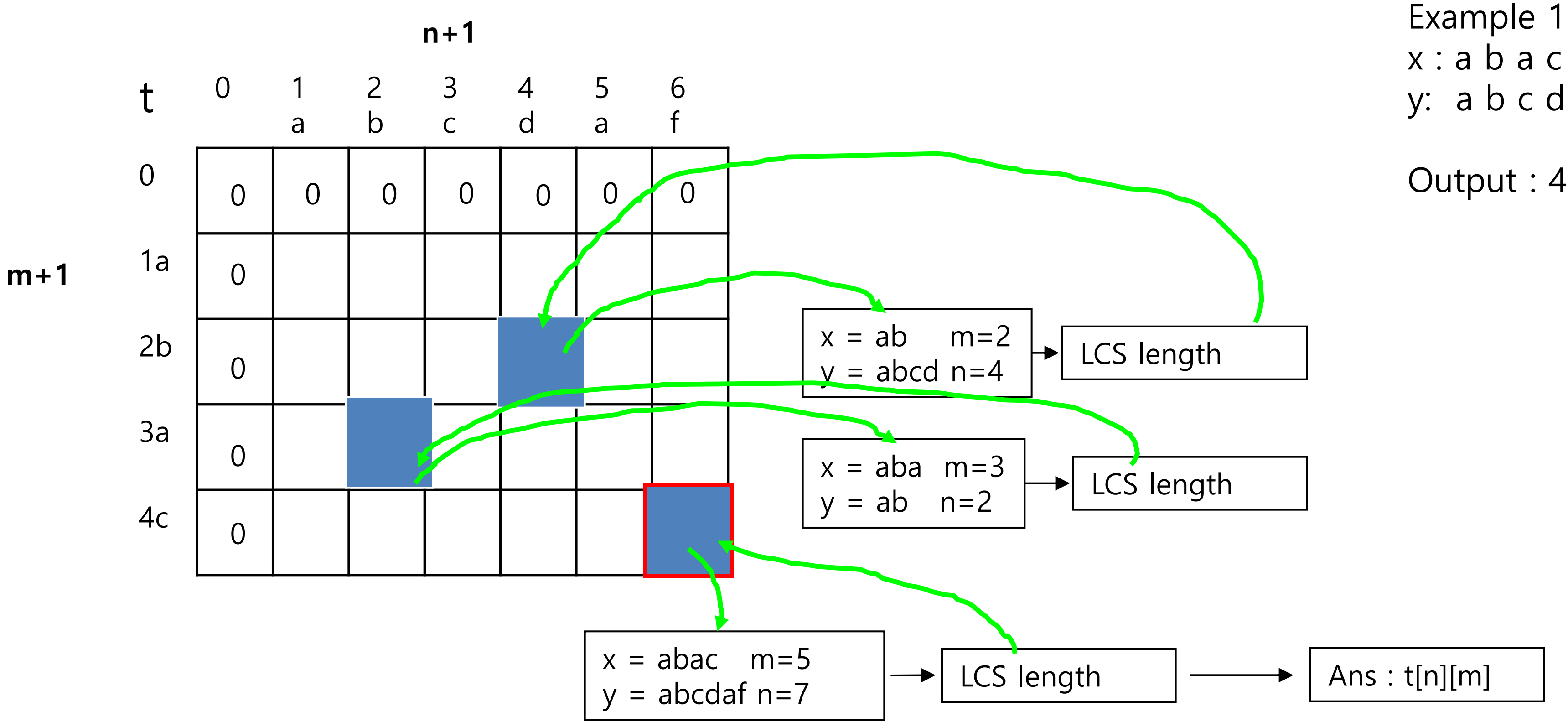
	0	1	2	3	4
0	0	0	0	0	0
1	0				
2	0				
3	0				

Now lets write the entire code

```
int LCS( String x, String y, int n , int m )
{
    // base case
    if ( n == 0 || m == 0 )
          return 0;

    // choice diagram
     if ( x[n-1] == y[m-1] )
        return 1 + LCS(x, y, n-1, m-1)
     else
        return max( LCS ( x, y, n , m-1) ,
LCS (x, y, n -1, m) )

}
```

LCS (Tabulation)

- What does each index in the table store ?
  - Each block in the table stores a result of a smaller sub problem



n+1

m+1

t	0	1 a	2 b	3 c	4 d	5 a	6 f
0	0	0	0	0	0	0	0
1a	0						
2b	0						
3a	0						
4c	0						

x = ab m=2
y = abcd n=4 → LCS length

x = aba m=3
y = ab n=2 → LCS length

x = abac m=5
y = abcdaf n=7 → LCS length → Ans : t[n][m]

Example 1:
x : a b a c -> m =4
y: a b c d a f -> n = 6

Output : 4

How to fill this table ?

LCS (Tabulation)

Recursion

Now lets write the entire code

```
int LCS( String x, String y, int n , int m )
{
      // base case
      if ( n == 0 || m == 0 )
              return 0;



      // choice diagram
       if ( x[n-1] == y[m-1] )
          return 1 + LCS(x, y, n-1, m-1)
       else
          return max( LCS ( x, y, n , m-1) , LCS (x, y, n -1, m) )

}
```

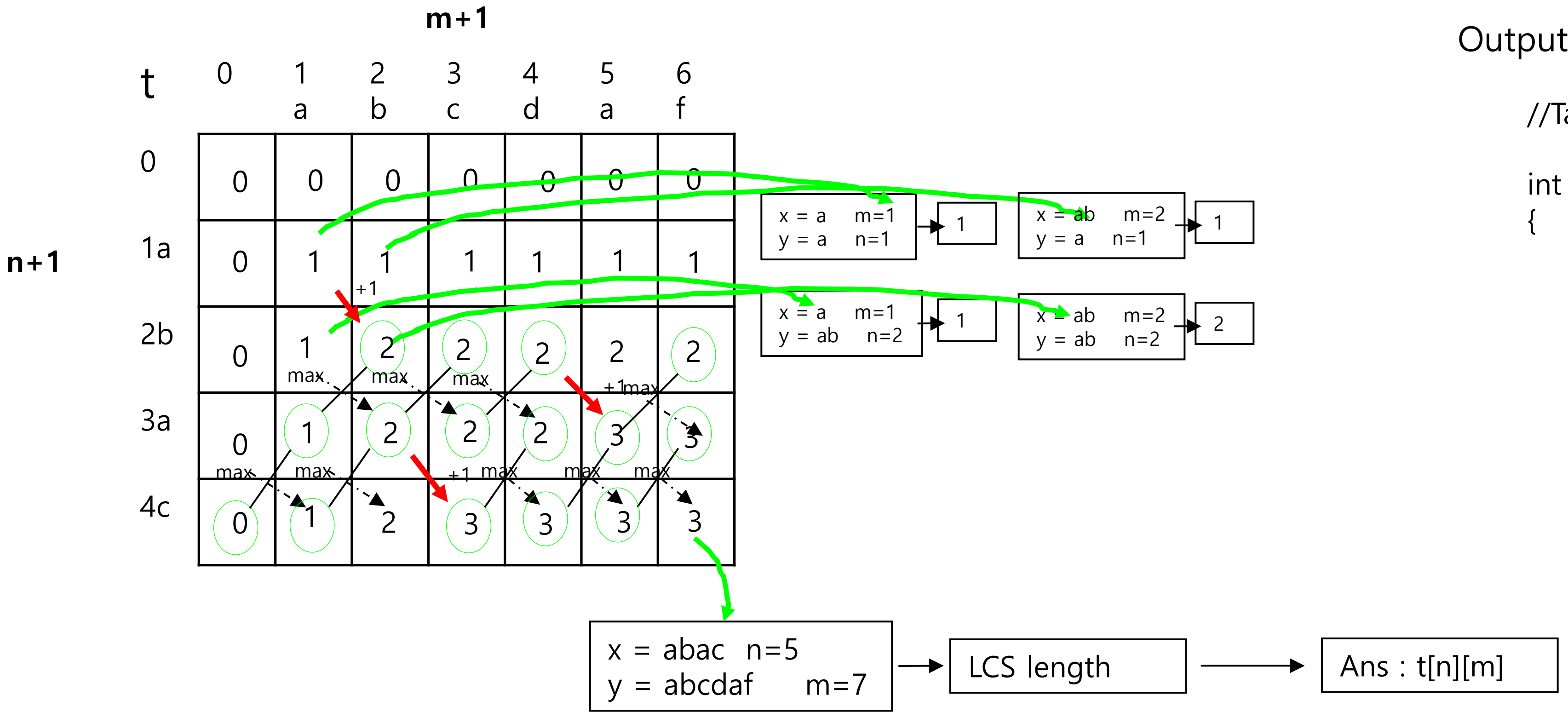
Tabulation   V[n+1][m+1]

Now lets write the entire code

```
int LCS( String x, String y, int n , int m )
{

     // base case
     for ( int i = 0; i<  n+1; i++ )
         for (int j =0; j< m+1; j++)
            if ( i == 0 || j ==0 )
               t[i][j] = 0;

     // choice diagram
       for ( int i = 1; i<  n+1; i++ )
         for (int j =1; j< m+1; j++)
              if ( x[i-1] == y[j-1] )
                  t[i][j] = 1 + t [i-1][j-1]
              else
                  t[i][j]= max( t [i][j-1] , t [i -1][j ] )

       return t[n][m]
}
```

LCS (Tabulation)

- Filling the table and obtaining the answer



m+1

n+1

t	0	1 a	2 b	3 c	4 d	5 a	6 f
0	0	0	0	0	0	0	0
1a	0	1	1	1	1	1	1
2b	0	1	2	2	2	2	2
3a	0	1	2	2	2	3	3
4c	0	1	2	3	3	3	3

x = a m=1
y = a n=1
→ 1

x = ab m=2
y = a n=1
→ 1

x = a m=1
y = ab n=2
→ 1

x = ab m=2
y = ab n=2
→ 2

x = abac n=5
y = abcdaf m=7
→ LCS length → Ans : t[n][m]

How to fill this table ?

Example 1:
x : a b a c -> m =4
y: a b c d a f -> n = 6

Output : 4

```
//Tabulation Code for LCS

int LCS( String x, String y, int n , int m )
{

    // base case
    for ( int i = 0; i<  n+1; i++ )
        for (int j =0; j< m+1; j++)
           if ( i == 0 || j ==0 )
              t[i][j] = 0;

    // choice diagram
      for ( int i = 1; i<  n+1; i++ )
        for (int j =1; j< m+1; j++)
           if ( x[i-1] == y[j-1] )
              t[i][j] = 1 + t [i-1][j-1]
           else
              t[i][j]= max( t [i][j-1] , t [i -1][j ] )

      return t[n][m]
}
```

Time complexity of LCS

Dynamic Programming

```
//Tabulation Code for LCS

int LCS( String x, String y, int n , int m )
{

    // base case
    for ( int i = 0; i<  n+1; i++ )
        for (int j =0; j< m+1; j++)
           if ( i == 0 || j ==0 )
              t[i][j] = 0;

    // choice diagram
      for ( int i = 1; i<  n+1; i++ )
        for (int j =1; j< m+1; j++)
             if ( x[i-1] == y[j-1] )
                 t[i][j] = 1 + t [i-1][j-1]
             else
                 t[i][j]= max( t [i][j-1] , t [i -1][j ] )

      return t[n][m]
}
```

Complexity – n^2

Example 1:
x : a b a c -> m =4
y: a b c d a f -> n = 6

Output : 4

Recursion

Now lets write the entire code

```
int LCS( String x, String y, int n , int m )
{
    // base case
    if ( n == 0 || m == 0 )
           return 0;

    // choice diagram
     if ( x[n-1] == y[m-1] )
        return 1 + LCS(x, y, n-1, m-1)
     else
        return max( LCS ( x, y, n , m-1) ,
LCS (x, y, n -1, m) )

}
```

Complexity – 2^n

Problems based on LCS

- Longest common substring
- Print LCS
- Shortest common super sequence
- Print SCS
- Min # of insertion and deletions A ->B Longest Repeating Subsequence
- Length of longest subsequence of A which is a substring in B
- Subsequence pattern matching
- Count how many times A appear as subsequence in B
- Longest Palindromic Subsequence
- Longest Palindromic Substring
- Count of Palindromic Substring
- Min # of Deletion in a string to make it a Palindrome
- Min # of insertion in a string to make it a Palindrome

Mid-Term Questions from DP

Subjective Questions from DP

1. MCM, Input for MCM will be given to you Input: p[] = {5, 4, 6, 2, 7}
   Write the recursive function int MatrixChainMultiplication (arr, I, j)
   Draw the recursive function tree and label the return Values

1. MCM, Input for MCM will be given to you Input: p[] = {5, 4, 6, 2, 7}
   Draw the DP table and fill in the values in the DP table for each sub problem

Note: similar kind of question can be asked for Knapsack and LCS as well