Codeforces

DYANAMIC PROGRAMMING

AJAY KUMAR

CONTENTS

STANDARD DP PROBLEMS AND THEIR APPROACH

KNAPSACK PROBLEM	Problem Description, approach, Tabulation Method
LONGEST COMMON SUBSEQUENCE	Problem Solution Approach Memoziation and Tabulation
LONGEST INCREASING SEQUENCE	
NLOGN SOLN OF LIS	
MATRIX CHAIN MULTIPLICATION	
KADANES ALGORITHM	

0/1

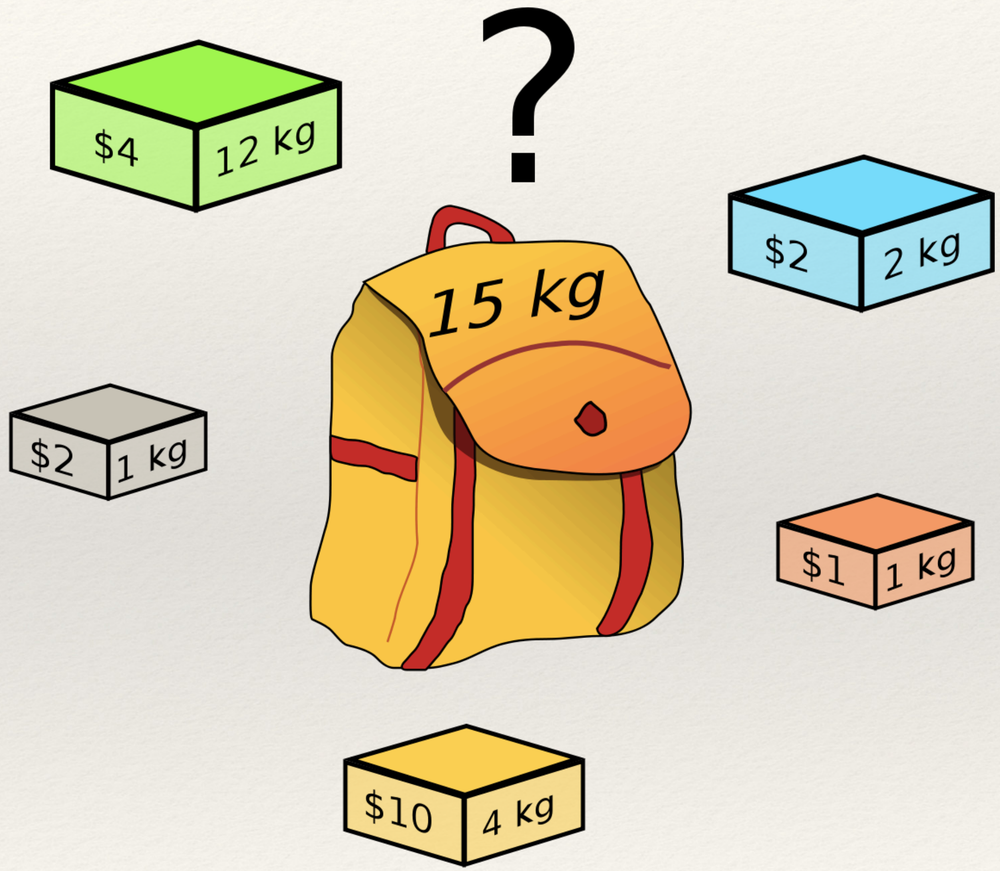
KNAPSACK PROBLEM

PICK OR NOT PICK

0/1 KNAPSACK

Given a Knapsack/Bag with W weight capacity and a list of N items with given v_i value and w_i weight.

Put these items in the knapsack in order to maximise the value of all the placed items without exceeding the limit of the Knapsack.



PROBLEM

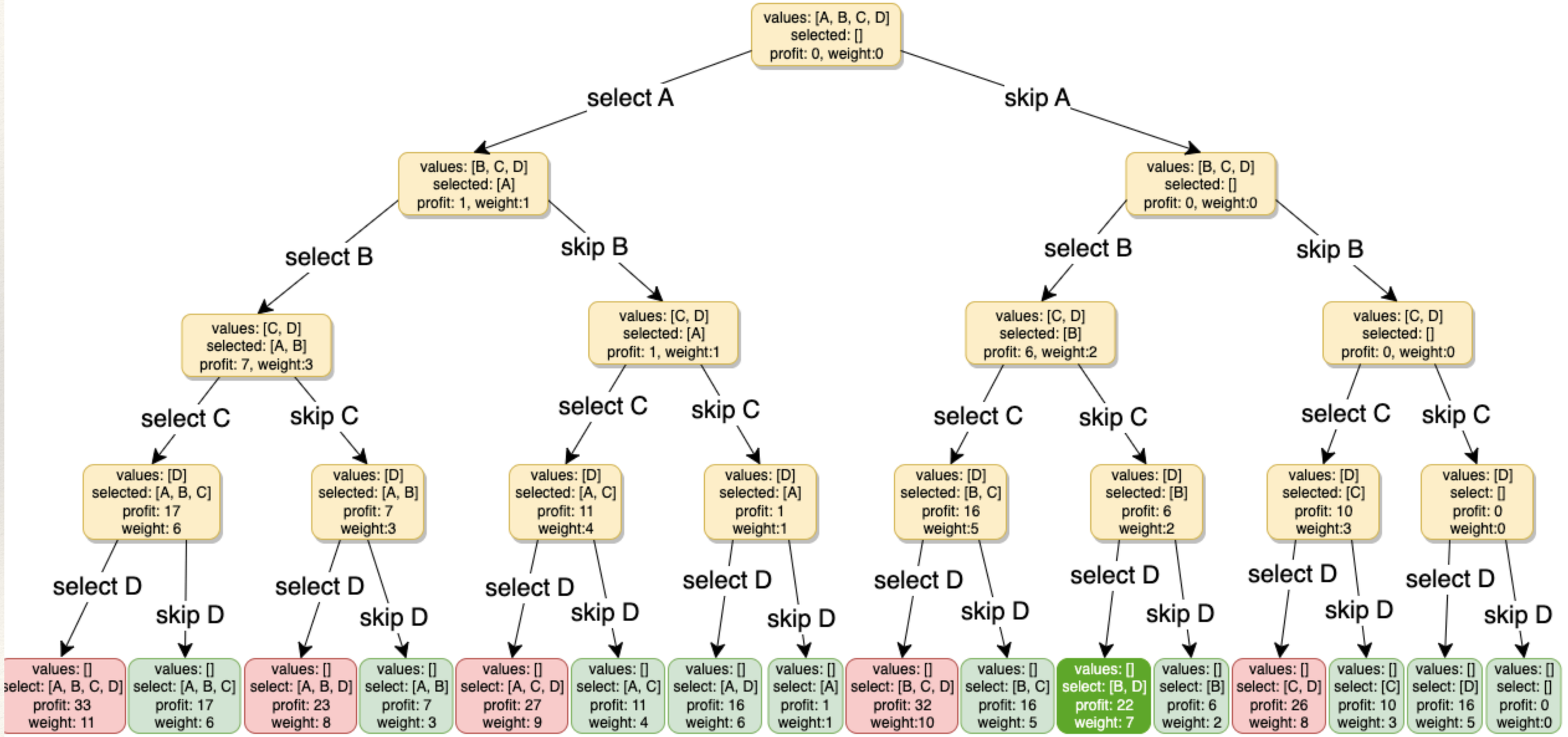
- We are given a knapsack of capacity c and a set of n objects numbered $1,2,\ldots,n$. Each object i has weight w_i and profit p_i.
- Let $v = [v_1, v_2, \ldots, v_n]$ be a solution vector in which $v_i = 0$ if object i is not in the knapsack, and $v_i = 1$ if it is in the knapsack.
- The goal is to find a subset of objects to put into the knapsack so $\sum_{i=1}^{n} w_i v_i \leq c$ (that is, the objects fit into the knapsack) and $\sum_{i=1}^{n} p_i v_i$ is maximized (that is, the profit is maximized).

Brute force

- A basic brute-force solution could be to try all combinations of the given items
- (as we did below), allowing us to choose the one with maximum profit and a weight that doesn't exceed 'C.'
- Take the example of four items (A, B, C, and D), as shown in the diagram below. To try all the combinations, our algorithm will look like:

items	A	B	C	D
profit	1	6	10	16
weight	1	2	3	5

Capacity: 7



Solution

- The naive method is to consider all 2^n possible subsets of the n objects and choose the one that fits into the knapsack and maximizes the profit.
- Let $F[i,x]$ be the maximum profit for a knapsack of capacity x using only objects $\{1,2,\ldots,i\}$. The DP formulation is:

$$F[i,x] = \begin{cases} 0 & x \geq 0, i = 0 \\ -\infty & x < 0, i = 0 \\ \max\{F[i-1,x], (F[i-1,x-w_i] + p_i)\} & 1 \leq i \leq n \end{cases}$$

```
class Solution
{
    public:
    //Function to return max value that can be put in knapsack of capacity W.
    int dp[1002][1002];
    int ans(int n,int w,int wt[],int val[]){
        if(n<0)return 0;
        if(dp[n][w]≠-1)return dp[n][w];
        if(w<wt[n])return dp[n][w]=ans(n-1,w,wt,val);
        else return dp[n][w]=max(val[n]+ans(n-1,w-wt[n],wt,val),ans(n-1,w,wt,val));
    }
    int knapSack(int W, int wt[], int val[], int n)
    {
       memset(dp,-1,sizeof(dp));
       return ans(n-1,W,wt,val);
    }
};
```

```
class Solution
{
    public:
    int knapSack(int W, int wt[], int val[], int n) {
        int dp[1001][1001] ;
        for(int i=0 ; i<=W ; i++){
            dp[0][i] = 0 ;
        }
        for(int j=0 ; j<=n ; j++){
            dp[j][0] = 0 ;
        }
        for(int i=1 ; i<=n ; i++){
            for(int j=1 ; j<=W ; j++){
                if(wt[i-1]<=j){
                    dp[i][j] = max(val[i-1]+dp[i-1][j-wt[i-1]],dp[i-1][j]) ;
                }
                else{
                    dp[i][j] = dp[i-1][j] ;
                }
            }
        }
        return dp[n][W] ;
    }
};
```

Complexity

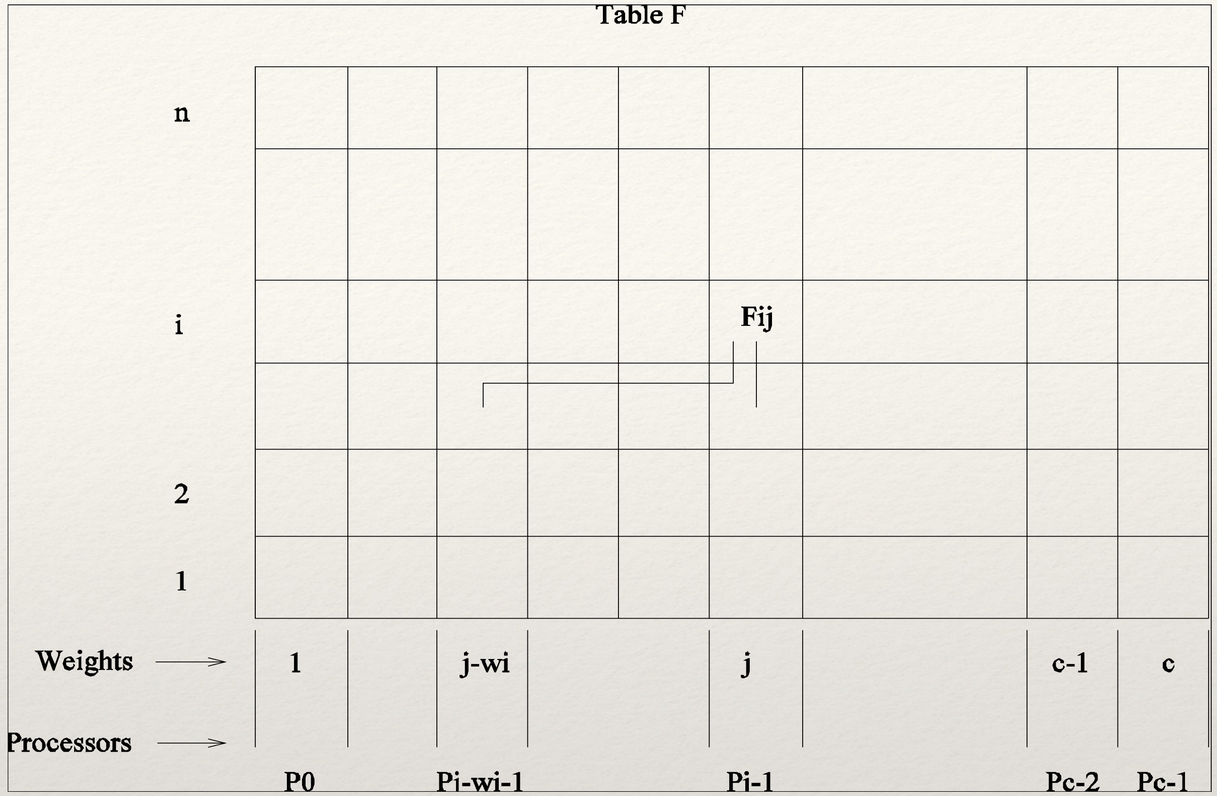
By Tabular Method

- Construct a table F of size $n \times c$ in row-major order.
- Filling an entry in a row requires two entries from the previous row: one from the same column and one from the column offset by the weight of the object corresponding to the row.
- Computing each entry takes constant time; the sequential run time of this algorithm is $\Theta(nc)$.

By Recursion

- The above algorithm will be using $O(N*C)$ space for the memoization array. Other than that, we will use O(N) space for the recursion call-stack. So the total space complexity will be O(N*C + N), which is asymptotically equivalent to O(N*C)

1. SUBSET SUM-(0/1 KNAPSACK)
2. COUNT OF SUBSET SUM-(0/1 KNAPSACK)
3. EQUAL SUM PARTITION-(0/1 KNAPSACK)
4. MINIMUM SUBSET SUM DIFFERENCE-(0/1 KNAPSACK)
5. TARGET SUM-(0/1 KNAPSACK)
6. NUMBER OF SUBSET WITH GIVEN DIFFERENCE-(0/1 KNAPSACK)
7. UNBOUNDED KNAPSACK-(BASIC)
8. ROD CUTTING PROBLEM (UNBOUNDED KNAPSACK)
9. Coin Change-1-(Maximum Number of Ways)-(Unbounded Knapsack)
10. Coin Change-2-(Minimum Number of Coins)-(Unbounded Knapsack)
11. Coin Change-3-(Maximum Number of Coins)-(Unbounded Knapsack)-(Maximize the Cut Segments)



02

LONGEST COMMON SUBESECUENSE

STRING DP

Longest Commmon subsequence

- we need to first know the number of possible different subsequences of a string with length n,
- i.e., find the number of subsequences with lengths ranging from 1,2,..n-1.
- Recall from theory of permutation and combination that number of combinations with 1 element are ${}^{n}C_{1}$. Number of combinations with 2 elements are ${}_{n}C^{2}$ and so forth and so on.
- We know that $nC^{0} + nC^{1} + nC^{2} + \ldots nC^{n} = 2^{n}$ So a string of length n has 2^{n}-1 different possible subsequences since we do not consider the subsequence with length 0.
- This implies that the time complexity of the brute force approach will be $O(n * 2^{n})$. Note that it takes O(n) time to check if a subsequence is common to both the strings

Algorithm

- Let *F[i,j]* denote the length of the longest common subsequence of the first *i* elements of *A* and the first *j* elements of *B*. The objective of the LCS problem is to find *F[n,m]*.
- We can write:

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

-

Optimal Substructure

- Let the input sequences be X[0..m-1] and Y[0..n-1] of lengths m and n respectively. And let L(X[0..m-1], Y[0..n-1]) be the length of LCS of the two sequences X and Y. Following is the recursive definition of L(X[0..m-1], Y[0..n-1]).
- If last characters of both sequences match (or X[m-1] == Y[n-1]) then L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])
- If last characters of both sequences do not match (or X[m-1] != Y[n-1]) then L(X[0..m-1], Y[0..n-1]) = MAX (L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2]))
- Examples:
  1) Consider the input strings "AGGTAB" and "GXTXAYB". Last characters match for the strings. So length of LCS can be written as:
  L("AGGTAB", "GXTXAYB") = 1 + L("AGGTA", "GXTXAY")

```
class Solution{
    public:
    int solver(int i,int j,string &str1,string & str2,vector<vector<int>>&dp){
        if(i<0 || j<0){
            return 0;
        }
        if(dp[i][j]≠-1){
            return dp[i][j];
        }
        if(str1[i]==str2[j]){
            return dp[i][j]=1+solver(i-1,j-1,str1,str2,dp);
        }
        return dp[i][j]=max(solver(i-1,j,str1,str2,dp),solver(i,j-1,str1,str2,dp));
    }
    int lcs(int x, int y, string s1, string s2){
        vector<vector<int>>dp(x,vector<int>(y,-1));
        return solver(x-1,y-1,s1,s2,dp);
    }
};
```

Tabular Method of Solving Longest Common Subsequence

- Consider the LCS of two amino-acid sequences H E A G A W G H E E and P A W H E A E.
- For the interested reader, the names of the corresponding amino-acids are A: Alanine, E: Glutamic acid, G: Glycine, H: Histidine, P: Proline, and W: Tryptophan.
- The *F* table for computing the LCS of the sequences. The LCS is A W H E E.

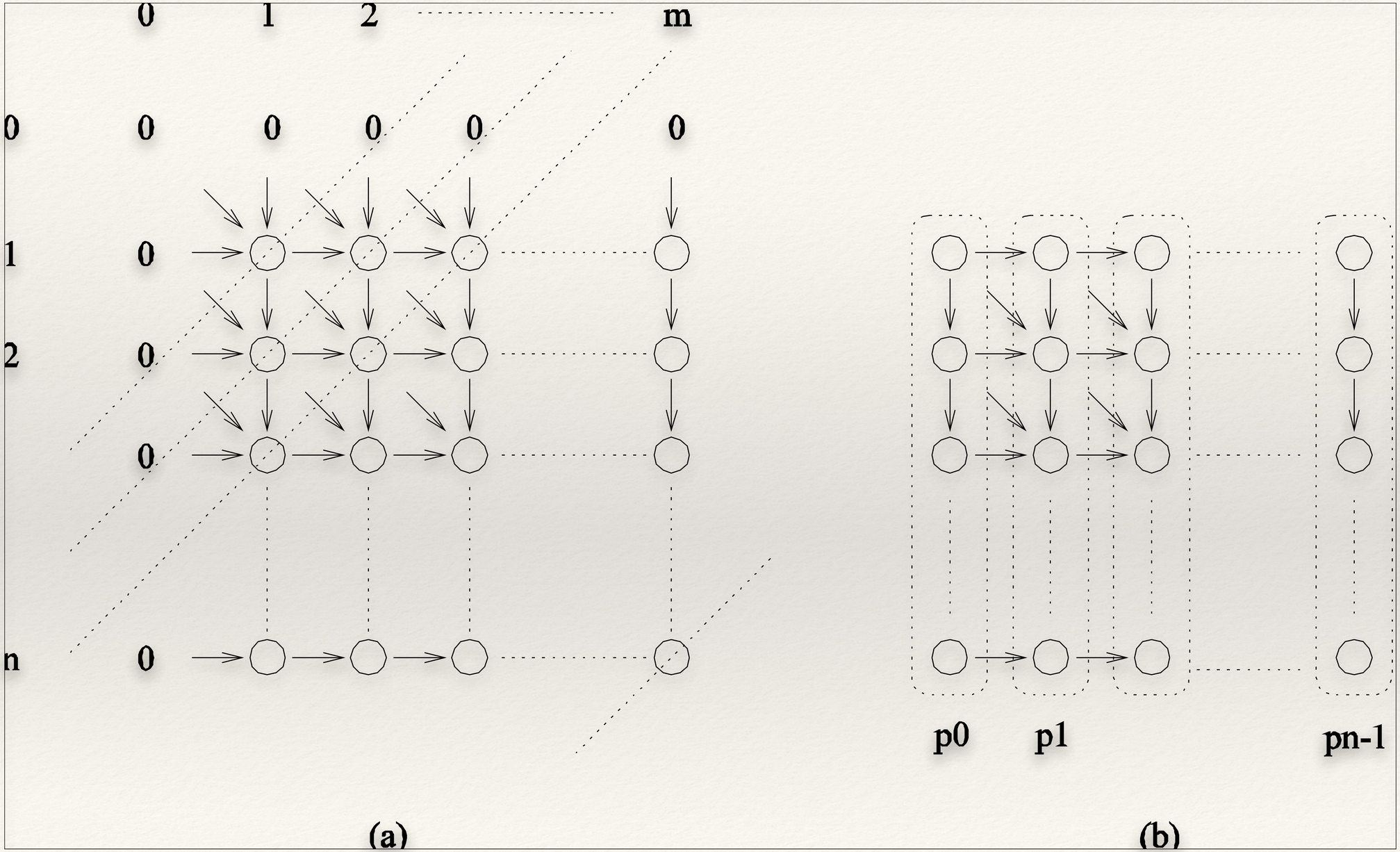
		H	E	A	G	A	W	G	H	E	E
	0	0	0	0	0	0	0	0	0	0	0
P	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	1	1	1	1	1	1	1	1
W	0	0	0	1	1	1	2	2	2	2	2
H	0	1	1	1	1	1	2	2	3	3	3
E	0	1	2	2	2	2	2	2	3	4	4
A	0	1	2	3	3	3	3	3	3	4	4
E	0	1	2	3	3	3	3	3	3	4	5

```
int lcs(char *X, char *Y, int m, int n){
    // intitalizing a matrix of size (m+1)*(n+1)
    int L[m + 1][n + 1];

    /* Following steps build L[m+1][n+1] in bottom up fashion. Note
        that L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1] */
    for(int i = 0; i ⩽ m; i++){
        for(int j = 0; j ⩽ n; j++){
            if(i == 0 || j == 0)
                L[i][j] = 0;

            else if(X[i - 1] == Y[j - 1])
                L[i][j] = L[i - 1][j - 1] + 1;

            else
                L[i][j] = max(L[i - 1][j], L[i][j - 1]);
        }
    }
    // L[m][n] contains length of LCS for X[0..n-1] and Y[0..m-1]
    return L[m][n];
}
```


0
1
2
m
0
0
0
0
0
0
1
0
2
0
0
n
0
p0
p1
pn-1
(a)
(b)

03

LONGEST INCREASING SUBESECUENSE

APPLICATION OF LCS

Longest Increase sequence

Dynamic Programming.

- If we closely observe the problem then we can convert this problem to longest Common Subsequence Problem.
- Firstly we will create another array of unique elements of original array and sort it. Now the longest increasing subsequence of our array must be present as a subsequence in our sorted array. That's why our problem is now reduced to finding the common subsequence between the two arrays.

Eg. arr =[50,3,10,7,40,80]

// Sorted array

arr1 = [3,7,10,40,50,80]

// LIS is longest common subsequence between the two arrays

ans = 4

The longest increasing subsequence is {3, 7, 40, 80}

Method 2-NlogN solution

1. If A[i] is smallest among all *end* candidates of active lists, we will start new active list of length 1.

2. If A[i] is largest among all *end candidates of* active lists, we will clone the largest active, list, and extend it by A[i].

3. If A[i] is in between, we will find a list with largest end element that is smaller than A[i]. Clone and extend this list by A[i]. We will discard all other lists of same length as that of this modified list

A[0] = 0. Case 1. There are no active lists, create one.
0.

A[1] = 8. Case 2. Clone and extend.
0.
0, 8.

A[2] = 4. Case 3. Clone, extend and discard.
0.
0, 4.
0, 8. Discarded

A[3] = 12. Case 2. Clone and extend.
0.
0, 4.
0, 4, 12.

A[4] = 2. Case 3. Clone, extend and discard.
0.
0, 2.
0, 4. Discarded.
0, 4, 12.
❖

```cpp
class Solution{
public:
    int maxLength(vector<int>s){
        int n=s.size();
        vector<int>v;
        for(int i=0;i<n;i++){
            auto it=lower_bound(v.begin(),v.end(),s[i]);
            if(it==v.end())v.push_back(s[i]);
            else {
                int pos=lower_bound(v.begin(),v.end(),s[i])-v.begin();
                v[pos]=s[i];
            }
        }

        return v.size();
    }
};
```

04

MATRIX CHAIN MULTIPLICATION

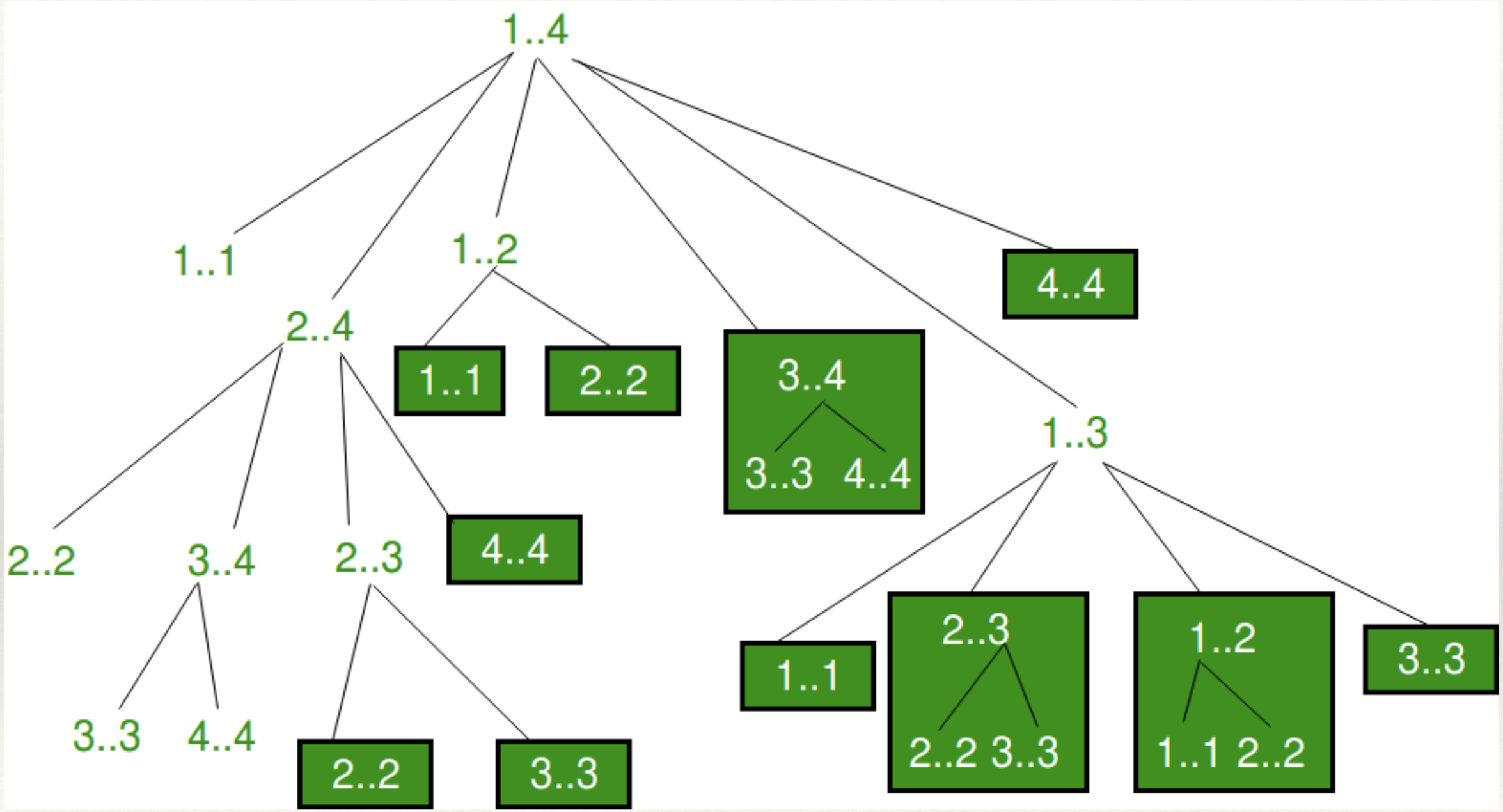
MATRIX DP

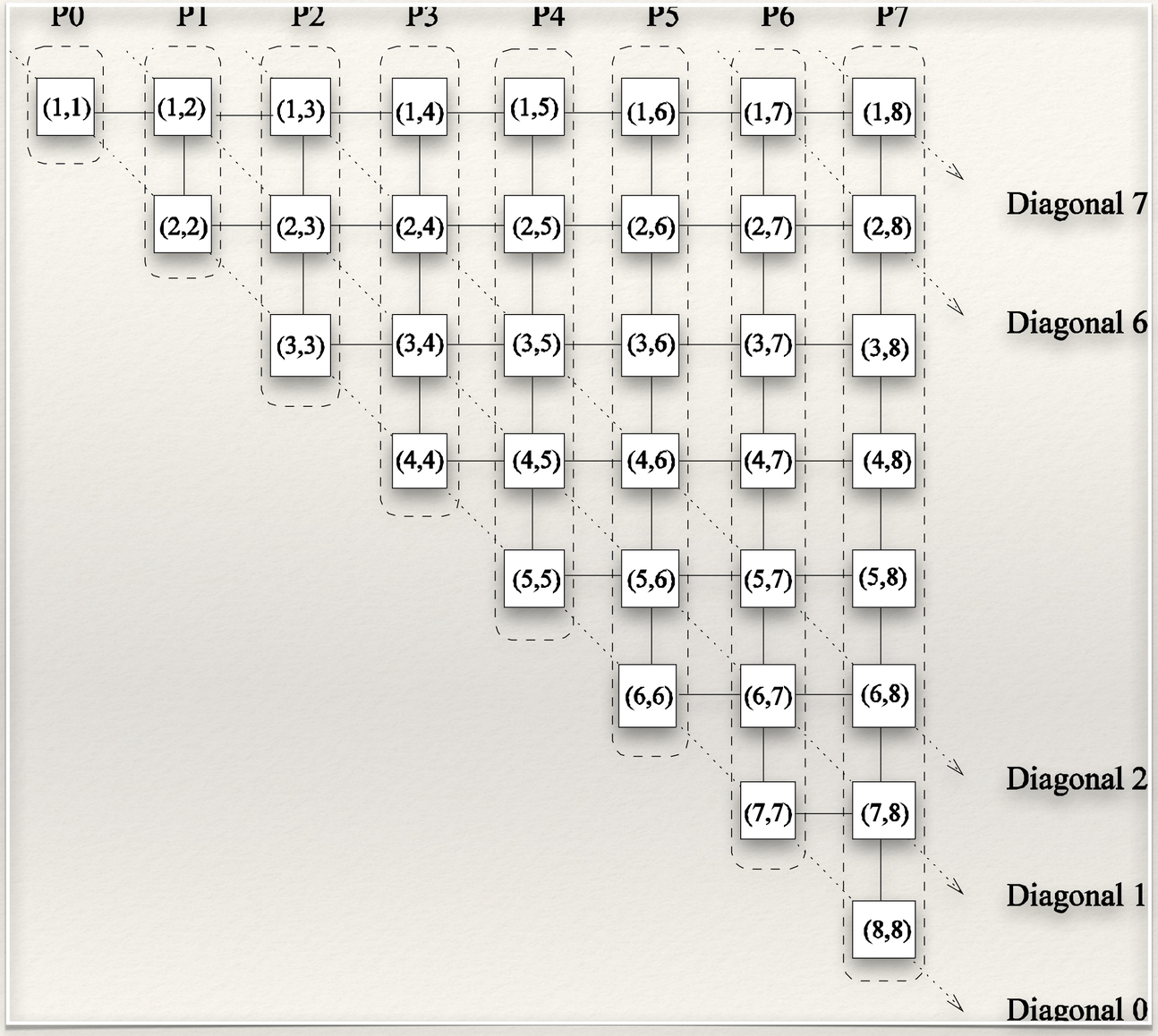
Matrix Chain Multiplication

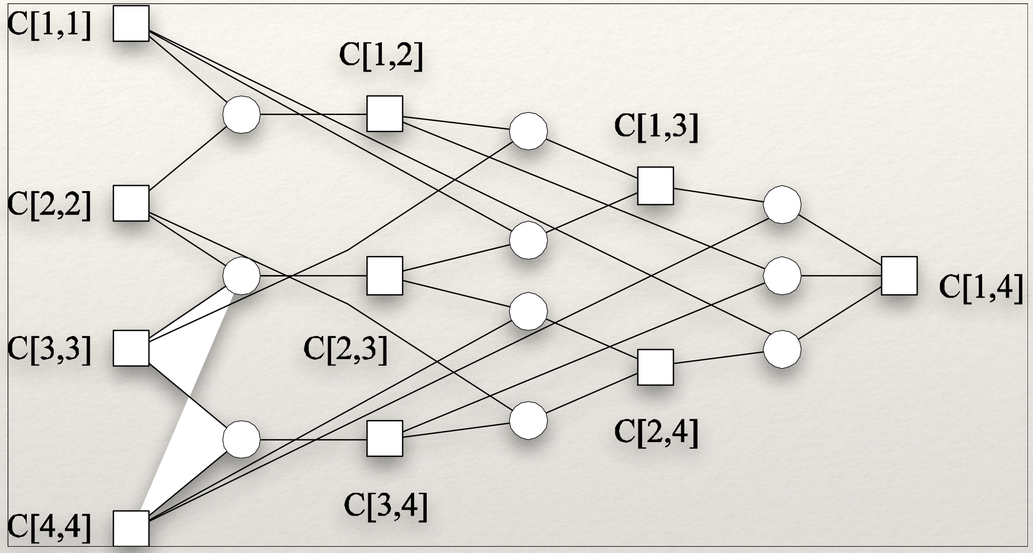
- When multiplying a sequence of matrices, the order of multiplication significantly impacts operation count.
- Let *C[i,j]* be the optimal cost of multiplying the matrices $A_i, \ldots A_j$.
- The chain of matrices can be expressed as a product of two smaller chains, $A_i, A_{i+1}, \ldots, A_k$ and $A_{k+1}, \ldots, A_j$.
- The chain $A_i, A_{i+1}, \ldots, A_k$ results in a matrix of dimensions $r_{i-1} \times r_k$, and the chain $A_{k+1}, \ldots, A_j$ results in a matrix of dimensions $r_k \times r_j$.
- The cost of multiplying these two matrices is $r_{i-1} r_k r_j$.

$$C[i,j] = \begin{cases} \min\limits_{i \le k < j} \{C[i,k] + C[k+1,j] + r_{i-1} r_k r_j\} & 1 \le i < j \le n \\ 0 & j = i, 0 < i \le n \end{cases}$$

-


1..4
1..1
2..4
1..2
4..4
2..2
3..4
2..3
4..4
1..1
2..2
3..4
3..3
4..4
1..3
3..3
4..4
2..2
3..3
1..1
2..3
2..2 3..3
1..2
1..1 2..2
3..3


P0
P1
P2
P3
P4
P5
P6
P7
(1,1)
(1,2)
(1,3)
(1,4)
(1,5)
(1,6)
(1,7)
(1,8)
(2,2)
(2,3)
(2,4)
(2,5)
(2,6)
(2,7)
(2,8)
(3,3)
(3,4)
(3,5)
(3,6)
(3,7)
(3,8)
(4,4)
(4,5)
(4,6)
(4,7)
(4,8)
(5,5)
(5,6)
(5,7)
(5,8)
(6,6)
(6,7)
(6,8)
(7,7)
(7,8)
(8,8)
Diagonal 7
Diagonal 6
Diagonal 2
Diagonal 1
Diagonal 0


C[1,1]
C[1,2]
C[1,3]
C[2,2]
C[1,4]
C[3,3]
C[2,3]
C[2,4]
C[3,4]
C[4,4]

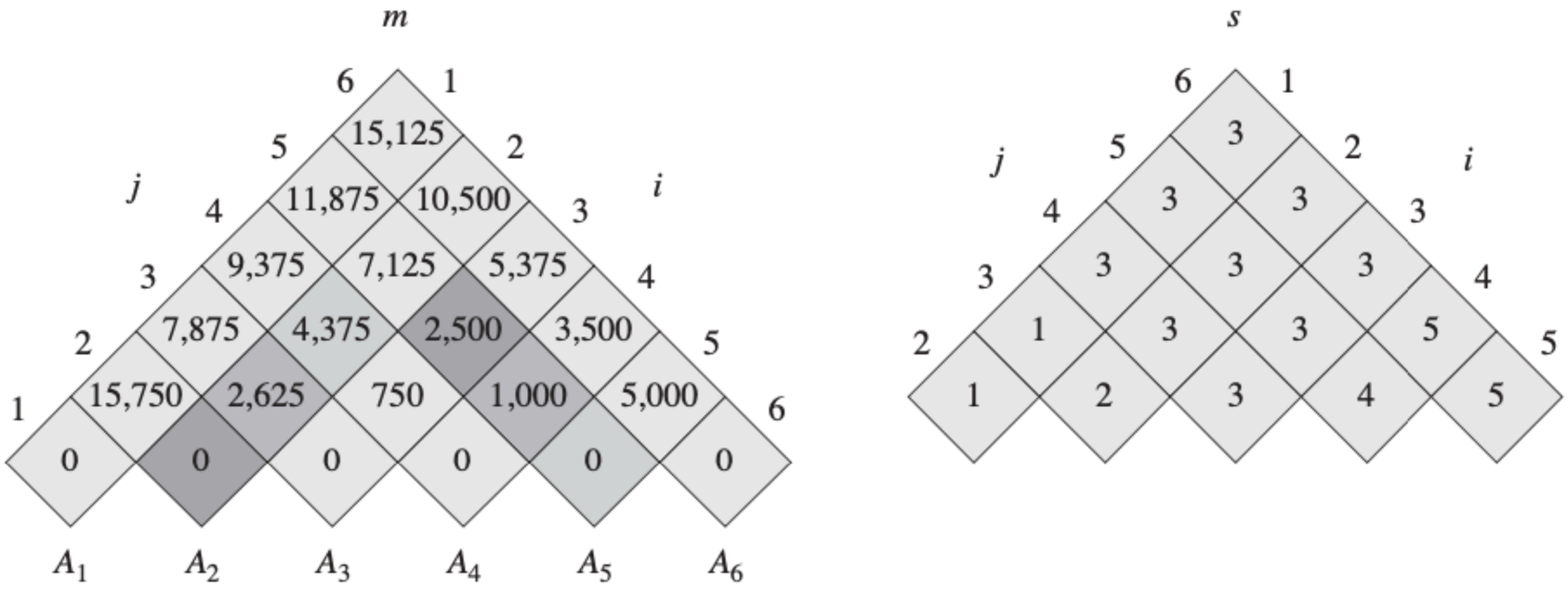


Figure 15.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

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

The tables are rotated so that the main diagonal runs horizontally. The m table uses only the main diagonal and upper triangle, and the s table uses only the upper triangle. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1, 6] = 15{,}125$. Of the darker entries, the pairs that have the same shading are taken together in line 10 when computing

$$
\begin{aligned}
m[2,5] &= \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 &= 0 + 2500 + 35 \cdot 15 \cdot 20 &= 13{,}000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 &= 2625 + 1000 + 35 \cdot 5 \cdot 20 &= 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 &= 4375 + 0 + 35 \cdot 10 \cdot 20 &= 11{,}375 \end{cases} \\
&= 7125\,.
\end{aligned}
$$

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

```
class Solution{
    public:
    int matrixMultiplication(int n,int arr[]){
    int dp[n+1][n+1];
    for(int i=1;i<n;i++){
        for(int j=1;j<n;j++){
            if(i==j){
                dp[i][j]=0;
            }else
            dp[i][j]=INT_MAX;
        }
    }
    for(int l=1;l<n;l++){
        for(int i=1;i<n-l;i++){
            int j=i+l;
            for(int k=i;k<j;k++){
                dp[i][j]=min(dp[i][j],dp[i][k]+dp[k+1][j]+arr[i-1]*arr[k]*arr[j]);
            }
        }
    }
    return dp[1][n-1];
  }
};
// Code implementation of above pseudo code
```

```
class Solution{
    public:
    int matrixMultiplication(int n,int arr[]){
    int dp[n+1][n+1];
    for(int i=1;i<n;i++){
        for(int j=1;j<n;j++){
            dp[i][j]=INT_MAX;
        }
    }
    for(int i=n-1;i>0;i--){
        for(int j=i;j<n;j++){
            if(i==j){
                dp[i][j]=0;
            }else{
                for(int k=i;k<j;k++){
                    dp[i][j]=min(dp[i][j],dp[i][k]+dp[k+1][j]+arr[i-1]*arr[k]*arr[j]);
                }
            }
        }
    }
       return dp[1][n-1];
    }
};
// Your can try in this way also
```

```cpp
class Solution{
public:
    int solve(int i,int j,int arr[],vector<vector<int>>&dp){
        if(i==j)return 0;
        if(dp[i][j]!=-1)return dp[i][j];
        int ans=INT_MAX;
        for(int k=i;k<j;k++){
            int temp_ans=solve(i,k,arr,dp)+solve(k+1,j,arr,dp)+arr[k]*arr[j]*arr[i-1];
            ans=min(temp_ans,ans);
        }
        return dp[i][j]=ans;
    }
    int matrixMultiplication(int N, int arr[])
    {
        // code here
        vector<vector<int>>dp(N+1,vector<int>(N+1,-1));
        return solve(1,N-1,arr,dp);
    }
};
// recursive solution also works
```

Dropping Egg problem

Given a certain amount of floors of a building (say f number of floors) and also given certain amount of eggs (say e number of eggs) …

What is the **least amount of egg drops** one has to perform to find out the **threshold floor**? (Threshold floor is one from which the egg starts breaking and also egg breaks for all the floors above. Also, if egg dropped from any floor below the threshold floor, it won't break.)

❖

Eggs\Floors	0	1	2	3	4	5	6
1	0	1	2	3	4	5	6
2	0	1					
3	0	1					

Base case is for 1 egg no of floor= no of chance

>>> eggDrop(e,f) = 1+min{max(eggDrop(e-1,k-1) , eggDrop(e,f-k)) , k in 1:f }

/ / Think we the way that either if we drop the egg from the kth floor it will break or it will not

If it will break then the egg-1 for k-1 floor and egg for the rest of the floor.

Eggs\Floors	0	1	2	3	4	5	6
1	0	1	2	3	4	5	6
2	0	1	2	2	3	3	3
3	0	1	2	2	3	3	3

```
int eggDrop(int n, int k){
    int eggFloor[n + 1][k + 1];
    int res;
    int i, j, x;
    for (i = 1; i <= n; i++) {
        eggFloor[i][1] = 1;
        eggFloor[i][0] = 0;
    }
    for (j = 1; j <= k; j++) eggFloor[1][j] = j;

    for (i = 2; i <= n; i++) {
        for (j = 2; j <= k; j++) {
            eggFloor[i][j] = INT_MAX;
            for (x = 1; x <= j; x++) {
                res = 1+ max(eggFloor[i - 1][x - 1],eggFloor[i][j - x]);
                if (res < eggFloor[i][j])
                    eggFloor[i][j] = res;
            }
        }
    }
    return eggFloor[n][k];
}
```

```
class Solution{
public:
    int solve(int i,int j,int arr[],vector<vector<int>>&dp){
        if(i==j)return 0;
        if(dp[i][j]!=-1)return dp[i][j];
        int ans=INT_MAX;
        for(int k=i;k<j;k++){
            int temp_ans=solve(i,k,arr,dp)+solve(k+1,j,arr,dp)+arr[k]*arr[j]*arr[i-1];
            ans=min(temp_ans,ans);
        }
        return dp[i][j]=ans;
    }
    int matrixMultiplication(int N, int arr[])
    {
        // code here
        vector<vector<int>>dp(N+1,vector<int>(N+1,-1));
        return solve(1,N-1,arr,dp);
    }
};
```

06

KADANE’S ALGORITHMS

MAXIMUM SUBARRAY SUM

Kadane's Algorithms

```
Initialize:
    max_so_far = INT_MIN
    max_ending_here = 0
Loop for each element of the array
  (a) max_ending_here = max_ending_here + a[i]
  (b) if(max_so_far < max_ending_here)
            max_so_far = max_ending_here
  (c) if(max_ending_here < 0)
            max_ending_here = 0
return max_so_far
```

```
int maxSubArraySum(int a[], int size)
{
    int max_so_far = INT_MIN, max_ending_here = 0;

    for (int i = 0; i < size; i++) {
        max_ending_here = max_ending_here + a[i];
        if (max_so_far < max_ending_here)
            max_so_far = max_ending_here;

        if (max_ending_here < 0)
            max_ending_here = 0;
    }
    return max_so_far;
}
```

07

LONGEST PALINDROMIC SUBSTRING

FIND MAXIMUM SUM OF SUBARRAY

Longest Common Substring

The brute force solution which comes into our mind is to pick all the substrings from a string that is possible and then we will check whether that substring is a palindrome or not.

Hence if we will talk about the implementation part of this approach then we just have to use two loops for finding all the substrings and then one more for checking whether the substance is a palindrome or not. Hence total time complexity of this approach will be N^3.

Efficient Approach-Dyanamic Programming

We can find some patterns in validating palindromes to avoid unnecessary re-computation and improve the brute force approach by memorizing the data.

Consider one example “cbebc”. If we already knew that “beb” is a palindrome, it is obvious that “cbebc” must be a palindrome since the two left and right end letters are the same.

$$P(i,j) = \begin{cases} \text{true}, & \text{if the substring } S_i \dots S_j \text{ is a palindrome} \\ \text{false}, & \text{otherwise.} \end{cases}$$

InterviewBit

$$P(i,j) = (P(i+1, j-1) \text{ and } S_i == S_j)$$

InterviewBit

$$P(i, i) = true$$

$$P(i, i+1) = (S_i == S_{i+1})$$

- Identify the subproblem then

```
int longestPalSubstr(string str)
{
    // get length of input string
    int n = str.size();

    // table[i][j] will be false if substring
    // str[i..j] is not palindrome.
    // Else table[i][j] will be true
    bool table[n][n];

    memset(table, 0, sizeof(table));

    // All substrings of length 1
    // are palindromes
    int maxLength = 1;

    for (int i = 0; i < n; ++i)
        table[i][i] = true;

    // check for sub-string of length 2.
    int start = 0;
    for (int i = 0; i < n - 1; ++i) {
        if (str[i] == str[i + 1]) {
            table[i][i + 1] = true;
            start = i;
            maxLength = 2;
        }

    }
    /for (int k = 3; k <= n; ++k) {
        // Fix the starting index
        for (int i = 0; i < n - k + 1; ++i) {
            // Get the ending index of substring from
            // starting index i and length k
            int j = i + k - 1;

            // checking for sub-string from ith index to
            // jth index if str[i+1] to str[j-1] is a
            // palindrome
            if (table[i + 1][j - 1] && str[i] == str[j]) {
                table[i][j] = true;

                if (k > maxLength) {
                    start = i;
                    maxLength = k;
                }
            }
        }
    }

    cout << "Longest palindrome substring is: ";
    printSubStr(str, start, start + maxLength - 1);

    // return length of LPS
    return maxLength;
}
```

How to write the tabulation code-striver

- Write down the base case
- Write down the changing states
- Copy the recurrence

05

PROBLEM INVOLVING THE GRIDS

Introduction

- Problem