

---

# DYNAMIC PROGRAMMING

CHAPTER

19



---

## 19.1 Introduction

In this chapter we will try to solve the problems for which we failed to get the optimal solutions using other techniques (say, *Divide & Conquer* and *Greedy* methods). Dynamic Programming (DP) is a simple technique but it can be difficult to master. One easy way to identify and solve DP problems is by solving as many problems as possible. The term *Programming* is not related to coding but it is from literature, and means filling tables (similar to *Linear Programming*).

## 19.2 What is Dynamic Programming Strategy?

Dynamic programming and memoization work together. The main difference between dynamic programming and divide and conquer is that in the case of the latter, sub problems are independent, whereas in DP there can be an overlap of sub problems. By using memoization [maintaining a table of sub problems already solved], dynamic programming reduces the exponential complexity to polynomial complexity ( $O(n^2)$ ,  $O(n^3)$ , etc.) for many problems. The major components of DP are:

- Recursion: Solves sub problems recursively.

- Memoization: Stores already computed values in table (*Memoization* means caching).

*Dynamic Programming = Recursion + Memoization*

### 19.3 Properties of Dynamic Programming Strategy

The two dynamic programming properties which can tell whether it can solve the given problem or not are:

- *Optimal substructure*: an optimal solution to a problem contains optimal solutions to sub problems.
- *Overlapping sub problems*: a recursive solution contains a small number of distinct sub problems repeated many times.

### 19.4 Can Dynamic Programming Solve All Problems?

Like Greedy and Divide and Conquer techniques, DP cannot solve every problem. There are problems which cannot be solved by any algorithmic technique [Greedy, Divide and Conquer and Dynamic Programming].

The difference between Dynamic Programming and straightforward recursion is in memoization of recursive calls. If the sub problems are independent and there is no repetition then memoization does not help, so dynamic programming is not a solution for all problems.

### 19.5 Dynamic Programming Approaches

Basically there are two approaches for solving DP problems:

- Bottom-up dynamic programming
- Top-down dynamic programming

#### Bottom-up Dynamic Programming

In this method, we evaluate the function starting with the smallest possible input argument value and then we step through possible values, slowly increasing the input argument value. While computing the values we store all computed values in a table (memory). As larger arguments are evaluated, pre-computed values for smaller arguments can be used.

#### Top-down Dynamic Programming

In this method, the problem is broken into sub problems; each of these sub problems is solved; and the solutions remembered, in case they need to be solved. Also, we save each computed value as the final action of the recursive function, and as the first action we check if pre-computed value exists.

## Bottom-up versus Top-down Programming

In bottom-up programming, the programmer has to select values to calculate and decide the order of calculation. In this case, all sub problems that might be needed are solved in advance and then used to build up solutions to larger problems. In top-down programming, the recursive structure of the original code is preserved, but unnecessary recalculation is avoided. The problem is broken into sub problems, these sub problems are solved and the solutions remembered, in case they need to be solved again.

**Note:** Some problems can be solved with both the techniques and we will see examples in the next section.

## 19.6 Examples of Dynamic Programming Algorithms

- Many string algorithms including longest common subsequence, longest increasing subsequence, longest common substring, edit distance.
- Algorithms on graphs can be solved efficiently: Bellman-Ford algorithm for finding the shortest distance in a graph, Floyd's All-Pairs shortest path algorithm, etc.
- Chain matrix multiplication
- Subset Sum
- 0/1 Knapsack
- Travelling salesman problem, and many more

## 19.7 Understanding Dynamic Programming

Before going to problems, let us understand how DP works through examples.

### Fibonacci Series

In Fibonacci series, the current number is the sum of previous two numbers. The Fibonacci series is defined as follows:

$$\begin{aligned} \text{Fib}(n) &= 0, & \text{for } n &= 0 \\ &= 1, & \text{for } n &= 1 \\ &= \text{Fib}(n-1) + \text{Fib}(n-2), & \text{for } n &> 1 \end{aligned}$$

The recursive implementation can be given as:

```
int RecursiveFibonacci(int n) {  
    if(n == 0) return 0;  
    if(n == 1) return 1;  
    return RecursiveFibonacci(n - 1) + RecursiveFibonacci(n - 2);  
}
```

Solving the above recurrence gives:

$$T(n) = T(n-1) + T(n-2) + 1 \approx \left(\frac{1+\sqrt{5}}{2}\right)^n \approx 2^n = O(2^n)$$

**Note:** For proof, refer to [Introduction](#) chapter.

### How does Memoization help?

Calling *fib*(5) produces a call tree that calls the function on the same value many times:

*fib*(5)  
*fib*(4) + *fib*(3)  
(*fib*(3) + *fib*(2)) + (*fib*(2) + *fib*(1))  
((*fib*(2) + *fib*(1)) + (*fib*(1) + *fib*(0))) + ((*fib*(1) + *fib*(0)) + *fib*(1))  
(((*fib*(1) + *fib*(0)) + *fib*(1)) + (*fib*(1) + *fib*(0))) + ((*fib*(1) + *fib*(0)) + *fib*(1))

In the above example, *fib*(2) was calculated three times (overlapping of subproblems). If *n* is big, then many more values of *fib* (sub problems) are recalculated, which leads to an exponential time algorithm. Instead of solving the same sub problems again and again we can store the previous calculated values and reduce the complexity.

*Memoization* works like this: Start with a recursive function and add a table that maps the function's parameter values to the results computed by the function. Then if this function is called twice with the same parameters, we simply look up the answer in the table.

**Improving:** Now, we see how DP reduces this problem complexity from exponential to polynomial. As discussed earlier, there are two ways of doing this. One approach is bottom-up: these methods start with lower values of input and keep building the solutions for higher values.

```

int fib[n];
int fib(int n) {
    // Check for base cases
    if(n == 0 || n == 1) return 1;
    fib[0] = 1;
    fib[1] = 1;
    for (int i = 2; i < n; i++)
        fib[i] = fib[i - 1] + fib[i - 2];
    return fib[n - 1];
}

```

The other approach is top-down. In this method, we preserve the recursive calls and use the values if they are already computed. The implementation for this is given as:

```

int fib[n];
int fibonacci( int n ) {
    if(n == 1)
        return 1;
    if(n == 2)
        return 1;
    if( fib[n] != 0) return fib[n] ;
    return fib[n] = fibonacci(n-1) + fibonacci(n -2) ;
}

```

**Note:** For all problems, it may not be possible to find both top-down and bottom-up programming solutions.

Both versions of the Fibonacci series implementations clearly reduce the problem complexity to  $O(n)$ . This is because if a value is already computed then we are not calling the subproblems again. Instead, we are directly taking its value from the table.

Time Complexity:  $O(n)$ . Space Complexity:  $O(n)$ , for table.

**Further Improving:** One more observation from the Fibonacci series is: The current value is the sum of the previous two calculations only. This indicates that we don't have to store all the previous values. Instead, if we store just the last two values, we can calculate the current value. The implementation for this is given below:

```

int fibonacci(int n) {
    int a = 0, b = 1, sum, i;
    for (i=0; i < n; i++) {
        sum = a + b;
        a = b;
        b = sum;
    }
    return sum;
}

```

Time Complexity:  $O(n)$ . Space Complexity:  $O(1)$ .

**Note:** This method may not be applicable (available) for all problems.

## Observations

While solving the problems using DP, try to figure out the following:

- See how the problems are defined in terms of subproblems recursively.
- See if we can use some table [memoization] to avoid the repeated calculations.

## Factorial of a Number

As another example, consider the factorial problem:  $n!$  is the product of all integers between  $n$  and 1. The definition of recursive factorial can be given as:

$$\begin{aligned}
 n! &= n * (n - 1)! \\
 1! &= 1 \\
 0! &= 1
 \end{aligned}$$

This definition can easily be converted to implementation. Here the problem is finding the value of  $n!$ , and the sub-problem is finding the value of  $(n - 1)!$ . In the recursive case, when  $n$  is greater than 1, the function calls itself to find the value of  $(n - 1)!$  and multiplies that with  $n$ . In the base case, when  $n$  is 0 or 1, the function simply returns 1.

```

int fact(int n) {
    if(n == 1) return 1;
    else if(n == 0) return 1;
    else // recursive case: multiply n by (n - 1) factorial
        return n * fact(n - 1);
}

```

The recurrence for the above implementation can be given as:  $T(n) = n \times T(n - 1) \approx O(n)$   
Time Complexity:  $O(n)$ . Space Complexity:  $O(n)$ , recursive calls need a stack of size  $n$ .

In the above recurrence relation and implementation, for any  $n$  value, there are no repetitive calculations (no overlapping of sub problems) and the factorial function is not getting any benefits with dynamic programming. Now, let us say we want to compute a series of  $m!$  for some arbitrary value  $m$ . Using the above algorithm, for each such call we can compute it in  $O(m)$ . For example, to find both  $n!$  and  $m!$  we can use the above approach, wherein the total complexity for finding  $n!$  and  $m!$  is  $O(m + n)$ .

Time Complexity:  $O(n + m)$ .

Space Complexity:  $O(\max(m, n))$ , recursive calls need a stack of size equal to the maximum of  $m$  and  $n$ .

**Improving:** Now let us see how DP reduces the complexity. From the above recursive definition it can be seen that  $fact(n)$  is calculated from  $fact(n - 1)$  and  $n$  and nothing else. Instead of calling  $fact(n)$  every time, we can store the previous calculated values in a table and use these values to calculate a new value. This implementation can be given as:

```
int facto[n];
int fact(int n) {
    if(n == 1) return 1;
    else if(n == 0)
        return 1;
    //Already calculated case
    else if(facto[n] != 0)
        return facto[n];
    else // recursive case: multiply n by (n - 1) factorial
        return facto[n] = n * fact(n - 1);
}
```

For simplicity, let us assume that we have already calculated  $n!$  and want to find  $m!$ . For finding  $m!$ , we just need to see the table and use the existing entries if they are already computed. If  $m < n$  then we do not have to recalculate  $m!$ . If  $m > n$  then we can use  $n!$  and call the factorial on the remaining numbers only.

The above implementation clearly reduces the complexity to  $O(\max(m, n))$ . This is because if the  $fact(n)$  is already there, then we are not recalculating the value again. If we fill these newly computed values, then the subsequent calls further reduce the complexity.

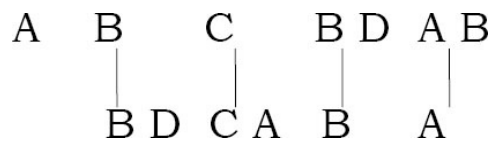
Time Complexity:  $O(\max(m, n))$ . Space Complexity:  $O(\max(m, n))$  for table.

## 19.8 Longest Common Subsequence

Given two strings: string  $X$  of length  $m$  [ $X(1..m)$ ], and string  $Y$  of length  $n$  [ $Y(1..n)$ ], find the longest common subsequence: the longest sequence of characters that appear left-to-right (but not necessarily in a contiguous block) in both strings. For example, if  $X = \text{"ABCBBDAB"}$  and  $Y = \text{"BDCABA"}$ , the  $LCS(X, Y) = \{\text{"BCBA"}, \text{"BDAB"}, \text{"BCAB"}\}$ . We can see there are several optimal solutions.

**Brute Force Approach:** One simple idea is to check every subsequence of  $X[1..m]$  ( $m$  is the length of sequence  $X$ ) to see if it is also a subsequence of  $Y[1..n]$  ( $n$  is the length of sequence  $Y$ ). Checking takes  $O(n)$  time, and there are  $2^m$  subsequences of  $X$ . The running time thus is exponential  $O(n \cdot 2^m)$  and is not good for large sequences.

**Recursive Solution:** Before going to DP solution, let us form the recursive solution for this and later we can add memoization to reduce the complexity. Let's start with some simple observations about the LCS problem. If we have two strings, say  $\text{"ABCBBDAB"}$  and  $\text{"BDCABA"}$ , and if we draw lines from the letters in the first string to the corresponding letters in the second, no two lines cross:



From the above observation, we can see that the current characters of  $X$  and  $Y$  may or may not match. That means, suppose that the two first characters differ. Then it is not possible for both of them to be part of a common subsequence - one or the other (or maybe both) will have to be removed. Finally, observe that once we have decided what to do with the first characters of the strings, the remaining sub problem is again a  $LCS$  problem, on two shorter strings. Therefore we can solve it recursively.

The solution to  $LCS$  should find two sequences in  $X$  and  $Y$  and let us say the starting index of sequence in  $X$  is  $i$  and the starting index of sequence in  $Y$  is  $j$ . Also, assume that  $X[i \dots m]$  is a substring of  $X$  starting at character  $i$  and going until the end of  $X$ , and that  $Y[j \dots n]$  is a substring of  $Y$  starting at character  $j$  and going until the end of  $Y$ .

Based on the above discussion, here we get the possibilities as described below:

- 1) If  $X[i] == Y[j]$  :  $1 + LCS(i + 1, j + 1)$
- 2) If  $X[i] \neq Y[j]$ .  $LCS(i, j + 1)$  // skipping  $j^{th}$  character of  $Y$
- 3) If  $X[i] \neq Y[j]$ .  $LCS(i + 1, j)$  // skipping  $i^{th}$  character of  $X$

In the first case, if  $X[i]$  is equal to  $Y[j]$ , we get a matching pair and can count it towards the total length of the  $LCS$ . Otherwise, we need to skip either  $i^{th}$  character of  $X$  or  $j^{th}$  character of  $Y$  and find the longest common subsequence. Now,  $LCS(i, j)$  can be defined as:



$$LCS(i, j) = \begin{cases} 0, & \text{if } i = m \text{ or } j = n \\ \text{Max}\{LCS(i, j + 1), LCS(i + 1, j)\}, & \text{if } X[i] \neq Y[j] \\ 1 + LCS[i + 1, j + 1], & \text{if } X[i] == Y[j] \end{cases}$$

LCS has many applications. In web searching, if we find the smallest number of changes that are needed to change one word into another. A *change* here is an insertion, deletion or replacement of a single character.

```
//Initial Call: LCSLength(X, 0, m-1, Y, 0, n-1);
int LCSLength( int X[], int i, int m, int Y[], int j, int n) {
    if (i == m || j == n)
        return 0;
    else if (X[i] == Y[j]) return 1 + LCSLength(X, i+1, m, Y, j+1, n);
    else return max( LCSLength(X, i+1, m, Y, j, n), LCSLength(X, i, m, Y, j+1, n));
}
```

This is a correct solution but it is very time consuming. For example, if the two strings have no matching characters, the last line always gets executed which gives (if  $m == n$ ) close to  $O(2^n)$ .

**DP Solution: Adding Memoization:** The problem with the recursive solution is that the same subproblems get called many different times. A subproblem consists of a call to `LCS_length`, with the arguments being two suffixes of  $X$  and  $Y$ , so there are exactly  $(i + 1)(j + 1)$  possible subproblems (a relatively small number). If there are nearly  $2^n$  recursive calls, some of these subproblems must be being solved over and over.

The DP solution is to check, whenever we want to solve a sub problem, whether we've already done it before. So we look up the solution instead of solving it again. Implemented in the most direct way, we just add some code to our recursive solution. To do this, look up the code. This can be given as:

```

int LCS[1024][1024];
int LCSLength( int X[], int m, int Y[], int n) {
    for( int i = 0; i <= m; i++ )
        LCS[i][n] = 0;
    for( int j = 0; j <= n; j++ )
        LCS[m][j] = 0;
    for( int i = m - 1; i >= 0; i-- ) {
        for( int j = n - 1; j >= 0; j-- ) {
            LCS[i][j] = LCS[i + 1][j + 1]; // matching X[i] to Y[j]
            if( X[i] == Y[j] )
                LCS[i][j]++; // we get a matching pair

            // the other two cases - inserting a gap
            if( LCS[i][j + 1] > LCS[i][j] )
                LCS[i][j] = LCS[i][j + 1];
            if( LCS[i + 1][j] > LCS[i][j] )
                LCS[i][j] = LCS[i + 1][j];
        }
    }
    return LCS[0][0];
}

```

First, take care of the base cases. We have created an *LCS* table with one row and one column larger than the lengths of the two strings. Then run the iterative DP loops to fill each cell in the table. This is like doing recursion backwards, or bottom up.

			$L[i][j]$	$L[i][j+1]$	
		$L[i+1][j]$		$L[i+1][j+1]$	

The value of  $LCS[i][j]$  depends on 3 other values ( $LCS[i + 1][j + 1]$ ,  $LCS[i][j + 1]$  and  $LCS[i + 1][j]$ ), all of which have larger values of  $i$  or  $j$ . They go through the table in the order of decreasing  $i$  and  $j$  values. This will guarantee that when we need to fill in the value of  $LCS[i][j]$ , we already know the values of all the cells on which it depends.

Time Complexity:  $O(mn)$ , since  $i$  takes values from 1 to  $m$  and  $j$  takes values from 1 to  $n$ .

Space Complexity:  $O(mn)$ .

**Note:** In the above discussion, we have assumed  $LCS(i,j)$  is the length of the  $LCS$  with  $X[i \dots m]$  and  $Y[j \dots n]$ . We can solve the problem by changing the definition as  $LCS(i,j)$  is the length of the  $LCS$  with  $X[1 \dots i]$  and  $Y[1 \dots j]$ .

**Printing the subsequence:** The above algorithm can find the length of the longest common subsequence but cannot give the actual longest subsequence. To get the sequence, we trace it through the table. Start at cell  $(0,0)$ . We know that the value of  $LC5[0][0]$  was the maximum of 3 values of the neighboring cells. So we simply recompute  $LC5[0][0]$  and note which cell gave the maximum value. Then we move to that cell (it will be one of  $(1,1)$ ,  $(0,1)$  or  $(1,0)$ ) and repeat this until we hit the boundary of the table. Every time we pass through a cell  $(i,j')$  where  $X[i] == Y[j]$ , we have a matching pair and print  $X[i]$ . At the end, we will have printed the longest common subsequence in  $O(mn)$  time.

An alternative way of getting path is to keep a separate table for each cell. This will tell us which direction we came from when computing the value of that cell. At the end, we again start at cell  $(0,0)$  and follow these directions until the opposite corner of the table.

From the above examples, I hope you understood the idea behind DP. Now let us see more problems which can be easily solved using the DP technique.

**Note:** As we have seen above, in DP the main component is recursion. If we know the recurrence then converting that to code is a minimal task. For the problems below, we concentrate on getting the recurrence.

## 19.9 Dynamic Programming: Problems & Solutions

**Problem-1** Convert the following recurrence to code.

$$T(0) = T(1) = 2$$
$$T(n) = \sum_{i=1}^{n-1} 2 \times T(i) \times T(i-1), \text{ for } n > 1$$

**Solution:** The code for the given recursive formula can be given as:

```

int f(int n) {
    int sum = 0;
    if(n==0 || n==1) //Base Case
        return 2;
    //recursive case
    for(int i=1; i < n; i++)
        sum += 2 * f(i) * f(i-1);
    return sum;
}

```

**Problem-2** Can we improve the solution to [Problem-1](#) using memoization of DP?

**Solution: Yes.** Before finding a solution, let us see how the values are calculated.

$$T(0) = T(1) = 2$$

$$T(2) = 2 * T(1) * T(0)$$

$$T(3) = 2 * T(1) * T(0) + 2 * T(2) * T(1)$$

$$T(4) = 2 * T(1) * T(0) + 2 * T(2) * T(1) + 2 * T(3) * T(2)$$

From the above calculations it is clear that there are lots of repeated calculations with the same input values. Let us use a table for avoiding these repeated calculations, and the implementation can be given as:

```

int f(int n) {
    T[0] = T[1] = 2;
    for(int i=2; i <= n; i++) {
        T[i] = 0;
        for (int j=1; j < i; j++)
            T[i] += 2 * T[j] * T[j-1];
    }
    return T[n];
}

```

Time Complexity:  $O(n^2)$ , two *for* loops. Space Complexity:  $O(n)$ , for table.

**Problem-3** Can we further improve the complexity of [Problem-2](#)?

**Solution: Yes,** since all sub problem calculations are dependent only on previous calculations, code can be modified as:

```

int f(int n) {
    T[0] = T[1] = 2;
    T[2] = 2 * T[0] * T[1];
    for(int i=3; i <= n; i++)
        T[i] = T[i-1] + 2 * T[i-1] * T[i-2];
    return T[n];
}

```

Time Complexity:  $O(n)$ , since only one *for* loop. Space Complexity:  $O(n)$ .

**Problem-4 Maximum Value Contiguous Subsequence:** Given an array of  $n$  numbers, give an algorithm for finding a contiguous subsequence  $A(i) \dots A(j)$  for which the sum of elements is maximum. **Example:**  $\{-2, 11, -4, 13, -5, 2\} \rightarrow 20$  and  $\{1, -3, 4, -2, -1, 6\} \rightarrow 7$

**Solution:**

**Input:** Array.  $A(1) \dots A(n)$  of  $n$  numbers.

**Goal:** If there are no negative numbers, then the solution is just the sum of all elements in the given array. If negative numbers are there, then our aim is to maximize the sum [there can be a negative number in the contiguous sum].

One simple and brute force approach is to see all possible sums and select the one which has maximum value.

```

int MaxContiguousSum(int A[], in n) {
    int maxSum = 0;
    for(int i = 0; i < n; i++)                // for each possible start point
        for(int j = i; j < n; j++)            // for each possible end point
            {
                int currentSum = 0;
                for(int k = i; k <= j; k++)
                    currentSum += A[k];
                if(currentSum > maxSum)
                    maxSum = currentSum;
            }
    }
    return maxSum;
}

```

Time Complexity:  $O(n^3)$ . Space Complexity:  $O(1)$ .

**Problem-5** Can we improve the complexity of [Problem-4](#)?

**Solution: Yes.** One important observation is that, if we have already calculated the sum for the subsequence  $i, \dots, j - 1$ , then we need only one more addition to get the sum for the subsequence  $i, \dots, j$ . But, the [Problem-4](#) algorithm ignores this information. If we use this fact, we can get an improved algorithm with the running time  $O(n^2)$ .

```
int MaxContiguousSum(int A[], int n) {
    int maxSum = 0;
    for( int i = 0; i < n; i++) {
        int currentSum = 0;
        for( int j = i; j < n; j++) {
            currentSum += a[j];
            if(currentSum > maxSum)
                maxSum = currentSum;
        }
    }
    return maxSum;
}
```

Time Complexity:  $O(n^2)$ . Space Complexity:  $O(1)$ .

**Problem-6** Can we solve [Problem-4](#) using Dynamic Programming?

**Solution: Yes.** For simplicity, let us say,  $M(i)$  indicates maximum sum over all windows ending at  $i$ .

Given Array, A: recursive formula considers the case of selecting  $i^{th}$  element

	.....	?	
--	-------	---	--

$A[i]$

To find maximum sum we have to do one of the following and select maximum among them.

- Either extend the old sum by adding  $A[i]$
- or start new window starting with one element  $A[i]$

$$M(i) = \text{Max} \begin{cases} M(i - 1) + A[i] \\ 0 \end{cases}$$

Where,  $M(i - 1) + A[i]$  indicates the case of extending the previous sum by adding  $A[i]$  and 0 indicates the new window starting at  $A[i]$ .

```

int MaxContiguousSum(int A[], int n) {
    int M[n] = 0, maxSum = 0;
    if(A[0] > 0)
        M[0] = A[0];
    else M[0] = 0;
    for( int i = 1; i < n; i++) {
        if( M[i-1] + A[i] > 0)
            M[i] = M[i-1] + A[i];
        else    M[i] = 0;
    }
    for( int i = 0; i < n; i++)
        if(M[i] > maxSum)
            maxSum = M[i];
    return maxSum;
}

```

Time Complexity:  $O(n)$ . Space Complexity:  $O(n)$ , for table.

**Problem-7** Is there any other way of solving [Problem-4](#)?

**Solution: Yes.** We can solve this problem without DP too (without memory). The algorithm is a little tricky. One simple way is to look for all positive contiguous segments of the array (*sumEndingHere*) and keep track of the maximum sum contiguous segment among all positive segments (*sumSoFar*). Each time we get a positive sum compare it (*sumEndingHere*) with *sumSoFar* and update *sumSoFar* if it is greater than *sumSoFar*. Let us consider the following code for the above observation.

```

int MaxContiguousSum(int A[], int n) {
    int sumSoFar = 0, sumEndingHere = 0;
    for(int i = 0; i < n; i++) {
        sumEndingHere = sumEndingHere + A[i];
        if(sumEndingHere < 0) {
            sumEndingHere = 0;
            continue;
        }
        if(sumSoFar < sumEndingHere)
            sumSoFar = sumEndingHere;
    }
    return sumSoFar;
}

```

**Note:** The algorithm doesn't work if the input contains all negative numbers. It returns 0 if all numbers are negative. To overcome this, we can add an extra check before the actual implementation. The phase will look if all numbers are negative, and if they are it will return maximum of them (or smallest in terms of absolute value).

Time Complexity:  $O(n)$ , because we are doing only one scan. Space Complexity:  $O(1)$ , for table.

**Problem-8** In [Problem-7](#) solution, we have assumed that  $M(i)$  indicates maximum sum over all windows ending at  $i$ . Can we assume  $M(i)$  indicates maximum sum over all windows starting at  $i$  and ending at  $n$ ?

**Solution:** Yes. For simplicity, let us say,  $M(i)$  indicates maximum sum over all windows starting at  $i$ .

Given Array, A: recursive formula considers the case of selecting  $i^{th}$  element

	.....	?	.....
--	-------	---	-------

$A[i]$

To find maximum window we have to do one of the following and select maximum among them.

- Either extend the old sum by adding  $A[i]$
- Or start new window starting with one element  $A[i]$

$$M(i) = \text{Max} \begin{cases} M(i+1) + A[i], & \text{if } M(i+1) + A[i] > 0 \\ 0, & \text{if } M(i+1) + A[i] \leq 0 \end{cases}$$

Where,  $M(i+1) + A[i]$  indicates the case of extending the previous sum by adding  $A[i]$ , and 0 indicates the new window starting at  $A[i]$ .

Time Complexity:  $O(n)$ . Space Complexity:  $O(n)$ , for table.

**Note:** For  $O(n \log n)$  solution, refer to the [Divide and Conquer](#) chapter.

**Problem-9** Given a sequence of  $n$  numbers  $A(1) \dots A(n)$ , give an algorithm for finding a contiguous subsequence  $A(i) \dots A(j)$  for which the sum of elements in the subsequence is maximum. Here the condition is we should not select *two* contiguous numbers.

**Solution:** Let us see how DP solves this problem. Assume that  $M(i)$  represents the maximum sum from 1 to  $i$  numbers without selecting two contiguous numbers. While computing  $M(i)$ , the decision we have to make is, whether to select the  $i^{th}$  element or not. This gives us two possibilities and based on this we can write the recursive formula as:



$$M(i) = \begin{cases} \text{Max}\{A[i] + M(i - 2), M(i - 1)\}, & \text{if } i > 2 \\ A[1], & \text{if } i = 1 \\ \text{Max}\{A[1], A[2]\}, & \text{if } i = 2 \end{cases}$$

- The first case indicates whether we are selecting the  $i^{\text{th}}$  element or not. If we don't select the  $i^{\text{th}}$  element then we have to maximize the sum using the elements 1 to  $i - 1$ . If  $i^{\text{th}}$  element is selected then we should not select  $i - 1^{\text{th}}$  element and need to maximize the sum using 1 to  $i - 2$  elements.
- In the above representation, the last two cases indicate the base cases.

Given Array, A: recursive formula considers the case of selecting  $i^{\text{th}}$  element

	.....	.....	?	
--	-------	-------	---	--

$A[i-2]$      $A[i-1]$      $A[i]$

```
int maxSumWithNoTwoContinuousNumbers(int A[], int n) {
    int M[n+1];
    M[0]=A[0];
    M[1]=(A[0]>A[1]?A[0]:A[1]);
    for(i=2, i<n; i++)
        M[i]= (M[i-1]>M[i-2]+A[i]? M[i-1]: M[i-2]+A[i]);
    return M[n-1];
}
```

Time Complexity:  $O(n)$ . Space Complexity:  $O(n)$ .

**Problem-10** In [Problem-9](#), we assumed that  $M(i)$  represents the maximum sum from 1 to  $i$  numbers without selecting two contiguous numbers. Can we solve the same problem by changing the definition as:  $M(i)$  represents the maximum sum from  $i$  to  $n$  numbers without selecting two contiguous numbers?

**Solution: Yes.** Let us assume that  $M(i)$  represents the maximum sum from  $i$  to  $n$  numbers without selecting two contiguous numbers:

Given Array, A: recursive formula considers the case of selecting  $i^{\text{th}}$  element

	?	.....	.....	
--	---	-------	-------	--

$A[i]$      $A[i+1]$      $A[i+2]$

As similar to [Problem-9](#) solution, we can write the recursive formula as:

$$M(i) = \begin{cases} \text{Max}\{A[i] + M(i + 2), M(i + 1)\}, & \text{if } i > 2 \\ A[1], & \text{if } i = 1 \\ \text{Max}\{A[1], A[2]\}, & \text{if } i = 2 \end{cases}$$

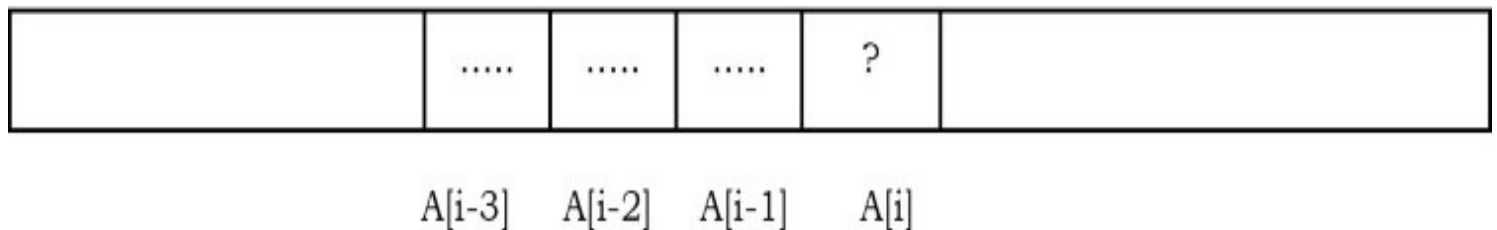
- The first case indicates whether we are selecting the  $i^{\text{th}}$  element or not. If we don't select the  $i^{\text{th}}$  element then we have to maximize the sum using the elements  $i + 1$  to  $n$ . If  $i^{\text{th}}$  element is selected then we should not select  $i + 1^{\text{th}}$  element need to maximize the sum using  $i + 2$  to  $n$  elements.
- In the above representation, the last two cases indicate the base cases.

Time Complexity:  $O(n)$ . Space Complexity:  $O(n)$ .

**Problem-11** Given a sequence of  $n$  numbers  $A(1) \dots A(n)$ , give an algorithm for finding a contiguous subsequence  $A(i) \dots A(j)$  for which the sum of elements in the subsequence is maximum. Here the condition is we should not select *three* continuous numbers.

**Solution:** Input: Array  $A(1) \dots A(n)$  of  $n$  numbers.

Given Array, A: recursive formula considers the case of selecting  $i^{\text{th}}$  element



Assume that  $M(i)$  represents the maximum sum from 1 to  $i$  numbers without selecting three contiguous numbers. While computing  $M(i)$ , the decision we have to make is, whether to select  $i^{\text{th}}$  element or not. This gives us the following possibilities:

$$M(i) = \text{Max} \begin{cases} A[i] + A[i - 1] + M(i - 3) \\ A[i] + M(i - 2) \\ M(i - 1) \end{cases}$$

- In the given problem the restriction is not to select three continuous numbers, but we can select two elements continuously and skip the third one. That is what the first case says in the above recursive formula. That means we are skipping  $A[i - 2]$ .
- The other possibility is, selecting  $i^{\text{th}}$  element and skipping second  $i - 1^{\text{th}}$  element. This is the second case (skipping  $A[i - 1]$ ).
- The third term defines the case of not selecting  $i^{\text{th}}$  element and as a result we should solve the problem with  $i - 1$  elements.

Time Complexity:  $O(n)$ . Space Complexity:  $O(n)$ .

**Problem-12** In [Problem-11](#), we assumed that  $M(i)$  represents the maximum sum from 1 to  $i$  numbers without selecting three contiguous numbers. Can we solve the same problem by changing the definition as:  $M(i)$  represents the maximum sum from  $i$  to  $n$  numbers without selecting three contiguous numbers?

**Solution: Yes.** The reasoning is very much similar. Let us see how DP solves this problem. Assume that  $M(i)$  represents the maximum sum from  $i$  to  $n$  numbers without selecting three contiguous numbers.

Given Array, A: recursive formula considers the case of selecting  $i^{th}$  element

	?	....	....	....	
	$A[i]$	$A[i+1]$	$A[i+2]$	$A[i+3]$	

While computing  $M(i)$ , the decision we have to make is, whether to select  $i^{th}$  element or not. This gives us the following possibilities:


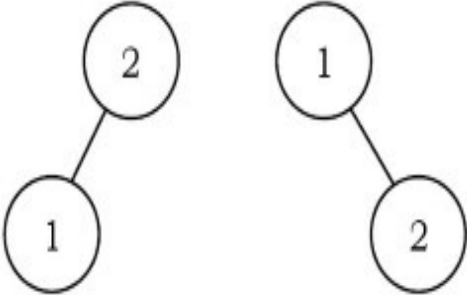
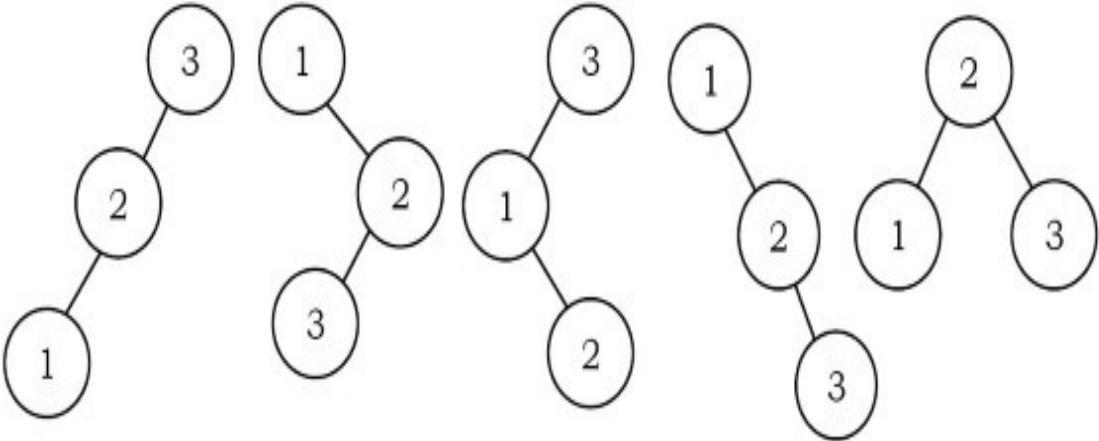
$$M(i) = \text{Max} \begin{cases} A[i] + A[i + 1] + M(i + 3) \\ A[i] + M(i + 2) \\ M(i + 1) \end{cases}$$

- In the given problem the restriction is to not select three continuous numbers, but we can select two elements continuously and skip the third one. That is what the first case says in the above recursive formula. That means we are skipping  $A[i + 2]$ .
- The other possibility is, selecting  $i^{th}$  element and skipping second  $i - 1^{th}$  element. This is the second case (skipping  $A[i + 1]$ ).
- And the third case is not selecting  $i^{th}$  element and as a result we should solve the problem with  $i + 1$  elements.

Time Complexity:  $O(n)$ . Space Complexity:  $O(n)$ .

**Problem-13 Catalan Numbers:** How many binary search trees are there with  $n$  vertices?

**Solution:** Binary Search Tree (BST) is a tree where the left subtree elements are less than the root element, and the right subtree elements are greater than the root element. This property should be satisfied at every node in the tree. The number of BSTs with  $n$  nodes is called *Catalan Number* and is denoted by  $C_n$ . For example, there are 2 BSTs with 2 nodes (2 choices for the root) and 5 BSTs with 3 nodes.

Number of nodes, $n$	Number of Trees
1	
2	
3	

Let us assume that the nodes of the tree are numbered from 1 to  $n$ . Among the nodes, we have to select some node as root, and then divide the nodes which are less than root node into left sub tree, and elements greater than root node into right sub tree. Since we have already numbered the vertices, let us assume that the root element we selected is  $i^{th}$  element.

If we select  $i^{th}$  element as root then we get  $i - 1$  elements on left sub-tree and  $n - i$  elements on right sub tree. Since  $C_n$  is the Catalan number for  $n$  elements,  $C_{i-1}$  represents the Catalan number for left sub tree elements ( $i - 1$  elements) and  $C_{n-i}$  represents the Catalan number for right sub tree elements. The two sub trees are independent of each other, so we simply multiply the two numbers. That means, the Catalan number for a fixed  $i$  value is  $C_{i-1} \times C_{n-i}$ .

Since there are  $n$  nodes, for  $i$  we will get  $n$  choices. The total Catalan number with  $n$  nodes can be given as:

$$C_n = \sum_{i=1}^n C_{i-1} \times C_{n-i}$$

```

int CatalanNumber( int n ) {
    if( n == 0 )
        return 1;
    int count = 0;
    for( int i = 1; i <= n; i++ )
        count += CatalanNumber (i -1) * CatalanNumber (n -i);
    return count;
}

```

Time Complexity:  $O(4^n)$ . For proof, refer [Introduction](#) chapter.

**Problem-14** Can we improve the time complexity of [Problem-13](#) using DP?

**Solution:** The recursive call  $C_n$  depends only on the numbers  $C_0$  to  $C_{n-1}$  and for any value of  $i$ , there are a lot of recalculations. We will keep a table of previously computed values of  $C_i$ . If the function *CatalanNumber()* is called with parameter **i**, and if it has already been computed before, then we can simply avoid recalculating the same subproblem.

```

int Table[1024];
int CatalanNumber( int n ) {
    if( Table[n] != 1 )
        return Table[n];
    Table[n] = 0;
    for( int i = 1; i <= n; i++ )
        Table[n] += CatalanNumber( i -1) * CatalanNumber(n -i);
    return Table[n];
}

```

The time complexity of this implementation  $O(n^2)$ , because to compute *CatalanNumber(n)*, we need to compute all of the *CatalanNumber(i)* values between 0 and  $n - 1$ , and each one will be computed exactly once, in linear time.

In mathematics, Catalan Number can be represented by direct equation as:  $\frac{(2n)!}{n!(n+1)!}$

**Problem-15 Matrix Product Parenthesizations:** Given a series of matrices:  $A_1 \times A_2 \times A_3 \times \dots \times A_n$  with their dimensions, what is the best way to parenthesize them so that it produces the minimum number of total multiplications. Assume that we are using standard matrix and not Strassen's matrix multiplication algorithm.

**Solution:** Input: Sequence of matrices  $A_1 \times A_2 \times A_3 \times \dots \times A_n$ , where  $A_i$  is a  $P_{i-1} \times P_i$ . The dimensions are given in an array P.

**Goal:** Parenthesize the given matrices in such a way that it produces the optimal number of multiplications needed to compute  $A_1 \times A_2 \times A_3 \times \dots \times A_n$ .

For the matrix multiplication problem, there are many possibilities. This is because matrix multiplication is associative. It does not matter how we parenthesize the product, the result will be the same. As an example, for four matrices A, B, C, and D, the possibilities could be:

$$(ABC)D = (AB)(CD) = A(BCD) = A(BC)D = ..$$

Multiplying  $(p \times q)$  matrix with  $(q \times r)$  matrix requires  $pqr$  multiplications. Each of the above possibilities produces a different number of products during multiplication. To select the best one, we can go through each possible parenthesization (brute force), but this requires  $O(2^n)$  time and is very slow. Now let us use DP to improve this time complexity. Assume that,  $M[i,j]$  represents the least number of multiplications needed to multiply  $A_i \dots A_j$ .

$$M[i, j] = \begin{cases} 0 & , \text{if } i = j \\ \text{Min}\{M[i, k] + M[k + 1, j] + P_{i-1}P_kP_j\} & , \text{if } i < j \end{cases}$$

The above recursive formula says that we have to find point  $k$  such that it produces the minimum number of multiplications. After computing all possible values for  $k$ , we have to select the  $k$  value which gives minimum value. We can use one more table (say,  $S[i,j]$ ) to reconstruct the optimal parenthesizations. Compute the  $M[i,j]$  and  $S[i,j]$  in a bottom-up fashion.

```

/* P is the sizes of the matrices, Matrix i has the dimension P[i-1] x P[i].
M[i,j] is the best cost of multiplying matrices i through j
S[i,j] saves the multiplication point and we use this for back tracing */
void MatrixChainOrder(int P[], int length) {
    int n = length - 1, M[n][n], S[n][n];
    for (int i = 1; i <= n; i++)
        M[i][i] = 0;

    // Fills in matrix by diagonals
    for (int l=2; l<= n; l++) { // l is chain length
        for (int i=1; i<= n -l+1; i++) {
            int j = i+l-1;
            M[i][j] = MAX_VALUE;
            // Try all possible division points i..k and k..j
            for (int k=i; k<=j-1; k++) {
                int thisCost = M[i][k] + M[k+1][j] + P[i-1]*P[k]*P[j];
                if(thisCost < M[i][j]) {
                    M[i][j] = thisCost;
                    S[i][j] = k;
                }
            }
        }
    }
}

```

**How many sub problems are there?** In the above formula,  $i$  can range from 1 to  $n$  and  $j$  can range from 1 to  $n$ . So there are a total of  $n^2$  subproblems, and also we are doing  $n - 1$  such operations [since the total number of operations we need for  $A_1 \times A_2 \times A_3 \times \dots \times A_n$  is  $n - 1$ ]. So the time complexity is  $O(n^3)$ .

Space Complexity:  $O(n^2)$ .

**Problem-16** For the [Problem-15](#), can we use greedy method?

**Solution:** Greedy method is not an optimal way of solving this problem. Let us go through some counter example for this. As we have seen already, greedy method makes the decision that is good locally and it does not consider the future optimal solutions. In this case, if we use Greedy, then we always do the cheapest multiplication first. Sometimes it returns a parenthesization that is not optimal.

**Example:** Consider  $A_1 \times A_2 \times A_3$  with dimensions  $3 \times 100$ ,  $100 \times 2$  and  $2 \times 2$ . Based on greedy we parenthesize them as:  $A_1 \times (A_2 \times A_3)$  with  $100 \cdot 2 \cdot 2 + 3 \cdot 100 \cdot 2 = 1000$  multiplications. But the optimal solution to this problem is:  $(A_1 \times A_2) \times A_3$  with  $3 \cdot 100 \cdot 2 + 3 \cdot 2 \cdot 2 = 612$

multiplications.  $\therefore$  we cannot use *greedy* for solving this problem.

**Problem-17 Integer Knapsack Problem [Duplicate Items Permitted]:** Given  $n$  types of items, where the  $i^{th}$  item type has an integer size  $s_i$  and a value  $v_i$ . We need to fill a knapsack of total capacity  $C$  with items of maximum value. We can add multiple items of the same type to the knapsack.

**Note:** For Fractional Knapsack problem refer to [Greedy Algorithms](#) chapter.

**Solution:** Input:  $n$  types of items where  $i^{th}$  type item has the size  $s_i$  and value  $v_i$ . Also, assume infinite number of items for each item type.

**Goal:** Fill the knapsack with capacity  $C$  by using  $n$  types of items and with maximum value.

One important note is that it's not compulsory to fill the knapsack completely. That means, filling the knapsack completely [of size  $C$ ] if we get a value  $V$  and without filling the knapsack completely [let us say  $C - 1$ ] with value  $U$  and if  $V < U$  then we consider the second one. In this case, we are basically filling the knapsack of size  $C - 1$ . If we get the same situation for  $C - 1$  also, then we try to fill the knapsack with  $C - 2$  size and get the maximum value.

Let us say  $M(j)$  denotes the maximum value we can pack into a  $j$  size knapsack. We can express  $M(j)$  recursively in terms of solutions to sub problems as follows:

$$M(j) = \begin{cases} \max\{M(j-1), \max_{i=1 \text{ to } n}(M(j-s_i)) + v_i\}, & \text{if } j \geq 1 \\ 0, & \text{if } j \leq 0 \end{cases}$$

For this problem the decision depends on whether we select a particular  $i^{th}$  item or not for a knapsack of size  $j$ .

- If we select  $i^{th}$  item, then we add its value  $v_i$  to the optimal solution and decrease the size of the knapsack to be solved to  $j - s_i$ .
- If we do not select the item then check whether we can get a better solution for the knapsack of size  $j - 1$ .

The value of  $M(C)$  will contain the value of the optimal solution. We can find the list of items in the optimal solution by maintaining and following "back pointers".

**Time Complexity:** Finding each  $M(j)$  value will require  $\Theta(n)$  time, and we need to sequentially compute  $C$  such values. Therefore, total running time is  $\Theta(nC)$ .

**Space Complexity:**  $\Theta(C)$ .

**Problem-18 0-1 Knapsack Problem:** For [Problem-17](#), how do we solve it if the items are not duplicated (not having an infinite number of items for each type, and each item is allowed to be used for 0 or 1 time)?



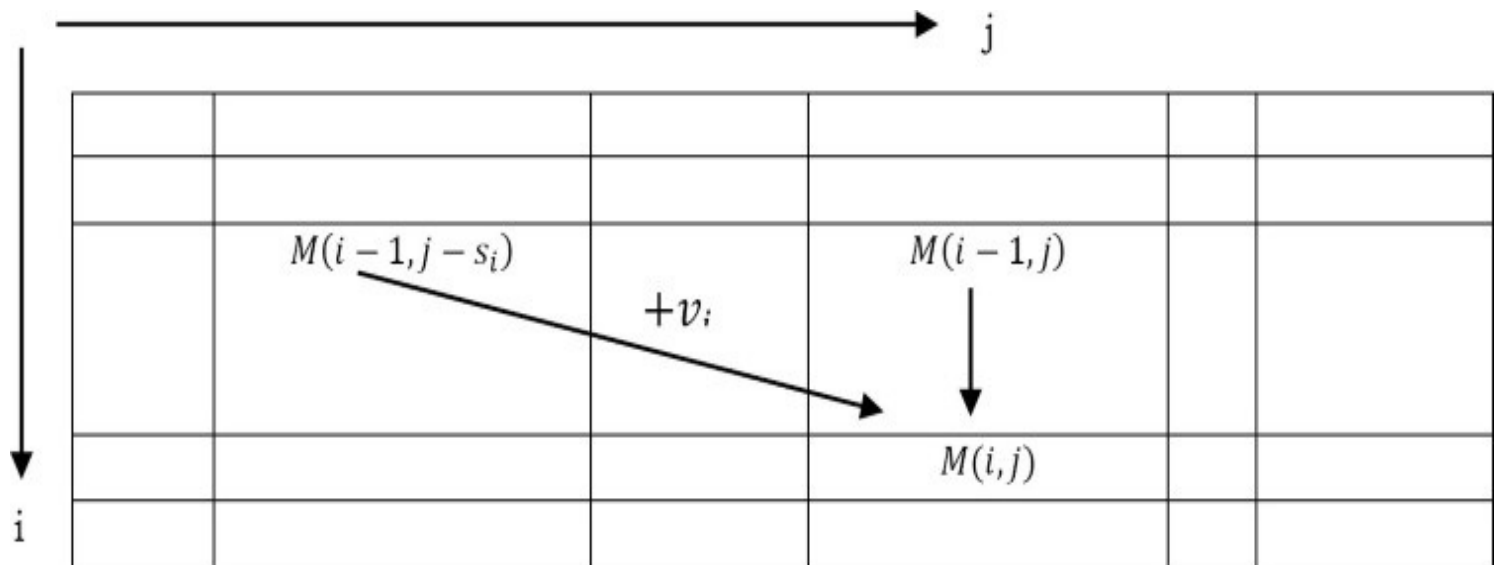
**Real-time example:** Suppose we are going by flight, and we know that there is a limitation on the luggage weight. Also, the items which we are carrying can be of different types (like laptops, etc.). In this case, our objective is to select the items with maximum value. That means, we need to tell the customs officer to select the items which have more weight and less value (profit).

**Solution:** Input is a set of  $n$  items with sizes  $s_i$  and values  $v_i$  and a Knapsack of size  $C$  which we need to fill with a subset of items from the given set. Let us try to find the recursive formula for this problem using DP. Let  $M(i,j)$  represent the optimal value we can get for filling up a knapsack of size  $j$  with items  $1 \dots i$ . The recursive formula can be given as:

$$M(i, j) = \text{Max}\{ \underbrace{M(i-1, j)}_{\substack{i^{\text{th}} \text{ item is} \\ \text{not used}}}, \underbrace{M(i-1, j - s_i) + v_i}_{\substack{i^{\text{th}} \text{ item is} \\ \text{used}}} \}$$

Time Complexity:  $O(nC)$ , since there are  $nC$  subproblems to be solved and each of them takes  $O(1)$  to compute. Space Complexity:  $O(nC)$ , where as Integer Knapsack takes only  $O(C)$ .

Now let us consider the following diagram which helps us in reconstructing the optimal solution and also gives further understanding. Size of below matrix is  $M$ .



Since  $i$  takes values from  $1 \dots n$  and  $j$  takes values from  $1 \dots C$ , there are a total of  $nC$  subproblems. Now let us see what the above formula says:

- $M(i-1, j)$ : Indicates the case of not selecting the  $i^{\text{th}}$  item. In this case, since we are not adding any size to the knapsack we have to use the same knapsack size for subproblems but excluding the  $i^{\text{th}}$  item. The remaining items are  $i-1$ .
- $M(i-1, j - s_i) + v_i$  indicates the case where we have selected the  $i^{\text{th}}$  item. If we add

the  $i^{th}$  item then we have to reduce the subproblem knapsack size to  $j - s_i$  and at the same time we need to add the value  $v_i$  to the optimal solution. The remaining items are  $i - 1$ .

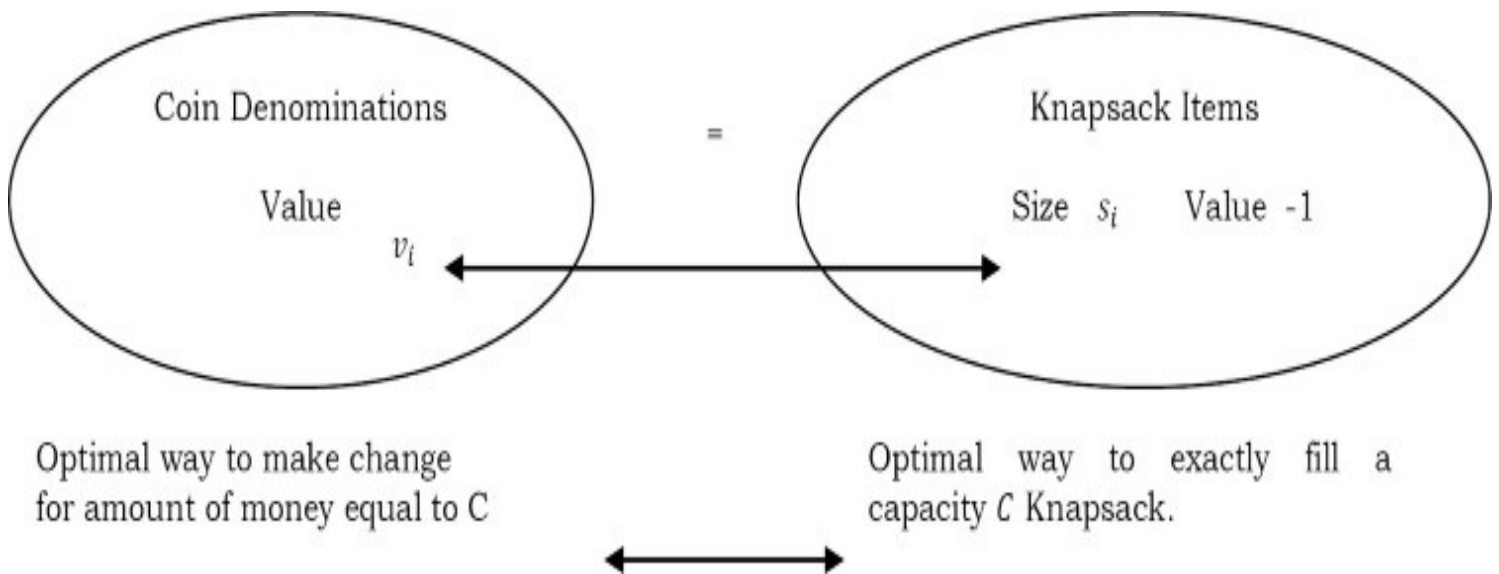
Now, after finding all  $M(i,j)$  values, the optimal objective value can be obtained as:  $Max_j\{M(n,j)\}$

This is because we do not know what amount of capacity gives the best solution.

In order to compute some value  $M(i,j)$ , we take the maximum of  $M(i - 1,j)$  and  $M(i - 1,j - s_i) + v_i$ . These two values ( $M(i,j)$  and  $M(i - 1,j - s_i)$ ) appear in the previous row and also in some previous columns. So,  $M(i,j)$  can be computed just by looking at two values in the previous row in the table.

**Problem-19 Making Change:** Given  $n$  types of coin denominations of values  $v_1 < v_2 < \dots < v_n$  (integers). Assume  $v_1 = 1$ , so that we can always make change for any amount of money  $C$ . Give an algorithm which makes change for an amount of money  $C$  with as few coins as possible.

**Solution:**



This problem is identical to the Integer Knapsack problem. In our problem, we have coin denominations, each of value  $v_i$ . We can construct an instance of a Knapsack problem for each item that has a sizes  $s_i$ , which is equal to the value of  $v_i$  coin denomination. In the Knapsack we can give the value of every item as  $-1$ .

Now it is easy to understand an optimal way to make money  $C$  with the fewest coins is completely equivalent to the optimal way to fill the Knapsack of size  $C$ . This is because since every value has a value of  $-1$ , and the Knapsack algorithm uses as few items as possible which correspond to as few coins as possible.

Let us try formulating the recurrence. Let  $M(j)$  indicate the minimum number of coins required to make change for the amount of money equal to  $j$ .

$$M(j) = \text{Min}_i \{M(j - v_i)\} + 1$$

What this says is, if coin denomination  $i$  was the last denomination coin added to the solution, then the optimal way to finish the solution with that one is to optimally make change for the amount of money  $j - v_i$  and then add one extra coin of value  $v_i$ .

```
int Table[128] ; //Initialization
int MakingChange(int n) {
    if(n < 0) return -1;
    if(n == 0)
        return 0;
    if(Table[n] != -1)
        return Table[n];
    int ans = -1;
    for ( int i = 0 ; i < num_denomination ; ++i )
        ans = Min( ans , MakingChange(n - denominations [ i ] ) ) ;

    return Table[ n ] = ans + 1 ;
}
```

Time Complexity:  $O(nC)$ . Since we are solving  $C$  sub-problems and each of them requires minimization of  $n$  terms. Space Complexity:  $O(nC)$ .

**Problem-20      Longest Increasing Subsequence:** Given a sequence of  $n$  numbers  $A_1 \dots A_n$ , determine a subsequence (not necessarily contiguous) of maximum length in which the values in the subsequence form a strictly increasing sequence.

**Solution:**

**Input:** Sequence of  $n$  numbers  $A_1 \dots A_n$ .

**Goal:** To find a subsequence that is just a subset of elements and does not happen to be contiguous. But the elements in the subsequence should form a strictly increasing sequence and at the same time the subsequence should contain as many elements as possible.

For example, if the sequence is (5,6,2,3,4,1,9,9,8,9,5), then (5,6), (3,5), (1,8,9) are all increasing sub-sequences. The longest one of them is (2,3,4,8,9), and we want an algorithm for finding it.

First, let us concentrate on the algorithm for finding the longest subsequence. Later, we can try printing the sequence itself by tracing the table. Our first step is finding the recursive formula.

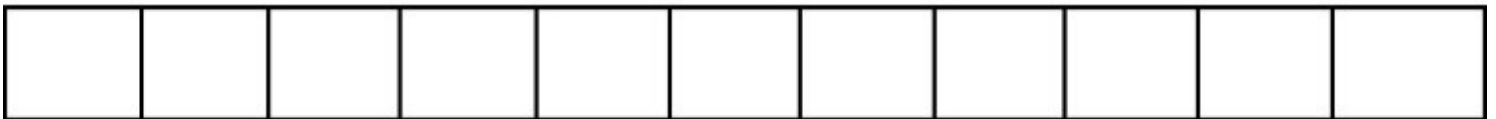
First, let us create the base conditions. If there is only one element in the input sequence then we don't have to solve the problem and we just need to return that element. For any sequence we can start with the first element ( $A[1]$ ). Since we know the first number in the LIS, let's find the second number ( $A[2]$ ). If  $A[2]$  is larger than  $A[1]$  then include  $A[2]$  also. Otherwise, we are done - the LIS is the one element sequence( $A[1]$ ).

Now, let us generalize the discussion and decide about  $i^{th}$  element. Let  $L(i)$  represent the optimal subsequence which is starting at position  $A[1]$  and ending at  $A[i]$ . The optimal way to obtain a strictly increasing subsequence ending at position  $i$  is to extend some subsequence starting at some earlier position  $j$ . For this the recursive formula can be written as:

$$L(i) = \text{Max}_{j < i \text{ and } A[j] < A[i]} \{L(j)\} + 1$$

The above recurrence says that we have to select some earlier position  $j$  which gives the maximum sequence. The 1 in the recursive formula indicates the addition of  $i^{th}$  element.

1        .....         $j$         .....         $i$



Now after finding the maximum sequence for all positions we have to select the one among all positions which gives the maximum sequence and it is defined as:

$$\text{Max}_i \{L(i)\}$$

```

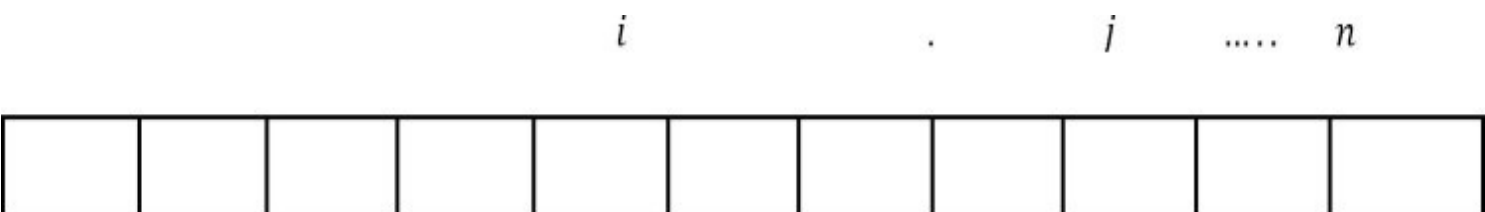
int LISTable [1024];
int LongestIncreasingSequence( int A[], int n ) {
    int i, j, max = 0;
    for ( i = 0; i < n; i++ )
        LISTable[i] = 1;
    for ( i = 0; i < n; i++ ) {
        for ( j = 0; j < i; j++ ) {
            if( A[i] > A[j] && LISTable[i] < LISTable[j] + 1 )
                LISTable[i] = LISTable[j] + 1;
        }
    }
    for ( i = 0; i < n; i++ ) {
        if( max < LISTable[i] )
            max = LISTable[i];
    }
    return max;
}

```

Time Complexity:  $O(n^2)$ , since two *for* loops. Space Complexity:  $O(n)$ , for table.

**Problem-21      Longest Increasing Subsequence:** In [Problem-20](#), we assumed that  $L(i)$  represents the optimal subsequence which is starting at position  $A[1]$  and ending at  $A[i]$ . Now, let us change the definition of  $L(i)$  as:  $L(i)$  represents the optimal subsequence which is starting at position  $A[i]$  and ending at  $A[n]$ . With this approach can we solve the problem?

**Solution: Yes.**



Let  $L(i)$  represent the optimal subsequence which is starting at position  $A[i]$  and ending at  $A[n]$ . The optimal way to obtain a strictly increasing subsequence starting at position  $i$  is going to be to extend some subsequence starting at some later position  $j$ . For this the recursive formula can be written as:

$$L(i) = \text{Max}_{j < i \text{ and } A[j] < A[i]} \{L(j)\} + 1$$

We have to select some later position  $j$  which gives the maximum sequence. The 1 in the recursive formula is the addition of  $i^{th}$  element. After finding the maximum sequence for all positions select

the one among all positions which gives the maximum sequence and it is defined as:

$$\text{Max}_i\{L(i)\}$$

```
int LISTable [1024];
int LongestIncreasingSequence( int A[], int n ) {
    int i, j, max = 0;
    for ( i = 0; i < n; i++ )
        LISTable[i] = 1;
    for(i = n - 1; i >= 0; i--) {
        // try picking a larger second element
        for( j = i + 1; j < n; j++ ) {
            if( A[i] < A[j] && LISTable [i] < LISTable [j] + 1 )
                LISTable[i] = LISTable[j] + 1;
        }
    }
    for ( i = 0; i < n; i++ ) {
        if( max < LISTable[i] )
            max = LISTable[i];
    }
    return max;
}
```

Time Complexity:  $O(n^2)$  since two nested *for* loops. Space Complexity:  $O(n)$ , for table.

**Problem-22** Is there an alternative way of solving *Problem-21*?

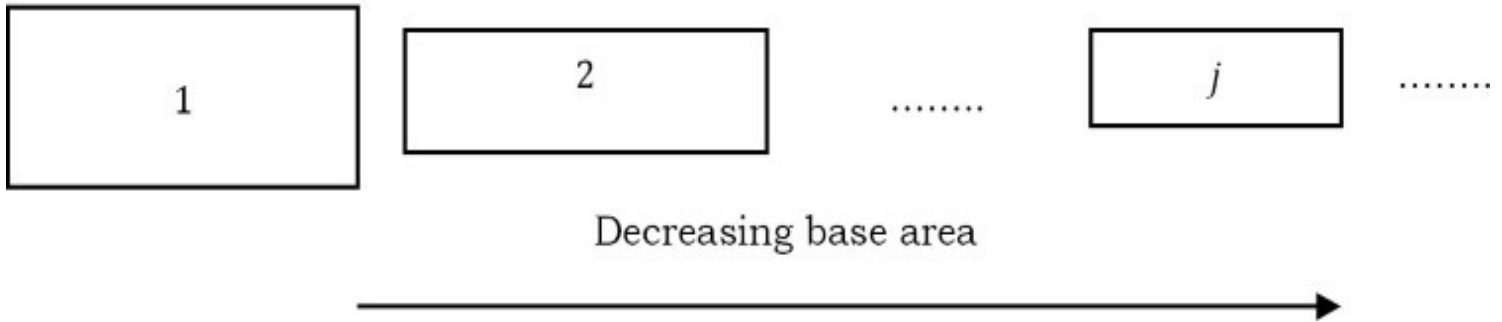
**Solution: Yes.** The other method is to sort the given sequence and save it into another array and then take out the “Longest Common Subsequence” (LCS) of the two arrays. This method has a complexity of  $O(n^2)$ . For LCS problem refer *theory section* of this chapter.

**Problem-23 Box Stacking:** Assume that we are given a set of  $n$  rectangular 3 – D boxes. The dimensions of  $i^{th}$  box are height  $h_i$ , width  $w_i$  and depth  $d_i$ . Now we want to create a stack of boxes which is as tall as possible, but we can only stack a box on top of another box if the dimensions of the 2 –D base of the lower box are each strictly larger than those of the 2 –D base of the higher box. We can rotate a box so that any side functions as its base. It is possible to use multiple instances of the same type of box.

**Solution:** Box stacking problem can be reduced to LIS [*Problem-21*].

**Input:**  $n$  boxes where  $i^{th}$  with height  $h_i$ , width  $w_i$  and depth  $d_i$ . For all  $n$  boxes we have to consider all the orientations with respect to rotation. That is, if we have, in the original set, a box with dimensions  $1 \times 2 \times 3$ , then we consider 3 boxes,

$$1 \times 2 \times 3 \Rightarrow \begin{cases} 1 \times (2 \times 3), \text{ with height 1, base 2 and width 3} \\ 2 \times (1 \times 3), \text{ with height 2, base 1 and width 3} \\ 3 \times (1 \times 2), \text{ with height 3, base 1 and width 2} \end{cases}$$



This simplification allows us to forget about the rotations of the boxes and we just focus on the stacking of  $n$  boxes with each height as  $h_i$  and a base area of  $(w_i \times d_i)$ . Also assume that  $w_i \leq d_i$ . Now what we do is, make a stack of boxes that is as tall as possible and has maximum height. We allow a box  $i$  on top of box  $j$  only if box  $i$  is smaller than box  $j$  in both the dimensions. That means, if  $w_i < w_j$  &  $d_i < d_j$ . Now let us solve this using DP. First select the boxes in the order of decreasing base area.

Now, let us say  $H(j)$  represents the tallest stack of boxes with box  $j$  on top. This is very similar to the LIS problem because the stack of  $n$  boxes with ending box  $j$  is equal to finding a subsequence with the first  $j$  boxes due to the sorting by decreasing base area. The order of the boxes on the stack is going to be equal to the order of the sequence.

Now we can write  $H(j)$  recursively. In order to form a stack which ends on box  $j$ , we need to extend a previous stack ending at  $i$ . That means, we need to put  $j$  box at the top of the stack [ $i$  box is the current top of the stack]. To put  $j$  box at the top of the stack we should satisfy the condition  $w_i > w_j$  and  $d_i > d_j$  [this ensures that the low level box has more base than the boxes above it]. Based on this logic, we can write the recursive formula as:

$$H(j) = \text{Max}_{i < j \text{ and } w_i > w_j \text{ and } d_i > d_j} \{H(i)\} + h_j$$

Similar to the LIS problem, at the end we have to select the best  $j$  over all potential values. This is because we are not sure which box might end up on top.

$$\text{Max}_j \{H(j)\}$$

Time Complexity:  $O(n^2)$ .

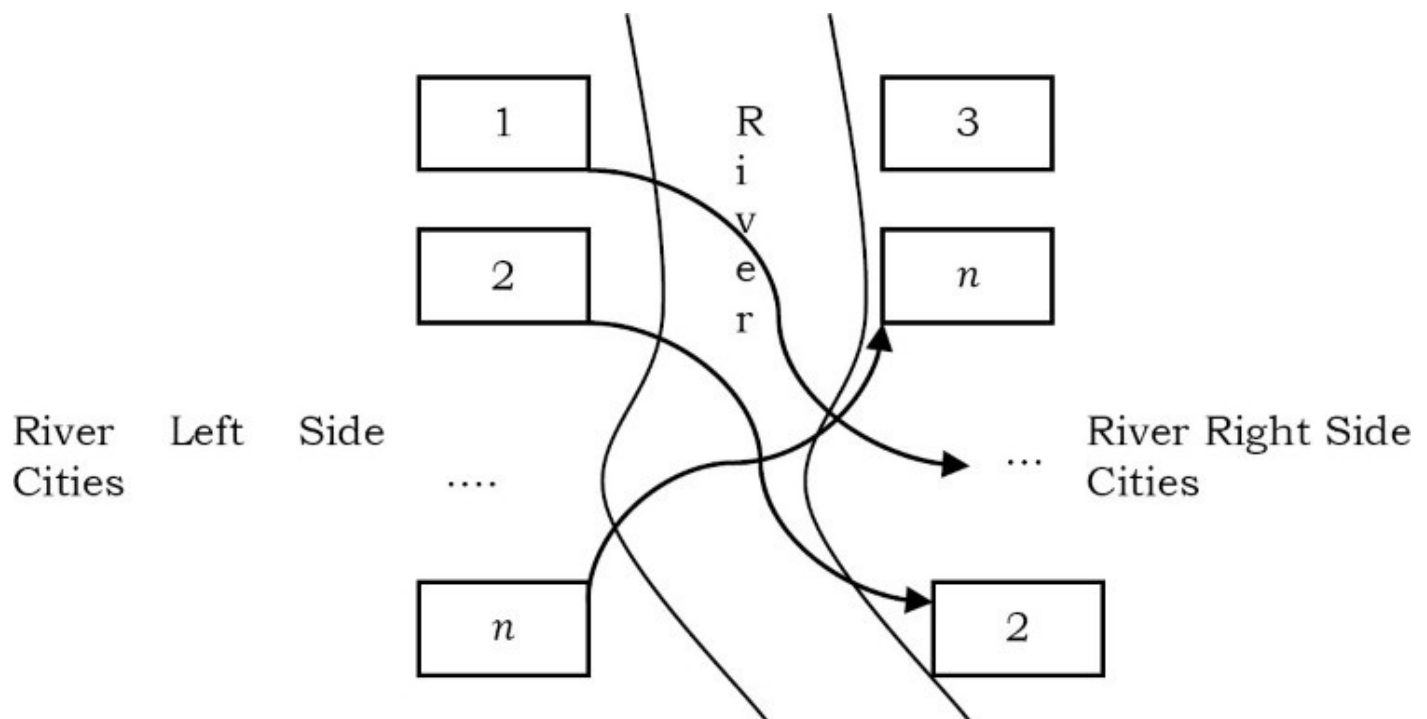
**Problem-24 Building Bridges in India:** Consider a very long, straight river which moves from north to south. Assume there are  $n$  cities on both sides of the river:  $n$  cities on the left of the river and  $n$  cities on the right side of the river. Also, assume that these cities are numbered from 1 to  $n$  but the order is not known. Now we want to connect as many left-

right pairs of cities as possible with bridges such that no two bridges cross. When connecting cities, we can only connect city  $i$  on the left side to city  $i$  on the right side.

**Solution:**

**Input:** Two pairs of sets with each numbered from 1 to  $n$ .

**Goal:** Construct as many bridges as possible without any crosses between left side cities to right side cities of the river.



To understand better let us consider the diagram below. In the diagram it can be seen that there are  $n$  cities on the left side of river and  $n$  cities on the right side of river. Also, note that we are connecting the cities which have the same number [a requirement in the problem]. Our goal is to connect the maximum cities on the left side of river to cities on the right side of the river, without any cross edges. Just to make it simple, let us sort the cities on one side of the river.

If we observe carefully, since the cities on the left side are already sorted, the problem can be simplified to finding the maximum increasing sequence. That means we have to use the LIS solution for finding the maximum increasing sequence on the right side cities of the river.

Time Complexity:  $O(n^2)$ , (same as LIS).

**Problem-25 Subset Sum:** Given a sequence of  $n$  positive numbers  $A_1 \dots A_n$ , give an algorithm which checks whether there exists a subset of  $A$  whose sum of all numbers is  $T$ ?

**Solution:** This is a variation of the Knapsack problem. As an example, consider the following array:



$$A = [3, 2, 4, 19, 3, 7, 13, 10, 6, 11]$$

Suppose we want to check whether there is any subset whose sum is 17. The answer is yes, because the sum of  $4 + 13 = 17$  and therefore  $\{4, 13\}$  is such a subset.

Let us try solving this problem using DP. We will define  $n \times T$  matrix, where  $n$  is the number of elements in our input array and  $T$  is the sum we want to check.

Let,  $M[i, j] = 1$  if it is possible to find a subset of the numbers 1 through  $i$  that produce sum/ and  $M[i, j] = 0$  otherwise.

$$M[i, j] = \text{Max}(M[i - 1, j], M[i - 1, j - A_i])$$

According to the above recursive formula similar to the Knapsack problem, we check if we can get the sum  $j$  by not including the element  $i$  in our subset, and we check if we can get the sum  $j$  by including  $i$  and checking if the sum  $j - A_i$  exists without the  $i^{\text{th}}$  element. This is identical to Knapsack, except that we are storing 0/1's instead of values. In the below implementation we can use binary OR operation to get the maximum among  $M[i - 1, j]$  and  $M[i - 1, j - A_i]$ .

```
int SubsetSum( int A[], int n, int T ) {
    int i, j, M[n+1][T+1];
    M[0][0]=0;
    for (i=1; i<= T; i++)
        M[0][i]= 0;
    for (i=1; i<=n; i++) {
        for (j = 0; j<= T; j++) {
            M[i][j] = M[i-1][j] || M[i-1][j - A[i]];
        }
    }
    return M[n][T];
}
```

**How many subproblems are there?** In the above formula,  $i$  can range from 1 to  $n$  and  $j$  can range from 1 to  $T$ . There are a total of  $nT$  subproblems and each one takes  $O(1)$ . So the time complexity is  $O(nT)$  and this is not polynomial as the running time depends on two variables [ $n$  and  $T$ ], and we can see that they are an exponential function of the other.

Space Complexity:  $O(nT)$ .

**Problem-26** Given a set of  $n$  integers and the sum of all numbers is at most  $if$ . Find the subset of these  $n$  elements whose sum is exactly half of the total sum of  $n$  numbers.

**Solution:** Assume that the numbers are  $A_1 \dots A_n$ . Let us use DP to solve this problem. We will

create a boolean array  $T$  with size equal to  $K + 1$ . Assume that  $T[x]$  is 1 if there exists a subset of given  $n$  elements whose sum is  $x$ . That means, after the algorithm finishes,  $T[K]$  will be 1, if and only if there is a subset of the numbers that has sum  $K$ . Once we have that value then we just need to return  $T[K/2]$ . If it is 1, then there is a subset that adds up to half the total sum.

Initially we set all values of  $T$  to 0. Then we set  $T[0]$  to 1. This is because we can always build 0 by taking an empty set. If we have no numbers in  $A$ , then we are done! Otherwise, we pick the first number,  $A[0]$ . We can either throw it away or take it into our subset. This means that the new  $T[]$  should have  $T[0]$  and  $T[A[0]]$  set to 1. This creates the base case. We continue by taking the next element of  $A$ .

Suppose that we have already taken care of the first  $i - 1$  elements of  $A$ . Now we take  $A[i]$  and look at our table  $T[]$ . After processing  $i - 1$  elements, the array  $T$  has a 1 in every location that corresponds to a sum that we can make from the numbers we have already processed. Now we add the new number,  $A[i]$ . What should the table look like? First of all, we can simply ignore  $A[i]$ . That means, no one should disappear from  $T[]$  - we can still make all those sums. Now consider some location of  $T[j]$  that has a 1 in it. It corresponds to some subset of the previous numbers that add up to  $j$ . If we add  $A[i]$  to that subset, we will get a new subset with total sum  $j + A[i]$ . So we should set  $T[j + A[i]]$  to 1 as well. That's all. Based on the above discussion, we can write the algorithm as:

```
bool T[10240];
bool SubsetHalfSum( int A[], int n ) {
    int K = 0;
    for( int i = 0; i < n; i++ )
        K += A[i];
    T[0] = 1;          // initialize the table
    for( int i = 1; i <= K; i++ )
        T[i] = 0;
    // process the numbers one by one
    for( int i = 0; i < n; i++ ) {
        for( int j = K - A[i]; j >= 0; j-- ) {
            if( T[j] )
                T[j + A[i]] = 1;
        }
    }
    return T[K / 2];
}
```

In the above code,  $j$  loop moves from right to left. This reduces the double counting problem. That means, if we move from left to right, then we may do the repeated calculations.

Time Complexity:  $O(nK)$ , for the two *for* loops. Space Complexity:  $O(K)$ , for the boolean table  $T$ .

**Problem-27** Can we improve the performance of [Problem-26](#)?

**Solution: Yes.** In the above code what we are doing is, the inner  $j$  loop is starting from  $K$  and moving left. That means, it is unnecessarily scanning the whole table every time.

What we actually want is to find all the 1 entries. At the beginning, only the  $0^{\text{th}}$  entry is 1. If we keep the location of the rightmost 1 entry in a variable, we can always start at that spot and go left instead of starting at the right end of the table.

To take full advantage of this, we can sort  $A[]$  first. That way, the rightmost 1 entry will move to the right as slowly as possible. Finally, we don't really care about what happens in the right half of the table (after  $T[K/2]$ ) because if  $T[x]$  is 1, then  $T[Kx]$  must also be 1 eventually – it corresponds to the complement of the subset that gave us  $x$ . The code based on above discussion is given below.

```
int T[10240];
int SubsetHalfSumEfficient( int A[], int n ) {
    int K = 0;
    for( int i = 0; i < n; i++ )
        K += A[i];
    Sort(A,n);
    T[0] = 1;          // initialize the table
    for( int i = 1; i <= sum; i++ )
        T[i] = 0;
    int R = 0; // rightmost 1 entry
    for( int i = 0; i < n; i++ ) {          // process the numbers one by one
        for( int j = R; j >= 0; j-- ) {
            if( T[j] )
                T[j + A[i]] = 1;
            R = min(K / 2, R + C[i] );
        }
    }
    return T[K / 2];
}
```

After the improvements, the time complexity is still  $O(nK)$ , but we have removed some useless steps.

**Problem-28** Partition problem is to determine whether a given set can be partitioned into two subsets such that the sum of elements in both subsets is the same [the same as the previous problem but a different way of asking]. For example, if  $A[] = \{1, 5,$

11, 5}, the array can be partitioned as {1, 5, 5} and {11}. Similarly, if  $A[] = \{1, 5, 3\}$ , the array cannot be partitioned into equal sum sets.

**Solution:** Let us try solving this problem another way. Following are the two main steps to solve this problem:

1. Calculate the sum of the array. If the sum is odd, there cannot be two subsets with an equal sum, so return false.
2. If the sum of the array elements is even, calculate  $sum/2$  and find a subset of the array with a sum equal to  $sum/2$ .

The first step is simple. The second step is crucial, and it can be solved either using recursion or Dynamic Programming.

**Recursive Solution:** Following is the recursive property of the second step mentioned above. Let  $subsetSum(A, n, sum/2)$  be the function that returns true if there is a subset of  $A[0..n-1]$  with sum equal to  $sum/2$ . The  $isSubsetSum$  problem can be divided into two sub problems:

- a)  $isSubsetSum()$  without considering last element (reducing  $n$  to  $n - 1$ )
- b)  $isSubsetSum$  considering the last element (reducing  $sum/2$  by  $A[n-1]$  and  $n$  to  $n - 1$ )

If any of the above sub problems return true, then return true.

$$subsetSum(A, n, sum/2) = isSubsetSum(A, n - 1, sum/2) \vee subsetSum(A, n - 1, sum/2 - A[n - 1])$$

```

// A utility function that returns true if there is a subset of A[] with sum equal to given sum
bool subsetSum (int A[], int n, int sum){
    if (sum == 0)
        return true;
    if (n == 0 && sum != 0)
        return false;

    // If last element is greater than sum, then ignore it
    if (A[n-1] > sum)
        return subsetSum (A, n-1, sum);
    return subsetSum (A, n-1, sum) || subsetSum (A, n-1, sum-A[n-1]);
}

// Returns true if A[] can be partitioned in two subsets of equal sum, otherwise false
bool findPartition (int A[], int n){
    // calculate sum of all elements
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += A[i];

    // If sum is odd, there cannot be two subsets with equal sum
    if (sum%2 != 0)
        return false;

    // Find if there is subset with sum equal to half of total sum
    return subsetSum (A, n, sum/2);
}

```

**Time Complexity:**  $O(2^n)$  In worst case, this solution tries two possibilities (whether to include or exclude) for every element.

**Dynamic Programming Solution:** The problem can be solved using dynamic programming when the sum of the elements is not too big. We can create a 2D array `part[][]` of size  $(sum/2) * (n + 1)$ . And we can construct the solution in a bottom-up manner such that every filled entry has a following property

*$part[i][j] = true$  if a subset of  $\{A[0], A[1], \dots, A[j-1]\}$  has sum equal to  $sum/2$ , otherwise false*

```

// Returns true if A[] can be partitioned in two subsets of equal sum, otherwise false
bool findPartition (int A[], int n){
    int sum = 0;
    int i, j;

    // calculate sum of all elements
    for (i = 0; i < n; i++)
        sum += A[i];
    if (sum%2 != 0)
        return false;
    bool part[sum/2+1][n+1];
    // initialize top row as true
    for (i = 0; i <= n; i++)
        part[0][i] = true;
    // initialize leftmost column, except part[0][0], as 0
    for (i = 1; i <= sum/2; i++)
        part[i][0] = false;

    // Fill the partition table in bottom up manner
    for (i = 1; i <= sum/2; i++) {
        for (j = 1; j <= n; j++) {
            part[i][j] = part[i][j-1];
            if (i >= A[j-1])
                part[i][j] = part[i][j] || part[i - A[j-1]][j-1];
        }
    }
    return part[sum/2][n];
}

```

Time Complexity:  $O(\text{sum} \times n)$ . Space Complexity:  $O(\text{sum} \times n)$ . Please note that this solution will not be feasible for arrays with a big sum.

**Problem-29 Counting Boolean Parenthesizations:** Let us assume that we are given a boolean expression consisting of symbols '*true*', '*false*', '*and*', '*or*', and '*xor*'. Find the number of ways to parenthesize the expression such that it will evaluate to *true*. For example, there is only 1 way to parenthesize '*true and false xor true*' such that it evaluates to *true*.

**Solution:** Let the number of symbols be  $n$  and between symbols there are boolean operators like and, or, xor, etc. For example, if  $n = 4$ , *T or F and T xor F*. Our goal is to count the numbers of ways to parenthesize the expression with boolean operators so that it evaluates to *true*. In the above case, if we use *T or ( (F and T) xor F )* then it evaluates to true.

$$T \text{ or } \{ (F \text{ and } T) \text{ xor } F \} = \text{True}$$

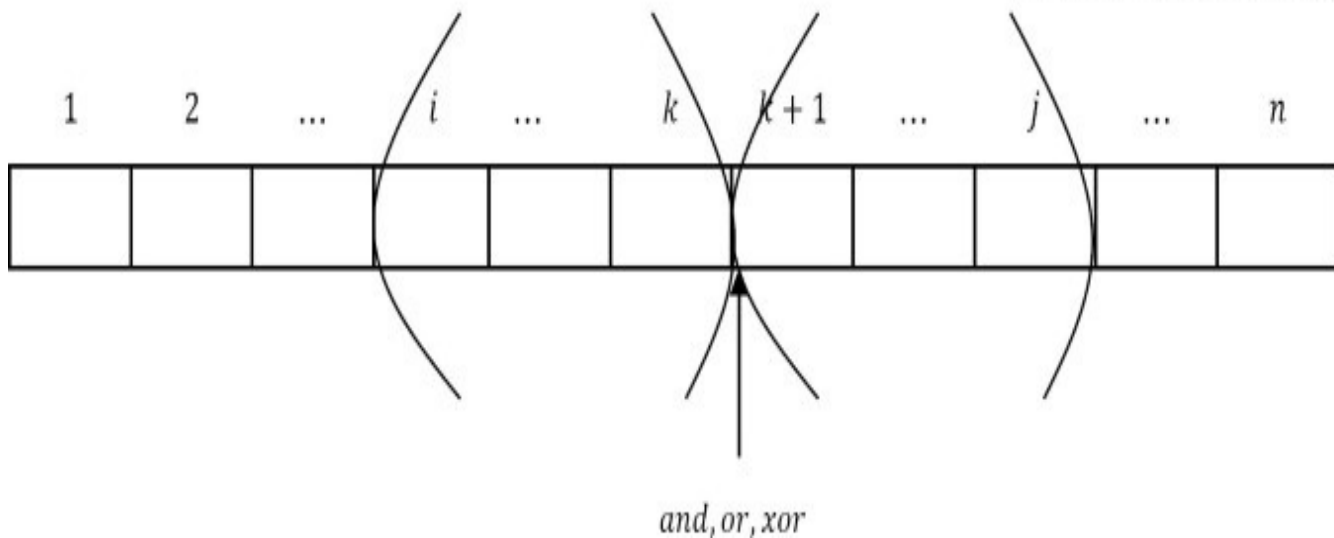
Now let us see how DP solves this problem. Let  $T(i,j)$  represent the number of ways to parenthesize the sub expression with symbols  $i \dots j$  [symbols means only  $T$  and  $F$  and not the operators] with boolean operators so that it evaluates to *true*. Also,  $i$  and  $j$  take the values from 1 to  $n$ . For example, in the above case,  $T(2,4) = 0$  because there is no way to parenthesize the expression  $F \text{ and } T \text{ xor } F$  to make it *true*.

Just for simplicity and similarity, let  $F(i,j)$  represent the number of ways to parenthesize the sub expression with symbols  $i \dots j$  with boolean operators so that it evaluates to *false*. The base cases are  $T(i,i)$  and  $F(i,i)$ .

Now we are going to compute  $T(i, i + 1)$  and  $F(i, i + 1)$  for all values of  $i$ . Similarly,  $T(i, i + 2)$  and  $F(i, i + 2)$  for all values of  $i$  and so on. Now let's generalize the solution.

$$T(i,j) = \sum_{k=i}^{j-1} \begin{cases} T(i,k)T(k+1,j), & \text{for "and"} \\ \text{Total}(i,k)\text{Total}(k+1,j) - F(i,k)F(k+1,j), & \text{for "or"} \\ T(i,k)F(k+1,j) + F(i,k)T(k+1,j), & \text{for "xor"} \end{cases}$$

Where,  $\text{Total}(i,k) = T(i,k) + F(i,k)$ .



What this above recursive formula says is,  $T(i,j)$  indicates the number of ways to parenthesize the expression. Let us assume that we have some sub problems which are ending at  $k$ . Then the total number of ways to parenthesize from  $i$  to  $j$  is the sum of counts of parenthesizing from  $i$  to  $k$  and from  $k + 1$  to  $j$ . To parenthesize between  $k$  and  $k + 1$  there are three ways: “and”, “or” and “xor”.

- If we use “and” between  $k$  and  $k + 1$ , then the final expression becomes *true* only when both are *true*. If both are *true* then we can include them to get the final count.
- If we use “or”, then if at least one of them is *true*, the result becomes *true*. Instead of including all three possibilities for “or”, we are giving one alternative where we are subtracting the “false” cases from total possibilities.

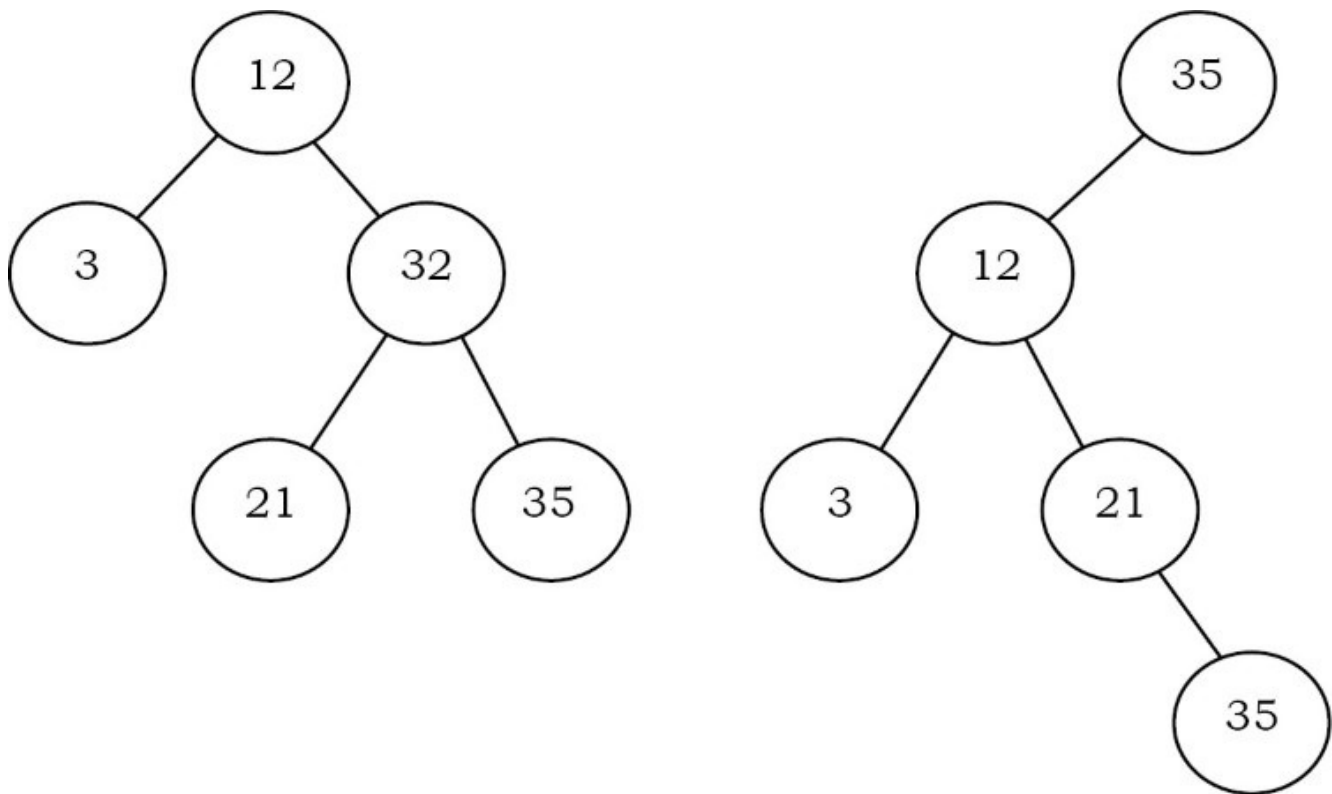
- The same is the case with “xor”. The conversation is as in the above two cases.

After finding all the values we have to select the value of  $k$ , which produces the maximum count, and for  $k$  there are  $i$  to  $j - 1$  possibilities.

**How many subproblems are there?** In the above formula,  $i$  can range from 1 to  $n$ , and  $j$  can range from 1 to  $n$ . So there are a total of  $n^2$  subproblems, and also we are doing summation for all such values. So the time complexity is  $O(n^3)$ .

**Problem-30 Optimal Binary Search Trees:** Given a set of  $n$  (sorted) keys  $A[1..n]$ , build the best binary search tree for the elements of  $A$ . Also assume that each element is associated with *frequency* which indicates the number of times that a particular item is searched in the binary search trees. That means we need to construct a binary search tree so that the total search time will be reduced.

**Solution:** Before solving the problem let us understand the problem with an example. Let us assume that the given array is  $A = [3, 12, 21, 32, 35]$ . There are many ways to represent these elements, two of which are listed below.



**Of the two, which representation is better?** The search time for an element depends on the depth of the node. The average number of comparisons for the first tree is:  $\frac{1+2+2+3+3}{5} = \frac{11}{5}$  and for the second tree, the average number of comparisons is:  $\frac{1+2+3+3+4}{5} = \frac{13}{5}$ . Of the two, the first tree gives better results.

If frequencies are not given and if we want to search all elements, then the above simple



calculation is enough for deciding the best tree. If the frequencies are given, then the selection depends on the frequencies of the elements and also the depth of the elements. For simplicity let us assume that the given array is  $A$  and the corresponding frequencies are in array  $F$ .  $F[i]$  indicates the frequency of  $i^{th}$  element  $A[i]$ . With this, the total search time  $S(\text{root})$  of the tree with  $\text{root}$  can be defined as:

$$S(\text{root}) = \sum_{i=1}^n (\text{depth}(\text{root}, i) + 1) \times F[i]$$

In the above expression,  $\text{depth}(\text{root}, i) + 1$  indicates the number of comparisons for searching the  $i^{th}$  element. Since we are trying to create a binary search tree, the left subtree elements are less than root element and the right subtree elements are greater than root element. If we separate the left subtree time and right subtree time, then the above expression can be written as:

$$S(\text{root}) = \sum_{i=1}^{r-1} (\text{depth}(\text{root}, i) + 1) \times F[i] + \sum_{i=1}^n F[i] + \sum_{i=r+1}^n (\text{depth}(\text{root}, i) + 1) \times F[i]$$

Where  $r$  indicates the position of the root element in the array.

If we replace the left subtree and right subtree times with their corresponding recursive calls, then the expression becomes:

$$S(\text{root}) = S(\text{root} \rightarrow \text{left}) + S(\text{root} \rightarrow \text{right}) + \sum_{i=1}^n F[i]$$

### Binary Search Tree node declaration

Refer to [Trees](#) chapter.

### Implementation:

```

struct BinarySearchTreeNode *OptimalBST(int A[], int F[], int low, int high) {
    int r, minTime = 0;
    struct BinarySearchTreeNode *newNode=(struct BinarySearchTreeNode *)
                                                malloc(sizeof(struct BinarySearchTreeNode));
    if(!newNode) {
        printf("Memory Error");
        return;
    }
    for (r =0, r <= n-1; r++) {
        root→left = OptimalBST(A, F, low, r-1);
        root→right = OptimalBST(A, F, r+1, high)
        root→data = A[r];
        if(minTime > S(root)) minTime = S(root);
    }
    return minTime;
}

```

**Problem-31 Edit Distance:** Given two strings  $A$  of length  $m$  and  $B$  of length  $n$ , transform  $A$  into  $B$  with a minimum number of operations of the following types: delete a character from  $A$ , insert a character into  $A$ , or change some character in  $A$  into a new character. The minimal number of such operations required to transform  $A$  into  $B$  is called the *edit distance* between  $A$  and  $B$ .

**Solution:**

**Input:** Two text strings  $A$  of length  $m$  and  $B$  of length  $n$ .

**Goal:** Convert string  $A$  into  $B$  with minimal conversions.

Before going to a solution, let us consider the possible operations for converting string  $A$  into  $B$ .

- If  $m > n$ , we need to remove some characters of  $A$
- If  $m = n$ , we may need to convert some characters of  $A$
- If  $m < n$ , we need to remove some characters from  $A$

So the operations we need are the insertion of a character, the replacement of a character and the deletion of a character, and their corresponding cost codes are defined below.

**Costs of operations:**

Insertion of a character	$C_i$
Replacement of a character	$C_r$

Now let us concentrate on the recursive formulation of the problem. Let,  $T(i, j)$  represents the minimum cost required to transform first  $i$  characters of  $A$  to first  $j$  characters of  $B$ . That means,  $A[1... i]$  to  $B[1... j]$ .

$$T(i, j) = \min \begin{cases} c_d + T(i - 1, j) \\ T(i, j - 1) + c_i \\ T(i - 1, j - 1), & \text{if } A[i] == B[j] \\ T(i - 1, j - 1) + c_r & \text{if } A[i] \neq B[j] \end{cases}$$

Based on the above discussion we have the following cases.

- If we delete  $i^{th}$  character from  $A$ , then we have to convert remaining  $i - 1$  characters of  $A$  to  $j$  characters of  $B$
- If we insert  $i^{th}$  character in  $A$ , then convert these  $i$  characters of  $A$  to  $j - 1$  characters of  $B$
- If  $A[i] == B[j]$ , then we have to convert the remaining  $i - 1$  characters of  $A$  to  $j - 1$  characters of  $B$
- If  $A[i] \neq B[j]$ , then we have to replace  $i^{th}$  character of  $A$  to  $j^{th}$  character of  $B$  and convert remaining  $i - 1$  characters of  $A$  to  $j - 1$  characters of  $B$

After calculating all the possibilities we have to select the one which gives the lowest cost.

**How many subproblems are there?** In the above formula,  $i$  can range from 1 to  $m$  and  $j$  can range from 1 to  $n$ . This gives  $mn$  subproblems and each one takes  $O(1)$  and the time complexity is  $O(mn)$ . Space Complexity:  $O(mn)$  where  $m$  is number of rows and  $n$  is number of columns in the given matrix.

**Problem-32 All Pairs Shortest Path Problem: Floyd's Algorithm:** Given a weighted directed graph  $G = (V, E)$ , where  $V = \{1, 2, \dots, n\}$ . Find the shortest path between any pair of nodes in the graph. Assume the weights are represented in the matrix  $C[V][V]$ , where  $C[i][j]$  indicates the weight (or cost) between the nodes  $i$  and  $j$ . Also,  $C[i][j] = \infty$  or  $-1$  if there is no path from node  $i$  to node  $j$ .

**Solution:** Let us try to find the DP solution (Floyd's algorithm) for this problem. The Floyd's algorithm for all pairs shortest path problem uses matrix  $A[1..n][1..n]$  to compute the lengths of the shortest paths. Initially,

$$A[i, j] = C[i, j] \text{ if } i \neq j \\ = 0 \text{ if } i = j$$

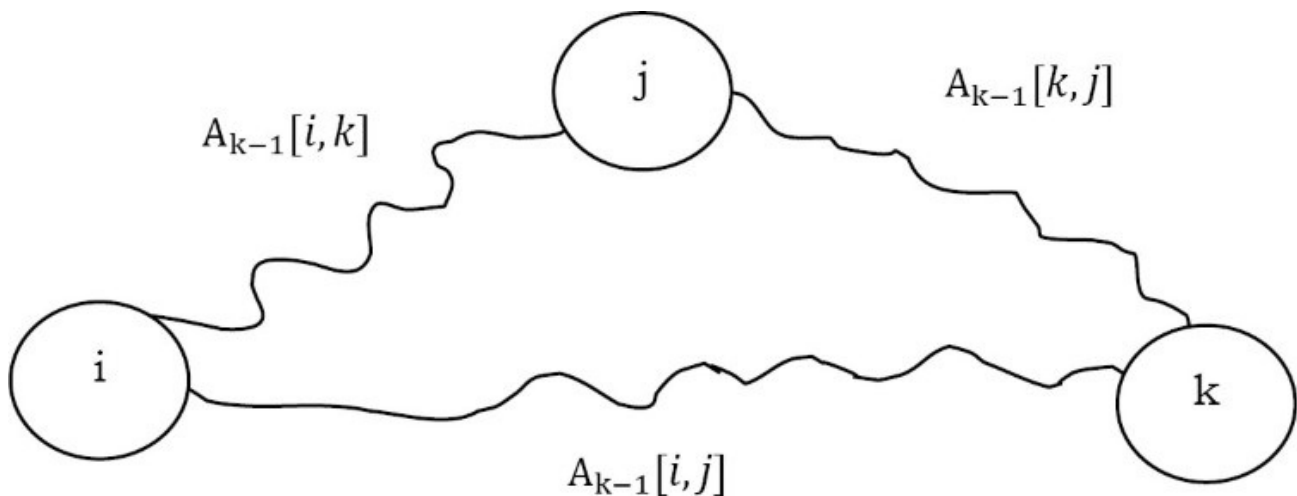
From the definition,  $C[i, j] = \infty$  if there is no path from  $i$  to  $j$ . The algorithm makes  $n$  passes over  $A$ . Let  $A_0, A_1, \dots, A_n$  be the values of  $A$  on the  $n$  passes, with  $A_0$  being the initial value.

Just after the  $k-1^{\text{th}}$  iteration,  $A_{k-1}[i,j]$  = smallest length of any path from vertex  $i$  to vertex  $j$  that does not pass through the vertices  $\{k+1, k+2, \dots, n\}$ . That means, it passes through the vertices possibly through  $\{1, 2, 3, \dots, k-1\}$ .

In each iteration, the value  $A[i][j]$  is updated with minimum of  $A_{k-1}[i,j]$  and  $A_{k-1}[i,k] + A_{k-1}[k,j]$ .

$$A[i,j] = \min \begin{cases} A_{k-1}[i,j] \\ A_{k-1}[i,k] + A_{k-1}[k,j] \end{cases}$$

The  $k^{\text{th}}$  pass explores whether the vertex  $k$  lies on an optimal path from  $i$  to  $j$ , for all  $i, j$ . The same is shown in the diagram below.



```
void Floyd(int C[], int A[], int n) {
    int i, j, k;
    for(i = 0; i <= n - 1; i++)
        for(j = 0; j <= n - 1; j++)
            A[i][j] = C[i][j];
    for(i = 0; i <= n - 1; i++)
        A[i][i] = 0;
    for(k = 0; k <= n - 1; k++) {
        for(i = 0; i <= n - 1; i++) {
            for(j = 0; j <= n - 1; j++)
                if(A[i][k] + A[k][j] < A[i][j])
                    A[i][j] = A[i][k] + A[k][j];
        }
    }
}
```

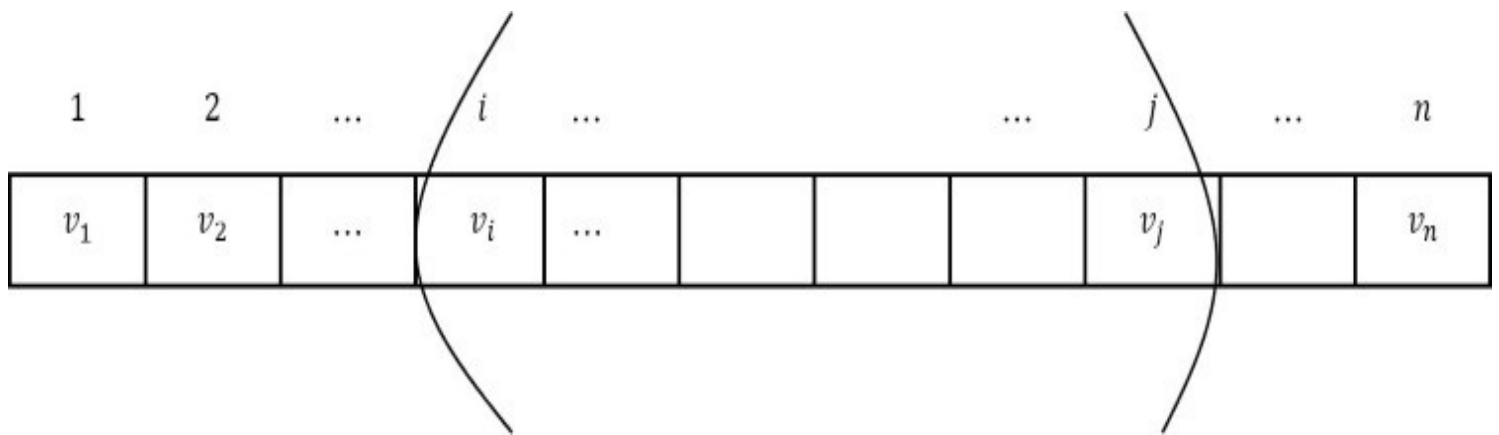
Time Complexity:  $O(n^3)$ .

**Problem-33 Optimal Strategy for a Game:** Consider a row of  $n$  coins of values  $v_1 \dots v_n$ , where  $n$  is even [since it's a two player game]. We play this game with the opponent. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Determine the maximum possible amount of money we can definitely win if we move first.

**Alternative way of framing the question:** Given  $n$  pots, each with some number of gold coins, are arranged in a line. You are playing a game against another player. You take turns picking a pot of gold. You may pick a pot from either end of the line, remove the pot, and keep the gold pieces. The player with the most gold at the end wins. Develop a strategy for playing this game.

**Solution:** Let us solve the problem using our DP technique. For each turn either we *or* our opponent selects the coin only from the ends of the row. Let us define the subproblems as:

$V(i,j)$ : denotes the maximum possible value we can definitely win if it is our turn and the only coins remaining are  $v_i \dots v_j$ .



Base Cases:  $V(i,i), V(i, i + 1)$  for all values of  $i$ .

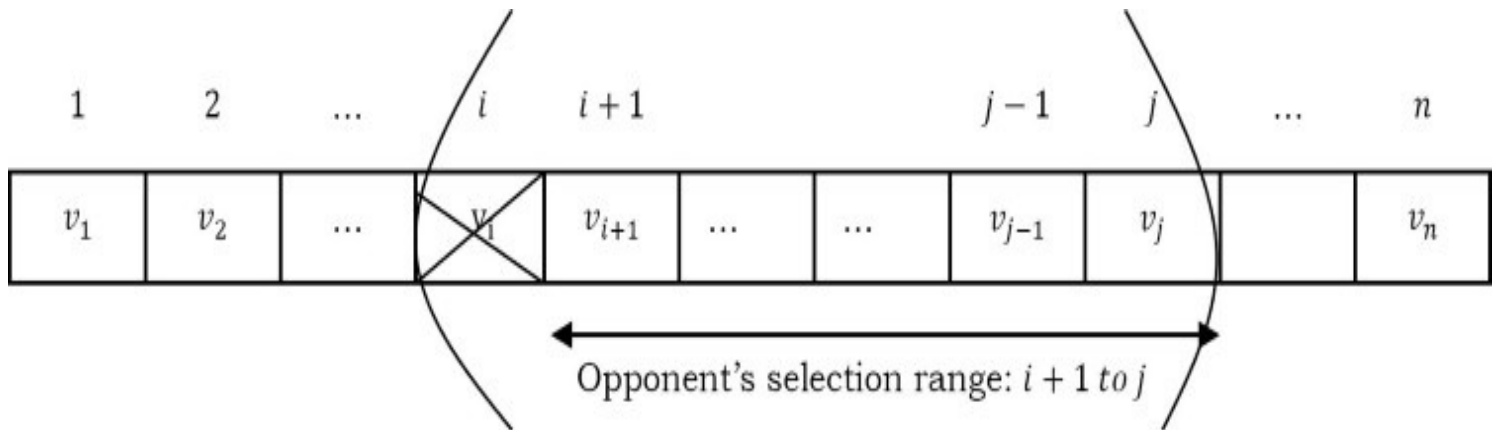
From these values, we can compute  $V(i, i + 2), V(i, i + 3)$  and so on. Now let us define  $V(i,j)$  for each sub problem as:

$$V(i, j) = \text{Max} \left\{ \text{Min} \left\{ \begin{matrix} V(i + 1, j - 1) \\ V(i + 2, j) \end{matrix} \right\} + v_i, \text{Min} \left\{ \begin{matrix} V(i, j - 2) \\ V(i + 1, j - 1) \end{matrix} \right\} + v_j \right\}$$

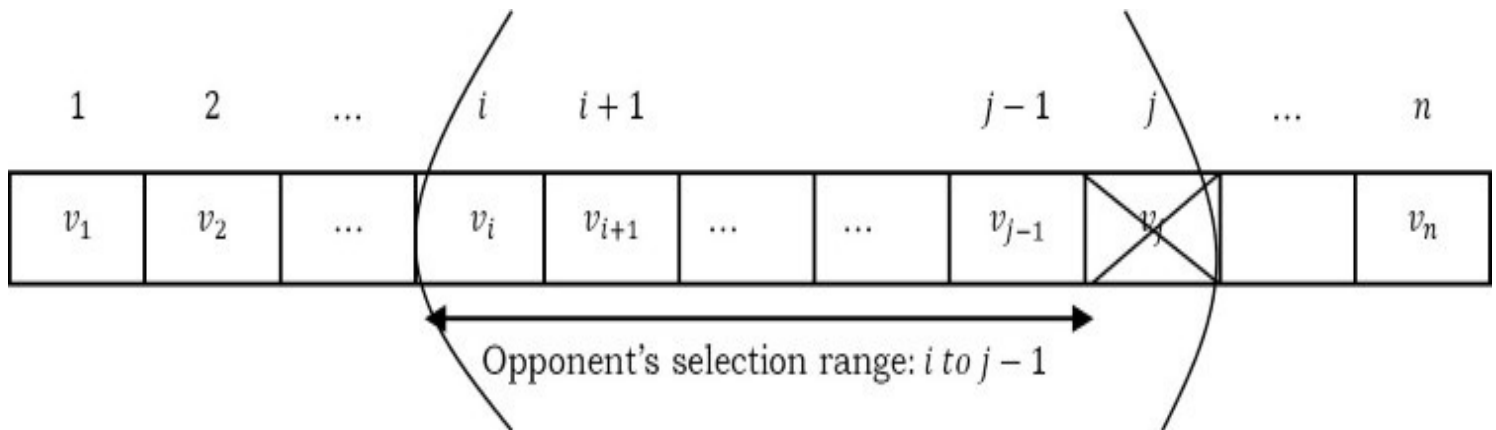
In the recursive call we have to focus on  $i^{\text{th}}$  coin to  $j^{\text{th}}$  coin ( $v_i \dots v_j$ ). Since it is our turn to pick the coin, we have two possibilities: either we can pick  $v_i$  or  $v_j$ . The first term indicates the case if we select  $i^{\text{th}}$  coin ( $v_i$ ) and the second term indicates the case if we select  $j^{\text{th}}$  coin ( $v_j$ ). The outer *Max* indicates that we have to select the coin which gives maximum value. Now let us focus on the terms:

- Selecting  $i^{\text{th}}$  coin: If we select the  $i^{\text{th}}$  coin then the remaining range is from  $i + 1$  to  $j$ . Since we selected the  $i^{\text{th}}$  coin we get the value  $v_i$  for that. From the remaining range

$i + 1$  to  $j$ , the opponents can select either  $i + 1^{th}$  coin or  $j^{th}$  coin. But the opponents selection should be minimized as much as possible [the *Min* term]. The same is described in the below figure.



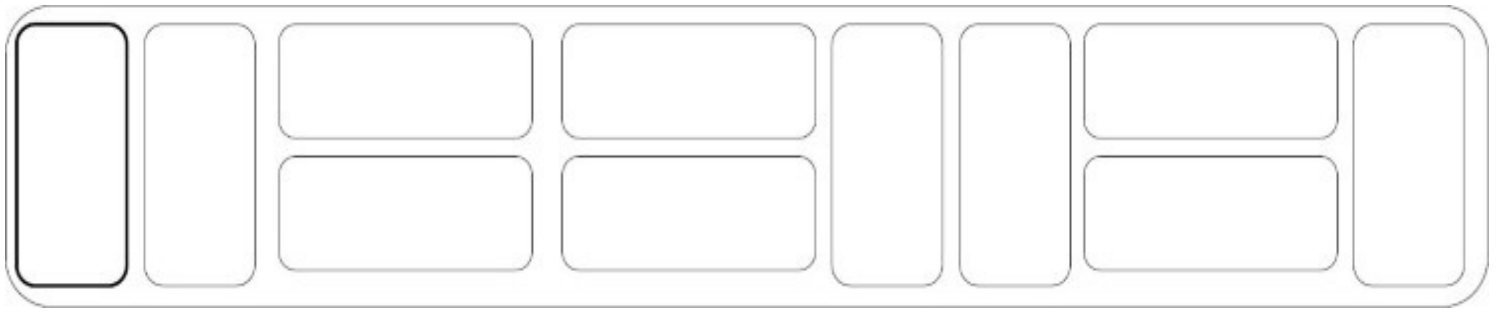
- Selecting the  $j^{th}$  coin: Here also the argument is the same as above. If we select the  $j^{th}$  coin, then the remaining range is from  $i$  to  $j-1$ . Since we selected the  $j^{th}$  coin we get the value  $v_j$  for that. From the remaining range  $i$  to  $j - 1$ , the opponent can select either the  $i^{th}$  coin or the  $j - 1^{th}$  coin. But the opponent's selection should be minimized as much as possible [the *Min* term].



**How many subproblems are there?** In the above formula,  $i$  can range from 1 to  $n$  and  $j$  can range from 1 to  $n$ . There are a total of  $n^2$  subproblems and each takes  $O(1)$  and the total time complexity is  $O(n^2)$ .

**Problem-34 Tiling:** Assume that we use dominoes measuring  $2 \times 1$  to tile an infinite strip of height 2. How many ways can one tile a  $2 \times n$  strip of square cells with  $1 \times 2$  dominoes?

**Solution:** Notice that we can place tiles either vertically or horizontally. For placing vertical tiles, we need a gap of at least  $2 \times 2$ . For placing horizontal tiles, we need a gap of  $2 \times 1$ . In this manner, the problem is reduced to finding the number of ways to partition  $n$  using the numbers 1 and 2 with order considered relevant [1]. For example:  $11 = 1 + 2 + 2 + 1 + 2 + 2 + 1$ .



If we have to find such arrangements for 12, we can either place a 1 at the end or we can add 2 in the arrangements possible with 10. Similarly, let us say we have  $F_n$  possible arrangements for  $n$ . Then for  $(n + 1)$ , we can either place just 1 at the end or we can find possible arrangements for  $(n - 1)$  and put a 2 at the end. Going by the above theory:

$$F_{n+1} = F_n + F_{n-1}$$

Let's verify the above theory for our original problem:

- In how many ways can we fill a  $2 \times 1$  strip: 1  $\rightarrow$  Only one vertical tile.
- In how many ways can we fill a  $2 \times 2$  strip: 2  $\rightarrow$  Either 2 horizontal or 2 vertical tiles.
- In how many ways can we fill a  $2 \times 3$  strip: 3  $\rightarrow$  Either put a vertical tile in the 2 solutions possible for a  $2 \times 2$  strip, or put 2 horizontal tiles in the only solution possible for a  $2 \times 1$  strip.  $(2 + 1 = 3)$ .
- Similarly, in how many ways can we fill a  $2 \times n$  strip: Either put a vertical tile in the solutions possible for  $2 \times (n - 1)$  strip or put 2 horizontal tiles in the solution possible for a  $2 \times (n - 2)$  strip.  $(F_{n-1} + F_{n-2})$ .
- That's how we verified that our final solution is:  $F_n = F_{n-1} + F_{n-2}$  with  $F_1 = 1$  and  $F_2 = 2$ .

**Problem-35 Longest Palindrome Subsequence:** A sequence is a palindrome if it reads the same whether we read it left to right or right to left. For example A, C, G, G, G, G, C, A. Given a sequence of length  $n$ , devise an algorithm to output the length of the longest palindrome subsequence. For example, the string A, G, C, T, C, B, M, A, A, C, T, G, G, A, M has many palindromes as subsequences, for instance: A, G, T, C, M, C, T, G, A has length 9.

**Solution:** Let us use DP to solve this problem. If we look at the sub-string  $A[i, \dots, j]$  of the string  $A$ , then we can find a palindrome sequence of length at least 2 if  $A[i] == A[j]$ . If they are not the same, then we have to find the maximum length palindrome in subsequences  $A[i + 1, \dots, j]$  and  $A[i, \dots, j - 1]$ .

Also, every character  $A[i]$  is a palindrome of length 1. Therefore the base cases are given by  $A[i, i] = 1$ . Let us define the maximum length palindrome for the substring  $A[i, \dots, j]$  as  $L(i, j)$ .

$$L(i, j) = \begin{cases} L(i + 1, j - 1) + 2, & \text{if } A[i] == A[j] \\ \text{Max}\{L(i + 1, j), L(i, j - 1)\}, & \text{otherwise} \end{cases}$$

$$L(i, i) = 1 \text{ for all } i = 1 \text{ to } n$$

```

int LongestPalindromeSubsequence(int A[], int n) {
    int max = 1;
    int i, k, L[n][n];
    for (i = 1; i <= n - 1; i++) {
        L[i][i] = 1;
        if (A[i] == A[i + 1]) {
            L[i][i + 1] = 1;
            max = 2;
        }
        else
            L[i][i + 1] = 0;
    }
    for (k = 3; k <= n; k++) {
        for (i = 1; i <= n - k + 1; i++) {
            j = i + k - 1;
            if (A[i] == A[j]) {
                L[i][j] = 2 + L[i + 1][j - 1];
                max = k;
            }
            else
                L[i][j] = max(L[i + 1][j - 1], L[i][j - 1]);
        }
    }
    return max;
}

```

Time Complexity: First 'for' loop takes  $O(n)$  time while the second 'for' loop takes  $O(n - k)$  which is also  $O(n)$ . Therefore, the total running time of the algorithm is given by  $O(n^2)$ .

**Problem-36 Longest Palindrome Substring:** Given a string  $A$ , we need to find the longest sub-string of  $A$  such that the reverse of it is exactly the same.

**Solution:** The basic difference between the longest palindrome substring and the longest palindrome subsequence is that, in the case of the longest palindrome substring, the output string should be the contiguous characters, which gives the maximum palindrome; and in the case of the longest palindrome subsequence, the output is the sequence of characters where the characters might not be contiguous but they should be in an increasing sequence with respect to their positions in the given string.



Brute-force solution exhaustively checks all  $n(n + 1) / 2$  possible substrings of the given  $n$ -length string, tests each one if it's a palindrome, and keeps track of the longest one seen so far. This has worst-case complexity  $O(n^3)$ , but we can easily do better by realizing that a palindrome is centered on either a letter (for odd-length palindromes) or a space between letters (for even-length palindromes). Therefore we can examine all  $n + 1$  possible centers and find the longest palindrome for that center, keeping track of the overall longest palindrome. This has worst-case complexity  $O(n^2)$ .

Let us use DP to solve this problem. It is worth noting that there are no more than  $O(n^2)$  substrings in a string of length  $n$  (while there are exactly  $2^n$  subsequences). Therefore, we could scan each substring, check for a palindrome, and update the length of the longest palindrome substring discovered so far. Since the palindrome test takes time linear in the length of the substring, this idea takes  $O(n^3)$  algorithm. We can use DP to improve this. For  $1 \leq i \leq j \leq n$ , define

$$L(i, j) = \begin{cases} 1, & \text{if } A[i] \dots A[j] \text{ is a palindrome substring,} \\ 0, & \text{otherwise} \end{cases}$$

$$L[i, i] = 1,$$

$$L[i, j] = L[i, i + 1], \text{ if } A[i] == A[i + 1], \text{ for } 1 \leq i \leq j \leq n - 1.$$

Also, for string of length at least 3,

$$L[i, j] = (L[i + 1, j - 1] \text{ and } A[i] = A[j]).$$

Note that in order to obtain a well-defined recurrence, we need to explicitly initialize two distinct diagonals of the boolean array  $L[i, j]$ , since the recurrence for entry  $[i, j]$  uses the value  $[i - 1, j - 1]$ , which is two diagonals away from  $[i, j]$  (that means, for a substring of length  $k$ , we need to know the status of a substring of length  $k - 2$ ).

```

int LongestPalindromeSubstring(int A[], int n) {
    int max = 1;
    int i, k, L[n][n];
    for (i = 1; i <= n-1; i++) {
        L[i][i] = 1;
        if (A[i] == A[i+1]) {
            L[i][i+1] = 1;
            max = 2;
        }
        else
            L[i][i+1] = 0;
    }
    for (k=3; k <= n; k++) {
        for (i = 1; i <= n-k+1; i++) {
            j = i + k - 1;
            if (A[i] == A[j] && L[i+1][j-1]) {
                L[i][j] = 1;
                max = k;
            }
            else
                L[i][j] = 0;
        }
    }
    return max;
}

```

Time Complexity: First for loop takes  $O(n)$  time while the second for loop takes  $O(n - k)$  which is also  $O(n)$ . Therefore the total running time of the algorithm is given by  $O(n^2)$ .

**Problem-37** Given two strings  $S$  and  $T$ , give an algorithm to find the number of times  $S$  appears in  $T$ . It's not compulsory that all characters of  $S$  should appear contiguous to  $T$ . For example, if  $S = ab$  and  $T = abadcb$  then the solution is 4, because  $ab$  is appearing 4 times in  $abadcb$ .

**Solution:**

**Input:** Given two strings  $S[1.. m]$  and  $T[1 ...m]$ .

**Goal:** Count the number of times that  $S$  appears in  $T$ .

Assume  $L(i,j)$  represents the count of how many times  $i$  characters of  $S$  are appearing in  $j$  characters of  $T$ .

$$L(i, j) = \text{Max} \begin{cases} 0, & \text{if } j = 0 \\ 1, & \text{if } i = 0 \\ L(i-1, j-1) + 1, & \text{if } S[i] == T[j] \\ L(i-1, j), & \text{if } S[i] \neq T[j] \end{cases}$$

If we concentrate on the components of the above recursive formula,

- If  $j = 0$ , then since  $T$  is empty the count becomes 0.
- If  $i = 0$ , then we can treat empty string  $S$  also appearing in  $T$  and we can give the count as 1.
- If  $S[i] == T[i]$ , it means  $i^{\text{th}}$  character of  $S$  and  $j^{\text{th}}$  character of  $T$  are the same. In this case we have to check the subproblems with  $i-1$  characters of  $S$  and  $j-1$  characters of  $T$  and also we have to count the result of  $i$  characters of  $S$  with  $j-1$  characters of  $T$ . This is because even all  $i$  characters of  $S$  might be appearing in  $j-1$  characters of  $T$ .
- If  $S[i] \neq T[i]$ , then we have to get the result of subproblem with  $i-1$  characters of  $S$  and  $j$  characters of  $T$ .

After computing all the values, we have to select the one which gives the maximum count.

**How many subproblems are there?** In the above formula,  $i$  can range from 1 to  $m$  and  $j$  can range from 1 to  $n$ . There are a total of  $mn$  subproblems and each one takes  $O(1)$ . Time Complexity is  $O(mn)$ .

Space Complexity:  $O(mn)$  where  $m$  is number of rows and  $n$  is number of columns in the given matrix.

**Problem-38** Given a matrix with  $n$  rows and  $m$  columns ( $n \times m$ ). In each cell there are a number of apples. We start from the upper-left corner of the matrix. We can go down or right one cell. Finally, we need to arrive at the bottom-right corner. Find the maximum number of apples that we can collect. When we pass through a cell, we collect all the apples left there.

**Solution:** Let us assume that the given matrix is  $A[n][m]$ . The first thing that must be observed is that there are at most 2 ways we can come to a cell - from the left (if it's not situated on the first column) and from the top (if it's not situated on the most upper row).

				$S[i-1][j]$	
				↓	
		$S[i][j-1]$	→	$S[i][j]$	

To find the best solution for that cell, we have to have already found the best solutions for all of the cells from which we can arrive to the current cell. From above, a recurrent relation can be easily obtained as:

$$S(i, j) = \left\{ A[i][j] + \text{Max} \begin{cases} S(i, j - 1), & \text{if } j > 0 \\ S(i - 1, j), & \text{if } i > 0 \end{cases} \right\}$$

$S(i, j)$  must be calculated by going first from left to right in each row and process the rows from top to bottom, or by going first from top to bottom in each column and process the columns from left to right.

```
int FindApplesCount(int A[][], int n, int m) {
    int S[n][m];
    for( int i = 1; i <= n; i++) {
        for(int j = 1; j <= m; j++) {
            S[i][j] = A[i][j];
            if(j > 0 && S[i][j] < S[i][j] + S[i][j-1])
                S[i][j] += S[i][j-1];
            if(i > 0 && S[i][j] < S[i][j] + S[i-1][j])
                S[i][j] += S[i-1][j];
        }
    }
    return S[n][m];
}
```

**How many such subproblems are there?** In the above formula,  $i$  can range from 1 to  $n$  and  $j$  can range from 1 to  $m$ . There are a total of  $nm$  subproblems and each one takes  $O(1)$ . Time Complexity is  $O(nm)$ . Space Complexity:  $O(nm)$ , where  $m$  is number of rows and  $n$  is number of columns in the given matrix.

**Problem-39** Similar to [Problem-38](#), assume that we can go down, right one cell, or even in a diagonal direction. We need to arrive at the bottom-right corner. Give DP solution to find the maximum number of apples we can collect.

**Solution: Yes.** The discussion is very similar to [Problem-38](#). Let us assume that the given matrix is  $A[n][m]$ . The first thing that must be observed is that there are at most 3 ways we can come to a cell - from the left, from the top (if it's not situated on the uppermost row) or from the top diagonal. To find the best solution for that cell, we have to have already found the best solutions for all of the cells from which we can arrive to the current cell. From above, a recurrent relation can be easily obtained:

$$S(i,j) = \begin{cases} A[i][j] + \text{Max} \begin{cases} S(i,j-1), & \text{if } j > 0 \\ S(i-1,j), & \text{if } i > 0 \\ S(i-1,j-1), & \text{if } i > 0 \text{ and } j > 0 \end{cases} \end{cases}$$

$S(i,j)$  must be calculated by going first from left to right in each row and process the rows from top to bottom, or by going first from top to bottom in each column and process the columns from left to right.

		$S[i-1][j-1]$	$S[i-1][j]$		
		$S[i][j-1]$	$S[i][j]$		

**How many such subproblems are there?** In the above formula,  $i$  can range from 1 to  $n$  and  $j$  can range from 1 to  $m$ . There are a total of  $mn$  subproblems and each one takes  $O(1)$ . Time Complexity is  $O(nm)$ .

Space Complexity:  $O(nm)$  where  $m$  is number of rows and  $n$  is number of columns in the given matrix.

**Problem-40 Maximum size square sub-matrix with all 1's:** Given a matrix with 0's and 1's, give an algorithm for finding the maximum size square sub-matrix with all 1s. For example, consider the binary matrix below.

```

0  1  1  0  1
1  1  0  1  0
0  1  1  1  0
1  1  1  1  0
1  1  1  1  1
0  0  0  0  0

```

The maximum square sub-matrix with all set bits is

```

1  1  1
1  1  1
1  1  1

```

**Solution:** Let us try solving this problem using DP. Let the given binary matrix be  $B[m][m]$ . The idea of the algorithm is to construct a temporary matrix  $L[][]$  in which each entry  $L[i][j]$  represents size of the square sub-matrix with all 1's including  $B[i][j]$  and  $B[i][j]$  is the rightmost

and bottom-most entry in the sub-matrix.

**Algorithm:**

- 1) Construct a sum matrix  $L[m][n]$  for the given matrix  $B[m][n]$ .
  - a. Copy first row and first columns as is from  $B[i][j]$  to  $L[i][j]$ .
  - b. For other entries, use the following expressions to construct  $L[i][j]$   
$$\text{if}(B[i][j] )$$
$$L[i][j] = \min(L[i][j - 1], L[i - 1][j], L[i - 1][j - 1]) + 1;$$
$$\text{else } L[i][j] = 0;$$
- 2) Find the maximum entry in  $L[m][n]$ .
- 3) Using the value and coordinates of maximum entry in  $L[i]$ , print sub-matrix of  $B[i][j]$ .

```

void MatrixSubSquareWithAllOnes(int B[][], int m, int n) {
    int i, j, L[m][n], max_of_s, max_i, max_j;
    // Setting first column of L[][]
    for(i = 0; i < m; i++)
        L[i][0] = B[i][0];
    // Setting first row of L[][]
    for(j = 0; j < n; j++)
        L[0][j] = B[0][j];
    // Construct other entries of L[][]
    for(i = 1; i < m; i++) {
        for(j = 1; j < n; j++) {
            if(B[i][j] == 1)
                L[i][j] = min(L[i][j-1], L[i-1][j], L[i-1][j-1]) + 1;
            else
                L[i][j] = 0;
        }
    }
    max_of_s = L[0][0]; max_i = 0; max_j = 0;
    for(i = 0; i < m; i++) {
        for(j = 0; j < n; j++) {
            if(L[i][j] > max_of_s){
                max_of_s = L[i][j];
                max_i = i;
                max_j = j;
            }
        }
    }
    printf("Maximum sub-matrix");
    for(i = max_i; i > max_i - max_of_s; i--) {
        for(j = max_j; j > max_j - max_of_s; j--)
            printf("%d", B[i][j]);
    }
}

```

**How many subproblems are there?** In the above formula,  $i$  can range from 1 to  $n$  and  $j$  can range from 1 to  $m$ . There are a total of  $nm$  subproblems and each one takes  $O(1)$ . Time Complexity is  $O(nm)$ . Space Complexity is  $O(nm)$ , where  $n$  is number of rows and  $m$  is number of columns in the given matrix.

**Problem-41 Maximum size sub-matrix with all 1's:** Given a matrix with 0's and 1's, give an algorithm for finding the maximum size sub-matrix with all 1s. For example, consider

the binary matrix below.

1	1	0	0	1	0
0	1	1	1	1	1
1	1	1	1	1	0
0	0	1	1	0	0

The maximum sub-matrix with all set bits is

1	1	1	1
1	1	1	1

**Solution:** If we draw a histogram of all 1's cells in the above rows for a particular row, then maximum all 1's sub-matrix ending in that row will be equal to maximum area rectangle in that histogram. Below is an example for 3<sup>rd</sup> row in the above discussed matrix [1]:

1	1	0	0	1	0
0	1	1	1	1	1
1	1	1	1	1	0
0	0	1	1	0	0

If we calculate this area for all the rows, maximum area will be our answer. We can extend our solution very easily to find start and end co-ordinates. For this, we need to generate an auxiliary matrix  $S[][]$  where each element represents the number of 1s above and including it, up until the first 0.  $S[][]$  for the above matrix will be as shown below:

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

Now we can simply call our maximum rectangle in histogram on every row in  $S[][]$  and update the maximum area every time. Also we don't need any extra space for saving  $S$ . We can update original matrix ( $A$ ) to  $S$  and after calculation, we can convert  $S$  back to  $A$ .



```

#define ROW 10
#define COL 10
int find_max_matrix(int A[ROW][COL]) {
    int max, cur_max = 0;
    //Calculate Auxiliary matrix
    for (int i=1; i<ROW; i++)
        for(int j=0; j<COL; j++) {
            if(A[i][j] == 1)
                A[i][j] = A[i-1][j] + 1;
        }
    //Calculate maximum area in S for each row
    for (int i=0; i<ROW; i++) {
        max = MaxRectangleArea(A[i], COL);           //Refer Stacks Chapter
        if(max > cur_max)
            cur_max = max;
    }
    //Regenerate Original matrix
    for (int i=ROW-1; i>0; i--)
        for(int j=0; j<COL; j++) {
            if(A[i][j])
                A[i][j] = A[i][j] - A[i-1][j];
        }
    return cur_max;
}

```

**Problem-42**      **Maximum sum sub-matrix:** Given an  $n \times n$  matrix  $M$  of positive and negative integers, give an algorithm to find the sub-matrix with the largest possible sum.

**Solution:** Let  $Aux[r, c]$  represent the sum of rectangular subarray of  $M$  with one corner at entry  $[1,1]$  and the other at  $[r,c]$ . Since there are  $n^2$  such possibilities, we can compute them in  $O(n^2)$  time. After computing all possible sums, the sum of any rectangular subarray of  $M$  can be computed in constant time. This gives an  $O(n^4)$  algorithm: we simply guess the lower-left and the upper-right corner of the rectangular subarray and use the  $Aux$  table to compute its sum.

**Problem-43**      Can we improve the complexity of [Problem-42](#)?

**Solution:** We can use the [Problem-4](#) solution with little variation, as we have seen that the maximum sum array of a 1 – D array algorithm scans the array one entry at a time and keeps a running total of the entries. At any point, if this total becomes negative, then set it to 0. This algorithm is called *Kadane's* algorithm. We use this as an auxiliary function to solve a two-dimensional problem in the following way.

```

public void FindMaximumSubMatrix(int[][] A, int n){
    //computing the vertical prefix sum for columns
    int[][] M = new int[n][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (j == 0)
                M[j][i] = A[j][i];
            else
                M[j][i] = A[j][i] + M[j - 1][i];
        }
    }
    int maxSoFar = 0, min, subMatrix;
    //iterate over the possible combinations applying Kadane's Alg.
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            min = 0;
            subMatrix = 0;
            for (int k = 0; k < n; k++) {
                if (i == 0)
                    subMatrix += M[j][k];
                else subMatrix += M[j][k] - M[i - 1][k];
                if(subMatrix < min)
                    min = subMatrix;
                if((subMatrix - min) > maxSoFar)
                    maxSoFar = subMatrix - min;
            }
        }
    }
}

```

Time Complexity:  $O(n^3)$ .

**Problem-44** Given a number  $n$ , find the minimum number of squares required to sum a given number  $n$ .

*Examples:*  $\min[1] = 1 = 1^2$ ,  $\min[2] = 2 = 1^2 + 1^2$ ,  $\min[4] = 1 = 2^2$ ,  $\min[13] = 2 = 3^2 + 2^2$ .

**Solution:** This problem can be reduced to a coin change problem. The denominations are 1 to  $\sqrt{n}$ . Now, we just need to make change for  $n$  with a minimum number of denominations.

**Problem-45 Finding Optimal Number of Jumps To Reach Last Element:** Given an array, start from the first element and reach the last by jumping. The jump length can be at most the value at the current position in the array. The optimum result is when you reach the goal

in the minimum number of jumps. **Example:** Given array  $A = \{2,3,1,1,4\}$ . Possible ways to reach the end (index list) are:

- 0,2,3,4 (jump 2 to index 2, and then jump 1 to index 3, and then jump 1 to index 4)
- 0,1,4 (jump 1 to index 1, and then jump 3 to index 4)

Since second solution has only 2 jumps it is the optimum result.

**Solution:** This problem is a classic example of Dynamic Programming. Though we can solve this by brute-force, it would be complex. We can use the LIS problem approach for solving this. As soon as we traverse the array, we should find the minimum number of jumps for reaching that position (index) and update our result array. Once we reach the end, we have the optimum solution at last index in result array.

**How can we find the optimum number of jumps for every position (index)?** For first index, the optimum number of jumps will be zero. Please note that if value at first index is zero, we can't jump to any element and return infinite. For  $n + 1^{th}$  element, initialize  $result[n + 1]$  as infinite. Then we should go through a loop from 0 ...  $n$ , and at every index  $i$ , we should see if we are able to jump to  $n + 1$  from  $i$  or not. If possible, then see if total number of jumps ( $result[i] + 1$ ) is less than  $result[n + 1]$ , then update  $result[n + 1]$ , else just continue to next index.

```

//Define MAX 1 less so that adding 1 doesn't make it 0
#define MAX 0xFFFFFFFF;
unsigned int jump(int *array, int n) {
    unsigned answer, int *result = new unsigned int[n];
    int i, j;
    //Boundary conditions
    if(n==0 || array[0] == 0)
        return MAX;
    result[0] = 0; //no need to jump at first element
    for (i = 1; i < n; i++) {
        result[i] = MAX; //Initialization of result[i]
        for (j = 0; j < i; j++) {
            //check if jump is possible from j to i
            if(array[j] >= (i-j)) {
                //check if better solution available
                if((result[j] + 1) < result[i])
                    result[i] = result[j] + 1; //updating result[i]
            }
        }
    }
    answer = result[n-1]; //return result[n-1]
    delete[] result;
    return answer;
}

```

The above code will return optimum number of jumps. To find the jump indexes as well, we can very easily modify the code as per requirement.

**Time Complexity:** Since we are running 2 loops here and iterating from 0 to  $i$  in every loop then total time takes will be  $1 + 2 + 3 + 4 + \dots + n - 1$ . So time efficiency  $O(n) = O(n * (n - 1)/2) = O(n^2)$ .

**Space Complexity:**  $O(n)$  space for result array.

**Problem-46** Explain what would happen if a dynamic programming algorithm is designed to solve a problem that does not have overlapping sub-problems.

**Solution:** It will be just a waste of memory, because the answers of sub-problems will never be used again. And the running time will be the same as using the Divide & Conquer algorithm.

**Problem-47** Christmas is approaching. You're helping Santa Claus to distribute gifts to children. For ease of delivery, you are asked to divide  $n$  gifts into two groups such that the weight difference of these two groups is minimized. The weight of each gift is a positive integer. Please design an algorithm to find an optimal division minimizing the value

difference. The algorithm should find the minimal weight difference as well as the groupings in  $O(nS)$  time, where  $S$  is the total weight of these  $n$  gifts. Briefly justify the correctness of your algorithm.

**Solution:** This problem can be converted into making one set as close to  $\frac{S}{2}$  as possible. We consider an equivalent problem of making one set as close to  $W = \left\lfloor \frac{S}{2} \right\rfloor$  as possible. Define  $FD(i, w)$  to be the minimal gap between the weight of the bag and  $W$  when using the first  $i$  gifts only. WLOG, we can assume the weight of the bag is always less than or equal to  $W$ . Then fill the DP table for  $0 \leq i \leq n$  and  $0 \leq w \leq W$  in which  $F(0, w) = W$  for all  $w$ , and

$$FD(i, w) = \min\{FD(i-1, w-w_i)-w_i, FD(i-1, w)\} \text{ if } \{FD(i-1, w-w_i) \geq w_i\} \\ = FD(i-1, w) \text{ otherwise}$$

This takes  $O(nS)$  time.  $FD(n, W)$  is the minimum gap. Finally, to reconstruct the answer, we backtrack from  $(n, W)$ . During backtracking, if  $FD(i, j) = FD(i-1, j)$  then  $i$  is not selected in the bag and we move to  $F(i-1, j)$ . Otherwise,  $i$  is selected and we move to  $F(i-1, j-w_i)$ .

**Problem-48** A circus is designing a tower routine consisting of people standing atop one another's shoulders. For practical and aesthetic reasons, each person must be both shorter and lighter than the person below him or her. Given the heights and weights of each person in the circus, write a method to compute the largest possible number of people in such a tower.

**Solution:** It is same as Box stacking and Longest increasing subsequence (LIS) problem.