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 of the cells on which it depends.

Time Complexity: $O(mn)$, since i takes values from 1 to m and 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 $X[i \ldots m]$ and $Y[j \ldots n]$. We can solve the problem by changing the definition as $LCS(i, j)$ is the length of the LCS with $X[1 \ldots i]$ and $Y[1 \ldots j]$.

Printing the subsequence: 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 $LCS[0][0]$ was the maximum of 3 values of the neighboring cells. So we simply recompute $LCS[0][0]$ 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 that record, for each cell, 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 DP technique.

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

19.8 Problems on Dynamic Programming

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 T(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 * T(i) * T(i-1);
        return sum;
}
```

Problem-2 Can we improve Problem-1 using memoization of DP?

Solution: **Yes.** Before going for solution, let us see how the values are being 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 table for avoiding these repeated calculations and the implementation can be given as:


```
int T(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 subproblem calculations are dependent only on previous calculations, code can be modified as:

```
int T(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 $A(i) \ldots A(j)$ for which the sum of elements is maximum.
Example: {-2, 11, -4, 13, -5, 2} → 20 and {1, -3, 4, -2, -1, 6} → 7

Solution: Input: Array $A(1) \ldots A(n)$ 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 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 MaxContigousSum(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 $i, \ldots, j-1$, then we need only one more addition to get the sum for the subsequence $i, \ldots, j$. 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 MaxContigousSum(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 $A[i]$

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

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 MaxContigousSum(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 little tricky. One simple way is to look for all positive contiguous segments of the array (*sumEndingHere*) and keep track of maximum sum

contiguous segment among all positive segments (*sumSoFar*). Each time we get a positive sum compare it with *sumSoFar* and update *sumSoFar* if it is greater than *sumSoFar*. Let us consider the following code for the above observation.

```
int MaxContigousSum(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: Algorithm doesn't work if the input contains all negative numbers. It returns 0 if all numbers are negative. For handling this we can add an extra check before actual implementation. The phase will look if all numbers are negative, 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]

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

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 $A[i]$.

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

Note: For $O(nlogn)$ solution refer *Divide and Conquer* chapter.

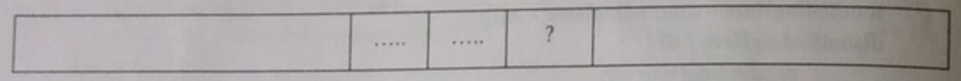
Problem-9 Given a sequence of n numbers $A(1) \dots A(n)$, give an algorithm for finding a contiguous subsequence $A(i) \dots A(j)$ for which the sum of elements in the subsequence is maximum. Here the condition is we should not select *two* contiguous numbers.

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 i^{th} element or not. This gives us two possibilities and based on this we can write the recursive formula as:

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

- The first case indicates whether we are selecting the i^{th} element or not. If we don't select the i^{th} element then we have to maximize the sum using the elements 1 to $i-1$. If i^{th} element is selected then we should not select $i-1^{th}$ 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^{th} element



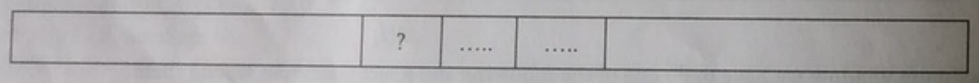

```
int maxSumWithNoTwoContinuousNumbers(int A[], int n) {
        int M[n+1];
        M[0]=A[0];
        M[1]=(A[0]>A[1]?A[0]:A[i]);
        for(i=2, i<n; i++)
                M[i]= (M[i-1]>M[i-2]+A[i]? M[i-1]: M[i-2]+A[i]);
        return M[n-1];
}
```

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

Problem-10 In Problem-9 solution, 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^{th} element



As similar to Problem-9 solution, we can write the recursive formula as:

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

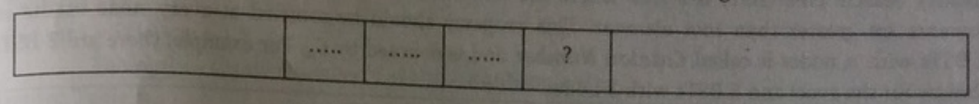
- The first case indicates whether we are selecting the i^{th} element or not. If we don't select the i^{th} element then we have to maximize the sum using the elements $i+1$ to n. If i^{th} element is selected then we should not select $i+1^{th}$ element need to maximize the sum using $i+2$ to n elements.
- 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 $A(1) \ldots A(n)$, give an algorithm for finding a contiguous subsequence $A(i) \ldots A(j)$ 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 $A(1) \ldots A(n)$ of n numbers.

Given Array, A: recursive formula considers the case of selecting i^{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^{th} element or not. This gives us the following possibilities:

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

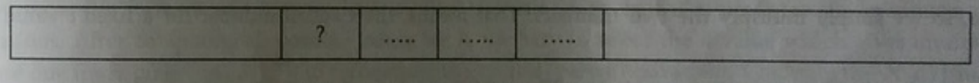
- In the given problem the restriction is not to select three continuous numbers, but we can select two elements continuously and skip the third one. That is what the first case says in the above recursive formula. That means we are skipping $A[i-2]$.
- The other possibility is, selecting i^{th} element and skipping second $i-1^{th}$ element. This is the second case (skipping $A[i-1]$).
- The third term defines the case of 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-12 In Problem-11 solution, 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



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:

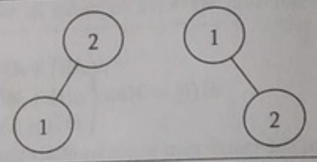
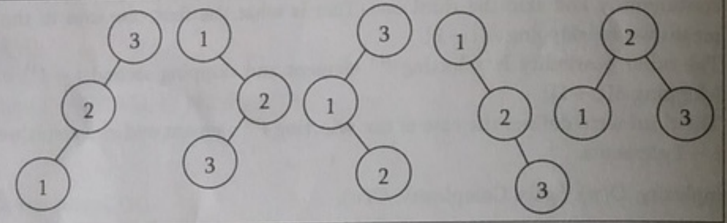
$$M(i) = Max\begin{cases} A[i] + A[i+1] + M(i+3) \\ A[i] + M(i+2) \\ M(i+1) \end{cases}$$

- 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^{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 root element and right subtree elements are greater than 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 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 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 is already 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: $A_1 \times A_2 \times A_3 \times \ldots \times A_n$ 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 $A_1 \times A_2 \times A_3 \times \ldots \times A_n$, 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 optimal number of multiplications needed to compute $A_1 \times A_2 \times A_3 \times \ldots \times A_n$.

For matrix multiplication problem, there could be many possibilities. This is because of the fact that 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 possibility produces different number of products during the multiplication. To select the best one, we can go through each possible parenthesizations (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 $A_i \cdots A_j$.

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

The above recursive formula says that we have to find the 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 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 $n-1$ such operations [since the total number of operations we need for $A_1 \times A_2 \times A_3 \times \ldots \times A_n$ are $n-1$]. 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 which 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 dimentions 3×100, 100×2 and 2×2. 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. ∴ 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 *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 tha knapsack with $C-2$ size and get the maximum value.

Let us say M(j) denote 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:

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

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 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 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 if the items are not duplicated (not having infinite number of items for each type and each item is allowed to use for 0 or 1 time)?

Real-time example: Suppose if we are going by flight, 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

selecting the items with maximum value. That means, we need to tell the customs officer to select the items which has more weight and less value (profit).

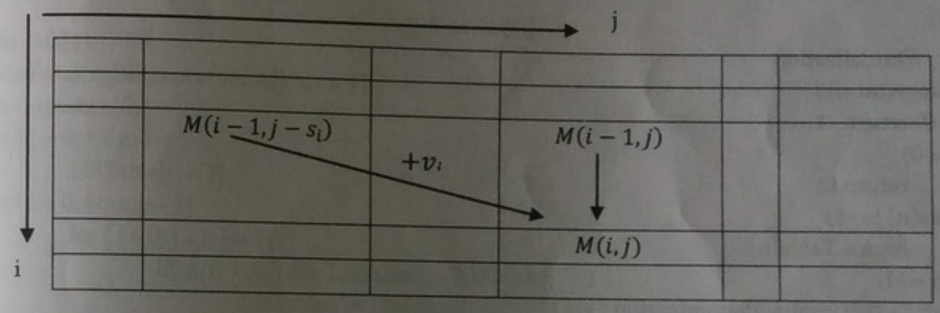
Solution: Input is a set of n items with sizes s_i and values v_i and a Kanpsack of size C which we need to fill with subset of items from the given set. Let us try to find the recursive formula for this problem using DP. Let $M(i,j)$ represents the optimal value we can get for filling up a knapsack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

$$M(i,j) = Max\{M(i-1,j), M(i-1,j-s_i) + v_i\}$$

(i^{th} item is not used; i^{th} item is 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 take values from $1 \ldots n$ and j takes values from $1 \ldots C$, 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 i^{th} item. In this case, since we are not adding any size to knapsack we have to use the same knapsack size for subproblems but excluding i^{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^{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 optimal solution. The remaining items are $i-1$.

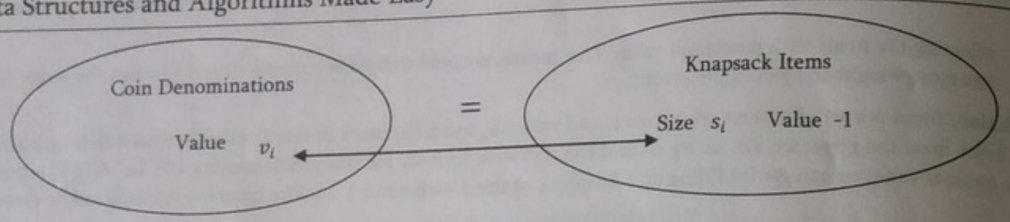
Now, after finding all $M(i,j)$ values the optimal objective value can be obtained as: $Max_j\{M(n,j)\}$
This is because we do not know what amount of capacity gives the best solution.

Inorder 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 $M(i-1,j-s_i)$) appears in previous row and also in some previous column. So, $M(i,j)$ can be computed just by looking at two values in the previous row of the table.

Problem-19 **Making Change:** Given n types of coin denominations of values $v_1 < v_2 < \ldots < v_n$ (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 has a size s_i, which is equal to the value of v_i coin denomination. In the Knapsack we can give value of every item as -1.

Now it is easy to understand that optimal way to make money C with 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 corresponds to as few coins as possible.



Optimal way to make change for amount of money equal to C ⟷ Optimal way to exactly fill a capacity C Knapsack.

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

$$M(j) = Min_i\{M(j - v_i)\} + 1$$

What this says is, if coin denomination i was the last denomination coin added to 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 $A_1 \ldots A_n$, 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 $A_1 \ldots A_n$.

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

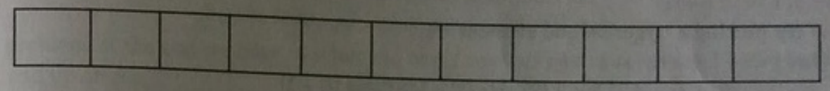
First, let us concentrate on the algorithm for finding the longest subsequence. Later, we can try printing the sequence itself by tracing the table. Our first step is finding the recursive formula. First, let us create the base conditions. If there is only one element in the input sequence then we don't have to solve the problem and just need to return that element. For any sequence we can start with the first element ($A[1]$). Since we know what 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, then 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 $L(i)$ represents 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 going to be to extend some subsequence starting at some earlier position j. For this the recursive formula can written as:

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

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

$$Max_i\{L(i)\}$$

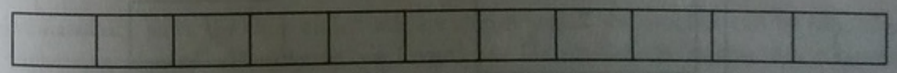
```
int LISTable [1024];
int LongestIncreasingSequence( int A[], int n ) {
        int i, j, max = 0;
        for ( i = 0; i < n; i++ )
                LISTable[i] = 1;
        for ( i = 0; i< n; i++ ) {
                for ( j = 0; j < i; j++ ) {
                        if( A[i] > A[j] && LISTable[i]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

i . j ..... n



Let $L(i)$ represents 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 written as:

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

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

$$Max_i\{L(i)\}$$

```
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.

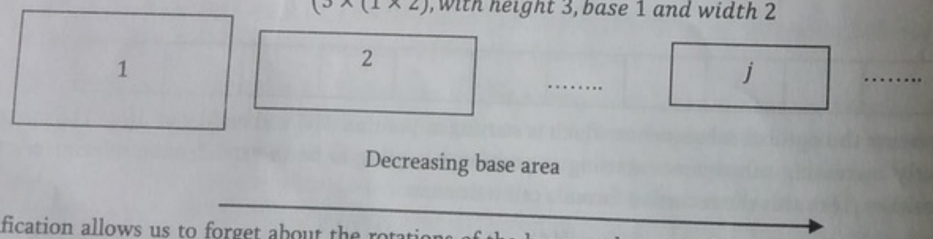
Problem-22 Is there any alternative way of solving the *Problem-21?*

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,

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



This simplification allows us to forget about the rotations of the boxes and we just focus on 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 which is as tall as possible which 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 LIS problem because stack of n boxes with ending box j is equal to finding a subsequence with 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 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 some previous stack which is ending at i. That means, we need to put j box on top of stack [i box is the current top of stack]. To put j box on top of stack we should satisfy the condition $w_i > w_j and\ d_i > d_j$ [these ensures the low level box has more base than boxes above it]. Based on this logic, we can write the recursive formula as:

$$H(j) = Max_{i<j\ and\ w_i>w_j and\ d_i>d_j}\{H(i)\} + h_i$$

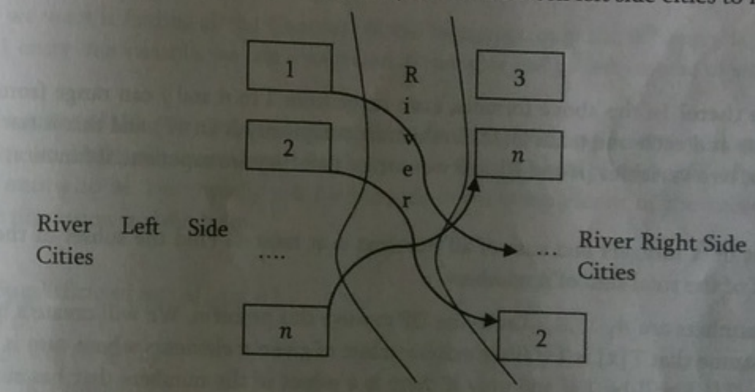
As similar to LIS problem, at the end we have to select the best j over all potential values. This is because we not sure which box might end up on top.

$$Max_j\{H(j)\}$$

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

Problem-24 **Building Bridges in India:** Consider a very big straight river which moves from north to south. Assume there are n cities on both sides of the river: n cities on left of the river and n cities on the right of the river. Also, assume that these cities were number 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 below diagram. In the diagram it can be seen that there are n cities on left side of river and n cities on right side of river. Also, note that we are connecting the cities which have the same number [requirement in problem]. Our goal is to map maximum cities on left side of river to the cities on the right side of the river without any cross edges. Just to make it simple, lest we sort the cities on one side of the river.

If we observe carefully, since the cities on left side are already sorted, the problem can be simplified to finding the maximum increasing sequence. That means, we have to use LIS solution for finding the maximum in 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 $A_1 \ldots A_n$, 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 if 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 an $n \times T$ 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 j and $M[i, j] = 0$ otherwise.

$$M[i, j] = Max(M[i - 1, j], M[i - 1, j - A_i])$$

The above recursive formula says that: as similar to 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 by checking if the sum $j - A_i$ exists without the i^{th} element. This is identical to Knapsack, except that we are storing a $0/1$'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 exponential function of the other. Space Complexity: $O(nT)$.

Problem-26 Given a set of n integers and sum of all numbers is at most K. 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 $A_1 \ldots A_n$. 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.

In general, 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's 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 adds 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 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 actually we want is finding all the 1 entries. At the beginning, only the 0^{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 doing useless work.

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 same [same as previous problem but 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 in other way. Following are the two main steps to solve this problem:

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

The first step is simple. The second step is crucial, 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 subproblems

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 the above subproblems return true, then return true.

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

```
// A utility function that returns true if there is a subset of A[] with sun 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 findPartiion (int A[], int n){
  // Calculate sum of the elements in Aay
  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 bottom up manner such that every filled entry has following property

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

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

  // Caculcate sun 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 botton 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($sum \times n$). Space Complexity: O($sum \times n$). Please note that this solution will not be feasible for arrays with 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 expressions 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 are n and between symbols there are boolean operators like *and, or, xor* etc.. For example, if with $n = 4$, *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 like *T or ((F and T) xor F)* then it evaluates to true.

$$T\ or(\ (F\ and\ T) xor\ F) = True$$

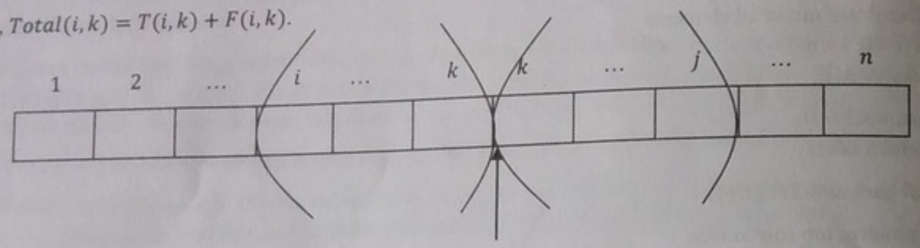
Now let us see how DP solves this problem. Let, $T(i, j)$ represents the numbers of ways to parenthesize the sub expression with symbols $i \ldots j$ [symbols means only T and F and not the operators] with boolean operators so that it evaluates to *true*. Also, i and j takes the values from 1 to n. For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

Just for simplicity and similarity, let $F(i, j)$ represents the numbers of ways to parenthesize the sub expression with symbols $i \ldots j$ 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.

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

Where, $Total(i,k)=T(i,k)+F(i,k)$.



and, or, xor

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*".

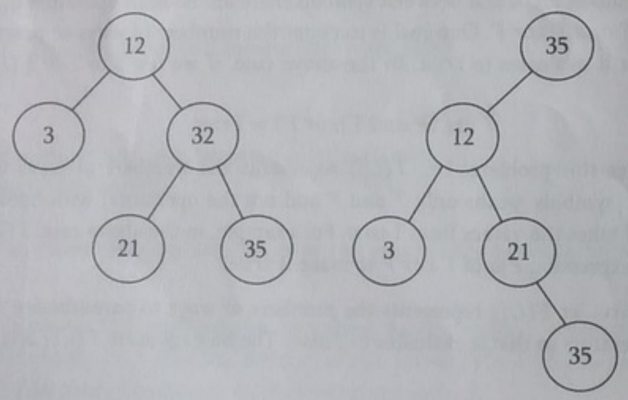
- If we use "*and*" between k and $k+1$ then the final expression becomes to *true* only when both of them are *true*. If both of them are *true* then we can include them to get the final count.
- If we use "*or*", then if at least one of them is *true* then the result becomes *true*. Instead of including all the three possibilities for "*or*", we are giving one alternative of that where we are subtracting the "false" cases from total possibilities.
- Same is the case with "*xor*". Conversation is as that of 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 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 $A = [3,12,21,32,35]$. To represent these elements there are many ways and below are two of them.



Among 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: $\frac{1+2+3+3+4}{5}=\frac{13}{5}$. Among the two, the first tree is giving 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(root) of the tree with *root* can be defined as:

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

In the above expression, $depth(root,i)+1$ indicates the number of comparisons for searching the i^{th} element. Since we are trying to create binary search tree, the left subtree elements are less than root element and 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:

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

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:

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

Binary **Search Tree node declaration**

Refer *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→left = 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 to B with minimal conversions.

Before going to solution, let us consider the possible operations for converting string A to 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 insertion of a character, replacement of a character and deletion of character and their corresponding cost codes are defined below.

Costs of operations:

Insertion of a character	c_i
Replacement of a character	c_r
Deletion of character	c_d

Now let us concentrate on 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, $A[1 \dots i]$ to $B[1 \dots j]$.

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

Based on 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 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 take $O(1)$ 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 finding 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,

$$A[i,j] = C[i,j] \ if\ i \neq j$$
$$= 0 \quad if\ i = j$$

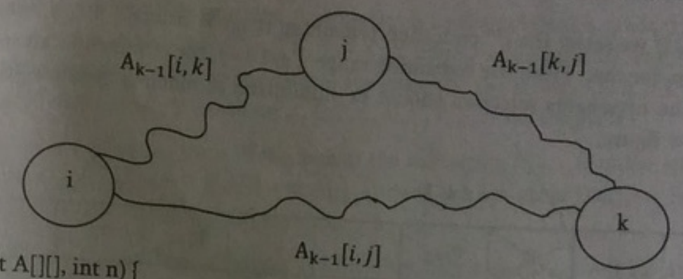
From the definition, $C[i,j] = \infty$ if there is no path from i to j. The algorithm makes n passes over A. Let $A_0, A_1, \dots, A_n$ be the values of A on the n passes, with A_0 being the initial value.

Just after the $k-1^{th}$ 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^{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 below diagram.



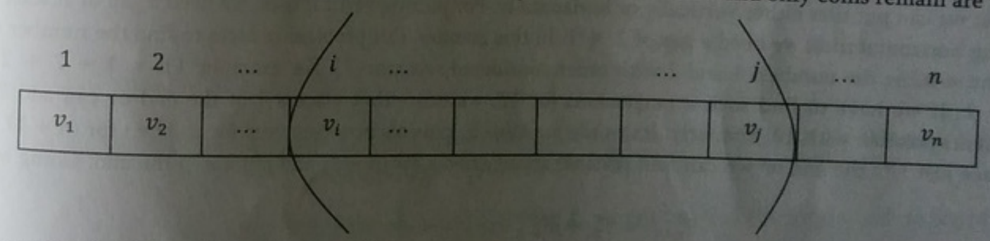

```
void Floyd(int C[][], int A[][], int n) {
        int i, j, k;
        for(i = 0, i <= n - 1;i + +)
                for(j = 0; j <= n - 1, j + +)
                        A[i][j] = C[i][j];
        for(i = 0;i <= n - 1;i + +)
                A[i][i] = 0;
        for(k = 0;k <= n - 1;k + +) {
    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

Solution: Let us solve the problem using our DP technique. In each turn either we *or* our opponent selects the coin only from 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 only coins remain are $v_i \dots v_j$.

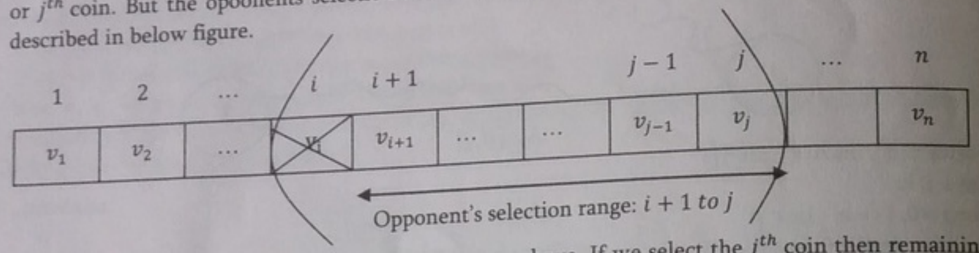


Base Cases: $V(i,i), V(i,i+1)$ for all values of i.
From these value, we can compute $V(i,i+2), V(i,i+3)$ and so on. Now let us define $V(i,j)$ for each sub problem as:

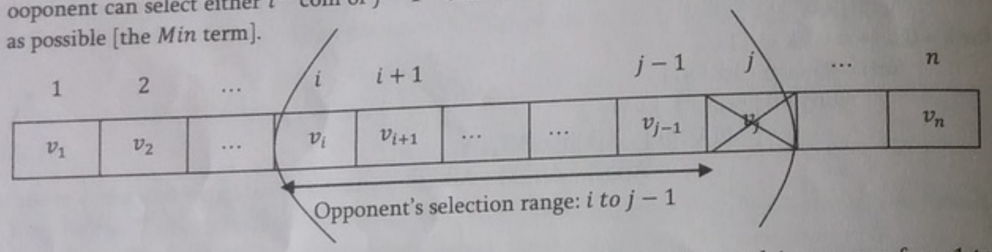
$$V(i,j) = Max\left\{Min\begin{Bmatrix} V(i+1,j-1) \\ V(i+2,j) \end{Bmatrix} + v_i, Min\begin{Bmatrix} V(i,j-2) \\ V(i+1,j-1) \end{Bmatrix} + v_j\right\}$$

In the recursive call we have to focus on i^{th} coin to j^{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^{th} coin (v_i) and second term indicates the case of selecting j^{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^{th} coin: If we select the i^{th} coin then remaining range is from $i+1$ *to* j. Since we selected i^{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 opoonents selection should be minimized as much as possible [the *Min* term]. Same is described in below figure.



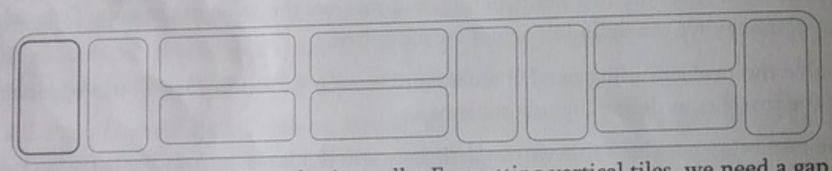
- Selecting j^{th} coin: Here also the argument is same as above. If we select the j^{th} coin then remaining range is from i to j − 1. Since we selected j^{th} coin we get the value v_j for that. From the remaining range i to j − 1, the ooponent can select either i^{th} coin or $j-1^{th}$ coin. But the opoonents 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 2×1 to tile an infinite strip of height 2. How many ways one can tile a $2 \times n$ strip of square cells with 1×2 dominoes?

Solution:



Notice that we can put tiles either vertically or horizontally. For putting vertical tiles, we need a gap of at least 2×2. For putting horizontal tiles, we need a gap of 2×1. In this manner, this problem reduces to find 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 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 in the end *or* we can find possible arrangements for $(n-1)$ and put a 2 in the end. Going by above theory:

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

Let's verify above theory for our original problem:

- In how many ways, we can fill a 2×1, strip: $1 \rightarrow$ Only one vertical tile.
- In how many ways, we can fill a 2×2, strip: $2 \rightarrow$ Either 2 horizontal or 2 vertical tiles.
- In how many ways, we can fill a 2×3, strip: $3 \rightarrow$ Either put a vertical tile in 2 solutions possible for 2×2 strip or put 2 horizontal tiles in only solution possible for 2×1 strip. $(2+1=3)$.
- Similarly in how many ways, we can fill a $2 \times n$, strip: Either put a vertical tile in solutions possible for $2X(n-1)$ strip or put 2 horizontal tiles in solution possible for $2 \times (n-2)$ strip. $(F_{n-1}+F_{n-2})$.
- That's how, we verified that our final solution: $F_n = F_{n-1} + F_{n-2}$ 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 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,..,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 same then we have to find the maximum length palindrome in subsequences A[i + 1,...,j] and A[i,...,j − 1].

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

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

```
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 longest palindrome substring and longest palindrome subsequence is that, in case of longest palindrome substring the output string should be the contiguous characters which gives the maximum palindrome and in case of longest palindrome subsequence the output is the sequence of characters where the characters might not be in contiguous but they should be in increasing sequence with respect to their positions in the given string.

Brute-force solution would be to exhaustively checking all $n(n+1)/2$ possible substrings of the given n-length string, test each one if it's a palindrome, and keep 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 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 $1 \le i \le j \le n$, define

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

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

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

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 appearing in T. It's not compulsory that all characters of S should appear contiguous in 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 string $S[1..m]$ and $T[1 \ldots m]$.

Goal: Count the number of times that S appearing in T.

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

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

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[j], means i^{th} character of S and j^{th} character of T are 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] ≠ T[j], then we have to get the result of subproblem with $i-1$ characters of S and j characters of T.

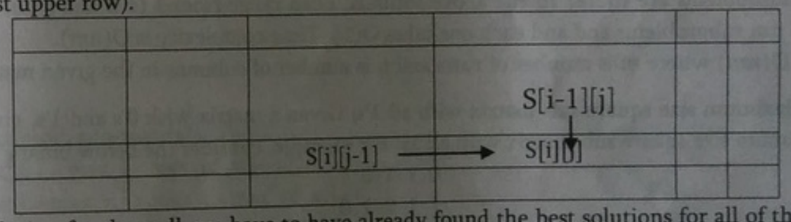
After computing all the values, we have to select the one which gives maximum count.

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 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 to 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).



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:

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

$S(i,j)$ 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;i<=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];
}
```

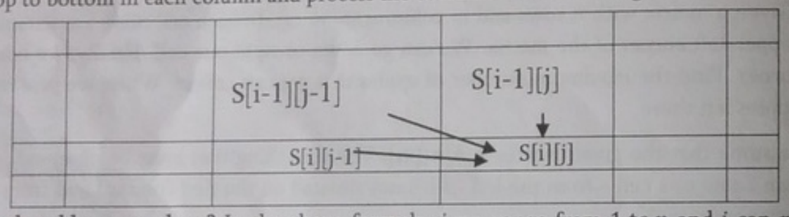
How many such subproblems are there? In the above formula, i can range from 1 *to* n and j can range from 1 *to* m. There are a total of nm subproblems and each one takes O(1). Time Complexity is O(nm). Space Complexity: O(nm), where m is number of rows and n is number of columns in the given matrix.

Problem-39 Similar to Problem-38, assume that, we can go down, right one cell or even in 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 much 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 most upper row) or from 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:

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

$S(i,j)$ 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.



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 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 below binary matrix.

```
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 an 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 bottommost entry in 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 it is from $B[\,][\,]$ to $L[\,][\,]$.
   b. For other entries, use following expressions to construct L[ ][ ]

   if($B[i][j]$)

   $$L[i][j] = min(L[i][j-1], L[i-1][j], L[i-1][j-1]) + 1;$$


```
            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 $B[\,][\,]$.

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

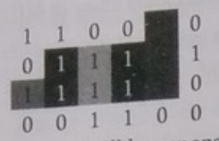
Problem-41 **Maximum size sub-matrix with all 1's:** Given a matrix with 0's and 1's, give an algorithm for finding the maximum size sub-matrix with all 1s. For example, consider the below binary matrix.

```
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 above rows for a particular row, then maximum all 1's sub-matrix ending in that row will the equal to maximum area rectangle in that histogram. Below is the example for 3^{rd} row in above discussed matrix [1]:



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 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 simple 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 Auxilary 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 Oriignal 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 $Aux[r, c]$ represents 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 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 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 the two dimensional problem in the following way [1].

```
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;
  int 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 coin change problem. The denominations are 1 to $\sqrt{n}$. Now, we just need to make change for n with 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. Optimum result is when you reach the goal in 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 but complexity in that case will be too high. We can use 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 at the end, we have the optimum solution at last index in result array.

How to find optimum number of jump for every position (index)? For first index, 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 $0 \ldots n$, and at every index i, we should see if we are able to jump to $n+1$ from there i or not. If possible then see if total number of jump ($result[i]+1$) 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 0xFFFFFFFE;
unsigned int jump(int *array, int n) {
        unsigned answer, int *result = new unsigned int[n];
        int i, j;
        //Boundry 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 is
                        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;
}
```

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 $1 + 2 + 3 + 4 + \ldots + n - 1$. So time efficiency $O(n) = O(n * (n-1)/2) = O(n^2)$.
Space Complexity: $O(n)$ space for result array.

COMPLEXITY CLASSES

Chapter-20

20.1 Introduction

In all previous chapters we have solved problems of different complexities. Some algorithms are having lower rate of growths and others are having higher rate of growth. The problems which are having lower rate of growth are called *easy* problems (or *easy solved problems*) and the problems which are having higher rate of growth are called *hard* problems (or *hard solved problems*). This classification is done based on the running time (or memory) that an algorithm takes for solving the problem.

Time Complexity	Name	Example	Problems
$O(1)$	Constant	Adding an element to the front of a linked list	Easy solved problems
$O(logn)$	Logarithmic	Finding an element in a binary search tree	
$O(n)$	Linear	Finding an element in an unsorted array	
$O(nlogn)$	Linear Logarithmic	Merge sort	
$O(n^2)$	Quadratic	Shortest path between two nodes in a graph	
$O(n^3)$	Cubic	Matrix Multiplication	
$O(2^n)$	Exponential	The Towers of Hanoi problem	Hard solved problems
$O(n!)$	Factorial	Permutations of a string	

Apart from the above there are lots of problems for which we do not know how to solve them. All the problems we have seen so far are the one which can be solved by computer in deterministic time. Before starting our discussion let us see the basic terminology we use in this chapter.

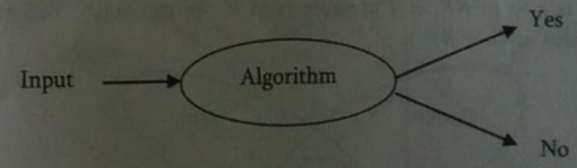
20.2 Polynomial/Exponential time

Exponential time means, in essence, trying every possibility (for example, backtracking algorithms) and they are very slow in nature. Polynomial time means having some clever algorithm to solve a problem, and we don't try every possibility. Mathematically, we can represent these as:

- Polynomial time is $O(n^k)$, for some k.
- Exponential time is $O(k^n)$, for some k.

20.3 What is Decision Problem?

A decision problem is a question with a *yes/no* answer and the answer depends on the values of input. For example, the problem "Given an array of n numbers check whether there are any duplicates or not?" is a decision problem. The answer for this problem can be either *yes* or *no* depending on values of input array.



20.4 Decision Procedure

For a given decision problem let us assume we have given some algorithm for solving it. The process of solving a given decision problem in the form of an algorithm is called a *decision procedure* for that problem.

20.5 What is a Complexity Class?

In computer science, in order to understand the problems for which solutions are not there, the problems are divided into classes and we call them as complexity classes. In complexity theory, a *complexity class* is a set of problems with related complexity. It is the branch of theory of computation that studies the resources required during computation to solve a given problem. The most common resources are time (how much time the algorithm takes to solve a problem) and space (how much memory it takes).

20.6 Types of Complexity Classes

P Class

The complexity class P is the set of decision problems that can be solved by a deterministic machine in polynomial time (P stands for polynomial time). P class problems are the set of problems whose solutions are easy to find.

NP Class

The complexity class NP (NP stands for non-deterministic polynomial time) is the set of decision problems that can be solved by a non-deterministic machine in polynomial time. NP class problems refers to a set of problems whose solutions are hard to find, but easy to verify.

For better understanding let us consider a college having 500 students. Also, assume that there are 100 rooms available for students. Selection of 100 students must be paired together in rooms, but the dean of students has a list of pairings of certain students who cannot room together (may be those students are punished ragging). The total possible number of pairings is too large. But the solutions (the list of pairings) provided to the dean, is easy to check for errors. If one of the prohibited pairs is on the list, that's an error. In this problem, we can see that checking every possibility is very difficult but the result is easy to validate.

That means, if someone gives us a solution to the problem, we can tell them whether it is right or not in polynomial time. Based on the above discussion, for NP class problems if the answer is *yes*, then there is a proof of this fact, which can be verified in polynomial time.

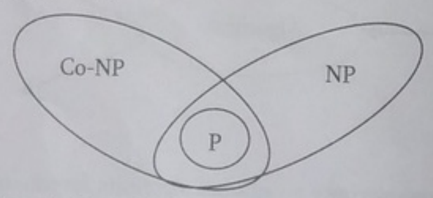
Co-NP Class

$Co-NP$ is the opposite of NP (complement of NP). If the answer to a problem in $Co-NP$ is *no*, then there is a proof of this fact that can be checked in polynomial time.

P	Solvable in polynomial time
NP	*Yes* answers can be checked in polynomial time
$Co-NP$	*No* answers can be checked in polynomial time

Relationship between P, NP and Co-NP

Every decision problem in P is also in NP. If a problem is in P, we can verify YES answers in polynomial time. Similarly, any problem in P is also in $Co-NP$.



One of the important open questions in theoretical computer science is whether or not $P = NP$. Nobody knows. Intuitively, it should be obvious that $P \neq NP$, but nobody knows how to prove it.

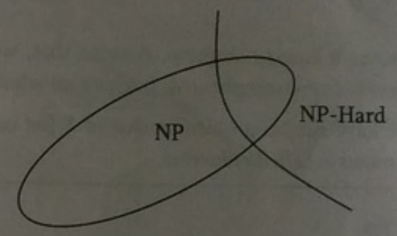
Another open question is whether NP and $Co-NP$ are different. Even if we can verify every YES answer quickly, there's no reason to think that we can also verify NO answers quickly. It is generally believed that $NP \neq Co-NP$, but again nobody knows how to prove it.

NP-hard Class

It is a class of problems such that every problem in NP reduces to it. All NP-hard problems are not in NP, so it takes a long time to even check them. That means, if someone gives us a solution for NP-hard problem, it takes a long time for us to tell check whether it is right or not.

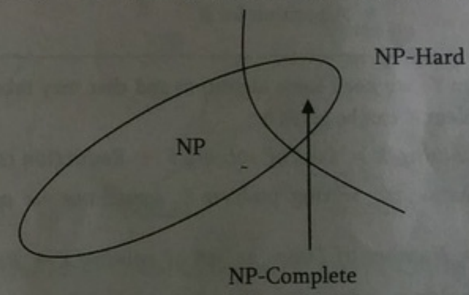
A problem K is NP-hard indicates that if a polynomial-time algorithm (solution) exists for K then a polynomial-time algorithm for every problem in NP. That means:

K is NP-hard implies that if K can be solved in polynomial time, then $P = NP$



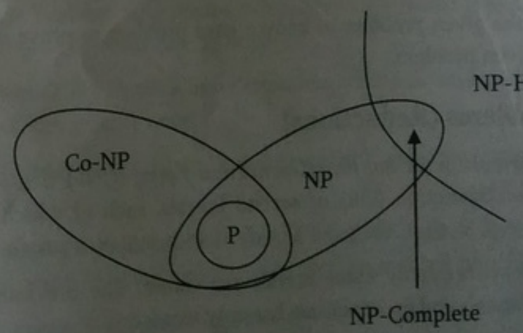
NP-complete Class

Finally, a problem is NP-complete if it is part of both NP-hard and NP. NP-complete problems are the hardest problems in NP. If anyone finds a polynomial-time algorithm for one NP-complete problem, then we can find polynomial-time algorithm for every NP-complete problem. This means that we can check an answer fast and every problem in NP reduces to it.



Relationship between P, NP Co-NP, NP-Hard and NP-Complete

From the above discussion, we can write the relationships between different components as shown below (remember, this is just an assumption).



The set of problems that are NP-hard is a strict superset of the problems that are NP-complete. Some problems (like the halting problem) are NP-hard, but not in NP. NP-hard problems might be impossible to solve in general. We can