

# Dynamic Programming - 2

---

## Longest Common Subsequence

**Problem Statement:** Given two strings  $S1$  and  $S2$  with lengths  $M$  and  $N$  respectively, find the length of the longest common subsequence.

A subsequence of a string  $S$  whose length is  $K$ , is a string containing characters in the same relative order as they are present in  $S$ , but not necessarily contiguous. Subsequences contain all the strings of length varying from 0 to  $K$ . For example, subsequences of string "abc" are -- ""(empty string), a, b, c, ab, bc, ac, abc.

### Explanation:

**Example1:** We are given with two strings  $s1 = \text{"adebc"}$  and  $s2 = \text{"dcadb"}$ .

"a", "d", "b", "c", "ad", "ab", "db", "dc" and "adb" are present as a subsequence in both the strings in which "adb" has the maximum length. There are no other common subsequences of length greater than 3 and hence the answer.

**Example2:** We are given with two strings  $s1 = \text{"abcdef"}$  and  $s2 = \text{"a1b2d3c4def"}$  then the longest common subsequence among these two is **abcdef**.

**Naive Approach:** If we look carefully at the first character of the strings then we can have two cases:

- **The first characters of the two strings are equal.**
  - This means we can call our LCS function on the remaining string and whatever answer we get, we can add 1 to it for the final answer, i.e., **1 + LCS ( $s1 + 1$ ,  $s2 + 1$ ).**

- **The first characters of the two strings are not equal.**
  - Consider an example in which s1 starts with **a** and s2 starts with **b**.

This creates three more possibilities:

- LCS starts with a.
- LCS starts with b.
- LCS starts with neither a nor b.

To take into consideration, the above three cases we need to find the answer from each of them and take the maximum for the final answer, i.e.,

**$\max(\text{LCS}(s1 + 1, s2), \text{LCS}(s1, s2 + 1))$ .**

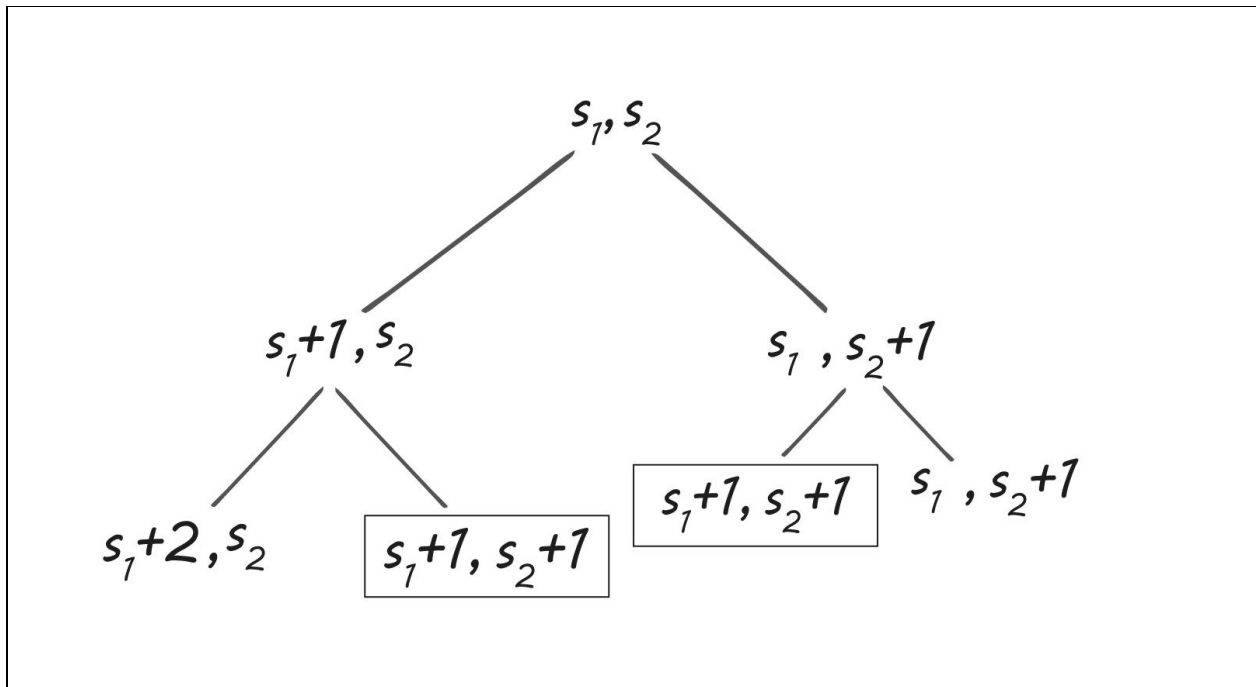
**Let's look at the code:**

```
#include<iostream>
using namespace std;

int lcs(char *s1, char *s2){
    if(s1[0] == '\0' || s2[0] == '\0'){
        return 0;
    }
    if(s1[0] == s2[0]){
        return 1 + lcs(s1+1, s2+1);
    } else {
        int option1 = lcs(s1, s2+1);
        int option2 = lcs(s1+1, s2);
        return max(option1, option2);
    }
}

int main(){
    char a[100];
    char b[100];
    cin >> a;
    cin >> b;
    cout << lcs(a,b) <<endl;
    return 0;
}
```

The above solution is very slow because there are multiple repetitive recursion calls happening.



To improve the above solution, we can use memoization, i.e., we can store the data so that we can use that data instead of making a recursive call that has already been executed. For that, we create a helper function that takes a 2D array which stores -1 initially to tell that work has not yet been done on that position. After the recursive call at that position has been made, we store the answer in the dp array.

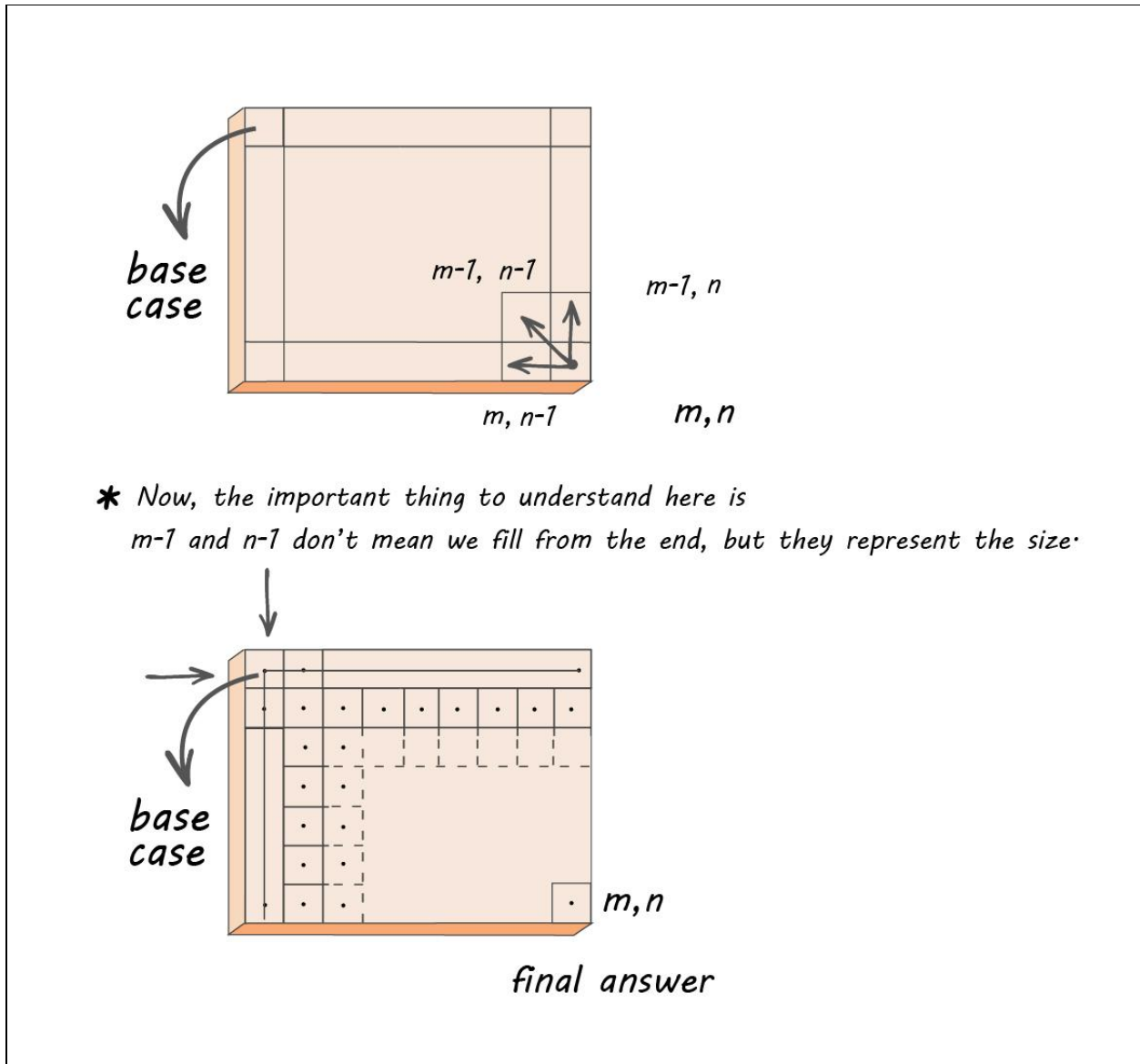
#### Code:

```
int lcs2Helper(char *s1, char *s2, int m, int n, int **dp){
    if(m == 0 || n==0){
        return 0;
    }
    // if the work has already been done then return the answer stored
    if(dp[m][n] > -1){
        return dp[m][n];
    }
}
```

```
}
int ans;
if(s1[0] == s2[0]){
    ans = 1 + lcs2Helper(s1+1, s2+1, m-1, n-1, dp);
} else {
    int option1 = lcs2Helper(s1, s2+1, m, n-1, dp);
    int option2 = lcs2Helper(s1+1, s2, m-1, n, dp);
    ans = max(option1, option2);
}
dp[m][n] = ans;
return ans;
}

int lcs2(char *s1, char *s2){
    int m = strlen(s1);
    int n = strlen(s2);
    int **dp = new int *[m+1];
    for(int i=0; i<=m; i++){
        dp[i] = new int[n+1];
        for(int j=0; j<=n; j++){
            dp[i][j] = -1; // initializing with -1
        }
    }
    int ans = lcs2Helper(s1, s2, m, n, dp);
    // deallocating the memory
    for(int i=0; i<=m; i++){
        delete []dp[i];
    }
    delete []dp;
    return ans;
}
```

## Iterative Approach:



## Code:

```
int lcsI(char* s1, char* s2) {
    int m = strlen(s1);
    int n = strlen(s2);
    int** dp = new int*[m + 1];
```

```
for (int i = 0; i <= m; i++) {
    dp[i] = new int[n + 1];
}
for (int i = 0; i <= m; i++) {
    dp[i][0] = 0;
}
for (int i = 0; i <= n; i++) {
    dp[0][i] = 0;
}

for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
        if (s1[m - i] == s2[n - j]) {
            dp[i][j] = 1 + dp[i - 1][j - 1];
        } else {
            dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
        }
    }
}

int ans = dp[m][n];
// delete 2d array
for (int i = 0; i <= m; i++) {
    delete [] dp[i];
}
delete [] dp;
return ans;
}
```

## Knapsack Problem

**Problem Statement:** Given the weights and values of 'N' items, we are asked to put these items in a knapsack, which has a capacity 'C'. The goal is to get the maximum value from the items in the knapsack. Each item can only be selected once, as we don't have multiple quantities of any item.

### Explanation:

#### For example:

**Items:** {Apple, Orange, Banana, Melon}  
**Weights:** {2, 3, 1, 4}  
**Values:** {4, 5, 3, 7}  
**Knapsack capacity:** 5

Possible combinations that satisfy the given conditions are:

Apple + Orange (total weight 5) => 9 value  
Apple + Banana (total weight 3) => 7 value  
Orange + Banana (total weight 4) => 8 value  
Banana + Melon (total weight 5) => 10 value

This shows that **Banana + Melon** is the best combination, as it gives us the maximum value, and the total weight does not exceed the capacity.

**Naive Approach:** First-of-all, let's discuss the brute-force-approach, i.e., the **recursive approach**. There are two possible cases for every item, either to put that item into the knapsack or not. If we consider that item, then its value will be contributed towards the total value, otherwise not. To figure out the maximum value obtained by maintaining the capacity of the knapsack, we will call recursion over these two cases simultaneously, and then will consider the maximum value obtained out of the two.

If we consider a particular weight 'w' from the array of weights with value 'v' and the total capacity was 'C' with the initial value 'Val', then the remaining capacity of the knapsack becomes 'C-w', and the value becomes 'Val + v'.

Let's look at the recursive code for the same:

```
int knapsack(int *weight, int *values, int n, int maxWeight) {
    // Base case : if the size of array is 0 or we are not able to add any
    // more weight to the knapsack
    if(n == 0 || maxWeight == 0) {
        return 0;
    }
    //If the particular weight's value extends limit of knapsack remaining
    // capacity, then we have to simply skip it
    if(weight[0] > maxWeight) {
        return knapsack(weight + 1, values + 1, n - 1, maxWeight);
    }

    // Recursive calls
    //1. Considering the weight
    int x = knapsack(weight + 1, values + 1, n - 1, maxWeight - weight[0])
    + values[0];
    // 2. Skipping the weight and moving forward
    int y = knapsack(weight + 1, values + 1, n - 1, maxWeight);

    // finally returning the maximum answer among the two
    return max(x, y);
}
```

Now, the memoization and DP approach is left for you to solve. For the code, refer to the solution tab of the same. Also, figure out the time complexity for the same by running the code over some examples and by dry running it.



## Subset Sum Problem

**Problem Statement:** Given an array of  $n$  integers, find if a subset of sum  $k$  can be formed from the given set. Print Yes or No.

### Explanation:

**Example:** given array =  $\{1, 2, 3\}$  and  $k = 4$ , so subset  $\{1, 3\}$  has a sum of elements equal to  $k$ . Therefore, our output will be true.

**Brute Force Approach:** The naive approach that can be thought of for this problem is to generate all the subsets and check whether they satisfy the constraints given or not. This is very similar to what we did in the knapsack problem.

**Iterative Approach:** Taking an example, **val =  $[1, 3, 5, 7, 9]$  and sum = 8.**

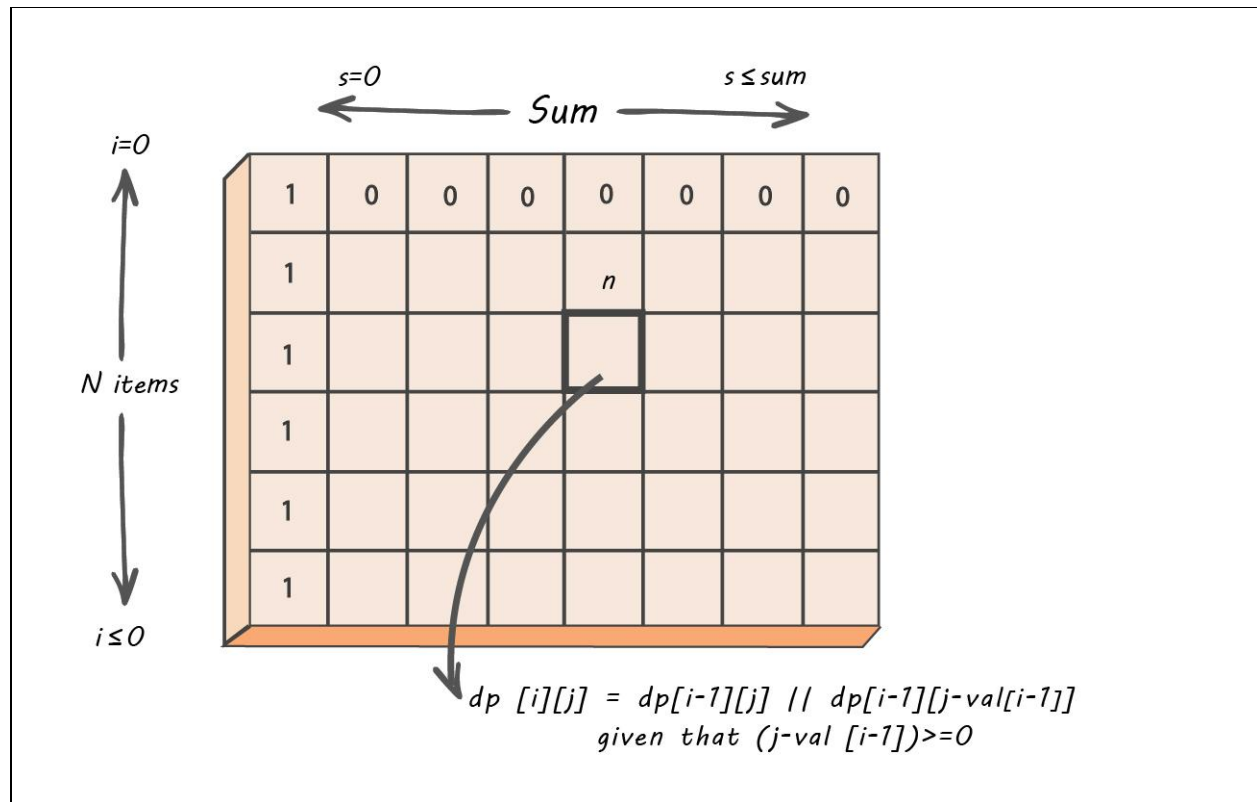
If we try to reduce this problem to a simpler one then there can be two cases:

- 9 is included in the subset sum :
  - For this case, the problem gets reduced to whether in  **$[1, 3, 5, 7]$**  there exists a subset whose sum is  **$8 - 9 = -1$** .
- 9 is not included in the subset sum:
  - For this case the problem gets reduced to whether in  **$[1, 3, 5, 7]$**  there exists a subset whose sum is **8**.

Therefore, we are going to create a dp array that stores the answer of  **$dp[i][j] = dp[i-1][j] \mid dp[i-1][j - val[i-1]]$** , given that  **$(j - val[i-1]) \geq 0$** .

The above expression means just what the two cases represent.

The first row of the dp array is initialized with 0 and the first column excluding the first element is initialized with 1, because the row is an indication of the items in the subset and the columns represent the sum values.



## Code:

```
#include<bits/stdc++.h>
using namespace std;

bool subsetSum(int *val, int n, int sum){

    bool **dp = new bool*[n+1];
    for(int i=0; i<=n; i++){
        dp[i] = new bool[sum+1];
    }
    // first row is initialized with false
    for(int i=0; i<=sum; i++){
        dp[0][i] = false;
    }
}
```

```
}
// first column is initialized with true
for(int i=0; i<=n; i++){
    dp[i][0] = true;
}
for(int i=1; i<=n; i++){
    for(int j=1; j<=sum; j++){

        dp[i][j] = dp[i-1][j];
        if(j>=val[i-1]){
            dp[i][j] = dp[i][j] || dp[i-1][j-val[i-1]];
        }
    }
}
bool ans = dp[n][sum];
// delete dp array
return ans;
}

int main(){
    int val[] = {1, 3, 5, 7, 9};
    int sum = 12;

    cout<<subsetSum(val, 5, sum);
    return 0;
}
```

**Optimized Approach:** In the above solution, for  $i^{\text{th}}$  row, we are using the previous row to compute or answer and the rows before that have no significance whatsoever. So, we could have made the dp array with just two rows and could have toggled around these two rows to calculate the solution.

**Code:**

```
bool subsetSum(int* val, int n, int sum){
```

```
bool** dp = new bool*[2];
for(int i=0;i<=1;i++){
    dp[i] = new bool[sum+1];
}

for(int i=0;i<=sum;i++){
    dp[0][i] = false;
}
dp[0][0] = true;

int flag = 1;

for(int i=1;i<=n;i++){
    for(int j=1;j<=sum;j++){

        dp[flag][j] = dp[flag^1][j];

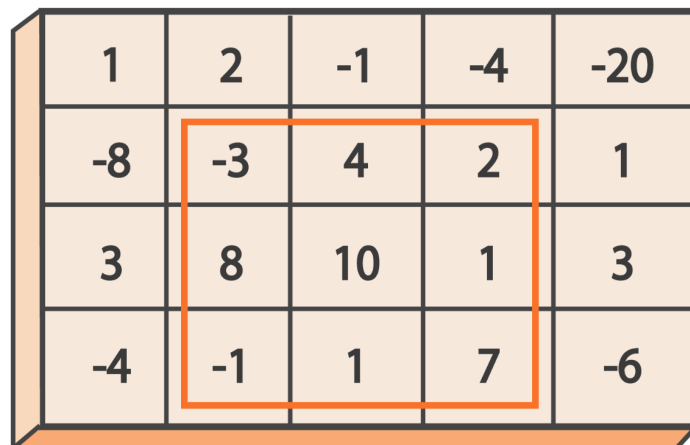
        if(j>=val[i-1]){
            dp[flag][j] = dp[flag][j] || dp[flag^1][j-val[i-1]];
        }
        flag = flag ^ 1;
    }
    bool ans = dp[flag^1][sum];
    // delete dp array
    return ans;
}
```

## Maximum Sum Rectangle

**Problem Statement:** Given a 2D array, find the maximum sum rectangle in it. In other words, find the maximum sum over all rectangles in the matrix.

### Explanation:

**Example:** in the following 2D array, the maximum sum subarray is highlighted with a blue rectangle and the sum of this subarray is 29.



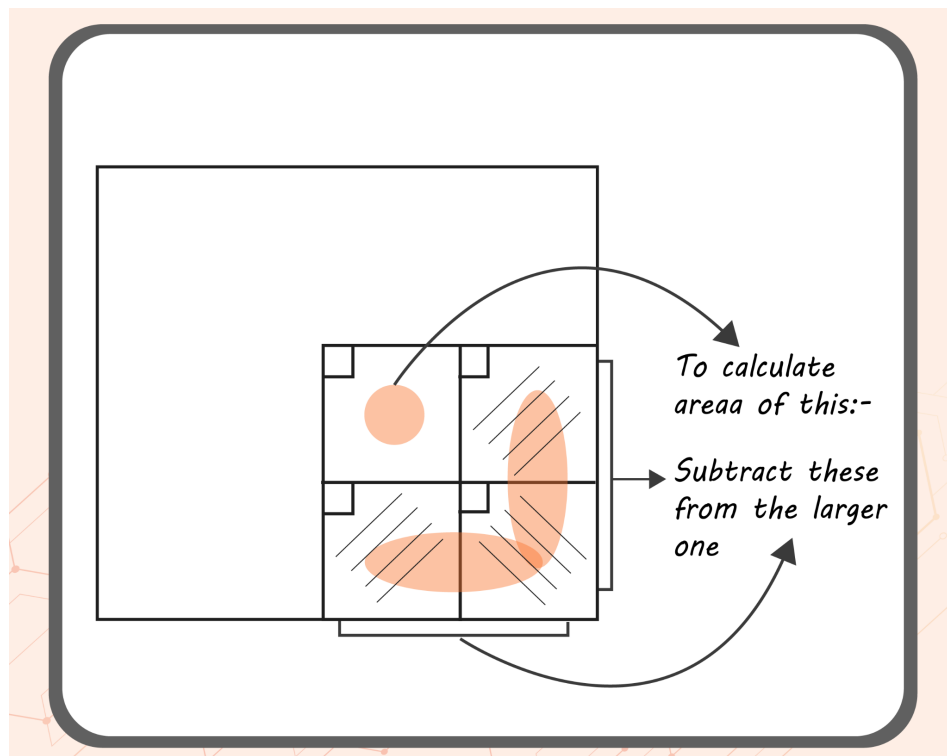
1	2	-1	-4	-20
-8	-3	4	2	1
3	8	10	1	3
-4	-1	1	7	-6

### Naive Approach:

We can take a particular cell  $[i][j]$  as the upper left corner of the rectangle, the bottom right corner will be the last cell given to us and we can store the area of the rectangle thus formed in the cell  $[i][j]$ . We can do this for every cell present in the outer rectangle.

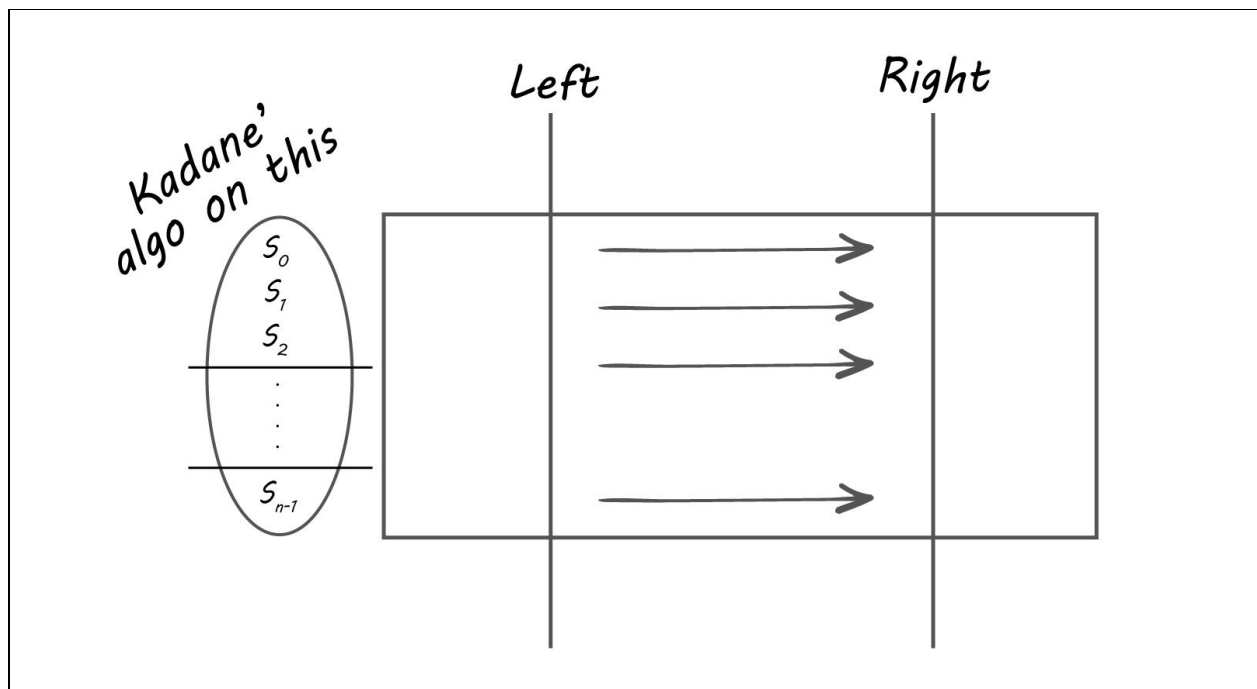
To calculate the area for a rectangle, that has no corner on any of the edges of the outer rectangle refer to the figure given below:

This approach will take  $O(N^4)$  time and hence is a very bad solution.



### Optimized Approach:

We can take two points left and right with  $O(N^2)$  possibilities, and in between those two points, we find the sum of rows. Now, suppose we have the sum of rows  $S_0, S_1, S_2$  and so on, then we can apply Kadane's algorithm  $O(N)$  on these to find the top and bottom points that will help us find the coordinates of the rectangle that contribute to the maximum sum.



Now, suppose our left point increments by 1 then we need not find the entire sum array for the rows again, instead what we can do is add the numbers of the respected added column to the sum array elements. Time taken for this sub problem will also be  $O(N)$ .

**Therefore, the overall time complexity of the problem will be  $O(N^3)$  since for every pair of left and right, we are doing  $O(2N)$  work.**