# Assignment-3

Rachit Parikh (CRS-2101)

Design and Analysis of Algorithms

**Disclaimer :** I declare that all the work presented in this assignment is my own work and I have only consulted the internet when it was absolutely necessary.

---

## Q-1

### A

Let us denote in set notation the number of permissible patterns for a single column. Any pattern valid or not would be a subset of $\{1, 2, 3, 4\}$, where a set $\{1, 3, 4\}$ means that the pebbles are placed on rows 1, 3 and 4. Keeping in mind this representation we can have following valid patterns for an isolated column. They are $\phi, \{1\}, \{2\}, \{3\}, \{4\}, \{1, 3\}, \{1, 4\}, \{2, 4\}$. Hence there are a total of 8 possible patterns which are valid for a single column. Other columns will have pebbles in adjacent.

I have listed out the compatible patterns below:

1. $\{1\}, \{3\}$
2. $\{2\}, \{4\}$
3. $\{1\}, \{4\}$
4. $\{1, 3\}, \{2, 4\}$
5. $\{1, 3\}, \{2\}$
6. $\{1, 3\}, \{4\}$
7. $\{1, 4\}, \{2\}$
8. $\{1, 4\}, \{3\}$
9. $\{2, 4\}, \{1\}$
10. $\{2, 4\}, \{3\}$

Now, $\phi$ is compatible with every other set. Let us define our subproblem $S(k, t)$ as the maximum sum obtained till column $k$ if the $k$th column type was $t$. $t$ can be any one of the 8 possible types as discussed above. We know that $(k-1)$th column would be of the type compatible to $t$. Let $c(i, j)$ be a tuple which indicates whether types $i$ and $j$ are compatible or not. If they are compatible, $c(i, j)$ would be true. So each subproblem $S(k, t)$ gives us the maximum sum obtained till column $k$ where the type of column is $t$.

## B

We have define the subproblem in first part. Now what we do is to find a recursion which will give the final output the total maximum sum that can obtained from all columns. Since the final result can be sum of any possible compatible patterns, we will try to find the maximum sum for each pattern at every step. So for a particular type $t$ and column $k$ we can define recursion as follows:

$$S(k,t) = \max_{c(t,q)=true} S(k-1,q) + \sum_{r \in t} A(r,k)$$

The above summation is over the checkerboard which is represented by matrix $A$. An entry $A(i,j)$ represents the number written on $i$th row and $j$th column in checkerboard $A$. For a type $t$ we are finding the maximum that can be obtained from its compatible type in $k-1$th column.
The pseudo-code is given below:

```
# This function just calculates the sum that can be obtained on adding rows in set type
# For a given column
# Let's say the type is {1, 3} then we will just add first and third element in column "col"
def value(col, pattern):
    return sum of type in col

# We are just initializing the first column sum for each type
def initialize():
    for type = 1 to 8:
        S[1][type] = value(1, type)

# Here we are using the subproblem define to find the maximum sum for all columns
def find_sum():

    # For each column
    for col = 2 to n:

        # For each pattern
        for t = 1 to 8:

        # Here we are iterating through all possible compatible types of 't'
            for comp_types of t:
                S[col][t] = max(S[col][t], S[col-1][com_types] + value(col, t));

    # answer can be 0 too if we are not placing any pebble
    ans = 0

    # S[n][t] represents maximum sum till last column if the last column was type t
    # We calculate max from all possible t and we will find the answer
    for t = 1 to 8:
        ans = max(ans, S[n][t])

    return ans
```

The explanation for the code is given in the comments itself. The correctness can easily be argued by seeing that we are just considering all possiblities and using dynamic programming we are storing the values of each possibility. Then we are taking maximum over all such possibilites which itself gives us the required sum. This is just smart brute force because we are storing values.

**Time Complexity :** There are two inner loops which run in almost constant time because the patterns are limited so they wont contribute to the total time complexity. The outer loop, loops through all possible columns $n$ and hence the total time complexity of the algorithm is $\mathcal{O}(n)$.

# Q-2

Let $A = \{g_1, g_2, \ldots, g_n\}$, we need to find a subset of $A$ such that value is maximized in touring that subset. We first define a subproblem $S(G, i)$ as, $S(G, i)$ = maximum value obtained on ending the tour at garage $g_i$ and only garages in set $G \subseteq A$ are traversed. Thus, $S(G, i)$ can be written as,

$$S(G, i) = \max_{g_u \in G}[S(G - \{i\}, u) - d_{ui} + v_i]$$

In the above recursion we are finding the maximum value for a single subset of $A$. For a particular subset $G \subseteq A$, we break that problem further by ending our tour at garage $g_i$ and find the maximum value possible to end the tour at $g_i$. So for a particular subset the final answer would be:

$$V(G) = \max_{g_i \in G}[S(G, i) - d_{i0}]$$

Here we substract $d_{i0}$ because we will be going back to home after completing the sale at garage $g_i$. We have found the maximum cost for the last garage, but what about the first garage? We will define the base cases as follows:

$$S(\{g_i\}, i) = v_i - d_{0i}$$

Now that we have defined the base case as well as the recursion, we will use dynamic programming to store the values so that we do not need to compute repeating values. This can be done by storing all the subsets in binary form. Any subset can be written as a binary string of length $n$ where if a garage $k$ is present in the subset, the $k$th position of that string is 1, otherwise it is 0. Thus, any subset can be represented by a number in the range 0 to $2^n - 1$. Also, for each subset we need to find for each element in that subset the maximum value possible if the tour ends at that element.

We can represent $S(G, i)$ by a matrix $M$ where entry $M(i, j)$ represents subproblem $S(G_i, j)$ where $G_i$ is the subset corresponding to the binary representation(of $i$) discussed earlier. This will help in making our algorithm efficient as we are storing the values in matrix and avoiding repetitive computations. Let us find time complexity for a single subset first. Assume $G$ is a subset of $S$ with $k$ elements. In the first iteration, for a single element of $G$ we will have to iterate through all the other $k - 1$ elements. But $i$ itself can be chosen in $k$ ways. This means that a subset with $k$ elements would require $\mathcal{O}(k^2)$ iterations.
Total number of subsets with $k$ elements = $\binom{n}{k}$
Hence total number of iterations for all subsets with $k$ elements = $\binom{n}{k}k^2$
So total time to fill the matrix $M$ would be :

$$T(n) = \binom{n}{1}1^2 + \binom{n}{2}2^2 + \cdots + \binom{n}{n}n^2$$

We solve the right side of equation as follows:

$$(1+x)^n = \binom{n}{0} + \binom{n}{1}x + \cdots + \binom{n}{n}x^n$$
$$\frac{d(1+x)^n}{dx} = \binom{n}{1} + \binom{n}{2}2x \cdots + \binom{n}{n}nx^{n-1}$$
$$nx(1+x)^{n-1} = \binom{n}{1}x + \binom{n}{2}2x^2 \cdots + \binom{n}{n}nx^n$$
$$\frac{d(nx(1+x)^{n-1})}{dx} = \binom{n}{1}1^2 + \binom{n}{2}2^2x \cdots + \binom{n}{n}n^2x^{n-1}$$
$$n(n-1)x(1+x)^{n-2} + n(1+x)^{n-1} = \binom{n}{1}1^2 + \binom{n}{2}2^2x \cdots + \binom{n}{n}n^2x^{n-1}$$
$$n(n-1)2^{n-2} + n2^{n-1} = \binom{n}{1}1^2 + \binom{n}{2}2^2 \cdots + \binom{n}{n}n^2 = T(n)$$

$$\implies T(n) = \mathcal{O}(n^2 2^n)$$

Filling out the matrix would take $\mathcal{O}(n^2 2^n)$ time. After filling the matrix we also need to search which combination would yield us the maximum value. This can be simply done by going through all the possiblities inside the matrix. This will take $\mathcal{O}(n2^n)$ time since we will go through all possible entries of the matrix $M$. Hence the total time complexity of the algorithm would be $\mathcal{O}(n^2 2^n) + \mathcal{O}(n2^n) = \mathcal{O}(n^2 2^n)$.

## Q-3

First we try to formulate the subproblem here. Let $T_i$ be a tree rooted at $v_i$. We define subproblem as follows:
Let, $S(u, 0)$ = weight of largest weight independent set $W_u$ in tree $T_u$ and $u \notin W_u$
$S(u, 1)$ = weight of largest weight independent set $W_u$ in tree $T_u$ and $u \in W_u$
Also, let us define the set of children of $u$ as $\mathcal{C}(u)$. Now, the recursion can be defined as follows:

$$S(u,0) = \sum_{v \in \mathcal{C}(u)} max\{S(v,0), S(v,1)\}$$
$$S(u,1) = w(u) + \sum_{v \in \mathcal{C}(u)} S(v,0)$$

In the above recursion what we are essentially doing is calculating the maximum weight that can be obtained by including or excluding a particular node. If for a node $u$, we are including it in our independent set then we cannot include its children in our independent set. So we will take the summation of largest weight independent sets of trees rooted at children nodes of $u$ without including them. Similarly, if we do not include $u$ in the independent set then we can either some, all or none of its children. So for each child of $u$, we take maximum of weight obtained by either including or not including the child.

Now, for the base case of recursion we consider for leaves. Assume for a leaf node $l$, we will take $S(l, 0) = 0$ because a tree rooted at leaf node is just a single node, so if we are not including it in independent set there is nothing in the set. If we are including it in the independent set then, $S(l, 1) = w(l)$ because it is the only node in the tree.

Our final answer for the given tree would be

$$ans = \max_{u \in V}[max\{S(u,0), S(u,1)\}]$$

We can solve this more efficiently by dynamic programming and storing $S(i,0)$ or $S(i,1)$ as array. Let $\mathcal{M}$ be a matrix of dimension $n \times 2$ where $\mathcal{M}[i][0]$ represents the subproblem $S(i,0)$ and $\mathcal{M}[i][1]$ represents the subproblem $S(i,1)$ and after filling up this matrix we just have to find the maximum element in this matrix $\mathcal{M}$.

**Time complexity :** For a single entry $\mathcal{M}(i,j)$, in worst case we need to go through $O(n)$ elements of the matrix. Since $j$ can only be 0 or 1 our total worse case time complexity would be $O(n)$.

## Q-4

A valid subarray is one which any two elements do not differ by $p$ percent. Before formulating an algorithm we state an important observation. For any range $(i,j)$ of a valid subarray we only need to check the difference between the *max* and *min* elements. So our strategy would revolve around maintaining maximum and minimum values of a subarray.

Another important observation here is that we can skip looking through some subarrays. For example, if I know $(i,j)$ is a valid subarray then $(i+1,j)$ is also a valid subarray. We use this observation and maintain two pointers *start* and *end* which will initially be at the starting element of the array. The *end* pointer will then traverse each element in the array and check whether a subarray is valid or not. If it is valid, we will keep increasing the *end* pointer and update the count of maximum length subarray found till now. If the subarray is invalid, we will move our *start* pointer which will check for the valid subarray property and we keep increasing the *start* pointer till we reach the *end* pointer. Once they are equal we know that valid subarray property will hold because an element will not differ from itself by more than $p$ percent. Then we repeat the strategy of moving the *end* pointer. All this is good, but how do we maintain the *max* and *min* for each subarray? We will maintain two deques which will have the property that one of them is in ascending order and other is in descending order. This will make sure that the top element of min deque is minimum and that of max deque is maximum.

Now that the basic idea is clear, I will present the pseudo code for the algorithm:

```
def initialize():
    min_deq = deque()
    max_deq = deque()
    start = 0
    end = 0
```

Here, we have just initialized the variables that we are going to use in our algorithm. The *min_deq* is a deque which will store the elements in increasing order and the *max_deq* is a deque which will store the elements in decreasing order. The top element of *min_deq* will be index of the minimum element for the subarray $(start, end)$, similarly the top element of *max_deq* will be index of the maximum element.

```
# This function just checks whether max and min elements are in p percentage range
def check(a, b):
    return a <= (1 + (p/100))*b


def find_max_subarray(A, start, end, min_deq, max_deq):
    # Loop 1
    while end < n:

        curr_end = A[end]

        # We are adding the index of curr_end (i.e. end) at the back of min_deq
        # But this might spoil the order of the min_deq
        # To maintain ascending order we will remove elements greater than curr_end from the back
        while (not min_deq.empty()) && (not A[min_deq.back] < curr_end)
            remove last element from min_deq

        # Now that we have removed all elements greater than A[end] from min_deq
        # We can now ensure that the order min_deq is maintained
        # So we push index of the current_end in min_deq
        push end in min_deq

        # We are adding the index of curr_end (i.e. end) at the back of max_deq
        # But this might spoil the order of the max_deq
        # To maintain descending order we will remove elements smaller than curr_end from the back
        while (not min_deq.empty()) && (not A[max_deq.back] > curr_end)
            remove last element from max_deq

        # Now that we have removed all elements smaller than A[end] from max_deq
        # We can now ensure that the order max_deq is maintained
        # So we push index of the current_end in max_deq
        push end in max_deq

        # These are the current minimum and maximum of the subarray [start....end]
        curr_min = A[min_deq.front]
        curr_max = A[max_deq.front]

        # Now we will check whether the property holds for our subarray
        # If the length was greater than that of any valid subarray we obtained before
        # Then we update the maximum_length variable
        # We will also update the end variable until we reach the last element of the array
        if check(curr_min, curr_max) == True:
            maximum_length = max(maximum_length, end - start + 1)
            end = end + 1

        # Since the property did not verify this is not a valid subarray
        # We will increment the start variable
        # So we have to ensure that the elements before the start index are removed as they are no longer
            part of subarray
        # We remove the front elements from both queues if they are not a part of subarray
        else:
            start = start + 1
            if not min_deq.empty() && min_deq.front < start:
                remove min_deq.front from min_deq
            if not max_deq.empty() && max_deq.front < start:
                remove max_deq.front from max_deq
```

I have explained the code above in the comments only. The proof of correctness for the above algorithm can be given by first considering the *min_deq* and *max_deq* dequeues. Their property is that they are always in increasing and decreasing orders (not literally, indices are stored corresponding to which the elements are in increasing and decreasing order).
Also, we are ensuring the valid subarray property holds by using check function every time. But how can we ensure that we have covered all the subsets? As I have explained earlier, we don't need to check for $(i+1, j)$ if $(i, j)$ is valid. That's why whenever the property is satisfied we increasing $j$ and not $i$. This will reduce the number of subarrays by great extent. For every valid $(i, j)$ subarray we have reduced $j-i+1$ subarrays. Similary $(i, j-1)$ is also a valid subarray and even in this we have reduced $j-i+1$ subarrays here. Hence, for every valid $(i, j)$ subarray we have in total reduced $2(j-i+1)$ subarrays.

**Time Complexity :** For both the deques any element in $A$ will only get added and deleted once. Also at each iteration either start or end is getting incremented and they can only go upto $n$. Thus the time complexity of this algorithm would be $\mathcal{O}(n)$.

# Q-5

**Assumption :** We are going to assume that we can repeat the vertices. Also we are going to assume that both $u$ and $v$ are part of the $k$ vertices.
Let the weighted directed graph be $G(V, E)$ and the weights corresponding to source $x$ and destination $y$ be represented as $wt(i, j)$. We need to find the shortest paths between vertices $u$ and $v$ with exactly $k$ nodes. We will be using Bellman Ford algorithm with new variable as edges. So we will just find $k + 1$ edge path which will have $k$ vertices.

We define subproblem as $S(i, j, l)$ which represents the shortest path from $i$ to $j$ with $l$ edges. This problem can be broken as:

$$S(i, j, l) = \min_{w \in \mathcal{N}(i)} \{S(i, j, l), S(w, j, l - 1) + wt(i, w)\}$$

Here $\mathcal{N}(i)$ represents neighbours of $i$. What we are doing is essentially checking the neighbours of the vertex $i$ and checking whether there is a path with $l - 1$ edges possible from it to $j$. This formulation is similar to Bellman Ford Algorithm, except here length is also taken into consideration. The code for the algorithm can be given below:

```
def initialize():
    for e = 0 to k + 1:
        for i = 0 to n:
            sp[i][i][e] = 0

def find_length(u, v):
    for edges = 1 to k+1:
        for i = 0 to n:
            for j = 0 to n:
                # Initialize remaining edges with infinite weight
                dp[i][j][e] = INFINITE

                # If directly connected, replace it by their actual weight
                if edges == 1 && wt[i][j] != INFINITE:
                    dp[i][j][e] = wt[i][j]

                # Otherwise use the solution for the subproblem
                else if edges > 1:

                    # We are just traversing through the neighbours of i
                    for w = 0 to n:
                        if i != w && j != w && wt[i][j] != INFINTE && dp[i][j][edges - 1] != INFINITE:
                            dp[i][j][e] = min(dp[i][j][e], wt[i][w] + dp[w][j][e-1])

    # We return dp[u][v][k+1] because we get path with k+1 edges meaning k vertices
    return dp[u][v][k+1]
```

Since the relaxation of repeating vertices is allowed our algorithm is correct. If it was not allowed, we would have to take all possible subsets which will keep a track of vertices we have covered and it would take exponential time.
**Time Complexity :** The above algorithm has two loops for source and destinations and one loop where we are traversing through neighbours of source. In worst case, that will also take $\mathcal{O}(n)$ time to run where $n$ is the number of vertices. Hence, the complexity for inner loop will be $\mathcal{O}(n^3)$, also we are going through all the possbile edge paths till $k + 1$ the total time complexity of the algorithm would be $\mathcal{O}(n^3 k)$

# Q-7

Let $a_i$ and $d_i$ be arrival and departure time for train $i$ respectively. We will need a new platform in the station whenever we have a train arriving but some train has not departed. So if there is some time $t$ before which $k$ trains have arrived but not departed till time $t$, and a new train is arriving we need $k+1$ platforms ($k$ for the previous trains and another one for new train arriving). Basically minimum number of stations

needed would be the maximum number of trains having overlapping intervals.

Let $A$ be the array of arrival time of trains and $D$ be the array consisting of departure times. We will try to design an algorithm to find the maximum overlap. Let us define a tuple $S$ such that $S =$ (time, train index, type). Here time can be anything (arrival or departure) and type of time $= 0$, if it is arrival time and 1 if it is departure. An example of one such tuple is $S(4, 2, 0)$, here what this tuple represents is that train 2 arrived at 4. For each element in $A$ and $D$ we first create a tuple and then sort it in non-decreasing order by time.

We have a total of $2n$ tuples. Let us define an array $\mathcal{N}$ which represents for each tuple the number of platforms required. The maximum element of $\mathcal{N}$ would be our answer. The pseudo code is attached below

```
for i = 1 to n
    create tuple S(A[i], i, 0)
    create tuple S(D[i], i, 1)

sort(S)

ans = INT_MIN
count = 0

for i = 1 to 2n
    if S[i].type == 0
        count = count - 1
    else
        count = count + 1
    ans = max(ans, count)
```

The above pseudo code can be explained as follows. First, we will create the tuple out of arrival and departure times for each train. Then we will sort the tuples. Now, what we do in the loop is that we keep a track of total number of trains arrived till now. We update the count variable in each loop. Once a train departs we decrease the count variable by 1 and once it arrives we increase the count variable by 1. The value *count* gives the number of overlaps at a particular time, thus the max number of overlaps will be stored in *ans*.

**Time Complexity :** Since we are combining the arrays and then sorting them it will take $\mathcal{O}(n \log n)$ time. Then we are just traversing the array which takes linear time, hence the total time complexity of this algorithm is $\mathcal{O}(n \log n)$

# Q-8

This problem is equivalent to the Travelling Salesman Problem. We can consider strings to be the nodes. The distance $d(i, j)$ from string $i$ to $j$ denotes extra number of characters to be added after the string $i$. Suppose string $s_i$ ="pajeet" and string $s_j$ ="jeetesh", then we will have to add "esh" behind the string "pajeet" to make a string for which both $s_j$ and $s_i$ are substrings and thus $d(i, j) = 3$. (Note that here order matters and thus $d(i, j)$ and $d(j, i)$ may not be similar). Our problem is to pass through all nodes (strings) such that the total distance is minimized and thus give us the minimum length string.

We will be using the Held-Karp algorithm to solve this problem. But to do this, first we need to calculate the distance matrix $D$. The algorithm for filling the matrix $D$ is given below:

```
check_suffix(x, y, k):
    for i = 0 to k-1:
        if y[i] != x[n-k+1]
            return false
        i = i+1
    return true

find_distance(x, y):
    max_common = 0
    for k = 1 to len(y):
        if(check_suffix(x, y, k) == true)
            max_common = max(max_common, k)
        k = k+1
    return len(y) - max_common
```

In the above code to find $d(x, y)$ we have used the $find\_distance$ function which first finds length of the maximum possible prefix of $y$ that is equal to the suffix of $x$. On substracting this value from length of $y$ what we get is the minimum number of characters that are needed to be added in the string $x$ so that the resulting string will have $x$ and $y$ as substrings. We are using brute force to calculate this thing. We iterate through all possible lengths of the common substring (prefix-suffix) and for each length we check whether it is possible to form a common substring or not. For cases where it is possible, we take a maximum out of all these cases and get the result.

Using this algorithm we can calculate the distance matrix $D$ for each pair of strings. This algorithm will take $\mathcal{O}(n^2)$ time where $n$ is the length of string $y$. For the whole matrix this algorithm will take $\mathcal{O}(n^4)$ time, which looks very expensive but is negligible in comparison to what's about to come next. The algorithm for finding the common substring which is prefix to $y$ and suffix to $x$ can be done more efficiently but I am skipping that algorithm for now.

Now that we have our distance matrix, we will be first represent each string by its index like we do in graphs. Since each of our string is a node and the distances we have already calculated. Now, this turns out to be standard TSP problem which we will solve using DP, but at every step we will have to record the order in which strings are taken. Pseudo code is given below:

```
no_of_subsets = 1 << n

for i = 1 to no_of_subsets:
    for j = 0 to n:
        if check_bit(i, j):
            prev = i - (1 << j)
            if(prev == 0)
                dp[i][j] = len(S[j])
            else
                for k = 0 to n:
                    if dp[prev][k] < INF && dp[prev][k] + d[k][j] < dp[i][j]:
                        dp[i][j] = dp[prev][k] + d[k][j];
                        seq[i][j] = k
        if i == (1<<n)-1 && dp[i][j] < min:
            min = dp[i][j]
            last = j


current_set = (1<<n) - 1
sequence = list()
i = 0
while current_set > 0:
    sequence[i] = last
    tmp = current_set
    current_set = current_set - (1 << last)
    last = dp[tmp][last]

ans = sequence[n-1]
for i = n-2 to 0:
    ans = remove_suffix(ans, sequence[i])
    i--
```

Here in the above TSP implementation the only thing different is the *sequence* array that we have defined which will store the sequence in which we should take the strings to minimize the lenght of superstring. To implement the $remove\_suffix$ function we will use the $find\_distance$ function and keep removing the common prefix in the incoming string on sequences. I will not be explaining the code here because it would make this answer very very long. To write this code, I have taken references from `https://en.wikipedia.org/wiki/Held-Karp_algorithm`. In this code we are keeping a track of sequence of strings in a different two dimensional array *seq*. After filling up the matrices *dp* and *seq*, we are going to iterate through the *seq* array as give in the loop below where we are storing the sequence in a new array *seq* where the first element of the array *sequence* represents the last string of the list to be added in the superstring.

**Time Complexity :** The time complexity of this algorithm would be $\mathcal{O}(n^2 2^n)$ because to fill the *dp* and *seq* table in the algorithm given above we have two loops within the loop where we traverse through all the subsets.

## Q-6

**Assumption :** The array contains non-negative integers.
We will try to first formulate the subproblem here. We only need to define sum of two sets because the last one would be fixed by these two. Let $A$ be the set of integers under consideration. We will define $S(i, j, l)$ as following:

$$S(i,j,l) = \begin{cases} \text{True} & \text{there exists 3 disjoint sets with sums } i, j, s_l - i - j \text{ made out of elements } A[1], \dots A[l] \\ \text{False} & \text{otherwise} \end{cases}$$

Here, $s_l$ gives the sum of first $l$ elements. Now, for any element $x$ of the array $A$ there are three possibilites :

1. $x \in I$
2. $x \in J$

3. $x \in K$

Also, if the sum of all elements of array is not divisible by 3 it is not possible to have disjoint sets with equal sum. Let the sum of all elements of our array be $3\mu$. Then each $I$, $J$ and $K$ should have sum $\mu$. So, the answer to our question lies in finding the truth value of $S(\mu, \mu, n)$. The C++ code is given in the next page.

```
// In this code I am not adding trivial checks like i-A[l] >= 0

bool S(int i, int j, int k, int l){

    // When we have traversed through all cases but not found a single match
    if(l < 0)
    return false;

    // If we get the required sum before traversing all the cases we output true
    // val is the total sum
    if(i==0 && j==0 && val-i-j==0)
    return true;

    // When the lth element enters first set - I
    bool f = S(i-A[l], j, l-1);

    // When the lth element enters second set - J
    bool s = S(i, j-A[l], l-1);

    // When the lth element enters third set - K
    bool t = S(i, j, l-1);

    return f || s || t;
}
```

Since we have taken the assumption that we will not be taking negative integers in this question, maximum values of $i$, $j$ and $k$ would be restricted to the sum of all elements. This was necessary because for some sub-problems we would be stuck in an infinite loop. Consider, going from 5 to -3 by adding -8 in it and then adding 8 in -3 to get back at 5.

We use dynamic programming to make the solution efficient. This can be done by taking an array $M(i, j, l)$ which will store boolean values for each case. So in implementation we start from $M(0, 0, 0)$ and start adding elements to $i$ and $j$ and we stop when the entries reach $\mu$. Let's say we have array $A = \{1, 1, 1\}$ then $M(1, 0, 1) = M(0, 1, 1) = M(0, 0, 1) = true$. What we are doing here is assigning first element to either of the three sets. Once we terminate the three loops we can check the element $M(\mu, \mu, n)$. If it is true, we know that we were able to reach $(\mu, \mu, \mu)$ from $(0, 0, 0)$ by taking all the $n$ elements of the array.

**Time Complexity :** We terminate the loops once we have exhausted all the elements. We also stop adding more to the particular sets once their limit $\mu$ is reached. So there will be two loops that will run for $\mathcal{O}(\sum a_i)$ steps. The third loop will be of the elements which will run for $\mathcal{O}(n)$ time. Let the total sum of the elements $\sum a_i = q$ Hence the total time complexity of the algorithm would be $\mathcal{O}(nq^2)$.