N meetings in one room

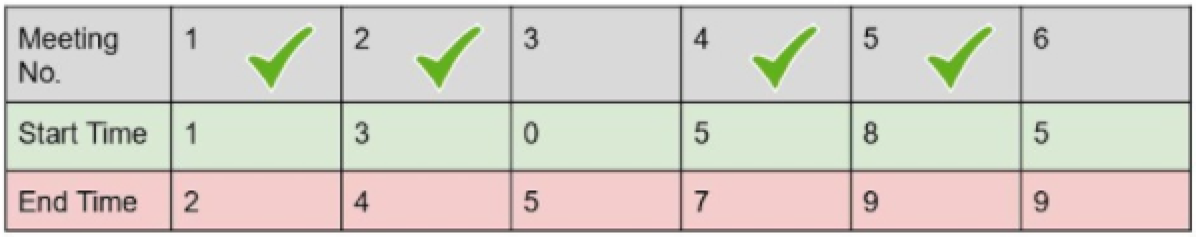
Problem Statement: There is **one** meeting room in a firm. You are given two arrays, start and end each of size N.For an index 'i', start[i] denotes the starting time of the ith meeting while end[i] will denote the ending time of the ith meeting. Find the maximum number of meetings that can be accommodated if only one meeting can happen in the room at a particular time. Print the order in which these meetings will be performed.

```
Example:

Input:  N = 6,  start[] = {1,3,0,5,8,5}, end[] =  {2,4,5,7,9,9}

Output: 1 2 4 5

Explanation: See the figure for a better understanding.
```



Meeting No.	1 ✓	2 ✓	3	4 ✓	5 ✓	6
Start Time	1	3	0	5	8	5
End Time	2	4	5	7	9	9

Solution:

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Initial Thought Process:-

Say if you have two meetings, one which gets over early and another which gets over late. Which one should we choose? If our meeting lasts longer the room stays occupied and we lose our time. On the other hand, if we choose a meeting that finishes early we can accommodate more meetings. Hence we should **choose meetings that end early** and utilize the remaining time for more meetings.

Approach:

To proceed we need a vector of three quantities: the starting time, ending time, meeting number. Sort this data structure in ascending order of end time.

We need a variable to store the answer. Initially, the answer is 1 because the first meeting can always be performed. Make another variable, say limit that keeps track of the ending time of the meeting that was last performed. Initially set limit as the end time of the first meeting.

Start iterating from the second meeting. At every position we have two possibilities:-

- If the start time of a meeting is strictly greater than ***limit*** we can perform the meeting. Update the answer.Our new ***limit*** is the ending time of the current meeting since it was last performed.Also update ***limit.***
- If the start time is less than or equal to ***limit*** ,skip and move ahead.

Let's have a dry run by taking the following example.
N = 6, start[] = {1,3,0,5,8,5}, end[] = {2,4,5,7,9,9}
Initially set answer =[1],limit = 2.
For Position 2 –
Start time of meeting no. 2 = 3 > limit. Update answer and limit.
Answer = [1, 2], limit = 4.
For Position 3 –
Start time of meeting no. 3 = 0 < limit.Nothing is changed.
For Position 4 –
Start time of meeting no. 4 = 5 > limit. Update answer and limit.
Answer = [1,2,4], limit = 7.
For Position 5 –
Start time of meeting no. 5 = 8 > limit.Update answer and limit.
Answer = [1,2,4,5], limit = 9.
For Position 6 –
Start time of meeting no. 6 = 8 < limit.Nothing is changed.
Final answer = [1,2,4,5]

Code:

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

struct meeting {
   int start;
   int end;
   int pos;
};

class Solution {
   public:
      bool static comparator(struct meeting m1, meeting m2) {
         if (m1.end < m2.end) return true;
         else if (m1.end > m2.end) return false;
         else if (m1.pos < m2.pos) return true;
         return false;
      }
   void maxMeetings(int s[], int e[], int n) {
      struct meeting meet[n];
      for (int i = 0; i < n; i++) {
         meet[i].start = s[i], meet[i].end = e[i], meet[i].pos = i +
1;
      }

      sort(meet, meet + n, comparator);

      vector < int > answer;

      int limit = meet[0].end;
      answer.push_back(meet[0].pos);

      for (int i = 1; i < n; i++) {
         if (meet[i].start > limit) {
            limit = meet[i].end;
            answer.push_back(meet[i].pos);
         }
      }
      cout<<"The order in which the meetings will be performed is
"<<endl;
      for (int i = 0; i < answer.size(); i++) {
         cout << answer[i] << " ";
      }

   }

};
int main() {
   Solution obj;
   int n = 6;
   int start[] = {1,3,0,5,8,5};
   int end[] = {2,4,5,7,9,9};
   obj.maxMeetings(start, end, n);
   return 0;}
```

Output:

```
The order in which the meetings will be performed is
1 2 4 5
```

Time Complexity: O(n) to iterate through every position and insert them in a data structure**.** **O(n log n)** to sort the data structure in ascending order of end time**. O(n)** to iterate through the positions and check which meeting can be performed.

Overall : O(n) +O(n log n) + O(n) ~O(n log n)

Space Complexity: O(n) since we used an additional data structure for storing the start time, end time, and meeting no.

Minimum number of platforms required for a railway

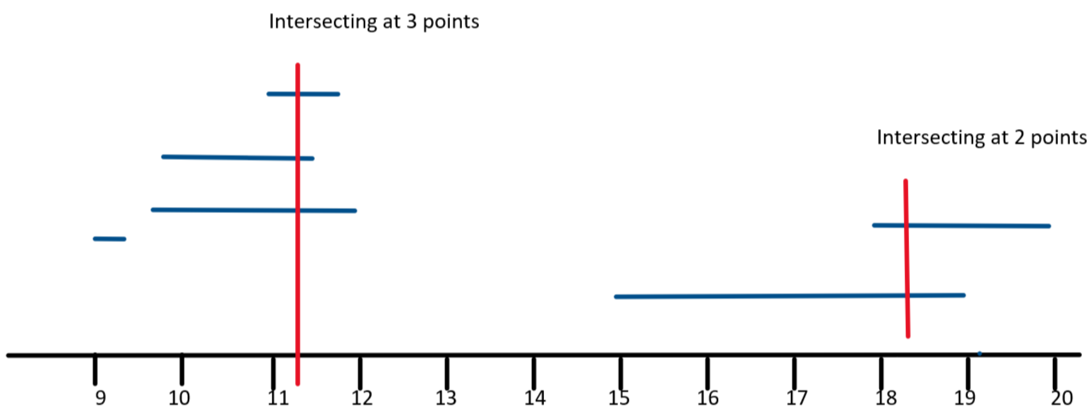
Problem Statement: We are given two arrays that represent the arrival and departure times of trains that stop at the platform. We need to find the minimum number of platforms needed at the railway station so that no train has to wait.

Examples 1:

```
Input: N=6,
arr[] = {9:00, 9:45, 9:55, 11:00, 15:00, 18:00}
dep[] = {9:20, 12:00, 11:30, 11:50, 19:00, 20:00}

Output:3
```

Explanation: There are at-most three trains at a time. The train at 11:00 arrived but the trains which had arrived at 9:45 and 9:55 have still not departed. So, we need at least three platforms here.



Example 2:

```
Input Format: N=2,
arr[]={10:20,12:00}
dep[]={10:50,12:30}

Output: 1
```

Explanation: Before the arrival of the new train, the earlier train already departed. So, we don't require multiple platforms.

Solution
Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Solution 1: **Naive Approach**
Intuition: Take each interval of arrival and departure one by one and count the number of overlapping time intervals. This can easily be done using nested for-loops. Maintain the maximum value of the count during the process and return the maximum value at the end.
Approach: We need to run two nested for-loops. Inside the inner loop count the number of intervals which intersect with the interval represented by the outer loop. As soon as the inner loop ends just update the maximum value of count and proceed with the next iteration of the outer loop. After the process ends we will get the maximum value of the count.
Code:
```
#include<bits/stdc++.h>
 using namespace std;

 int countPlatforms(int n,int arr[],int dep[])
 {
    int ans=1; //final value
    for(int i=0;i<=n-1;i++)
    {
        int count=1; // count of overlapping interval of only this
iteration
        for(int j=i+1;j<=n-1;j++)
        {
            if((arr[i]>=arr[j] && arr[i]<=dep[j]) ||
           (arr[j]>=arr[i] && arr[j]<=dep[i]))
           {
               count++;
           }
        }
        ans=max(ans,count); //updating the value
    }
    return ans;
 }

 int main()
 {
    int arr[]={900,945,955,1100,1500,1800};
    int dep[]={920,1200,1130,1150,1900,2000};
    int n=sizeof(dep)/sizeof(dep[0]);
    cout<<"Minimum number of Platforms required
"<<countPlatforms(n,arr,dep)<<endl;
 }
```

Output: Minimum number of Platforms required 3
Time Complexity: O(n^2) (due to two nested loops).
Space Complexity: O(1) (no extra space used).

Solution 2: **Efficient Approach** [Sorting]
Intuition: At first we need to sort both the arrays. When the events will be sorted, it will be easy to track the count of trains that have arrived but not departed yet. Total platforms needed at one time can be found by taking the difference of arrivals and departures at that time and the maximum value of all times will be the final answer.
Approach: At first we need to sort both the arrays. When the events will be sorted, it will be easy to track the count of trains that have arrived but not departed yet. Total platforms needed at one time can be found by taking the difference of arrivals and departures at that time and the maximum value of all times will be the final answer. If(arr[i]<=dep[j]) means if arrival time is less than or equal to the departure time then- we need one more platform. So increment count as well as increment i. If(arr[i]>dep[j]) means arrival time is more than the departure time then- we have one extra platform which we can reduce. So decrement count but increment j. Update the ans with max(ans, count) after each iteration of the while loop.
Code:
```
#include<bits/stdc++.h>
 using namespace std;

 int countPlatforms(int n,int arr[],int dep[])
 {
    sort(arr,arr+n);
    sort(dep,dep+n);

    int ans=1;
    int count=1;
    int i=1,j=0;
    while(i<n && j<n)
    {
        if(arr[i]<=dep[j]) //one more platform needed
        {
            count++;
            i++;
        }
        else //one platform can be reduced
        {
            count--;
            j++;
        }
        ans=max(ans,count); //updating the value with the current
maximum
    }
    return ans;
 }

 int main()
 {
    int arr[]={900,945,955,1100,1500,1800};
    int dep[]={920,1200,1130,1150,1900,2000};
    int n=sizeof(dep)/sizeof(dep[0]);
    cout<<"Minimum number of Platforms required
"<<countPlatforms(n,arr,dep)<<endl;
 }
```

Output:
Minimum number of Platforms required 3
Time Complexity: O(nlogn) Sorting takes O(nlogn) and traversal of arrays takes O(n) so overall time complexity is O(nlogn).
Space complexity: O(1) (No extra space used).

Job Sequencing Problem

Problem Statement: You are given a set of N jobs where each job comes with a **deadline** and **profit**. The profit can only be earned upon completing the job within its deadline. Find the **number of jobs** done and the **maximum profit** that can be obtained. Each job takes a **single unit** of time and only **one job** can be performed at a time.
Examples

```
Example 1:

Input: N = 4, Jobs = {(1,4,20),(2,1,10),(3,1,40),(4,1,30)}

Output: 2 60

Explanation: The 3rd job with a deadline 1 is performed during the
first unit of time .The 1st job is performed during the second unit
of time as its deadline is 4.
Profit = 40 + 20 = 60

Example 2:

Input: N = 5, Jobs = {(1,2,100),(2,1,19),(3,2,27),(4,1,25),(5,1,15)}

Output: 2 127

Explanation: The  first and third job both having a deadline 2 give
the highest profit.
Profit = 100 + 27 = 127
```

Solution

***Disclaimer**: Don’t jump directly to the solution, try it out yourself first.*

Approach: The strategy to maximize profit should be to pick up jobs that offer **higher profits.** Hence we should **sort** the jobs in descending order of profit. Now say if a job has a deadline of 4 we can perform it anytime between day 1-4, but it is preferable to perform the job on its **last day**. This leaves enough empty slots on the previous days to perform other jobs.

Basic Outline of the approach:-

- Sort the jobs in descending order of profit.
- If the maximum deadline is x, make an array of size x .Each array index is set to -1 initially as no jobs have been performed yet.
- For every job check if it can be performed on its last day.
- If possible mark that index with the job id and add the profit to our answer.
- If not possible, loop through the previous indexes until an empty slot is found.

DRY RUN:

Job ID	Profit	Deadline
3	40	2
4	30	2
1	20	4
2	10	1

-1	-1	-1	-1

Profit = 90 , No of jobs done = 3

Code:

```
#include<bits/stdc++.h>
using namespace std;
// A structure to represent a job
struct Job {
   int id; // Job Id
   int dead; // Deadline of job
   int profit; // Profit if job is over before or on deadline
};
class Solution {
   public:
      bool static comparison(Job a, Job b) {
         return (a.profit > b.profit);
      }
   //Function to find the maximum profit and the number of jobs done
   pair < int, int > JobScheduling(Job arr[], int n) {

      sort(arr, arr + n, comparison);
      int maxi = arr[0].dead;
      for (int i = 1; i < n; i++) {
         maxi = max(maxi, arr[i].dead);
      }

      int slot[maxi + 1];

      for (int i = 0; i <= maxi; i++)
         slot[i] = -1;

      int countJobs = 0, jobProfit = 0;

      for (int i = 0; i < n; i++) {
         for (int j = arr[i].dead; j > 0; j--) {
            if (slot[j] == -1) {
               slot[j] = i;
               countJobs++;
               jobProfit += arr[i].profit;
               break;
            }
         }
      }

      return make_pair(countJobs, jobProfit);
   }
};
int main() {
   int n = 4;
   Job arr[n] = {{1,4,20},{2,1,10},{3,2,40},{4,2,30}};

   Solution ob;
   //function call
   pair < int, int > ans = ob.JobScheduling(arr, n);
   cout << ans.first << " " << ans.second << endl;

   return 0;
}
```

Output: 3 90
Time Complexity: O(N log N) + O(N*M).
O(N log N) for sorting the jobs in decreasing order of profit. **O(N*M)** since we are iterating through all **N** jobs and for every job we are checking from the last deadline, say **M** deadlines in the worst case.
Space Complexity: O(M) for an array that keeps track on which day which job is performed if **M** is the maximum deadline available.

Fractional Knapsack Problem : Greedy Approach

Problem Statement: The weight of **N** items and their corresponding values are given. We have to put these items in a knapsack of weight **W** such that the **total value** obtained is **maximized.**
Note: We can either take the item as a whole or break it into smaller units.

```
Example:

Input: N = 3, W = 50, values[] = {100,60,120}, weight[] = {20,10,30}.

Output: 240.00

Explanation: The first and second items  are taken as a whole  while
only 20 units of the third item is taken. Total value = 100 + 60 + 80
= 240.00
```

Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*
Approach:
The greedy method to maximize our answer will be to pick up the items with higher values. Since it is possible to break the items as well we should focus on picking up items having higher value /weight first. To achieve this, items should be sorted in decreasing order with respect to their value /weight. Once the items are sorted we can iterate. Pick up items with weight lesser than or equal to the current capacity of the knapsack. In the end, if the weight of an item becomes more than what we can carry, break the item into smaller units. Calculate its value according to our current capacity and add this new value to our answer.
Let's understand with an example:-
N = 3, W = 50, values[] = {100,60,120}, weight[] = {20,10,30}.
The value/weight of item 1 is **(100/20) = 5**,for item 2 is **(60/10) = 6** and for item 3 is **(120/30) = 4.**
Sorting them in decreasing order of value/weight we have

Item No.	1	2	3
Value	60	100	120
Weight	10	20	30

Initially capacity of bag(W) = 50, value = 0
Item 1 has a weight of 10, we can pick it up.
Current weight = 50 – 10 = 40 ,Current value = 60.00
Item 2 has a weight of 20 , we can pick it up.
Current weight = 40 – 20 = 20 ,Current value = 60.00 + 100.00 = 160.00
Item 3 has a weight of 30 , but current knapsack capacity is 20.Only a fraction of it is chosen.
Current weight = 20 – 20 = 0 ,Final value = 160.00 + (120/30)*20 = 240.00
Code:
```
#include <bits/stdc++.h>

using namespace std;

struct Item {
   int value;
   int weight;
};
class Solution {
   public:
      bool static comp(Item a, Item b) {
         double r1 = (double) a.value / (double) a.weight;
         double r2 = (double) b.value / (double) b.weight;
         return r1 > r2;
      }
   // function to return fractionalweights
   double fractionalKnapsack(int W, Item arr[], int n) {

      sort(arr, arr + n, comp);

      int curWeight = 0;
      double finalvalue = 0.0;

      for (int i = 0; i < n; i++) {

         if (curWeight + arr[i].weight <= W) {
            curWeight += arr[i].weight;
            finalvalue += arr[i].value;
            } else {
               int remain = W - curWeight;
               finalvalue += (arr[i].value / (double) arr[i].weight) *
(double) remain;
               break;
            }
         }

         return finalvalue;

   }
};
int main() {
   int n = 3, weight = 50;
   Item arr[n] = { {100,20},{60,10},{120,30} };
   Solution obj;
   double ans = obj.fractionalKnapsack(weight, arr, n);
   cout << "The maximum value is " << setprecision(2) << fixed <<
ans;
   return 0;
}
```

Output:
The maximum value is 240.00
Time Complexity: O(n log n + n). O(n log n) to sort the items and **O(n)** to iterate through all the items for calculating the answer.
Space Complexity: O(1), no additional data structure has been used.

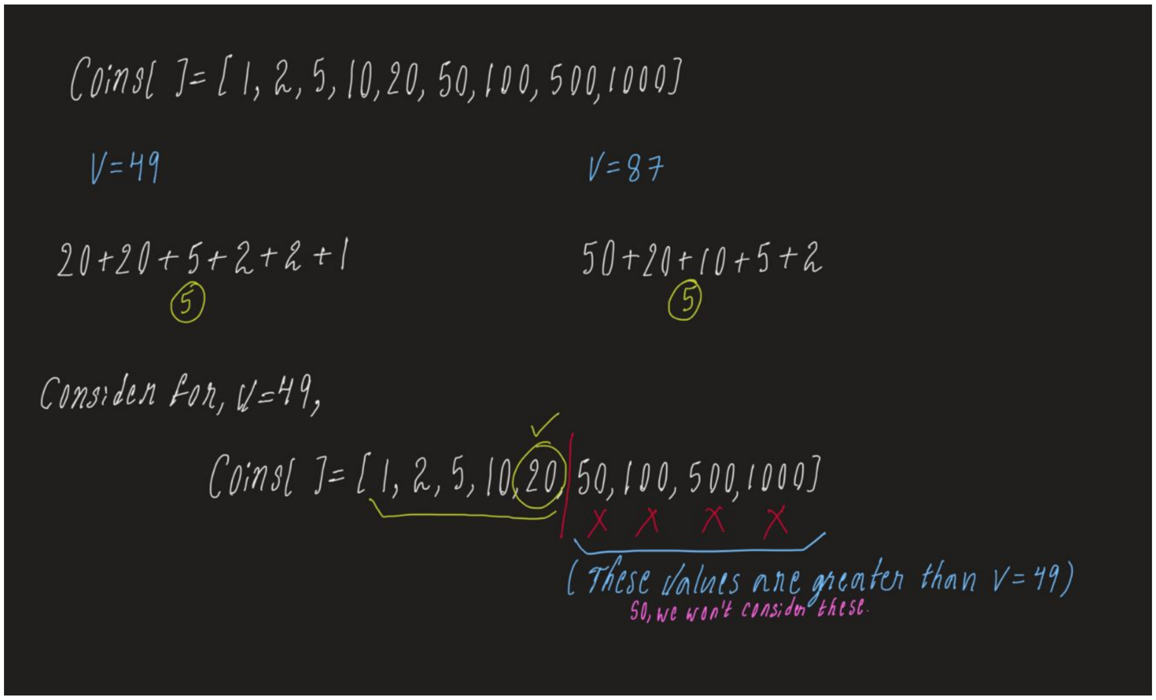
Find minimum number of coins

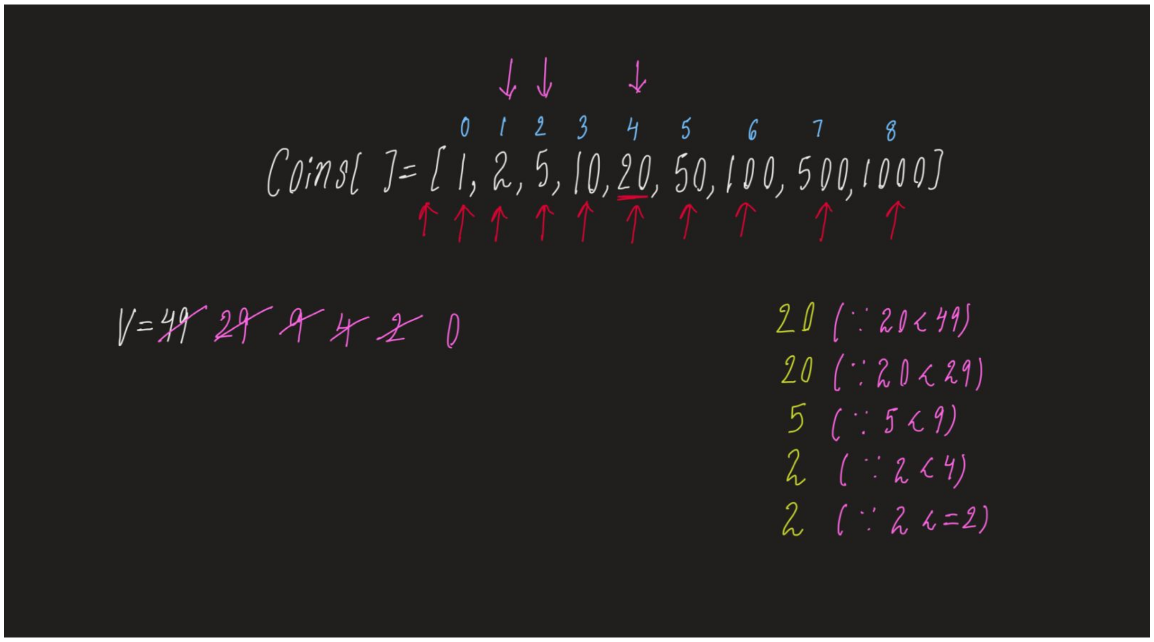
Problem Statement: Given a value V, if we want to make a change for V Rs, and we have an infinite supply of each of the denominations in Indian currency, i.e., we have an infinite supply of { 1, 2, 5, 10, 20, 50, 100, 500, 1000} valued coins/notes, what is the minimum number of coins and/or notes needed to make the change.
Examples:

```
Example 1:
Input: V = 70
Output: 2
Explaination: We need a 50 Rs note and a 20 Rs note.
Example 2:
Input: V = 121
Output: 3
Explaination: We need a 100 Rs note, a 20 Rs note and a 1 Rs coin.
```

Solution:
Disclaimer: *Don't jump directly to the solution, try it out yourself first.*
Solution: Greedy Algorithm

Approach: We will keep a pointer at the end of the array i. Now **while(V >= coins[i])** we will reduce V by coins[i] and add it to the ans array.
We will also ignore the coins which are greater than V and the coins which are less than V. We consider them and reduce the value of V by coins[I].


Coins[ ]= [1, 2, 5, 10, 20, 50, 100, 500, 1000]
V=49
V=87
20+20+5+2+2+1
5
50+20+10+5+2
5
Consider for, V=49,
Coins[ ]= [1, 2, 5, 10, 20, 50, 100, 500, 1000]
(These values are greater than V=49)
so, we won't consider these.



Code:

```
#include<bits/stdc++.h>
using namespace std;
int main() {
  int V = 49;
  vector < int > ans;
  int coins[] = {1, 2, 5, 10, 20, 50, 100, 500, 1000};
  int n = 9;
  for (int i = n - 1; i >= 0; i--) {
    while (V >= coins[i]) {
      V -= coins[i];
      ans.push_back(coins[i]);
    }
  }
  cout<<"The minimum number of coins is "<<ans.size()<<endl;
  cout<<"The coins are "<<endl;
  for (int i = 0; i < ans.size(); i++) {
    cout << ans[i] << " ";
  }

  return 0;
}
```

Output:

```
The minimum number of coins is 5
The coins are
20 20 5 2 2
```

Time Complexity:O(V)
Space Complexity:O(1)

N meetings in one room

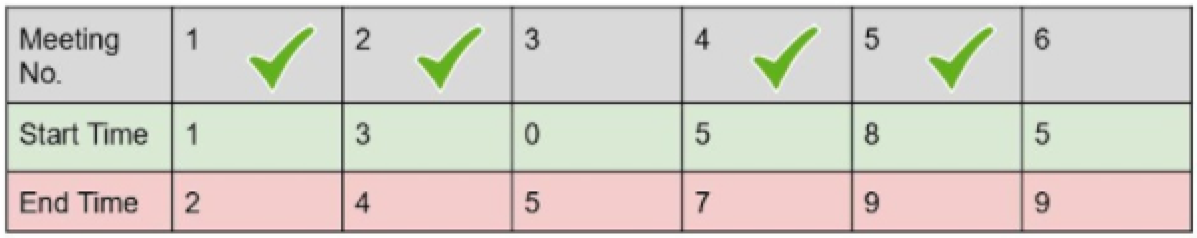
Problem Statement: There is **one** meeting room in a firm. You are given two arrays, start and end each of size N.For an index ‘i’, start[i] denotes the starting time of the ith meeting while end[i] will denote the ending time of the ith meeting. Find the maximum number of meetings that can be accommodated if only one meeting can happen in the room at a particular time. Print the order in which these meetings will be performed.

Example:

Input: `N = 6,  start[] = {1,3,0,5,8,5}, end[] =  {2,4,5,7,9,9}`

Output: `1 2 4 5`

Explanation: `See the figure for a better understanding.`



Meeting No.	1 ✓	2 ✓	3	4 ✓	5 ✓	6
Start Time	1	3	0	5	8	5
End Time	2	4	5	7	9	9

Solution:

***Disclaimer**: Don't jump directly to the solution, try it out yourself first.*

Initial Thought Process:-

Say if you have two meetings, one which gets over early and another which gets over late. Which one should we choose? If our meeting lasts longer the room stays occupied and we lose our time. On the other hand, if we choose a meeting that finishes early we can accommodate more meetings. Hence we should **choose meetings that end early** and utilize the remaining time for more meetings.

Approach:

To proceed we need a vector of three quantities: the starting time, ending time, meeting number. Sort this data structure in ascending order of end time.

We need a variable to store the answer. Initially, the answer is 1 because the first meeting can always be performed. Make another variable, say limit that keeps track of the ending time of the meeting that was last performed. Initially set limit as the end time of the first meeting.

Start iterating from the second meeting. At every position we have two possibilities:-

- If the start time of a meeting is strictly greater than ***limit*** we can perform the meeting. Update the answer.Our new ***limit*** is the ending time of the current meeting since it was last performed.Also update ***limit.***
- If the start time is less than or equal to ***limit*** ,skip and move ahead.

Let's have a dry run by taking the following example.

N = 6, start[] = {1,3,0,5,8,5}, end[] = {2,4,5,7,9,9}

Initially set answer =[1],limit = 2.

For Position 2 –

Start time of meeting no. 2 = 3 > limit. Update answer and limit.

Answer = [1, 2], limit = 4.

For Position 3 –

Start time of meeting no. 3 = 0 < limit.Nothing is changed.

For Position 4 –

Start time of meeting no. 4 = 5 > limit. Update answer and limit.

Answer = [1,2,4], limit = 7.

For Position 5 –

Start time of meeting no. 5 = 8 > limit.Update answer and limit.

Answer = [1,2,4,5], limit = 9.

For Position 6 –

Start time of meeting no. 6 = 8 < limit.Nothing is changed.

Final answer = [1,2,4,5]

Code:

```
#include <bits/stdc++.h>
using namespace std;
struct meeting {
   int start;
   int end;
   int pos;
};
class Solution {
   public:
      bool static comparator(struct meeting m1, meeting m2) {
         if (m1.end < m2.end) return true;
         else if (m1.end > m2.end) return false;
         else if (m1.pos < m2.pos) return true;
         return false;
      }
   void maxMeetings(int s[], int e[], int n) {
      struct meeting meet[n];
      for (int i = 0; i < n; i++) {
         meet[i].start = s[i], meet[i].end = e[i], meet[i].pos = i +
1;
      }

      sort(meet, meet + n, comparator);

      vector < int > answer;

      int limit = meet[0].end;
      answer.push_back(meet[0].pos);

      for (int i = 1; i < n; i++) {
         if (meet[i].start > limit) {
            limit = meet[i].end;
            answer.push_back(meet[i].pos);
         }
      }
      cout<<"The order in which the meetings will be performed is
"<<endl;
      for (int i = 0; i < answer.size(); i++) {
         cout << answer[i] << " ";
```

```
        }

    }

};
int main() {
    Solution obj;
    int n = 6;
    int start[] = {1,3,0,5,8,5};
    int end[] = {2,4,5,7,9,9};
    obj.maxMeetings(start, end, n);
    return 0;
}
```

Output:
The order in which the meetings will be performed is
1 2 4 5
Time Complexity: O(n) to iterate through every position and insert them in a data structure**.** **O(n log n)** to sort the data structure in ascending order of end time**. O(n)** to iterate through the positions and check which meeting can be performed.
Overall : O(n) +O(n log n) + O(n) ~O(n log n)
Space Complexity: O(n) since we used an additional data structure for storing the start time, end time, and meeting no.