Activity Selection Problem

The Activity Selection Problem is an optimization problem which deals with the selection of non-conflicting activities that needs to be executed by a single person or machine in a given time frame.

Each activity is marked by a start and finish time. Greedy technique is used for finding the solution since this is an optimization problem.

What is Activity Selection Problem?

Let's consider that you have n activities with their start and finish times, the objective is to find solution set having **maximum number of non-conflicting activities** that can be executed in a single time frame, assuming that only one person or machine is available for execution.

Some **points to note** here:

- It might not be possible to complete all the activities, since their timings can collapse.
- Two activities, say **i** and **j**, are said to be non-conflicting if $s_i >= f_j$ or $s_j >= f_i$ where s_i and s_j denote the starting time of activities **i** and **j** respectively, and f_i and f_j refer to the finishing time of the activities **i** and **j** respectively.
- **Greedy approach** can be used to find the solution since we want to maximize the count of activities that can be executed. This approach will greedily choose an activity with earliest finish time at every step, thus yielding an optimal solution.

Input Data for the Algorithm:

- $act[]$ array containing all the activities.
- $s[]$ array containing the starting time of all the activities.
- $f[]$ array containing the finishing time of all the activities.

Ouput Data from the Algorithm:

- $sol[]$ array refering to the solution set containing the maximum number of non-conflicting activities.

Steps for Activity Selection Problem

Following are the steps we will be following to solve the activity selection problem,

Step 1: Sort the given activities in ascending order according to their finishing time.

Step 2: Select the first activity from sorted array act[] and add it to sol[] array.

Step 3: Repeat steps 4 and 5 for the remaining activities in act[].

Step 4: If the start time of the currently selected activity is greater than or equal to the finish time of previously selected activity, then add it to the sol[] array.

Step 5: Select the next activity in act[] array.

Step 6: Print the sol[] array.

Activity Selection Problem Example

Let's try to trace the steps of above algorithm using an example:

In the table below, we have 6 activities with corresponding start and end time, the objective is to compute an execution schedule having maximum number of non-conflicting activities:

Start Time (s)	Finish Time (f)	Activity Name
5	9	a1
1	2	a2
3	4	a3
0	6	a4
5	7	a5
8	9	a6

A possible **solution** would be:

Step 1: Sort the given activities in ascending order according to their finishing time.

The table after we have sorted it:

Start Time (s)	Finish Time (f)	Activity Name
1	2	a2
3	4	a3
0	6	a4
5	7	a5
5	9	a1
8	9	a6

Step 2: Select the first activity from sorted array act[] and add it to the sol[] array, thus **sol = {a2}**.

Step 3: Repeat the steps 4 and 5 for the remaining activities in act[].

Step 4: If the start time of the currently selected activity is greater than or equal to the finish time of the previously selected activity, then add it to sol[].

Step 5: Select the next activity in act[]

For the data given in the above table,

A. Select activity **a3**. Since the start time of **a3** is greater than the finish time of **a2** (i.e. $s(a3) > f(a2)$), we add **a3** to the solution set. Thus **sol = {a2, a3}**.

B. Select **a4**. Since $s(a4) < f(a3)$, it is not added to the solution set.

C. Select **a5**. Since $s(a5) > f(a3)$, **a5** gets added to solution set. Thus **sol = {a2, a3, a5}**

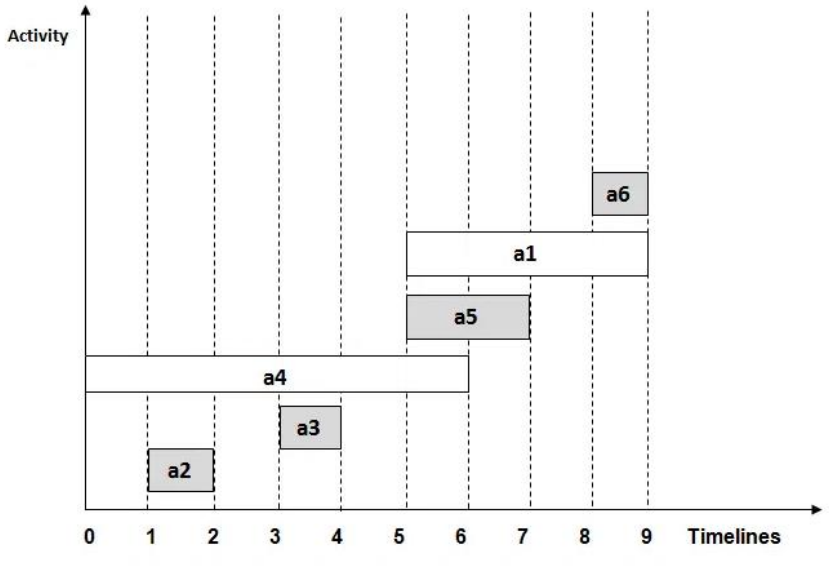
D. Select **a1**. Since $s(a1) < f(a5)$, **a1** is not added to the solution set.

E. Select **a6**. **a6** is added to the solution set since $s(a6) > f(a5)$. Thus **sol = {a2, a3, a5, a6}**.

Step 6: At last, print the array sol[]

Hence, the execution schedule of maximum number of non-conflicting activities will be:

Start Time (s)	Finish Time (f)	Activity Name
1	2	a2
3	4	a3
0	6	a4
5	7	a5
5	9	a1
8	9	a6



(1,2), (3,4), (5,7), (8,9)

In the above diagram, the selected activities have been highlighted in grey.

Implementation of Activity Selection Problem Algorithm

Now that we have an overall understanding of the activity selection problem as we have already discussed the algorithm and its working details with the help of an example, following is the C++ implementation for the same.

Note: The algorithm can be easily written in any programming language.

```
#include <bits/stdc++.h>
using namespace std;
#define N 6            // defines the number of activities
// Structure represents an activity having start time and finish time.
struct Activity
{
   int start, finish;
};

// This function is used for sorting activities according to finish time
bool Sort_activity(Activity s1, Activity s2)
{
   return (s1.finish< s2.finish);
}

void print_Max_Activities(Activity arr[], int n)
{
   // Sort activities according to finish time
         sort(arr, arr+n, Sort_activity);
         cout<< "Following activities are selected \n";
   // Select the first activity
   int i = 0;
         cout<< "(" <<arr[i].start<< ", " <<arr[i].finish << ")\n";
   // Consider the remaining activities from 1 to n-1
   for (int j = 1; j < n; j++)
   {
         // Select this activity if it has start time greater than or equal to the finish time of previously selected activity
      if (arr[j].start>= arr[i].finish)
         {
         cout<< "(" <<arr[j].start<< ", "<<arr[j].finish << ") \n";
```

```
            i = j;
            }
    }
}

// Driver program
int main()
{
    Activity arr[N];
            for(int i=0; i<=N-1; i++)
            {
            cout<<"Enter the start and end time of "<<i+1<<" activity \n";
                    cin>>arr[i].start>>arr[i].finish;
    }
            print_Max_Activities(arr, N);
    return 0;
}
```

Time Complexity Analysis

Following are the scenarios for computing the time complexity of Activity Selection Algorithm:

- **Case 1**: When a given set of activities are already sorted according to their finishing time, then there is no sorting mechanism involved, in such a case the complexity of the algorithm will be $O(n)$
- **Case 2**: When a given set of activities is unsorted, then we will have to use the `sort()` method defined in **bits/stdc++** header file for sorting the activities list. The time complexity of this method will be $O(nlogn)$, which also defines complexity of the algorithm.

Real-life Applications of Activity Selection Problem

Following are some of the real-life applications of this problem:

- Scheduling multiple competing events in a room, such that each event has its own start and end time.
- Scheduling manufacturing of multiple products on the same machine, such that each product has its own production timelines.
- Activity Selection is one of the most well-known generic problems used in Operations Research for dealing with real-life business problems.