Dash
All
Articles
Videos
Problems
Quiz
Contest
Prev
Next

Activity Selection Problem

You are given n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.

Example:

Example 1 : Consider the following 3 activities sorted by by finish time.

```
     start[]  =  {10, 12, 20};
     finish[] =  {20, 25, 30};
```

A person can perform at most **two** activities. The maximum set of activities that can be executed is {0, 2} [These are indexes in start[] and finish[]]

Example 2 : Consider the following 6 activities sorted by finish time.

```
     start[]  =  {1, 3, 0, 5, 8, 5};
     finish[] =  {2, 4, 6, 7, 9, 9};
```

A person can perform at most **four** activities. The maximum set of activities that can be executed is {0, 1, 3, 4} [These are indexes in start[] and finish[]]

The greedy choice is to always pick the next activity whose finish time is least among the remaining activities and the start time is more than or equal to the finish time of the previously selected activity. We can sort the activities according to their finishing time so that we always consider the next activity as minimum finishing time activity.
1) Sort the activities according to their finishing time
2) Select the first activity from the sorted array and print it.
3) Do the following for the remaining activities in the sorted array.
.....a) If the start time of this activity is greater than or equal to the finish time of the previously selected activity then select this activity and print it.

C++ Java

```java
// The following implementation assumes that the activities
// are already sorted according to their finish time
import java.util.*;
import java.lang.*;
import java.io.*;

class ActivitySelection
{
    // Prints a maximum set of activities that can be done by a single
    // person, one at a time.
    //  n   -->  Total number of activities
    //  s[] -->  An array that contains start time of all activities
    //  f[] -->  An array that contains finish time of all activities
    public static void printMaxActivities(int s[], int f[], int n)
    {
    int i, j;

    System.out.print("Following activities are selected : \n");

    // The first activity always gets selected
    i = 0;
    System.out.print(i+" ");
```

```
        // Consider rest of the activities
        for (j = 1; j < n; j++)
        {
            // If this activity has start time greater than or
            // equal to the finish time of previously selected
            // activity, then select it
            if (s[j] >= f[i])
            {
                System.out.print(j+" ");
                i = j;
             }
         }
        }

        // driver program to test above function
        public static void main(String[] args)
        {
        int s[] =  {1, 3, 0, 5, 8, 5};
        int f[] =  {2, 4, 6, 7, 9, 9};
        int n = s.length;

        printMaxActivities(s, f, n);
        }

    }
```


Output

```
Following activities are selected n0 1 3 4
```

How does Greedy Choice work for Activities sorted according to finish time?

Dash
All
Articles
Videos
Problems
Quiz
Contest

Let the given set of activities be S = {1, 2, 3, ...n} and activities are sorted by finish time. The greedy choice is to always pick activity 1. How come activity 1 always provides one of the optimal solutions. We can prove it by showing that if there is another solution B with the first activity other than 1, then there is also a solution A of the same size with activity 1 as the first activity. Let the first activity selected by B be k, then there always exist A = {B - {k}} U {1}.

(Note that the activities in B are independent and k has the smallest finishing time among all. Since k is not 1, finish(k) >= finish(1)).

How to implement when given activities are not sorted?

We create a structure/class for activities. We sort all activities by finish time (Refer sort in C++ STL). Once we have activities sorted, we apply the same algorithm.

Below image is an illustration of the above approach:

Initially : arr[] = { { 5,9 } , { 1,2 } , { 3,4 } , { 0,6 } , { 5,7 } , { 8,9 } }

Step 1:
Sort the array according to finish time
arr[] = { { 1,2 } , { 3,4 } , { 0,6 } , { 5,7 } , { 8,9 } , { 5,9 } }

Step 2:
Print first activity and make i = 0
print = ({ 1,2 })

Step 3:
arr[] = { { 1,2 } , { 3,4 } , { 0,6 } , { 5,7 } , { 8,9 } , { 5,9 } }
j
Start[j] >= finish[i]. print({ 3,4 })
make i = j , j++

Step 4:
arr[] = { { 1,2 } , { 3,4 } , { 0,6 } , { 5,7 } , { 8,9 } , { 5,9 } }
j
Start[j] < finish[i]. j++

Step 5:
arr[] = { { 1,2 } , { 3,4 } , { 0,6 } , { 5,7 } , { 8,9 } , { 5,9 } }
j
Start[j] >= finish[i]. print ({ 5,7 })
make i = j , j++

Step 6:
arr[] = { { 1,2 } , { 3,4 } , { 0,6 } , { 5,7 } , { 8,9 } , { 5,9 } }
j
make i = j , j++

Step 6:
arr[] = { { 1,2 } , { 3,4 } , { 0,6 } , { 5,7 } , { 8,9 } , { 5,9 } }
Start[j] < finish[i].

Below is the implementation of the above approach:

C++ Java

```java
// Java program for activity selection problem
// when input activities may not be sorted.
import java.io.*;
import java.util.*;

// A job has a start time, finish time and profit.
class Activity
{
  int start, finish;

  // Constructor
  public Activity(int start, int finish)
  {
    this.start = start;
    this.finish = finish;
  }
}

// class to define user defined comparator
class Compare
{

  // A utility function that is used for sorting
  // activities according to finish time
  static void compare(Activity arr[], int n)
  {
    Arrays.sort(arr, new Comparator<Activity>()
                {
                  @Override
```

```
                    public int compare(Activity s1, Activity s2)
                    {
                      return s1.finish - s2.finish;
                    }
                  });
  }
}

// Driver class
class GFG {

  // Returns count of the maximum set of activities that
  // can
  // be done by a single person, one at a time.
  static void printMaxActivities(Activity arr[], int n)
  {
    // Sort jobs according to finish time
    Compare obj = new Compare();
    obj.compare(arr, n);
    System.out.println(
      "Following activities are selected :");

    // The first activity always gets selected
    int i = 0;
    System.out.print("(" + arr[i].start + ", "
                     + arr[i].finish + "), ");

    // Consider rest of the activities
    for (int j = 1; j < n; j++)
    {

      // If this activity has start time greater than
      // or equal to the finish time of previously
      // selected activity, then select it
      if (arr[j].start >= arr[i].finish)
      {
        System.out.print("(" + arr[j].start + ", "
                         + arr[j].finish + "), ");
        i = j;
      }
    }
```

```
    }

    // Driver code
    public static void main(String[] args)
    {

      int n = 6;
      Activity arr[] = new Activity[n];
      arr[0] = new Activity(5, 9);
      arr[1] = new Activity(1, 2);
      arr[2] = new Activity(3, 4);
      arr[3] = new Activity(0, 6);
      arr[4] = new Activity(5, 7);
      arr[5] = new Activity(8, 9);

      printMaxActivities(arr, n);
    }
}

// This code is contributed by Dharanendra L V.
```


Output:

```
Following activities are selected
(1, 2), (3, 4), (5, 7), (8, 9),
```

Time Complexity: It takes O(n log n) time if input activities may not be sorted. It takes O(n) time when it is given that input activities are always sorted.

Using STL we can solve it as follows:

CPP Java


```java
// java program for the above approach
import java.io.*;
import java.lang.*;
import java.util.*;

class GFG {

  // Pair class
  static class Pair {

    int first;
    int second;

    Pair(int first, int second)
    {
      this.first = first;
      this.second = second;
    }
  }

  static void SelectActivities(int s[], int f[])
  {

    // Vector to store results.
    ArrayList<Pair> ans = new ArrayList<>();

    // Minimum Priority Queue to sort activities in
    // ascending order of finishing time (f[i]).
    PriorityQueue<Pair> p = new PriorityQueue<>(
      (p1, p2) -> p1.first - p2.first);

    for (int i = 0; i < s.length; i++) {
      // Pushing elements in priority queue where the
      // key is f[i]
```

```
    Pair it = p.poll();
    int start = it.second;
    int end = it.first;
    ans.add(new Pair(start, end));

    while (!p.isEmpty()) {
      Pair itr = p.poll();
      if (itr.second >= end) {
        start = itr.second;
        end = itr.first;
        ans.add(new Pair(start, end));
      }
    }
    System.out.println(
      "Following Activities should be selected. \n");

    for (Pair itr : ans) {
      System.out.println(
        "Activity started at: " + itr.first
        + " and ends at  " + itr.second);
    }
  }

  // Driver Code
  public static void main(String[] args)
  {

    int s[] = { 1, 3, 0, 5, 8, 5 };
    int f[] = { 2, 4, 6, 7, 9, 9 };

    // Function call
    SelectActivities(s, f);
  }
}

// This code is contributed by Kingash.
```

Dash
All
Articles
Videos
Problems
Quiz
Contest

Output

```
Following Activities should be selected.

Activity started at: 1 and ends at  2
Activity started at: 3 and ends at  4
Activity started at: 5 and ends at  7
Activity started at: 8 and ends at  9
```