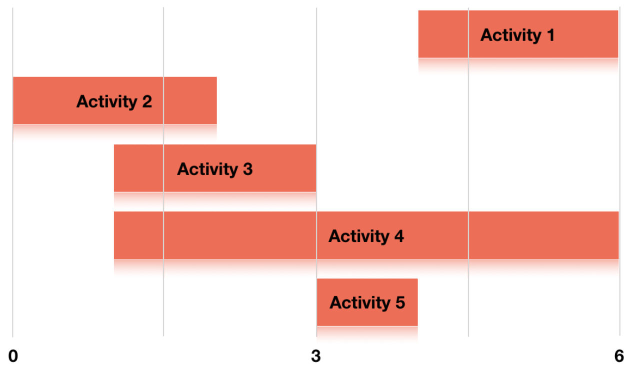
Activity Selection Problem | Greedy Algorithm

Activity selection problem is a problem in which a person has a list of works to do. Each of the activities has a starting time and ending time. We need to schedule the activities in such a way the person can complete a maximum number of activities. Since the timing of the activities can collapse, so it might not be possible to complete all the activities and thus we need to schedule the activities in such a way that the maximum number of activities can be finished.

Look at the following table containing activities, and their start and end time.

Activity (a_i)	1	2	3	4	5
s_i	4	0	1	1	3
f_i	6	2	3	6	4

Here, s_i and f_i are the starting and the finishing time of the activity a_i.

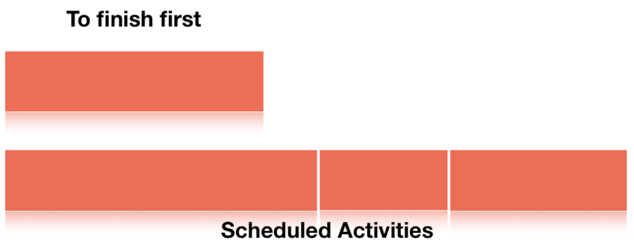


Greedy Approach to the Problem

We want to adjust the maximum number of activities to be completed. So, choosing the activity which is going to finish first will leave us maximum time to adjust the later activities. This is basically an intuition that greedily choosing the activity with earliest finish time will give us an optimal solution. Thus, our next task is to verify this intuition.

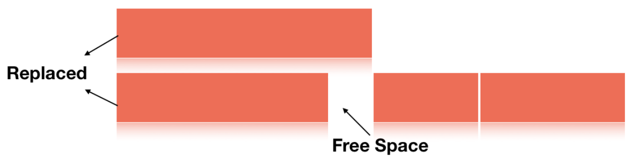
Verifying Greedy Will Work

Suppose, we have adjusted some activities in the given time slot and claiming that this solution is the optimal solution and the element first to be finished is not in this solution.



Now we can replace the first activity in the slot with the element with first finish time without any consequences. As we are only concerned with the number of activities and by replacing the activity, the number of activities will be the same.

(/add_ques



This is even going to give us some free time in the slot which can be used to further optimize the problem. So, we need the element of least finish time in our optimal solution and thus we just theoretically verified that making a greedy choice will lead us to the optimal solution.

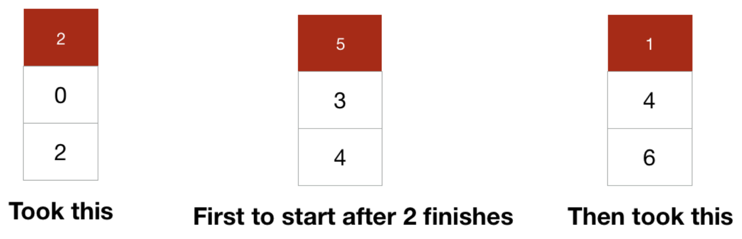
Solving the Problem

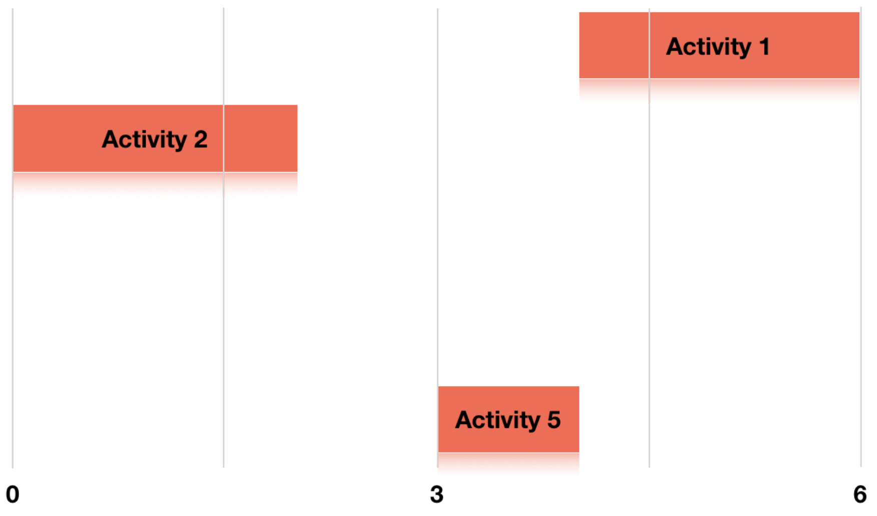
Now we know that greedily choosing the activity with the least finish time will give us the optimal solution, so our first task would be to sort the activities with their finishing times.

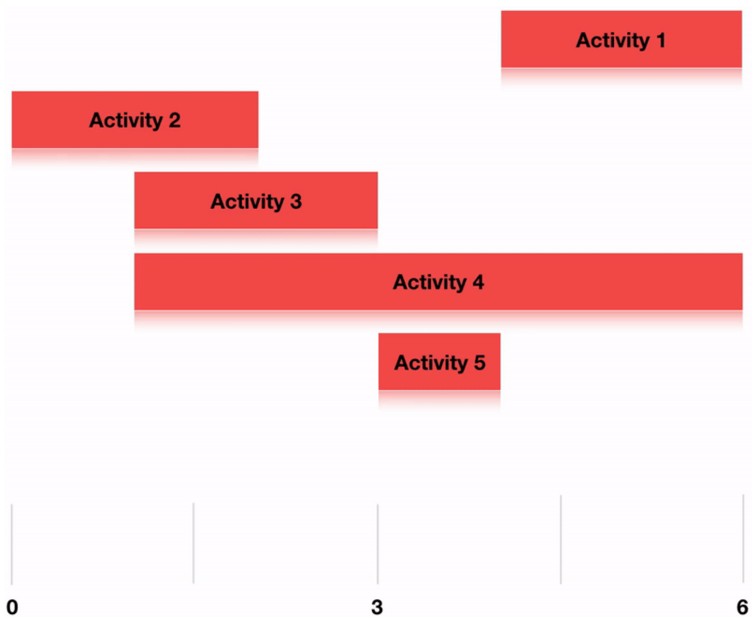
Activity (a_i)	2	3	5	1	4
s_i	0	1	3	4	1
f_i	2	3	4	6	6

Sorted according to finish time

To take the element with the least finish time, we will iterate over the list of the activities and will select the first activity and then we will select the activity which is starting next after the currently selected activity finishes.







So, let's write the code for the same.

Code for Activity Selection Problem

We need information about the activities to get started. So, we will start by passing the arrays containing the starting times and finishing times to our function - `ACTIVITY-SELECTION(a, s, f)`. Here, a is the array storing the activities numbers, s and f are the arrays of starting times and finishing times respectively.

We are assuming that these arrays are sorted according to the finish time of the activities. We know that we are going to start scheduling the activities by taking the first activity first. So, let's make an array which will have all the activities of the optimal solution and add the first activity to it i.e., `A = [a[1]]`. We also need to keep a track of the last element in the solution to see which element is going to start next after this activity finishes. So, we will store the index of this activity in a variable - `k = 1`.

Now, we need to iterate over the activities and choose the next activity which is finishing first after the completion of the first activity. As we have already included the first element, we will start our iteration from the 2nd element - `for i in 2 to a.length`.

We will find the activity which is going to start next after the last activity in the solution finishes. So, let first compare it with the current activity in the iteration - `if s[i] >= f[k]`. If this condition is true, then we will add this activity in our solution - `A.append(a[i])` and then point k to this - `k = i`.

```
for i in 2 to a.length
  if s[i] >= f[k]
    A.append(a[i])
    k = i
```

At last, we will just return this array A - `return A`.

```
ACTIVITY-SELECTION(a, s, f)
  A = [a[1]]
  k = 1

  for i in 2 to a.length
    if s[i] >= f[k]
      A.append(a[i])
      k=i
  return A
```

C **Python** **Java**

```c
#include <stdio.h>
#include <stdlib.h>

// function will return an array A
int* activity_selection(int a[], int s[], int f[], int n) {
  int* A = malloc(sizeof(int)*n); // array A of size n
  A[0] = 0; //array will start from index 1. So, fake item at index 0
  A[1] = a[1];

  int k=1;
  int i;
  int iter = 1;

  for(i=2; i<=n; i++) {
    if(s[i] >= f[k]) {
      //appending array A
      iter++;
      A[iter] = a[i];
      k=i;
    }
  }

  /*
    Making first element of the array A (index 0) equal to iter
    just to know the length of the array when used in different function.
  */
  A[0] = iter;
  return A;
}

int main() {
  //Arrays staring from 1. Elements at index 0 are fake
  int a[] = {0, 2, 3, 5, 1, 4};
  int s[] = {0, 0, 1, 3, 4, 1};
  int f[] = {0, 2, 3, 4, 6, 6};
  int *p = activity_selection(a, s, f, 5);

  int i;
  //p[0] is the length upto which activities are stored
  for(i=1; i<=p[0]; i++) {
    printf("%d\n",p[i]);
  }
  return 0;
}
```

Analysis of the Algorithm

We can clearly see that the algorithm is taking a $\Theta(n)$ running time, where n is the number of activities. Also if the arrays passed to the function are not sorted, we can sort them in $O(n \lg n)$ time.

Let's move to the next chapter and solve an interesting problem using the greedy algorithm.

“ Each day, I come in with a positive attitude, trying to get better. ”

- Stefon Diggs

PREV (/course/algorithms-greedy-algorithm/) (/course/algorithms-egyptian-fraction/) NEXT