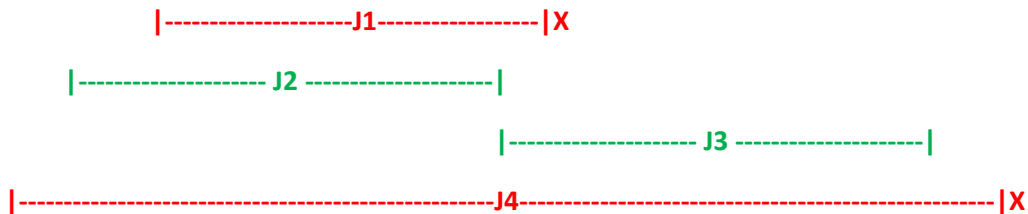


## HW2 Solution

1. There are alternatives to CLRS3's greedy choice of the earliest finishing time. For each of the following alternatives, prove or disprove that it constructs an optimal schedule (assume ties are broken arbitrarily) and briefly outline the implementation costs:

**Example of Earliest finishing time:**

Job	Start Time - $s[i]$	End time - $f[i]$	Duration
J1	5	10	5
J2	3	9	6
J3	9	16	7
J4	2	18	6

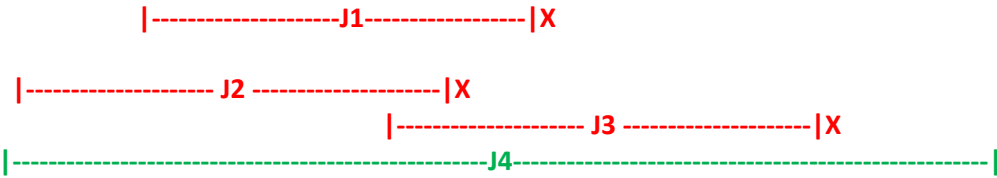


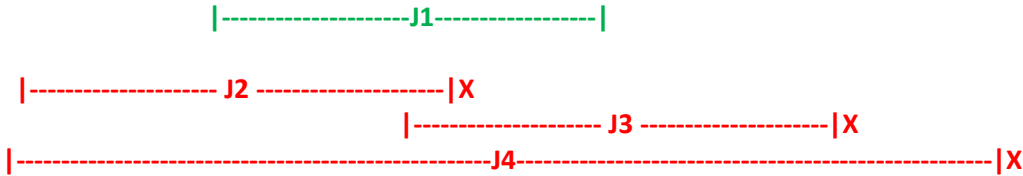
**Earliest finishing time:**

The earliest finishing time is less of **J2**. So **J2** is first job that is to be scheduled. Job **J3** is scheduled next. Since jobs **J1 & J4** conflicts with the start and end time of job **J2**, they are discarded. Job **J3** is scheduled next. Therefore, optimal solution for the earliest finishing time job scheduler is 2.

**Optimal schedule – 2 jobs scheduled**

(a)	<p><b>Choose the job that ends latest</b></p> <p><b>After Sorting based on finishing time:</b></p> <p>Only single jobs is scheduled based on jobs that ends latest.</p> <p>Consider the above example for job scheduling:</p>
-----	---

	<p>The finishing time of last scheduled interval is the job that ends latest;</p> <ul style="list-style-type: none"> <li>• In the above example, Job <b>J4</b> has minimum starting time and latest finishing time.</li> <li>• Therefore, job <b>J4</b> will be scheduled at first place.</li> <li>• Eliminating <b>J1, J2 &amp; J3</b> as they conflict <b>J4</b></li> <li>• <b>So only J4 will be scheduled based on latest finishing time.</b></li> </ul> <p><b>Pseudo Code:</b></p> <p>S = set of input intervals <math>\{(s[i], f[i])\}</math> // <math>s[]</math> – starting time of jobs , <math>f[]</math> – end time of jobs  Sort elements in set S based on finishing time  While S is not null;  q = latest finishing time job from set S  Output job in q  Discard conflicting intervals from set S that overlaps with q</p> <p>Sorting intervals – <math>O(n \log n)</math>  Traversing the list to output the first element that starts after last elements finishing time – <math>O(n)</math></p> <p>Therefore, time complexity = <b><math>O(n \log n)</math></b></p> <p>The above analysis proves that, jobs which ends latest <b>does not provide an optimal schedule.</b></p>
(b)	<p><b>Choose the job that starts first</b></p> <p>Job scheduling:</p>  <p>Consider the above example for job scheduling:</p> <ul style="list-style-type: none"> <li>• In the above example, Job <b>J4</b> has minimum starting time of 2.</li> <li>• If we start with <b>J4</b>, we need to discard the conflicting jobs and proceed further.</li> <li>• Jobs <b>J1, J2 &amp; J3</b> conflicts with <b>J4's</b> end time, so discarding it leaves with job <b>J4</b>.</li> <li>• <b>So, total 1 jobs J4 will be scheduled.</b></li> </ul> <p><b>Pseudo Code:</b></p> <p>S = set of input intervals <math>\{(s[i], f[i])\}</math> // <math>s[]</math> – starting time of jobs , <math>f[]</math> – end time of jobs  Sort elements in set S based on minimum starting time <math>s[i]</math>  While S is not null;  q = latest finishing time job from set S</p>

	<p>Output job in q Discard conflicting intervals from set S that overlaps with q</p> <p>Sorting – <math>O(n \log n)</math> Traversing the list to output – <math>O(n)</math></p> <p><u>Therefore, time complexity = <math>O(n \log n)</math></u></p> <p>This is not the optimal solution. It doesn't match with the earliest finishing time optimal solution.</p>
(c)	<p><b>Choose the job of shortest duration</b></p>  <p><b>1 jobs will be scheduled based on shortest duration.</b></p> <p>Consider the above example for job scheduling:</p> <ul style="list-style-type: none"> <li>• In the above example, Jobs <b>J1</b> has shortest duration and minimum start time.</li> <li>• Jobs <b>J2, J3 &amp; J4</b> will be discarded due to conflicting issues.</li> <li>• <b>Therefore, just 1 jobs will be scheduled.</b></li> </ul> <p>Sorting – <math>O(n \log n)</math> Traversing the list – <math>O(n)</math></p> <p><u>Therefore, time complexity = <math>O(n \log n)</math></u></p> <p>The optimal solution for this case is 1 which doesn't match with the ideal optimal solution of Earliest finishing time. <b>Therefore, this scenario isn't optimal solution.</b></p>

2. Consider a weighted form of the activity selection problem in which each of the  $n$  activities is weighted by its importance (the weight is unrelated to the starting and finishing times and to the duration). We want to choose a subset of non-conflicting activities with the highest total weight.

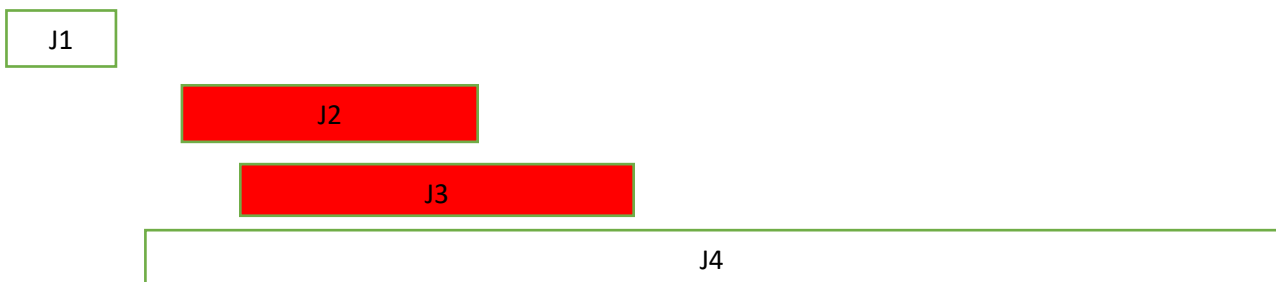
**Answer:**

Consider there are  $N$  number of jobs that are to be scheduled. Based on the weight assigned by the importance, the optimal schedule will result into highest total weight.

Example:  $N = 4$

Job	Start Time	Finish Time	Weight
J1	2	3	40
J2	4	6	70
J3	7	19	200
J4	3	100	300

Job scheduler based on minimum start time

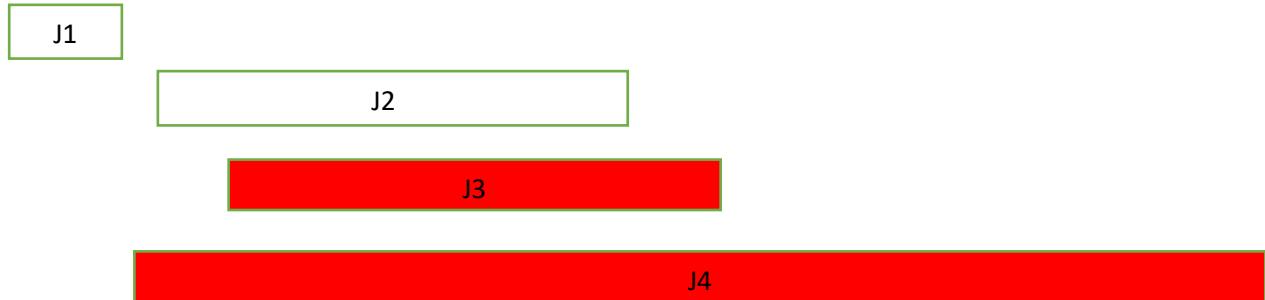


Consider above job scheduling example;

- Job **J1** will be scheduled first as it has minimum start time of 2
- Jobs **J3 & J2** are conflicting with **J4**, so they are to be discarded
- Job **J4** will be scheduled next as it has starting time of 3
- Therefore total weight after scheduling would be **weight (J1) + weight (J4) ->  $300 + 40 = 340$**
- **So, the optimal schedule based on highest total weight will be 340**

(a) Prove that the heuristic of choosing the earliest finishing time does not always result in an optimal schedule.

**Job scheduling based on earliest finishing time.**



Considering above scenario;

- Job **J1** has earliest finishing time. So, **J1** is scheduled at first place. **Weight = 40**
- Job **J2** starts after **J1**, as **J2** has next earliest finishing time. **Weight = 70**
- Jobs **J3 & J4** is being discarded because of conflicting issue.
- Therefore, final weight after job scheduling turns out to be **weight (J1) + weight (J2) -> 40+70 = 110**
- Considering the optimal schedule based on highest total weight and minimum start time, this job scheduling based on earliest finishing time is not optimal.
- **Hence proved that the heuristic of choosing the earliest finishing time does not always result in an optimal schedule.**

(b) Find an  $O(n^2)$  algorithm to compute an optimal schedule.

**Algorithm:**

Function MaxWeight(jobs, n)

Compare the jobs based on the finishing time

- sort (jobs, jobs+n, jobCompare); // jobs<start time, finish time,weight>
- Store weight of corresponding jobs into an array say table[]

Assign initial job weight at 0<sup>th</sup> location

table[0] = jobs[0].weight; // stores weight for jobs[i]

for (int i <- 1; i < n; i++)

currentweight = jobs[i].weight;

Compare corresponding jobs and discard conflicting jobs

for (int j <- n-1; j >= 0; j--)

{

if (a[j].finish <= a[n-1].start)

return j;

}

return -1;

if ( jobs aren't conflicting )

currentweight <- currentweight + table[i];

table[i] <- max(currentweight, table[i-1]);

//update the weight to get maximum total weight

// maximum weight of including & excluding jobs

return table[i];

Sorting –  $O(n \log n)$

Two for loops are executed –  $O(n^2)$ .

**Therefore, time complexity –  $O(n^2)$**

3. Your new hybrid car that uses fuel extremely efficiently, but can only travel 100 miles on a single battery. The car's fuel is stored in a single-use battery, which must be replaced after at most 100 miles. The actual fuel is virtually free, but the batteries are expensive and can only be installed by licensed battery-replacement technicians. Thus, even if you decide to replace your battery early, you must still pay full price for the new battery to be installed. Moreover, because these batteries are in high demand, no one can afford to own more than one battery at a time. You are trying to get from San Francisco to New York City on the new Inter-Continental Super-Highway, which runs in a direct line between these two cities. There are several fueling stations along the way; each station charges a different price for installing a new battery. Before you start your trip, you carefully print the Wikipedia page listing the locations and prices of every fueling station on the way. Given this information, how do you decide the best places to stop for fuel? More formally, suppose you are given two arrays  $D[1 \dots n]$  and  $C[1 \dots n]$ , where  $D[i]$  is the distance from the start of the highway to the  $i$ th station, and  $C[i]$  is the cost to replace your battery at the  $i$ th station. Assume that your trip starts and ends at fueling stations (so  $D[1] = 0$  and  $D[n]$  is the total length of your trip), and that your car starts with an empty battery (so you must install a new battery at station 1).

- (a) Describe, prove correct, and analyze a greedy algorithm to find the minimum number of refueling stops needed to complete your trip. Prove that your algorithm is correct.

**Answer:**

```
MinStops(D[1..n])
{
  s <- 1                      // number of stops so far
  last <- 1                   // current destination

  for i <- 2 to n-1
    if D[i+1] - last >= 100
      s <- s+1
      last <- D[i]
  return s
}
```

- $n$  is the number of stations &  $D[1..n]$ ,  $C[1..n]$  contains arbitrary inputs of size  $n$
- Let  $g_1$  be the first location chosen by the greedy algorithm after the trip begins from San Francisco to New York City
- Consider Set  $S$  as the optimal schedule of stops and  $s_1, s_2$  be the first two stops in  $S$
- We must have  $s_1 \leq g_1$ , as greedy algorithm refuels as late as possible.
- So replace  $s_1$  by  $g_1$  and let this be a part of  $S$ 's schedule
- $(s_2 - g_1) \leq (s_2 - s_1) \leq 100$
- As  $S'$  and  $S$  contains same number of stops,  $S'$  is optimal schedule
- The inductive hypothesis implies that greedy algorithm computes minimum number of stops for fueling from  $g_1$ . Thus, we conclude that greedy algorithm computes optimal schedule for whole trip

**Time complexity –  $O(n)$**

(b) Show that your greedy algorithm in part (a) does not minimize the total cost of travel.

In the above case (a), following the greedy algorithm only minimum stops were considered. The total cost of travel isn't computed or considered. In the below example;

Station	1	2	3	4	5	6
Distance	1	90	100	180	170	180
Cost (\$)	80	60	90	40	100	50

- If we go according to Greedy algorithm in part (a), function **MinStops()** computes minimum stations/stops only based on distance. In that case, we would stop at **station 3** as  $D[3] = 100$
- Cost of fueling at **station 3** is \$120
- If we consider cost and distance each time before stopping at station, then we would probably end-up fueling at **station 2** instead of **station 3**, since the distance  $D[i] < 100$  and  $C[i] < C[i+1]$
- If we stop at **station 2** then cost of fueling will be \$60 instead of 90\$. Cost of **station 2** is much more less than cost of fueling at **station 3**.
- Therefore, Greedy algorithm in part (a) doesn't minimize the cost of travel. It only focuses on distance and minimum stop computation

(c) Describe an efficient algorithm to compute the locations of the fuel stations you should stop at that does minimize the total cost of travel.

**Algorithm:**

function Travel( $D[1..n], C[1..n]$ )

for ( $i \leftarrow 1; i < n; i++$ )

```

    A[i] <- D[i]/ C[i]           //Calculate distance to cost ratio;
    last <- 1                   // current destination
    A[last] <- A[i]             // this will store the maximum ratio

```

for( $i=1; i < n; i++$ )

```

    if( $D[i]-last \geq 100$ )
        Max = FindMaximum(A,last,i);
        last <- Max

```

return station[last] // return minimum cost station

**Time complexity – two for loops running –  $O(n^2)$**