CS 430 - ASSIGNMENT - 1

Consider following example

1.a)

JOB ID	START TIME	END TIME	DURATION
a	1	3	2
b	2	3	1
c	3	6	3
d	5	7	2
e	6	8	2

Sorting by minimum duration, we will first schedule job b. Removing conflicting jobs, the next minimum job is d. There are no other conflicting jobs so the jobs executed will be

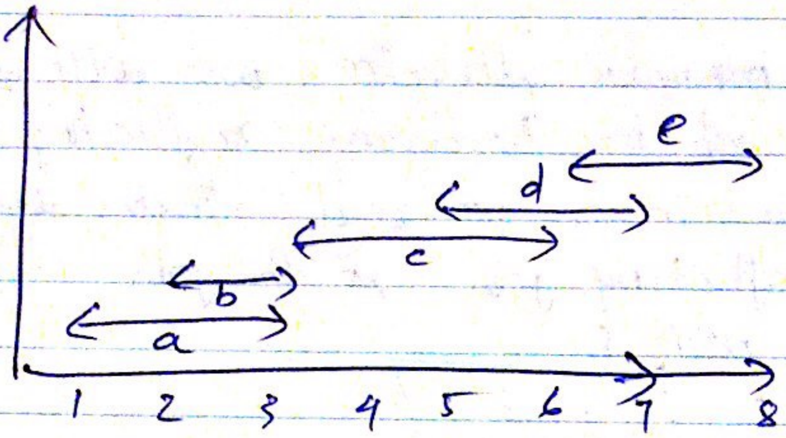
$\boxed{\{b, d\}}$

Whereas, if duration is not considered, we can schedule $\boxed{\{a, c, e\}}$

∴ scheduling based on duration will not give best result.

1.b) Job $\underline{b}$ overlaps with Job $\underline{a}$ ~~and~~
Job $\underline{d}$ overlaps with Job $\underline{c}$
Job $\underline{e}$ overlaps with Job $\underline{d}$

Diagramatically



~~Job e is the only job with no overlapping~~

~~Both~~ With no overlapping only jobs a and c can be done.
But the optimal solution is $\{a, c, e\}$

This shows that scheduling jobs ~~based~~ with lowest conflict doesn't provide optimal solution.

② Internally, sort all the jobs by using start time. For the first job we will allocate one processor. Add the processor to a processor list along with the time when processor will be free [end time of the job]. For the remaining jobs, if the start time is greater than the free time for any processor in the processor list, then we will allocate the job to that processor else we will add a new processor to the processor list.

Algorithm:

jobs ← list of jobs with start and end time, sorted based on start time

processor list ← list of processors with free time for each.

```
ScheduleJobs(jobs)
{
processorlist Add (new processor (jobs[0]
                                  endtime ))
for (i =1 ; i< jobs length ; i++)
  {
   int j ;
```

```
for (j=0 ; j< processorlist.length ;j++)
{
  jobs[i]
  if (starttime > processorlist[j].freetime)
   {
    processorlist [j]. freetime = jobs[i].endtime;
     break;
   }
  if (j == processorlist.length)
    processorlist.add (new processor (jobs [i]endtime))
}

 return processorlist.length

}
```

③ Position the first distribution box at l_1. The next distribution box should be placed on location l_k such that $l_k - l_1 > d$.
Placing the consequtive boxes in a similar fashion we will have the minimum number of boxes.

Algorithm:

l [ ] ← array of location
d ← distance covered by each box.
n ← no. of houses.

```
int [ ] findpositions (int[ ] l, int d, int n)
  {
      int [ ] positions
      position [0] = l [0]
      int prev_position = 0 ;
      for (i=1 ; i<n ; i++)
         {
            if (l [i] - positions [prev_position] > d)
                positions [prev_position ++] = l [i]
         }
  }
```

④ The idea behind Huffman coding is to find a way to compress the storage of data using variable length codes.

Our standard model of storing data uses fixed length codes. Considering varying frequency of characters, this method is inefficient.
For Eg: letter "e" occurs 15 times more frequently than letter "f".

In this case a lesser bit code for letter "e" than letter "f" is efficient since it shortens our overall message length.

Huffman coding provides a way to take advantage of varying character frequencies in a particular file. On an average, Huffman coding can shrink a file from 10% to 30% more than standard fixed length algorithm, depending on character distribution.

$\boxed{\text{More Skewed distribution} \xrightarrow{\text{gives}} \text{better Huffman Coding}}$

For 2^k characters, length of Huffman code $\rightarrow$ k

and c

In our case, the given data file contains sequence of 8-bit characters such that all 256 characters are equally common.

Maximum character frequency < 2 minimum character frequency

This gives you less skewed distribution of characters.

∴ Huffman coding is not efficient when compared to standard fixed length code.