CS345 : Algorithms II

Ankur Kumar(14109) , Tushar Vatsal(14766)

Theoretical Assignment 4

1 A Job Scheduling Problem

Part 1

To prove: There is an optimal solution J in which the jobs in J are scheduled in the increasing order of their deadlines.
Solution: Consider an optimal solution J in which jobs are not scheduled in the increasing order of their deadlines. Therefore $\exists$ two consecutive jobs in J, say job i and j, such that $d_i > d_j$ but i is scheduled before job j. Also suppose job i was scheduled at time T.
Now consider a sequence of jobs J' which has the same sequence of jobs as J except that job j comes before job i now. After doing so, all the jobs before job j in J' are still schedulable since they have same start time and finish time when scheduled as in J.
Job j was scheduled at time $T + t_i$ in J. $\Rightarrow T + t_i + t_j \leq d_j$
Job j will be scheduled at time T in J' and will finish at time $T + t_j$. From above $T + t_j \leq d_j \Rightarrow$ Job j is schedulable in J'.
Job i will be scheduled at time $T + t_j$ in J' and will finish at $T + t_j + t_i$. From above $T + t_i + t_j \leq d_j$. Since $d_i > d_j \Rightarrow T + t_j + t_i < d_i \Rightarrow$ Job i is also schedulable in J'
All the jobs after job i in J' are schedulable since they have the same start time and finish time when scheduled as in J.
From above, all the jobs in J' are schedulable. Therefore J' is also optimal as number of schedulable jobs is same as that of J which is optimal.
Thus we can carry out this swap operation numerous times in J resulting in a sequence which has all the jobs scheduled in increasing order of deadlines. Also this final sequence will also be optimal as we have proved above that the sequence of jobs obtained after applying one swap operation on an optimal job is optimal.

Part 2

Problem Statement: To give an algorithm for finding an optimal solution. Also the running time should be polynomial in the number of jobs n, and the maximum deadline $D = max_i\ d_i$
Algorithm: Algorithm for an optimal solution having jobs schedulable in increasing order of deadlines.
Step 1: Sort the jobs in increasing order of deadline. Let us call this sorted sequence of jobs to be J.
Note: After sorting the jobs, we are renaming the jobs, i.e. when we shall say job i from now onwards it refers to *ith* job in the sorted sequence J. It may not be the same job i before sorting the set of jobs. Therefore d_i and t_i will now be deadline and processing time for *ith* job in sorted sequence J.
Step 2: Coming up with a notation for recursive formulation.
Sched(i, T): Length of schedulable subset of maximum size(available to scheduled starting at time T) among set of jobs $i, i + 1, \ldots., n - 1, n$
Recursive Definition:
Base Case: Sched(n, T) = 1 if $T + t_n \leq d_n$, 0 otherwise

if $T + t_i > d_i$ **then**
  Sched(i, T) = Sched($i + 1$, T)
else
  Sched(i, T) = max(Sched($i + 1$, T), 1+Sched($i + 1$, $T + t_i$))
end if

Proof of Above Optimal Substructure Property

As said earlier, we will be looking at optimal job having all the jobs scheduled in increasing order of deadline.
Case 1: if $T + t_i > d_i$ **then** Sched(i, T) is Sched($i + 1$, T)
In this case minimum time by which job i will end, will be $T + t_i$(by scheduling job i first among all the jobs i, i+1 ... upto n). But since $T + t_i > d_i$, job i is not schedulable. Hence the optimal subset for jobs i to n can not contain job i. Therefore the optimal solution for current instance will come from set of jobs i+1 to n. Hence optimal solution for smaller instance(jobs $i + 1$ to n) with same start time T is the solution for optimal solution of current instance.

Case 2: if $T + t_i <= d_i$ **then** Sched(i, T) is either Sched($i + 1$, T) or 1 + Sched($i + 1$, $T + t_i$)
In this case job i can be scheduled. So job i can be in optimal solution for jobs i to n.

Suppose job i is not there in the optimal solution for jobs i to n. Therefore optimal solution for current instance will be coming from set of jobs i+1 to n. Hence optimal solution for smaller instance(jobs $i+1$ to n) with same start time T is the solution for optimal solution of current instance, i.e. Sched($i,\ T$) will be Sched($i+1,\ T$) in this case.
Now suppose job i is in optimal solution. Since the jobs in optimal solution are scheduled in increasing order of their deadlines, job i will be scheduled first among jobs i to n(these are sorted in increasing order of deadline). Job i finishes at time $T + t_i$. After that the remaining jobs for optimal solution will come from set of jobs i+1 to n. Hence job i with optimal solution for smaller instance(jobs $i+1$ to n) with same start time $T + t_i$ will be the solution for optimal solution of current instance, i.e. Sched($i,\ T$) is 1 + Sched($i+1,\ T+t_i$).
Hence for case 2, Sched($i,\ T$) is either Sched($i+1,\ T$) or 1 + Sched($i+1,\ T+t_i$)

Iterative Approach for Sched[$i,\ T$]

We will make a matrix, named Table, of size n x (D-s+1) (row i for job i and column j for time j, time ranges from s to D) where, n = number of jobs, s = given start time, D = $max_i\ d_i$ and Table[$i,\ T$] = Sched[$i,\ T$]
The first while loop below computes the whole Table matrix. We get the size of our optimal solution as Table[1, s]. We create an array Schedulable[] of this size to store the optimal subset of jobs. This is computed in the last while loop by tracing back how Table[1, s] will be computed recursively if we have value for each recursive term that follows.

Algorithm 1: Algorithm for an optimal solution

```
Result: Array Schedulabe[] contains an optimal schedulable subset
Sort set of jobs in increasing order of deadlines;
i ← n-1;
Fill row n in Table according to base case described in recursive definition;
while i > 0 do
    T ← s;
    while T ≤ D do
        if T + t_i > d_i then
            Table[i, T] = Table[i + 1, T];
        else
            Table[i, T] = max(Table[i + 1, T], 1+Table[i + 1, T + t_i]);
        end
        T ← T + 1;
    end
    i ← i − 1;
end
count ← Table[1, s];
Create an array Schedulable[] of size count;
i ← 1;
j ← 0;
T ← s;
while j < n do
    if T + t_i > d_i then
        i ← i+1;
    else
        if Table[i + 1, T] > 1 + Table[i + 1, T + t_i] then
            i ← i+1;
        else
            Schedulable[j] ← i;
            i ← i+1;
            T ← T + t_i;
        end
    end
    j ← j+1;
end
```

Time Complexity

Base case, i.e. to fill nth rows takes O(n) time.
The first while loop has n*(D-s+1) iterations(including the ineer while loop), each iteration for O(1) time. ⇒ **O(nD)** time
The last loop has n iteratons, each in O(1) time. ⇒ O(n) time
Total time = O(n) + Time to sort n jobs + O(nD) + O(n) = **Time to sort n jobs + O(nD)**
Now if D ≤ n, n jobs can be sorted using counting sort in O(n) time. ⇒ Total time = O(n) + O(nD) = **O(nD)**
If D > n, then n jobs can be sorted in nlog n time. ⇒ Total time = O(n$logn$) + O(nD) = **O(nD)**
Thus, **Time Complexity = O(nD)**

2 Buying and selling shares

2.1 Notations

→ Days are numbered $i = 1, 2, ..., n$
→ There is a price $p(i)$ per share of the stock on i^{th} day.
→ Given a variable k, a $k - shot\ strategy$ is a collection of m pair of days $(b_1, s_1), ..., (b_m, s_m)$ where $0 \le m \le k$ and $1 \le b_1 < s_1 < b_2 < s_2 < .... < b_m < s_m \le n$
Return of $k - shot\ strategy$

$$Profit(k) = 1000 \sum_{i=1}^{m} (p(s_i) - p(b_i))$$

Aim : To maximize this profit for a given k
→ $Rec\_profit(i, l)$ =Maximum profit using $l - shot\ strategy$ from i^{th} day to n^{th}(last) day.
Aim : To find $Rec\_profit(1, k)$

2.2 Recursive Formulation of $Rec\_profit(i, l)$

$$Rec\_profit(i, l) = max\{Rec\_profit(i + 1, l), max_{i<j\le n}(Rec\_profit(j + 1, l - 1) + p(j) - p(i))\}$$

Base Case:
$Rec\_profit(n, l) = 0\ \forall\ l \in \{0, 1, 2, ..., k\}$
$Rec\_profit(n + 1, l) = 0\ \forall\ l \in \{0, 1, 2, ..., k\}$
$Rec\_profit(i, 0) = 0\ \forall\ i \in \{1, 2, 3, ..., n\}$

2.3 Proof of Recursive Formulation

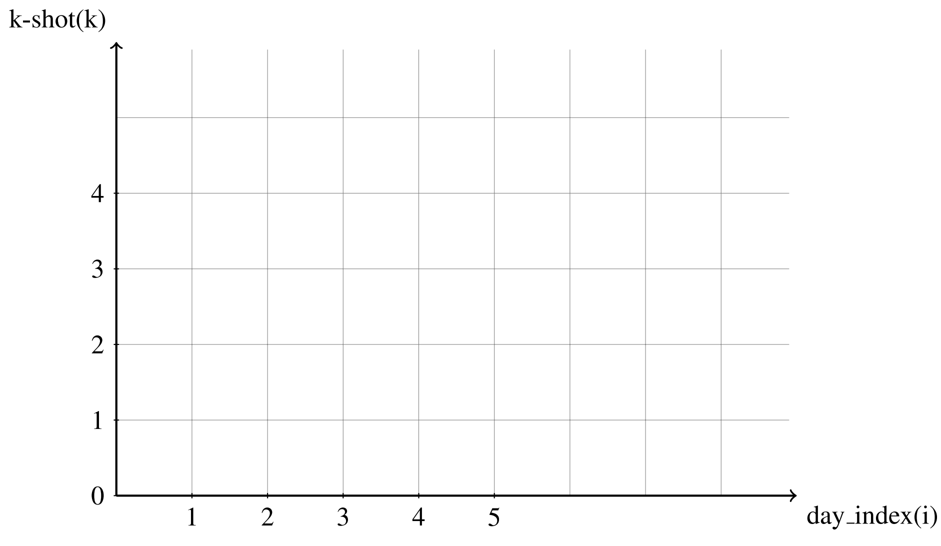
$$Rec\_profit(i, l) = max\{Rec\_profit(i + 1, l), max_{i<j\le n}(Rec\_profit(j + 1, l - 1) + p(j) - p(i))\}$$

$Rec\_profit(i, l)$ = Maximum profit from i^{th} day to n^{th} day using $l - shot\ strategy$.
There are two possible cases :
Case1 : Suppose no transaction occurs at i^{th} day then maximum profit from i^{th} day to n^{th} day using $l - shot\ strategy$ is same as maximum profit from $i + 1^{th}$ day to n^{th} day using $l - shot\ strategy$. This means $Rec\_profit(i, l) = Rec\_profit(i + 1, l)$
Case2 : Suppose we buy share on i^{th} day then I have to sell this share on j^{th} day for some j such that $i < j \le n$. For selling this share bought on i^{th} day, we have $n - i$ possible days. Suppose I sell this on some day $j > i$, then maximum possible profit is sum of $(p(j) - p(i))$(profit due to this transaction) and maximum possible profit from $j + 1^{th}$ day to n^{th} day using $l - 1\ shot\ strategy$. This means maximum possible profit if we sell this on j^{th} day for some $j > i$ = $Rec\_profit(j + 1, l - 1) + p(j) - p(i)$. Thus maximum possible profit if we sell this share brought on i^{th} day on any day $j > i = max_{i<j\le n}(Rec\_profit(j + 1, l - 1) + p(j) - p(i))$
Since we have to maximize the term $Rec\_profit(i, l)$ either by doing any transaction at i^{th} day or not doing it, I will take

$$Rec\_profit(i, l) = max\{Rec\_profit(i + 1, l), max_{i<j\le n}(Rec\_profit(j + 1, l - 1) + p(j) - p(i))\}$$

2.4 Implementation using Dynamic Programming

Build a table of $k + 1$ rows and $n + 1$ columns where k and n have their usual meanings



Let us start calling $Rec\_profit(i,l) = T(i,l)$ from now just to simplify our life.

$$T(i,l) = max\{T(i+1,l), max_{i<j\leq n}(T(j+1,l-1) + p(j) - p(i))\}$$

To calculate any entry $T(i,l)$, we should know $T(i+1,l)$ and all the entries to the right of the block $T(i+1,l-1)$ in the grid shown above.
First we will assign all the entries of n^{th} and $n+1^{th}$ column 0 and all the entries of 0^{th} row 0 according to the base case stated above. And then we will calculate the entries column-wise starting from second-last column from bottom to top. This will ensure while calculating any entry $T(i,l)$ we will know beforehand all other entries used for calculating it.

Space Complexity : $O(n(k+1)) = O(nk+n) = O(nk)$
Time Complexity :
Assigning last column : $O(k+1) = O(k)$
Assigning last row : $O(n)$
As we know that

$$T(i,l) = max\{T(i+1,l), max_{i<j\leq n}(T(j+1,l-1) + p(j) - p(i))\}$$

Thus time required to calculate any block $T(i,l) = c(1) + c(n-i) = c(n-i+1)$, where c is a constant.
Thus total time to calculate all the remaining entries of the table :

$$\sum_{i=1}^{n-1}\sum_{l=1}^{k} c(n-i+1) = ck\sum_{i=1}^{n-1}(n-i+1) = O(n^2k)$$

Thus the entire table can be computed in $O(n^2k) + O(n) + O(k) = O(n^2k)$ time. This is a polynomial time algorithm in n and k.

2.5 Towards $O(nk)$ time complexity

However we can improve this to $O(nk)$ time complexity. Take a look over the term $T(i,l)$=

$$max\{T(i+1,l), max_{i<j\leq n}(T(j+1,l-1) + p(j) - p(i))\}$$

If this term can be calculated in $O(1)$ time then we are done.
Let's say $max_{i<j\leq n}(T(j+1,l-1) + p(j) - p(i)) = P(i,l)$
$\Rightarrow P(i,l) = (max_{i<j\leq n}(T(j+1,l-1) + p(j)) - p(i)$
Say $max_{i<j\leq n}(T(j+1,l-1) + p(j) = P'(i,l)$
Now $P(i,l) = P'(i,l) - p(i)$
To find $P(i,l)$, we need $P'(i,l)$ and $p(i)$
While calculating i^{th} column of the table we will keep an array M of size k such that $M[l] = P'(i,l)$
Initalize all the elements of $M = 0$
After initializing n^{th} column and 0^{th} row of the grid start calculating the grid T from $n-1^{th}$ to the last column according to these rule :
$T(i,l) = max\{T(i+1,l), M[l] - p(i)\}$. After calculating $T(i,l)$ update $M[l] = max\{M[l], T(i+1,l-1) + p(i)\}$
Thus we are able to calculate every entry in $O(1)$ time and thus total time complexity goes to $O(nk)$