Assignment - 6
CS 5084

1. 

a) - In this case, graph is a path and we can use a Greedy Algorithm to find solution.
- We can start with an empty set, the repeatedly add heaviest remaining node, to the set, and remove its neighbours (nodes next to it) from consideration.
- Continue until no nodes remain. However, we may not always find the set of maximum total weight.
- As an example, if we consider a path with nodes of weight 2, 3, 2, then as per algorithm we'll have to pick middle node i.e. the one with weight 3, but maximum weight idependent set consists of first & third nodes (i.e. 4 → 2+2)

b) - In this case, we are given an algorithm that works by dividing the nodes into two sets, one consists of odd-numbered nodes and other cosisting of all even-numbered nodes. Both of these sets are independent sets, meaning that no two nodes in either set are next to each other
- The algorithm then determines which of these two sets has a greater total weight and returns that set as a solution.
- But for example, consider a path with nodes of weight 3, 1, 2 and 3. The algorithm would pick the first and third nodes (total weight 5) from odd numbered set, but max. wt. independent set consist of the first & fourth nodes (total 6)

c) - To find solution, we start looking at the first 2 nodes of path. We can either include the first node in our independent set or not. If we include it we can't include the second node, so total weight of independent set is the weight of the first node.
- If we don't include the first node, we move on to the second node and repeat the process.
- For subsequent node, we look at whether its better to include it in the independent set or not.
- If we include it we can't include the previous node, so total weight of independent set is :-
[the weight of current node] +
[the weight of the node 2 steps back]
If we don't include it, the total weight of the independent set is the same as the previous node.
- We keep track of the maximum weight independent set we've found so far, and at the end, we can trace back through the computations of find the nodes that are included in set.
- Since we spend constant time from start to end in 'n' steps the algorithm runs in O(n) time.

2. a) Algorithm has some drawbacks like it may consider high stress job too early and better jobs after that.
Lets us consider following example.

	W1	W2	W3
h	1	5	10
l	2	2	2

here, algorithmically it would take high stress job in week 2, but unique optimal solution would take a low stress job in week1, then again nothing in week 2 but in the end high stress job in week 3

2. b) There are 2 ways to solve this problem, One way is to use recurrence relation where we define optimal solution as max. revenue in first week.
- We can compute all optimal values by invoking a recurrence relation from 1 to n (no. of i/p's) and the actual sequence of jobs can be reconstructed by tracking back through the set of optimal values.
- Another way to solve this problem is to define subproblems Lmax & Hmax which are max. revenue achievable first 'x' weeks, given that team chooses lowstress / high stress job in x^{th} week resp.
- The we can compute these values by invoking recurrence relation from 1 to n & actual sequence of jobs can be constructed by comparing the final values of Lmax & Hmax.

3. a) given algorithm,

While there is an edge out of node w
  choose the edge (w, v_j)
    for which j is as small as possible
  set $w = v_j$
  Increase L by 1
end while
Return L as the length of longest path.

The graph nodes v_1, v_2, v_3, v_4, v_5 with edges
$(v_1\ v_2)$ $(v_1\ v_3)$ $(v_2\ v_5)$ $(v_3\ v_4)$ $(v_4\ v_5)$
The algorithm will return 2 corresponding to the path of edges (v_1, v_2) $(v_2\ v_5)$, while optimum is 3 using path (v_1, v_3) $(v_3\ v_4)$ $(v_4\ v_5)$

3. b) We can use Dynamic Programming here

```
Longest path(n)
 array A[1...n]
 A[1] = 0
 for (i=2...n)
 {
   M = -∞
   for all edges(i,j)
      if A[j] ≠ -∞
         if A < A[j]+1 then
            A = A[j]+1
         endif
      endif
   end for
   A[i] = A }
Return A[n] as l(longest path)
```

Running time
$\Rightarrow O(n^2)$

4. a) - lets consider an example where operating cost in NY is low for first and third months, but high in second month.
Meanwhile operating cost in SF is high for First & third months but low for second month.
- The optimal plan in this case would be [NY,NY,NY], since switching cities incurs a fixed cost. However, the algorithm given in the question would switch to SF in second month, results total higher cost.

b) consider following ex,
let Moving cost = 10

$\{N_1\ N_2\ N_3\ N_4\} = \{1, 100, 1, 100\}$

$\{S_1\ S_2\ S_3\ S_4\} = \{100, 1, 100, 1\}$

here in one month its cheaper to operate in NY & in next month its cheaper to operate in SF
- If we start with NY 1st month then SF 2nd month.
to return to NY for the 3rd month, it incurs a moving cost.
- Similarly to operate in SF for the 4th month the business needs to move back to SF & incur another moving cost M.
∴ Any optimal plan must include at least 3 moves, as moving fewer times would result in higher operation cost.
In plan [NY, SF, NY, SF] (1+1+1+1+20=34)
2 times moving cost

4. c) The following Algorithm works by observing that the optimal plan will either end in NY / SF. If it ends in NY then optimal plan for 1st n-1 months will either end in NY/SF with a move from SF to NY & similarly if it ends in SF.

So we can recursively compute the cost of optimal plan for each city at the end of the month.

- let optimal soln N = 0
  — " — S = 0

We compute optimal plan ending in NY & SF by following steps:

1) cost of running the business in the current city (N_i or S_i)
+ cost of the optimal plan for the first i-1 months ending in the same city
+ cost of moving

2) cost of running business in the other city (S_i or N_i)
+ Cost of optimal plan for first i-1 ending in the other city
+ cost of moving from other city to current city

Finally, we'll return min. of cost of optimal plan ending in NY and the cost of the optimal plan ending in SF

The algorithm runs $O(n)$ time, as each step takes const. time.

17.a) -The algorithm is flawed, it looks at each day's price and checks if it is higher than the previous day's price; if it is, the algorithm adds 1 to the length of the rising trend.
- But, algo. does not find the longest rising trend. As an example, lets consider sequence of prices 1, 4, 2, 3. The longest rising trend in this sequence is 1, 2, 3 but the algorithm above would return a length of 2 w.r.t. subsequence 1, 4

b) We can use dynamic programming.

- Let's say we have a sequence of no. 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5.
- we want to find longest rising trend in seq.
- By definition given in the question, a rising trend is a subsequence where each no. is strictly greater than the prev. no. It can start at any position in the sequence, but it must start with the first no.

- So in this case we'll start with no. 3, longest rising trend that starts with 3 is 3, 4, 5, 9. this is the subsequence of length 4.
- lets move on to second no, i.e. 1 but there is no rising trend that starts with 1 so we skip.
- lets move on to third no. (4). the longest rising trend starts with 4 is 4, 5, 9 with subsequence of length 3.

- We continue this way, looking at each no. in turn and finding the longest rising trend that starts with that no.
- At the end, we take max. length of all rising trends we found, in this case it is 3, 4, 5, 9 with length 4.
- Thus we have running time of $O(n^2)$ since we need to compute the values for each price in sequence.

28. a) - If J is optimal subset then by definition all jobs in J can be scheduled to meet their deadline.
- lets consider problem of scheduling to minimize max. lateness from class but considering all jobs in J only.
- by defn of J minimum lateness is 0
⇒ all jobs are on time.
- But the greedy algorithm of scheduling jobs in order of their deadline, which is optimal for minimizing max. lateness.
- Hence, proved ordering jobs in J by the deadline generates feasible schedule for this set of jobs.

28. b) - Similar to the subset sum problem with respective subproblems, our initial approach involves considering subproblems utilizing a subset of jobs $\{1 \dots m\}$ ordered by increasing deadline, where $d_1 \leq d_2 \leq d_3 \dots \leq d_n = D$.
- We need to examine 2 cases; either job n is accepted or it is not. If it is not, the problem is reduced to the subproblem using only the first $n-1$ times.
- If it is accepted, we define subproblems such that all jobs must be completed by $D - t_n$ in order for machine to finish job 'n' by deddlines.
- let optimal sol^n (d, m) be max. subset of requests in the set $\{1 \dots m\}$ can be completed by deadline d.

- We can buildup values for the subproblems using the following function. & with running time $O(n^2D)$

```
$ (n, D)
  Array M[0...n][0...D]
  Array S[0....n][0...D]
  for d = 0 to D
      M[0,d] = 0
      S[0,d] = {}
  End for
  For   m = 1 ... n
    For d = 0 to D
      If M[m-1, d] > M[m-1, d-tm] + vm
          M[m,d] = M[m-1,d]
          S[m,d] = S[m-1,d]
      Else
          M[m,d] = M[m-1, d-tm] + vm
          S[m,d] = S[m-1, d-tm] U {m}
      End for
    End for
  End for
Return M[n,D], S[n,D]
```