Problem 1. *(10 pts) Develop an efficient algorithm to find the length of the longest path in a directed graph* $G = (V, E)$.

Hint:

1. *First, observe that if the graph G has a directed cycle, then the maximum path length is infinite, since you can construct a path that keeps going around this cycle forever. Therefore, first devise an algorithm to test whether G contains a directed cycle (and output* ∞ *if it does). Prove the correctness of your algorithm for this step.*
2. *If G contains no cycles, it must be a DAG. Provide an efficient algorithm to output the maximum path length in a DAG. Prove the correctness of your algorithm for this step.*
3. *Analyze the overall running time of your algorithm.*

1.) To observe whether the Graph G has a direct cycle, check if G has a topolgical ordering. The inductive proof from textbook page 102. If we can, then we know G is a toplogical, return Turn. otherwise return False.

⇒ initialize graph G={}, list=[]
initialize maximum path length L=0.
pick any node v in total nodes N.
while i <= len(n) // find whethe is a cycle.
loop complexity/ big O(n)
if node v != N[i]
if n not in list;
list.append(v).
delet v from N
else:
it is a cycle.

else; return No.
End if.
End while

correctness: a direct cycle cannot have a toplogical ordering, because a directed cycle has no root nodes. Since the sudo code terminate once there are no root nodes, it will eventually stop if the input contains a directed cycle. Conversely, when the input doesn't contain a directed cycle, it produces a toplogical ordering of the nodes.

Time complexity: To bound the running time of this algorithm, we note

Problem 4:
let
Ji -> the first job on the list.
Ti -> the time of first job on the list.
Wi -> the weight of first job on the list.
S(i->j) -> Ordering from jobs from i to j. (Sgima Wi* Ci (S))

Q1.
Counterexample:

Assume: J1 and J2 have same duration time. J1 is W1 = 5, T1 = 100;
and J2 is W2 = 4, T2 = 2; In this case, W1 > W2; then:

CASE 1:
J1 goes first, J2 goes after, we get:
C1 = W1 * T1
C2 = W2 * (T1 + T2)
S1 = W1 * C1 + W2 * C2
   = W1 * T1 + W2 * (T1 + T2)
   = 5 * 100 + 4 * (100 + 2)
   = 908

CASE 2:
J2 goes first, then J1, we get:
C1 = W2 * T2
C2 = W1 * (T1 + T2)
S2 = W1 * C1 + W2 * C2
   = W2 * T2 + W1 * (T1 + T2)
   = 4 * 2 + 5 * (100 + 2)
   = 518
Here W1 > W2, however, S1 > S2

W1 / T1 = 5/100 = .05
W2 / T2 = 4/2 = 2
Therefore, case 2 has less time when W1 bigger and goes after W2.

COUNTEREXAMPLE FOR Q2:

We assume that W1 = 2, W2 = 10, T1 = 2, T2 = 3; then,
Case1, J1 goes first, J2 goes after, we get:
C1 = W1 * T1
C2 = W2 * (T1 + T2)
S1 = W1 * C1 + W2 * C2
   = W1 * T1 + W2 * (T1 + T2)
   = 2 * 1 + 10 * (2 + 3)
   = 52

Case2: J2 goes first, then J1, we get:

C1 = W2 * T2

C2 = W1 ∗ (T1 + T2)
S2 = W1 ∗ C1 + W2 ∗ C2
   = W2 ∗ T2 + W1 ∗ (T1 + T2)
   = 10 ∗ 3 + 2 ∗ (2 + 3)
   = 40

Here T1 < T2 and T1 goes first; however, S1 > S2

W1 / T1 = 10/2 = 5
W2 / T2 = 5/2 = 2.5

Therefore, T1 with shorter time goes first has a worse case than T2 goes first

---

Q3:
We have proof that the optimal scheduling for this job neither the heaviest-weight first, nor the shortest-job first. Therefore,
Base on the observation, when W1/T1 = W2/T2 will be an optimal scheduling.
Proof by exchange argument:

Let assume that W1 = 4, W2 = 6, T1 = 2, T2 = 3; and they did not think W1 / T1 = W2 / T2
will have a optimality scheduling.
J1 goes first, J2 goes after, we get:
C1 = W1 ∗ T1
C2 = W2 ∗ (T1 + T2)
S1 = W1 ∗ C1 + W2 ∗ C2
   = W1 ∗ T1 + W2 ∗ (T1 + T2)
   = 4 ∗ 2 + 6 ∗ (2 + 3)
   = 38

After they swap order,
J2 goes first, then J1, we get:

C1 = W2 ∗ T2
C2 = W1 ∗ (T1 + T2)
S2 = W1 ∗ C1 + W2 ∗ C2
   = W2 ∗ T2 + W1 ∗ (T1 + T2)
   = 6 ∗ 3 + 4 ∗ (2 + 3)
   = 38

Hence S1 = S2, they get S1 = S2.

than identifying a node v with no incoming edges, use loop to check all nodes., can be done in O(n) time. since the algorithm at most check n+1 (node) if it is a circle. so the time complexity is O(n)

⇒ 2. initialize s is all nodes with no coing edges.

If graph no cycle,

let dist[] = {0, 0, 0, ...} // Breadth First search O(n+m)

while L is not empty

w = L. dequeue

For each node(V, u)

dist[v] = dist[u]+1;

endFor

endwhile.

return max(dist)

proof: Assume it has a longer path than this, it will has a node that this algorithm didn't detect. However, this algorithm use Breadth First search detected all the possible nodes, Thus. it is the longest path.

Problem 2. *(10 pts) Given a sequence q of n numbers, give an $O(n^2)$ algorithm to find the longest monotone increasing subsequence in q. As always, prove the correctness of your algorithm and analyze its running time.*

Hint: Find a graph representation for this problem and use a graph algorithm (from class, lab or homework) to solve it.

Solution: graph Algorithm.

Create a graph, G, with node representing each number. ↵ if there are multiple numbers that connect the same number, create nodes for each distinct number

Now create two types of edges: connection edges and edges going forward. start by going throught the first node, let say N_1 creating a directed edge from the node of first number in the Graph N_1 to other number in Graph represented by it respective node N_2, and an edge in the opposite direction.. Additionlly, create directed edges to the pairs that represent the same number at the next later node; This creates at most 2n more forward edges (at most one per node.

step2. we use the algorithm from question 1. whether it is a DAG. In other words, this graph is an DAG.

pseudo code:

```
Initialize graph G={}, list=[]
initialize maximum path length L=0
pick any node v in total nodes N.
while i <= len(n)                    // find whethe is a cycle.
                                     loop complexity: big O(n)
   if node v != N[i]
      if n not in list:
         list.append(v)
         delet v from N
      else:
         it is a cycle.
   else: return No
   End if
End while
```

Initialize S is all nodes with no coing edges.

If graph no cycle.

```
let dist[] = {0, 0, 0, ...}   // Breadth First search
                                 O(n+m)
while L is not empty
    w = L. dequeue
    For each node(V, u)
        dist[v] = dist[u]+1;
    endFor
endwhile.
return max(dist)
```

Correctness: a direct cycle cannot have a toplogical ordering, because a directed cycle has no root nodes. Since the sudo code terminate once there are no root nodes, it will eventually stop if the input contains a directed cycle. Conversely, when the input doesn't contain a directed cycle, it produces a toplogical ordering of the nodes.

proof: Assume it has a longer path than this, it will has a node that this algorithm didn't detect. However, this algorithm use Breadth First search detected all the possible nodes. Thus, it is the longest path

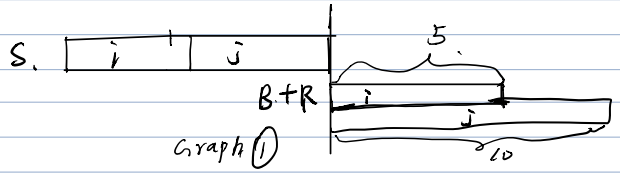
3. Your friend is working as a camp counselor, and he is in charge of organizing activities for a set of junior-high-school-age campers. One of his plans is the following mini-triathalon exercise: each contestant must swim 20 laps of a pool, then bike 10 miles, then run 3 miles. The plan is to send the contestants out in a staggered fashion, via the following rule: the contestants must use the pool one at a time. In other words, first one contestant swims the 20 laps, gets out, and starts biking. As soon as this first person is out of the pool, a second contestant begins swimming the 20 laps; as soon as he or she is out and starts biking, a third contestant begins swimming . . . and so on.)

Each contestant has a projected *swimming time* (the expected time it will take him or her to complete the 20 laps), a projected *biking time* (the expected time it will take him or her to complete the 10 miles of bicycling), and a projected *running time* (the time it will take him or her to complete the 3 miles of running). Your friend wants to decide on a *schedule* for the triathalon: an order in which to sequence the starts of the contestants. Let's say that the *completion time* of a schedule is the earliest time at which all contestants will be finished with all three legs of the triathalon, assuming they each spend exactly their projected swimming, biking, and running times on the three parts. (Again, note that participants can bike and run simultaneously, but at most one person can be in the pool at any time.) What's the best order for sending people out, if one wants the whole competition to be over as early as possible? More precisely, give an efficient algorithm that produces a schedule whose completion time is as small as possible.

Solution: let R → running. , B → biking. S → swimming.

R_i, B_i, S_i are participant i Running, biking and swimming,

In this competition, swimming is for one person a time, so we will let swim goes first for every participants. For the whole competition to be finished as early as possible, we choice the longest R + B swim first. then the second longest R+B swim second .... and so on. After each one of them finish swim, they are going to R or B immidatelly, until the competition to be over. this is the optimal algorithm we use.



Graph ①

proof:

Use inverse argument to proof this algorithm, we assume that

i and j is two participants and $R_i + B_i < R_j + B_j$ (graph①), we say that they didnot usethe order for optimal solution. in the competition that inclusin i and j participants. After they swap order and i goes to swim first. then j follow i, j will finish his competition longer than previous schedule he used to. as $R_i + B_i < R_j + B_j$

Problem 4. *(20 pts)[programming assignment]*

Consider a different version of the job scheduling problem that minimizes the weighted sum of job completion times. In this version, each of n jobs has a weight w_i and a duration t_i, and all must be scheduled in serial order, starting at time zero (with no slack). A schedule S is therefore an ordering of the jobs, and the completion time of job i in S, $C_i(S)$, is defined to be the sum of the durations t_k of all jobs (including i) that precede i in S. Find the schedule $S*$ that minimizes over all S: $\sum_{i=1}^{n} w_i C_i(S)$.

1. *Provide a counterexample to the optimality of ordering by heaviest-weight first. [1 pt]*
2. *Provide a counterexample to the optimality of ordering by shortest-job first. [1 pt]*
3. *Design and analyze an efficient algorithm for computing an optimal schedule.∗ Hint: use an exchange argument, similar to the one used for the minimizing lateness problem for the proof of optimality.*

(∗) For this problem you need to submit your code with comments explaining the steps and with regard to its running time. You also need to provide answers to parts (1) and (2) as well as the proof of correctness for part (3) on your pdf with the answers to the other questions.

T (

2 = T 3

10 5)

0 0 - 10

$\overline{t_1}'$ =

6 $t_2 = 3.$

$\overline{2}$ $= 18 + 4(5)$

w_1 $8 = 3s$

t