Parth Ray

Lab 4

1. Optimizing Cell Tower Placement:
    a. Place a tower 4 miles after the first house. Then keep placing towers 4 miles after a house if the house is not already within range of a tower.

```
OptimizingCellTower(M[1…n])
    LOC[1] = M[1] + 4;
    k = 2;
    For i from 2 to n:
        If M[i] is not within the bounds of LOC[k - 1] then
            LOC[k] = M[i] + 4
            k++
    Return LOC;
```

    b. Proof of correctness: Placing a cell tower 4 miles after a house with no service produces the minimum number of cell towers with full coverage.

        Setup: Assume LOC*[$k_{1...}k_n$] is the optimal solution to the tower placement problem with n number of towers being placed for coverage of each house and let LOC[$k_1…k_m$] be the set of tower locations given by our greedy algorithm.

    Proposition: Let P(k) be the proposition LOC*[k] ≤ LOC[k]. Want to show that this is true for all k≥1 up to n.

    Base Case: Show P(1) is true. LOC*[1] ≤ LOC[1].

    Proof of Base Case: LOC*[1] ≤ LOC[1] is true since the LOC[1] is placed at the max range away from M[1], thus LOC*[1] must be less than or equal to LOC[1] in order to provide service to the first house.

    Inductive Case: Show that P(k) is true → P(k + 1) is true for any 1≤k<m. Thus we need to show that LOC*[k] ≤ LOC[k] → LOC*[k + 1] ≤ LOC[k + 1].

    Proof of Inductive Case:

    LOC[k + 1] is the tower location that is 4 miles after the location of the next house that is not covered.

    Claim: LOC[k + 1] is placed as far down or further down the road than LOC*[k + 1].

    1. LOC[k + 1] ≥ LOC[k] since we are placing the towers in increasing

order.

2. LOC[k] ≥ LOC*[k] by the inductive hypothesis

3. LOC[k + 1] ≥ LOC*[k]

4. LOC*[k + 1] ≥ LOC*[k] in order for it to be in the optimal solution,

you cannot have overlapping towers.

Since, LOC[k + 1] and LOC*[k + 1] are both ≥ LOC*[k] and LOC[k + 1] is placed as far down the road possible, in order to cover the house before it, LOC[k + 1] ≥ LOC*[k + 1]. Thus by Principle of Mathematical Induction, P(k) is true for k = 1..m.

Final Step: Show that m = n. Suppose that m > n. But this is impossible since you know by the above proposition that LOC*[k] ≤ LOC[k]. Since, we are placing towers at the furthest possible location, the greedy algorithm will stay ahead and make it impossible for m to be greater than n. Thus LOC[] has the same number of towers as the optimal solution and is itself optimal.

c. The algorithm complexity is O(n) since I only have to go through the list of houses once and place towers as I traverse the list of the houses.

2. Optimizing Road Placements

a. Place a road 2 miles after the first device. Then keep placing roads 2 miles after a device if the device is not already within 2 miles of a road.

```
OptimizingRoads (m[1…n])
    LOC[1] = m[1] + 2;
    k = 2;
    For i from 2 to n:
        If m[i] is not within the bounds of LOC[k - 1] then
            LOC[k] = m[i] + 4
            k++
    Return LOC;
```

a. Proof of correctness: Placing a road access 2 miles after a device with road access within 2 miles produces the minimum number of access roads.

Setup: Assume LOC*[$k_{1...}k_n$] is the optimal solution to the access roads placement problem with n number of access roads being placed for coverage of each device and let LOC[$k_1…k_m$] be the set of access road locations given by our greedy algorithm.

Proposition: Let P(k) be the proposition LOC*[k] ≤ LOC[k]. Want to show that

this is true for all k≥1 up to n.

Base Case: Show P(1) is true. LOC*[1] ≤ LOC[1].

Proof of Base Case: LOC*[1] ≤ LOC[1] is true since the LOC[1] is placed at the

max range away from m[1], thus LOC*[1] must be less than or equal to LOC[1] in order to provide access to the first device.

Inductive Case: Show that P(k) is true → P(k + 1) is true for any $1 \leq k < m$. Thus we need to show that LOC*[k] ≤ LOC[k] → LOC*[k + 1] ≤ LOC[k + 1].

Proof of Inductive Case:

LOC[k + 1] is the access road location that is 4 miles after the location of the next device that is not reachable.

Claim: LOC[k + 1] is placed as far down or further down the pipeline than LOC*[k + 1].

2. LOC[k + 1] ≥ LOC[k] since we are placing the access roads in increasing order.

2. LOC[k] ≥ LOC*[k] by the inductive hypothesis

3. LOC[k + 1] ≥ LOC*[k]

4. LOC*[k + 1] ≥ LOC*[k] in order for it to be in the optimal solution, you cannot have overlapping access roads.

Since, LOC[k + 1] and LOC*[k + 1] are both ≥ LOC*[k] and LOC[k + 1] is placed as far down the road possible, in order to reach the device before it, LOC[k + 1] ≥ LOC*[k + 1]. Thus by Principle of Mathematical Induction, P(k) is true for k = 1..m.

Final Step: Show that m = n. Suppose that m > n. But this is impossible since you know by the above proposition that LOC*[k] ≤ LOC[k]. Since, we are placing access roads at the furthest possible location, the greedy algorithm will stay ahead and make it impossible for m to be greater than n. Thus LOC[] has the same number of access roads as the optimal solution and is itself optimal.

c. The algorithm complexity is O(n) since I only have to go through the list of devices once and place roads as I traverse the list of the devices.

A. Scheduling to Minimize Lateness

1. **A counter example for Shortest Job First.** It must consist of only 2 jobs specified by their duration and deadlines that shows that ***Shortest slack time first*** does not always result in finding the optimal schedule
   a. job_1 $l_1 = 2$ $d_1 = 2$
   b. job_2 $l_2 = 1$ $d_2 = 4$

   Fill in the following table:

Schedule	Lateness of first job in the schedule	Lateness of second job in the schedule	Maximum of the latenesses
Shortest Job First schedule goes here	3-2= 1	1-4= -3	1
Optimal ordering (j1, j2)	2-2= 0	3-4= -1	-1

2. **A counter example for Shortest Slack Time First.** It must consist of only 2 jobs specified by their duration and deadlines, that is
   a. job_1 $l_1 = 3$ $d_1 = 3$ //d-1 = 0
   b. job_2 $l_2 = 1$ $d_2 = 2$ //d-1 = 1

   Fill in the following table:

Schedule	Lateness of first job in the schedule	Lateness of second job in the schedule	Maximum of the latenesses
Shortest Slack Time schedule goes here	3-3=0	4-2=2	2
Optimal Ordering (j2, j1)	1-2=-1	4-3=1	1

3. A proof that ***Earliest deadline first*** finds an optimal schedule using the basically the same steps used in proof **Largest weight to length ratio first** produces an optimal schedule for the Minimize total weighted time to completion problem
   a. Any additional assumptions you need to make (usually you wait to see what you need for the proof and add as needed as long as they do not change the generality of what you are trying to prove,
   b. Set up the notation for the optimal and greedy schedules
   c. Argue that if greedy is not an optimal schedule (does not minimize the maximum lateness) then two jobs in any optimal schedule must be ***out of order***.
   d. Swap the jobs and show that the **maximum lateness of the new schedule cannot be worst than the optimal schedule**. This is different (!!) in that in the TWTC problem we could show that objective function decreased and thus was better. Thus resulting in a contradiction

e. So now life is difficult since we cannot just conclude we have a contradiction. To reach the conclusion, you need to continue swapping jobs until you actually reach the greedy solution and show that it has the same maximum lateness
f. Hence the greedy solution is an optimal solution!!

Submit a Proof using the above steps. (Note it is possible to make assumptions about the jobs that would allow you to follow the proof used for TWTC more closely, but you are **not** allowed to do that here. The point it to expose you to a different style of proof that must be used in some situations.

Proof of correctness: Earliest deadline algorithm finds the minimum lateness of a set of jobs.

Assume that there is an ordering $\sum^*$ that is the optimal solution but is different from $\sum$ the ordering given by the greedy algorithm.

If $\sum^*$ is not $\sum$, it must have two consecutive jobs: i, j where j has the earlier deadline but i is before j in $\sum^*$. Swapping the two jobs does not affect the max lateness of the jobs before and after the two jobs i and j. Since we know that j has the earlier deadline, then $d_i > d_j$. Let C be the completion time of the jobs before i. Then, before the swap our max lateness would look like: $\max\{\max(L_{\text{before } i}), (C + l_i) - d_i, (C + l_i + l_j) - d_j, \max(L_{\text{after } j})\}$ and the max lateness after the swap would look like: $\max\{\max(L_{\text{before } j}), (C + l_j) - d_j, (C + l_j + l_i) - d_i, \max(L_{\text{after } i})\}$. Let's compare the max before the swap and after the swap assuming that max lateness before i and after j is lower than the lateness of i and j. Thus, we have $\max_{\text{before}} \{(C + l_i) - d_i, (C + l_i + l_j) - d_j\}$ which would result in the max lateness being $(C + l_i + l_j) - d_j$ since $d_j < d_i$. Now, we compare this max lateness value to that of $\max_{\text{after}} \{(C + l_j) - d_j, (C + l_j + l_i) - d_i\}$. First, $(C + l_i + l_j) - d_j$ must be greater than $(C + l_j) - d_j$, since their difference results in l_i which is greater than zero. Then, $(C + l_i + l_j) - d_j$ is also greater than $(C + l_j + l_i) - d_i$, since their difference results in $d_i - d_j$ which is greater than zero. Thus, since the $\max_{\text{before}}$ is greater than the possible maximums for $\max_{\text{after}}$, the new swapped schedule cannot be worse than the optimal schedule. Now we know that every time we swap two consecutive jobs out of order, the max lateness is can get any larger. These steps are repeatable for all two consecutive jobs that are out of order ($d_i > d_j$). Therefore, we can continue swapping in $\sum^*$, until no more pairs are out of order. Then, we will get that $\sum^*$ is equal to $\sum$ since both will be ordered from earliest deadline and therefore have the same max lateness that is minimized.

B. **Minimize Time to Completion:** Given n jobs j_1 to j_n with duration times d_1 to d_n, that need to be run on a single processor. What is the best way to schedule the jobs to minimize the time to completion. Time to completion here is defined to be the same as in the Total Weighted Time to Completion problem covered in the screencasts.

A. Specify the algorithm. Clearly describe how your algorithm would order the jobs.
Order the jobs in increasing order of duration times with shortest duration time being the first job in our ordered list. This should minimize the time to completion for the set of jobs.
Ex| job_1: $d_1 = 3$ $job_2 = d_2 = 5$
TTC for {1,2} = 3 + 8 = 11
TTC for {2,1} = 5 + 8 = 13

B. Prove this algorithm always finds an optimal solution.

Proof of correctness: Shortest duration algorithm finds the minimum time to completion of a set of jobs.

Assume that there is an ordering $\sum^*$ that is the optimal solution but is different from $\sum$ the ordering given by the greedy algorithm. If the greedy solution is not the optimal solution, the optimal solution must be out of order. If $\sum^*$ is not $\sum$, it must have two consecutive jobs: i, j where $d_i > d_j$ and i is before j in $\sum^*$ but i has the longer duration. Let C be the completion time of the jobs before i. Then, before swapping i and j, our CT would be: $job_i = C + d_i$ and $job_j = C + d_i + d_j$. Then after the swap the CT would be: $job_i = C + d_i + d_j$ and $job_j = C + d_j$. Now if we take the difference between the TTC of the two jobs before and after the swap we get $d_i - d_j$ which is greater than 0. Thus, the TTC after the swap was smaller than the TTC before the swap. Therefore, $\sum^*$ cannot be the optimal solution since it did not produce the set of jobs with a minimized time to completion. Thus, after the swap we obtain the solution given by the greedy solution, which is a job ordering of shortest duration. Therefore, the after ordering given by the greedy solution produces the minimum time to completion for a set of jobs.

C. Give the algorithm's asymptotic complexity.

Sort in increasing duration: O(n log n) using quick sort