Assignment 3: Greedy Algorithms

> Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: _____ Wisc id: _____

Greedy Algorithms

1. In one or two sentences, describe what a greedy algorithm is. Your definition should be informal, something you could share with a non computer scientist.

   Solution: A "greedy" algorithm only makes decisions which yield the highest immediate reward, but does not take into account how the decision will affect future states and their rewards. This is essentially treating every decision the algorithm makes as if that decision is the only one affecting the final result.

2. There are many different problems all described as "scheduling" problems. In the following questions, pay attention to the details of the problem setup, as they will change each time!

   (a) Let each job have a start time, an end time, and a value. We want to schedule as much value of non-conflicting jobs as possible. Use a counterexample to show that Earliest Finish First (the greedy algorithm we used for jobs with all equal value) does NOT work in this case.

   Solution: Two jobs: The first job runs from time 0-2 and is worth 1. The second job runs from time 1-3 and is worth 2. Earliest Finish First selects the first job and scores 1, but the optimum is the second job and total value 2.

   (b) *Kleinberg, Jon. Algorithm Design (p. 191, q. 7)* Now let each job consist of two durations. A job i must be preprocessed for p_i time on a supercomputer, and then finished for f_i time on a standard PC. There are enough PCs available to run all jobs at the same time, but there is only one supercomputer (which can only run a single job at a time). The completion time of a schedule is defined as the earliest time when all jobs are done running on both the supercomputer and the PCs. Give a polynomial time algorithm that finds a schedule with the earliest completion time possible.

   Solution: Longest Finish Time First
   We sort all jobs by their finish times (ignoring preprocessing time) in $O(N \log N)$ time. Then we run jobs through the supercomputer in descending finish times. We assign each job to a standard PC for finishing as soon as it is done preprocessing.

(c) Prove the correctness and efficiency of your algorithm from part (b).

Solution: LFTF runs in $O(N \log N)$ time as described above as this is the time required to sort, and selecting jobs from the sorted list can be done in $O(N)$ time.

For correctness, suppose some optimal algorithm OPT chooses job x immediately before job y at some point, but $f_x \leq f_y$. (This is an inversion from LFTF.) We claim that exchanging jobs x and y results in an optimal schedule.

First, no other job's completion time is affected, because x, y has the same preprocessing duration as y, x. Because we preprocess y earlier as a result of the exchange, only job x may finish later. If we call the end of the combined preprocessing time p, then job x completes at $p + f_x$ after the exchange. But job y completed at $p + f_y$ before the exchange, and $f_x \leq f_y$. Thus, no job may increase the overall completion time and we may exchange all jobs into LFTF order while maintaining optimality.

3. *Kleinberg, Jon. Algorithm Design (p. 190, q. 5)*

   (a) Consider a long straight road with houses scattered along it. We want to place cell phone towers along the road so that every house is within four miles of at least one tower. Give an efficient algorithm that achieves this goal using the minimum possible number of towers.

   > **Solution:** Proceed along the road from one end. Every time we reach a house not already in range of a tower, proceed four miles farther and plant a tower there.

   (b) Prove the correctness of your algorithm.

   > **Solution:**
   >
   > We write that our greedy algorithm places m towers in positions $g_1 < g_2 < \cdots = g_m$, and that an optimal solution places $n \leq m$ towers in positions $o_1 < o_2 < \cdots < o_n$. We then proceed with one proof by induction and one direct proof: the first showing that $o_i \leq g_i$ for all i from 1 to n, and the second showing that the greedy algorithm is optimal.
   >
   > For the first proof, our induction is on the statement that $o_k \leq g_k$ for some value $k \leq n$. Our base case, $k = 1$, is simple: the greedy approach places the first tower as far along the road as possible (i.e. with as large a value as possible for g_1) while still being within range of the first house. To show the inductive hypothesis, we assume that $o_k \leq g_k$ for some $k < n$ and let x_h be the position of the first house too far along the road to be reached by greedy tower k (i.e. the smallest house position such that $x_h > g_k + 4$). Since $o_1 < o_2 < \cdots < o_k \leq g_k < x_h - 4$, the optimal approach must place another tower within range of x_h, so $o_{k+1} \leq x_h + 4$. The greedy approach, however, always places the next tower at position $g_{k+1} x_h + 4 \geq o_{k+1}$, so, if $o_k \leq g_k$ for some $k < n$, $o_{k+1} \leq g_{k+1}$. As such, by induction, we have that $o_k \leq g_k$ for all $k \leq n$.
   >
   > To prove the optimality of the greedy algorithm, we first note that the greedy algorithm gives service to all houses with positions up to $g_k + 4$ with towers 1 through k (for any $k \leq m$): any house from positions $g_k - 4$ to $g_k + 4$ is satisfied by the tower at g_k, and no house further back than $g_k - 4$ can be given service by a later tower (as the g_i values are strictly increasing in i). We also observe that the optimal approach cannot give service to any houses after $o_n + 4$ with towers 1 through k, since all values for o_i are less than or equal to o_n. As such, all houses have positions less than or equal to $o_n + 4 \leq g_n + 4$, so all houses are given service by the first n towers placed by the greedy algorithm. As such, the greedy algorithm places only n towers, so $n = m$ and the greedy algorithm is optimal.

4. *Kleinberg, Jon. Algorithm Design (p. 197, q. 18)* Your friends are planning to drive north from Madison to the town of Superior, Wisconsin over winter break. They have drawn a directed graph with nodes representing potential stops and edges representing the roads between them.

   They have also found a weather forecasting site that can accurately predict how long it will take to traverse one of the edges on their graph, given the starting time t. This is important because some of the roads on their graph are affected strongly by the seasons and by extreme weather. It's guaranteed that it never takes negative time to traverse an edge, and that you can never arrive earlier by starting later.

   (a) Design an algorithm your friends can use to plot the quickest route. You may assume that they start at time $t = 0$, and that the predictions made by the weather forecasting site are accurate.

   > **Solution:** Run Dijkstra's shortest paths algorithm, starting at Madison. Whenever we add node x to our shortest paths tree with shortest path $M \to x$ of length t, we query the forecasting site for the lengths of all edges outgoing from x at time t. Edge "lengths" change as a function of time, but since we can never reach the next node earlier by departing later, the key property behind Dijkstra's algorithm still holds: if an unvisited node x can be reached earlier than any other unvisited node, the shortest path through x does not go through any other unvisited nodes, since these can only be reached after we could reach x.
   >
   > To make recovering the path easier, each entry in the shortest path tree record includes both the shortest path distance to x and also the node y such that (y, x) was the last edge used.

(b) Demonstrate how your algorithm works using a small example with 6 nodes. Your demonstration should include any data structures you maintain during the execution of your algorithm and any queries you make to the weather forecasting site. For example, if your algorithm maintains a "current path" that grows from (M)adison to (S)uperior, you might show something like the following table:

Path	Total time
M	0
M,A	2
M,A,E	5
M,A,E,F	6
M,A,E	5
M,A,E,H	10
M,A,E,H,S	13
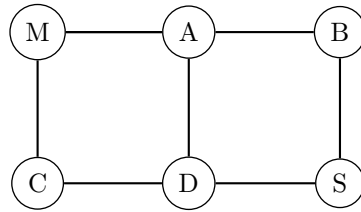
Solution:

Shortest paths:

Node	Distance	Predecessor	Priority queue (edges out of the shortest paths tree)
M	0		MA0:4,MC0:6
A	4	M	MC0:6,AD4:6,AB4:11
C	6	M	AD4:6,AB4:11
D	6	A	AB4:11,DS6:11
B	11	A	DS6:11
S	11	D	

Reconstructed path: M,A,D,S

Note: Priority queue holds queries of the form M to A at time t: t+edge time.

Coding Question

5. Implement the optimal algorithm for interval scheduling (for a definition of the problem, see the Greedy slides on Canvas) in either C, C++, C#, Java, or Python. Be efficient and implement it in $O(n \log n)$ time, where n is the number of jobs.

   The input will start with an positive integer, giving the number of instances that follow. For each instance, there will be a positive integer, giving the number of jobs. For each job, there will be a pair of positive integers i and j, where $i < j$, and i is the start time, and j is the end time.

   A sample input is the following:

   ```
   2
   1
   1 4
   3
   1 2
   3 4
   2 6
   ```

   The sample input has two instances. The first instance has one job to schedule with a start time of 1 and an end time of 4. The second instance has 3 jobs.

   For each instance, your program should output the number of intervals scheduled on a separate line. Each output line should be terminated by a newline. The correct output to the sample input would be:

   ```
   1
   2
   ```