Solutions to Homework Six

CSE 101

1. *Another scheduling problem.* Here's the idea: do the quickest jobs first.

   `Sort the` t_i
   `Output` $1 \leq i \leq n$ `in increasing order of` t_i

   The total time taken is $O(n \log n)$.

   To show that this is optimal, we'll prove a basic property of this scheduling problem.

   Claim. *Suppose $t_i < t_j$. Consider any schedule in which job j is done before job i. Then, swapping jobs i and j (and leaving the rest of the schedule unchanged) yields a smaller total waiting time.*

   Proof. Let S be the schedule in which j is done before i. Divide this schedule into three phases: (1) jobs before j, (2) jobs starting with j but before i, and finally (3) the remaining jobs starting with i.

   The swap yields a new schedule, call it $\overline{S}$, in which phase-one jobs and phase-three jobs have exactly the same waiting times as in S. But the jobs in the middle phase have their waiting times shrunk by $t_j - t_i$. Therefore, $\overline{S}$ is better than S. □

2. *Two-coloring a graph.* Some observations about two-coloring a graph G:

   - Different connected components of G can be handled separately.
   - Fix any vertex u. If there is a valid two-coloring in which u is `white`, then there is also a valid two-coloring in which it is `black` (just flip all colors in u's connected component).
   - Therefore, if G is two-colorable, then in each connected component, we can pick any node and color it `black`; thereafter the colors of the other nodes in that component are fully determined.

   The algorithm:

   1. Find the connected components of G.
   2. For each connected component C:
      - Pick a vertex in that component.
      - Run breadth-first search starting at that vertex (this search will be limited to C).
      - Color all nodes in C at even distance `black` and all nodes at odd distance `white`.
   3. For each edge in G:
      - If the endpoints have the same color, halt and output "not two-colorable".

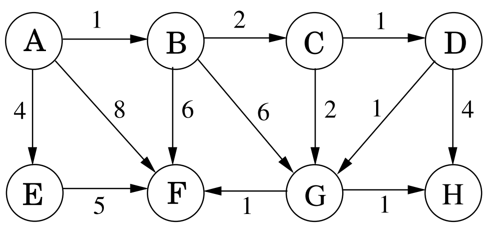
      Output "two-colorable".

   Each of the steps (1)–(3) is linear-time, and thus the overall running time is linear.

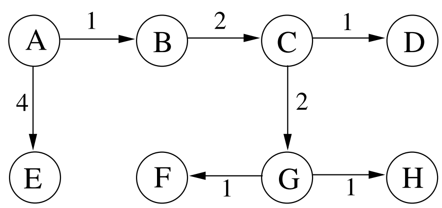
   Justification: Suppose, first, that G is two-colorable. Then, by the remarks above, the algorithm will discover a valid coloring, and this will be validated in step (3).

   Conversely, if G is not two-colorable, then whatever coloring is obtained in steps (1)–(2) is necessarily invalid. This will be detected in step (3).

3. *Textbook problem 4.1.*



A	B	C	D	E	F	G	H
: $\boxed{0}$	∞	∞	∞	∞	∞	∞	∞
0	$\boxed{1}$	∞	∞	4	8	∞	∞
0	1	$\boxed{3}$	∞	4	7	7	∞
0	1	3	$\boxed{4}$	4	7	5	∞
0	1	3	4	$\boxed{4}$	7	5	8
0	1	3	4	4	7	$\boxed{5}$	8
0	1	3	4	4	$\boxed{6}$	5	6
0	1	3	4	4	6	5	$\boxed{6}$
0	1	3	4	4	6	5	6



4. *Non-optimality of greedy set cover.* There is a counterexample in Section 5.4 of the textbook.

5. (a) One way Alice can choose a set of guests S is to first let S contain all n people, and then eliminate any people *that absolutely have to be eliminated.*

   Which people are these? *Anybody with less than five friends.*

   This suggests a simple algorithm:

```
Initialize S to contain all n people
While S contains a person p with fewer than five friends in S:
   Remove p from S
```

Claim. *Let S^* be any optimal solution. Then throughout the execution of the algorithm, S^* is always contained in S.*

Proof. Our initial setting of S certainly contains S^*. And any person p we subsequently eliminate has less than five friends in S (and thus less than five friends in S^*) and so cannot be in S^*. Since we never eliminate a person in S^*, set S always contains S^*. □

Moreover, every node in the final set S it has at least five neighbors in S. Therefore $S = S^*$.

(b) Here's a simple linear-time implementation. The set S is maintained as a Boolean array `invite`.

```
Q = (empty queue) // people to be eliminated
For each person p:
   invite[p] = true // Boolean array that indicates who is invited
   set friends[p] to the the number of friends of p
   if friends[p] < 5:
      inject(Q, p) // mark p for elimination
      invite[p] = false

While Q is not empty:
   p = eject(Q)
   for all friends q of p:
```

```
friends[q] = friends[q] − 1
if friends[q] < 5 and invite[q] = true:
    inject(Q, q)
    invite[q] = false
```

Every person p to be eliminated ends up in the queue at some stage. When p is pulled off the queue, it is officially eliminated, in the sense that its neighbors have their friend-count decremented. Moreover, p is only added to the queue once; thereafter `invite`$[p]$ becomes `false`.

The form of the input is rather like the adjacency list of a graph. Setting the `friends` array is like computing the degree of every node in that graph: linear time. Likewise, the innermost loop is like iterating through the neighbors of a specific node in the graph.

Thus the overall running time is the same as that of a basic graph search algorithm like DFS, that is, $O(n + m)$, where m is the number of friend-pairs (edges).

6. A natural approach: on your first tank of gas, go as far as possible within M miles; that is, go up to the largest $m_i \leq M$. On your second tank, go to the largest m_j with $m_j - m_i \leq M$, and so on.

   Here's the pseudocode.

```
i = 1 // index of current position
while i < n:
    j = i // index of next gas stop
    while j < n and m_{j+1} − m_i ≤ M:
        j = j + 1
    output ‘‘stop at m_j’’
    i = j
```

   The running time is $O(n)$: the indices i, j each do a single pass through the array of mile-posts.

   To see why this strategy is optimal, let $S_1, S_2, \ldots$ denote the mileage posts at which we end up stopping. For instance, if our third stop is at m_{10}, then $S_3 = m_{10}$.

   Let $T_1, T_2, \ldots$ be any other valid solution: any sequence of stops that doesn't run out of gas. We'll show that our solution (S_k) is at least as good as (T_k).

   Claim. *$S_k \geq T_k$ for all k.*

   Proof. We can prove this by induction on k. It certainly holds for $k = 1$, by the way in which we choose the first stopping point.

   So let's say it holds up to the first k stops (that is, $S_k \geq T_k$), and let's look at $k+1$. Since $T_{k+1} - T_k \leq M$ and $S_k \geq T_k$, it follows that $T_{k+1} - S_k \leq M$. By definition, $S_{k+1} - S_k$ is the longest stretch starting at S_k that is at most M miles long. Therefore, $S_{k+1} \geq T_{k+1}$. □

7. Given a set of intervals, let's sort them by ending-point so that we have $[\ell_1, u_1], \ldots, [\ell_n, u_n]$ where $u_1 \leq u_2 \leq \cdots \leq u_n$.

   Here's the idea: we need at least one point in the very first interval, $[\ell_1, u_1]$. We might as well take this point to be u_1, because it touches as least as many other intervals as any other point in $[\ell_1, u_1]$. Then we can recurse.

```
Let I be the set of n intervals
Let C = {} (selected points)
While I is not empty:
    Find the smallest u_i in I
    Add u_i to C
    Remove intervals containing u_i from I
```

To understand why this greedy strategy is optimal, we show that there is always an optimal solution in which the leftmost point is u_1. With this in place, we can then remove every interval that u_1 touches, leaving a smaller version of the original problem; and recurse.

Claim. *Let $x_1 < x_2 < \cdots < x_m$ be any solution, that is, a set of points that touches all the intervals. Then swapping x_1 with u_1 also yields a solution.*

Proof. The leftmost point, x_1, must satisfy $x_1 \leq u_1$; otherwise none of the points touches $[\ell_1, u_1]$.

The intervals touched by x_1 all start before x_1, and thus before u_1; and they end after u_1, since u_1 is the leftmost ending point. Therefore, any interval touched by x_1 is also touched by u_1, and the substitution $x_1 \to u_1$ also generates a valid solution. □

The greedy algorithm can be implemented in time $O(n \log n)$; do you see how?