Midterm Practice

These problems were designed to be used as a "take-home" exam, which, normally, allows students to work for 24 hours. So, if you cannot finish all these problems in 2-3 hours, it would still be okay.

1. **Finding the smallest triangle** (10 point)
   You are given the positions of n points in the plane. How would you find a triangle with the shortest perimeter? We call such triangle the *smallest* one. This is an example of the kind of problem that occurs in such application areas as air traffic control, the analysis of galaxies in astronomy, and the layout of integrated circuits. In this problem, we will develop a divide-and-conquer algorithm that solves the problem much more efficiently than naïve methods. You don't have to finish all the parts in this problem to score some points; you can skip any part.

   (a) Consider the naïve algorithm that simply computes the distance between every triplet of points and keeps track of the smallest triangle found. Show that the running time of this algorithm is $O(n^3)$. (1 pt)

   (b) Figure 1 shows a template for a divide-and-conquer algorithm for this problem. (For simplicity, the algorithm only outputs the shortest perimeter found; it will be clear shortly how to modify it so that it outputs the smallest triangle.)

Algorithm Smallest_Triangle(P)
{P *is a set of* n *points in the plane,* $n \geq 2$ *a power of two;*
the points are assumed to be sorted by x*-coordinate and by* y*-coordinate*}
if $|P| = 3$ then output the perimeter d of the triangle formed by the three points in P
else
  divide P by x-coordinate into sets P_L, P_R, each of size $\frac{n}{2}$, using a vertical line Λ
  $d_L :=$ Smallest_Triangle(P_L)
  $d_R :=$ Smallest_Triangle(P_R)
  $d := \min\{d_L, d_R\}$
(∗) check whether there exists a triplet of points p_1, p_2, p_3, at least one in P_L
    and at least one in P_R, from which the perimeter of the triangle formed
    is less than d; let d' be the minimum such perimeter found,
    and $d' = +\infty$ if no such pair exists
  output $\min\{d, d'\}$

Figure 1: Template for a smallest triangle algorithm. The implementation of step (∗) is left deliberately vague; you will fill in the details of this in parts (c)–(g). In the "divide" step, it is possible that one or more points may lie on the vertical dividing line Λ. In this case we are free to allocate these points arbitrarily to P_L or P_R, as long as we ensure an even split. "Sorted by x-coordinate and by y-coordinate" means that there are two sorted lists containing the same set of points, one in x-order and one in y-order. You need not worry for now about how these two lists are constructed for P_L and P_R.

Prove that the algorithm Smallest_Triangle is correct, i.e., for any input set P of n points ($n \geq 2$ a power of two), the algorithm outputs the minimum distance between any pair of points in P. (1pt)

(c) Let's now consider the implementation of step (∗). Denote by S the subset of P consisting of all points in a vertical strip of width d centered on the dividing line Λ. Show that in this step, it is enough to consider only triplet of points p_1, p_2, p_3 where $p_1 \in P_L \cap S$, $p_2 \in P_R \cap S$, and $p_3 \in (P_L \cup P_R) \cap S$. (1pt)

(d) Prove that in any square box of side $d/4$ there can be at most two points from P_L and at most two points from P_R. (1pt)

(e) Now use parts (c) and (d) to prove that, for each point $q \in P_L \cap S$, there can be at most 16 points p' and p'' in $P_R \cap S$ which, together with p, form a triangle of perimeter less than d. [The number 16 may be an over-estimate: if you can do better, that is good but not important for our purposes.] (2pt)

(f) From part (e), it should be reasonably clear that the $(*)$ step requires at most $16^2 n$ triangle perimeter computations. Now assume that the $(*)$ step can be implement to run in time $O(n)$. Write a running time recurrence for the algorithm and show that it runs in time $O(n \log n)$. (2pt)

(g) Last part, we claimed that the $(*)$ step could be done in time $O(n)$. This assumes that, for any point p we can identify the 16 (or fewer) possible candidates q without performing any distance computations. Explain how to do this efficiently for all points p, using the fact that the points are sorted by y-coordinate. (2pt)

2. **Connections** (5 points)
You are given a circle C on a plane with n points in circular order on its boundary: $1, 2, \ldots, n$. You want to connect m pairs of points $(a_1, b_1), (a_2, b_2), \ldots, (a_m, b_m)$ with copper wires. To connect point i and j, you can place the wire connecting them entirely inside the circle, or entirely outside the circle. However, you don't want any two wires to touch.

Given a set of required connection pairs, your task is to find an algorithm that determine if it is possible to connect them all. And if possible, you must specify for each pair whether to put it inside of outside the circle.

Show that this problem can be solved as a 2-SAT problem. What are your variables? What is the Boolean formula? What is the running time of your algorithm?

[Bonus problem] (1 point) Find a faster algorithm. (You don't need to give or prove the algorithm. Only reference is need, e.g., the name of problem that the algorithm solves.)

Hint for bonus problem: it's a linear time algorithm.

3. **Searching known queries** (10 points)
We are given a list of n numbers incrementally. While receiving numbers, we also have to answer query of the form: "is x in the list up to now?" (Here is an example: receive numbers 10,30,20; receive query 'is 10?'—yes; receive query 'is 15?'—no; receive numbers: 14,15,17; receive query 'is 15?'—yes; receive query 'is 30?'—yes.)

This problem can be solved by storing the numbers in a balanced binary search tree. The total running time is $O(n \log n + m \log n)$, where m is the number of queries, because each insertion into the tree takes $O(\log n)$ and each query can be supported in $O(\log n)$ time.

In this problem, we consider a special case of this problem. In this case, the number of inputs n is much smaller than the number of queries m. Also, suppose that we know Q, the sequence of all m queries, in advance and can spend *any time* to preprocess Q.

Describe a data structure performs the previous task in time $O(n \log m + m)$, i.e., the time needed to process one input data (insertion) is $O(\log m)$ and the query time is $O(1)$. How much time do you need for preprocessing?

4. **Good networks** (10 points)
Tree networks are useful because for each pair of nodes, there exists only one path connecting them in the networks. In this problem, we look at other kinds of networks with similar property.

A directed graph $G = (V, E)$ *satisfies our condition* when for any pair of nodes u and v, if v is reachable from u, there is a unique *simple* path from u to v (i.e., there is only one path from u to v). Give an algorithm which runs in time $O(mn)$ that checks if a given directed graph satisfies our condition.

Hint: Use dfs. What kind of edges that should not be in our graph?

5. **Shortest paths when edges are on and off** (10 points)
Given a directed graph $G = (V, E)$ with non-negative length function $l : E \to \mathbb{R}$. Given a source s, we can use Dijkstra's algorithm to find shortest path from s to every nodes.

Suppose that there is a further condition enforcing for each edge as follows. We first consider the "length" on each edge as a traveling time. We start from s at time $t = 0$. For each edge $e = (u, v)$, we have parameters $p(e)$ and $r(e)$, which define the possible time one can entering the edge at u

to move to v, i.e., the time when the edge is opened, as follows[1] The edge is closed at time 0. It is opened from $r(e)$ to $r(e) + p(e)$, then it is closed from time $r(e) + p(e)$ to $r(e) + 2p(e)$. In general, for any non-negative integer i, it is open from $r(e) + 2ip(e)$ to $r(e) + (2i + 1)p(e)$, and it is closed from $r(e) + (2i + 1)p(e)$ to $r(e) + (2i + 2)p(e)$. When the edge is closed any one traveling on the edge can continue to move, but no one would allowed to enter the edge until it is opened again. We want to compute the shortest traveling time from s to every node.

For example, suppose that we have a network with 2 edges as follow:

```
s--------->u--------->t
```

The parameters are: $l(s,u) = l(u,t) = 10$, $r(s,u) = 2, p(s,u) = 3$, and $r(u,t) = 0, p(u,t) = 8$. The shortest traveling time from s to u is 12, and to t is 26.

Show how to modify Dijkstra's algorithm to solve this problem. Prove the correctness and its running time.

6. **Matching/maximum flows** (5 points)
Consider the *maximum bipartite matching* problem described as follows. Given two sets of nodes A and B and a set of edges $E \subseteq A \times B$. Note that edges only connect nodes in A with nodes in B. A *matching* M is a subset of edges such that each node is adjacent to at most one edge in M. We want to find the *maximum matching,* i.e, the matching with the maximum size.

   One can use the maximum flow algorithm to find the maximum matching as follows. We create a source node s and a sink node t. Then, we create an edge connecting s to each node in A with capacity 1, and an edge connect each node in B to t with capacity 1. Also, put capacity 1 to all original edges. Note that the value of the maximum flow from s to t equals to the size of the maximum bipartite matching, and edges with flow value of 1 correspond to edges in the matching. (See any textbooks.)

   We use Ford-Fulkerson algorithm to find a maximum flow. Each step, we augment one unit of flow from s to t and maintain a matching. When we cannot augment any more flow, the current matching is maximum. A node is called *matched* if it is adjacent to some edges in the matching. Show that if a node is matched at any step, it remain matched through out the execution of the algorithm.

[1] This condition represents traffic lights, in a simpler form.