Remarks: All the graphs here are without self loops and parallel or anti-parallel edges. In all the algorithms, always explain their correctness and analyze their complexity. The complexity should be as small as possible. A correct algorithm with large complexity, may not get full credit.

Choose 5 out of the next 6 questions.

Question 1: In this question, you are given a procedure $A$ that finds the median in an $n$ element array (the middle element after sorting) in $O(n)$ time.

1. Give an algorithm that uses $A$ to find the $k$-largest element in the array, for a given $k$. Which algorithmic method are you using?

   **Answer:** We use divide and conquer. We find the median $x$ in $O(n)$ operations. Then we go over $A$ and split $A$ to $A_s$, the elements smaller or equal to $x$, and $A_l$, the elements larger than $x$. This is similar to quicksort. Now, we recurse only on $A_s$ or on $A_l$. The idea is the following: if $k \leq n/2$, then the $k$ largest element belongs to $A_s$ otherwise, it belongs to $A_l$. However, if we recurse on $A_l$, we need to call the recursion with $k - n/2$. This is because, $n/2$ elements (the elements of $A_s$) are discarded from the recursion. This makes the $k$ only the $k - n/2$ largest in the new array. When the array in the recursion reaches size 1, we stop, and the single element in the array is the $k$-largest.

   The correctness is by induction (left to you). The time complexity: The number of operations is $O(n) + O(n/2) + O(n/4) + \ldots$. This is because at every iteration, the remaining array is halved. Note that $O(n) + O(n/2) + O(n/4) + \ldots = O(n + n/2 + n/4 + \ldots) = O(2n) = O(n)$. Thus, the running time is $O(n)$.

2. Let $M$ and $m$, be, respectively, the maximum and minimum in the array. Show that there are two elements $z_1, z_2 \in A$ so that

   $$|z_1 - z_2| \leq \frac{M - m}{n - 1}.$$

   **Answer:** Let's say that for the sake of the answer, we sort $A$ into $A_s = [a_1, a_2, \ldots, a_n]$, with $a_i \leq a_{i+1}$. Note that the sum

   $$(a_2 - a_1) + (a_3 - a_2) + (a_4 - a_3) + \ldots + (a_n - a_{n-1}),$$

   equals $a_n - a_1$. Because the array is sorted, $a_n = M$ and $a_1 = m$. So $(a_2 - a_1) + (a_3 - a_2) + (a_4 - a_3) + \ldots + (a_n - a_{n-1}) = M - m$. Thus, the average of the numbers $(a_2 - a_1), (a_3 - a_2), (a_4 - a_3), \ldots + (a_n - a_{n-1})$ is $(M - m)/(n - 1)$ (as there are $n - 1$ terms $a_{i+1} - a_i$.) Thus, one of the differences (the minimum $a_{i+1} - a_i$) is at most the average $(M - m)/(n - 1)$ (because the minimum is always at most the average).
3. Give an algorithm that finds such $z_1$ and $z_2$

**Answer:** From the above answer, it follows that sorting the array using Merge-Sort, and then going over the array and choosing the $i$ so that $a_{i+1} - a_i$ is minimum, solves the problem in time $n \log n$. This gets almost all the points (all besides 2 points). There is a way to do this in $O(n)$. (I may print the $O(n)$ solution for that later.)

**Question 2:** Solve the weighted version of question 4 in exercise 1. The lines have weights.

The goal is to find a perfect matching collection of lines (touching all the $p_i$ exactly once and all $q_j$ at most once) of maximum weight.

**Answer:** We may assume without loss of generality that the $p_i$ and $q_i$ are sorted by increasing value of their $y$-coordinate.

We define $k \cdot t$ subproblems $P_{i,j}$ of our problem. Problem $P_{i,j}$ is the problem of solving the perfect matching problem for the sets $\{p_1, \ldots , p_i\}$ versus $\{q_1, \ldots , q_j\}$. This means that only $p_1$ needs to be matched. The value of $M[1,j]$ is the value of the the largest edge $(p_1,q_s)$ for $s \leq j$. If $p_1$ is not connected to any of $q_1, \ldots , q_s$, then you define $M[1,j] = -\infty$.

For a vertex $p_j$, let $\text{high}(j)$ be the highest index, such that a line between $p_j$ and $q_{\text{high}(j)}$ exists. For example, of $p_7$ is joined to $q_7$ but not to $q_8, \ldots , q_t$, then $\text{high}(j) = 7$.

The first choice for a line for $p_j$ should be the line from $p_j$ to $q_{\text{high}(j)}$, because this leaves the largest room for non-intersecting lines.

We now say how to compute $M[i,j]$, $i \leq j$. We split into two possibilities. Either $p_i$ is joined to $q_{\text{high}(i)}$ in the optimum solution. In this case, the weight $w(p_i,q_{\text{high}(i)})$ is part of the optimum weight. In addition, the rest of the matching must be between $p_s, s \leq i - 1$, and $q_s, s \leq \text{low}(i) - 1$. This is because there can not be an intersection (for example, any line to the $\text{high}(i) + 1$ q vertex, would intersect the line between $p_i$ and $q_{\text{high}(i)}$.) Thus, in this case $M[i,j] = w((p_i,q_{\text{low}(i)})) + M[i-1, \text{low}(i) - 1]$.

The second case is that $p_i$ is not joined to $q_{\text{high}(i)}$. In this case, the matching takes place between $p_1, \ldots , p_i$ and $q_1, \ldots , q_{\text{low}(i)-1}$. This is because the highest possible point to be joined to $p_i$ is $q_{\text{high}(i)-1}$. Thus, in this case, $M[i,j] = M[i, \text{low}(i) - 1]$.

In summary:

$$M[i,j] = \max\{w((p_i,q_{\text{low}(i)})) + M[i-1, \text{low}(i) - 1], M[i, \text{low}(i) - 1]\}.$$

The matrix is filled by two for loops. The external one goes over $i = 1$ to $k$ and the other $j = 1$ to $t$. The above equation is used to fill $M$. The running time is $O(k \cdot t)$. 

2
Question 3:

1. Give an algorithm that takes an undirected graph, and adds the minimum possible number of edges so that the graph will have an Euler cycle

   **Answer:** Any graph has an even number of odd degree vertices. Let $2k$ be the number of odd degree vertices in our graph. You need to make all the degrees even, in order for an Euler cycle to exist.

   Divide the odd degree vertices into pairs $\{v_1,v_2\}, \{v_3,v_4\}, \ldots, \{v_{2k-1},v_{2k}\}$. Add all the edges $(v_i,v_{i+1})$ between the above pairs. This gives $k$ edges.

   $k$ is also the minimum, as if you add $t$ edges, these edges must touch all the $v_i$ vertices of odd degree. Every edge can touch at most two new vertices, thus you need that $2t \geq 2k$. Thus, $t \geq k$.

   The time complexity is that of computing the degrees: $O(E)$.

2. Give an algorithm that takes as input an undirected graph. The algorithm should give a single direction to every edge, and minimize $\max_{v \in V} |d_{in}(v) - d_{out}(v)|$.

   **Answer:** Let $2k$ be the number of odd degree vertices in $V$. Add $k$ edges and make all the degrees even (this edges are deleted in the sequel). Now, the graph admits an Euler cycle. Give directions to the edges according to the direction of the cycle (if you go over an edge $(u,v)$ in the cycle, give the edge direction from $u$ to $v$).

   Note that the number of times we enter a vertex $v$ equals the number of times we leave a vertex. This means that in the new graph (after we have added the $k$ edges) we can make $d_{in}(v) = d_{out}(v)$, for all $v$.

   Remove back the edges that you have added. Thus, now we get that $|d_{in}(v) - d_{out}(v)| \leq 1$. Thus we can always make the degrees “almost” equal. Note that for a vertex of odd degree, the difference will be at least one in any case (why?). Thus a difference of 1 is the best possible.

**Question 4:** Let $G$ be a directed graph with $V = \{1, \ldots, n\}$. Give an algorithm that for each $i \in V$, computes the smallest $j$, $j \neq i$ reachable from $i$.

   **Answer:** It is possible to do $n$ times of BFS. When doing BFS from $j$, record the lowest reachable vertex from $j$. This requires $O(E \cdot V)$ running time.

   The following is better. We first check which are the vertices that can reach 1. In order to do so, you start with reversing the directing of the edges of the graph. Let $G_R$ be the reversed graph. Now, you do a BFS from 1 in the reverse graph. All the vertices reachable from 1 in the reverse graph, can reach 1 in $G$.

   Let $U_1$ be the set of all vertices that can reach 1. Note that there are no directed edges in $G$ from $v \in \bar{U}_1$ (the complement of $U_1$) into $U_1$, as such edges would imply that $v$ can reach 1 as well. Thus, all the directed edges that go from a $U_1$ vertex to a $\bar{U}_1$ vertex in $G$, actually start at the $U_1$ vertex, and end at the $\bar{U}_1$ vertex. These edges can not help any vertex getting out of $\bar{U}_1$ or reaching another vertex in $\bar{U}_1$. Thus, we remove $U_1$ and all the edges touching $U_1$ from $G_R$. These edges can not help the next BFS iterations. In the remaining graph, we check which are the set $U_2$ that can reach
2, and later remove $U_2$ from the graph $G_R$, and so on. Note that every vertex and edge is scanned once in the algorithm. For example, in the first iteration, the edges in $G$ that are scanned are only the ones internal to $U_1$. This gives an $O(E + V)$ running time.

**Question 5:** Give an algorithm that gets as input a weighted graph $G(V, E, w)$, with $w(e) \geq 0$ for all $e \in E$. The algorithm gets in addition an edge $e_1 = (v, u)$ and two vertices $a, b$. The algorithm should check if there is a path of length $\text{dist}(a, b)$ from $a$ to $b$ that goes through $e$.

**Answer:** A path as required first goes from $a$ to $u$ or $v$, the goes along the edge from $u$ to $v$ or vice-versa, (depending on the first choice) and then continues to $b$.

The shortest paths that are of this type are:

Use the shortest path from $a$ to $v$, go to $u$ via $e$, and now take the shortest path from $u$ to $b$. The other possibility is to go from $a$ to $u$, to cross $e$ and to go to $b$.

Thus, we should check if $\text{dist}(a, b) = \min\{\text{dist}(a, u) + w(u, v) + \text{dist}(v, b), \text{dist}(a, v) + w(v, u) + \text{dist}(u, b)\}$. If so, the answer is yes and otherwise, the answer in no. The running time is like Dijkstra.

**Question 6:** Give an algorithm that gets as input an undirected graph $G(V, E)$ and a vertex $v \in V$. The algorithm should produce the shortest simple cycle containing $v$.

**Answer:** We use a variant of BFS. In BFS, vertices get labeled by $i + 1$ by vertices labeled $i$. Note that if a neighbor of $w$ tries to relabel $w$, and $w$ is already labeled, this means a cycle. See Figure 1. Say, for example that in the graph of the figure, $d$

![Figure 1: An example](image)

labels $c$ (by 3). When $b$ tries to relabel $c$, a cycle has been closed (since $b$ and $d$ both
have a path to $v$, the paths $b$ to $v$ and $d$ to $v$ must intersect. They intersect at $a$ in our example).

So, as soon as a vertex is relabeled, a cycle has been found. However, it does not mean that the cycle contain $v$. In our example, the paths from $b$ to $v$ and $d$ to $v$ meet, before they reach $v$. Thus, if we try to include the vertex $v$ in this cycle, (adding the edge $(v, a)$ twice) the cycle including $v$ that we get, is not simple.

Note that as long as we do not detect a cycle, the BFS graph is a tree. See for example figure 2. The dotted edges have not been scanned yet. The other edges form a tree.

![Figure 2: An example](image)

As long as, say, $e$ is not rediscovered (by the dotted edge $(d, e)$) it will remain a tree. What is important to keep, is to which child of the tree a vertex $w$ belong to. For example, $c$ currently belongs to the subtree of $a$ and so does $d$. $f$ and $e$ belong to the subtree of $b$. Note that the cycle $(a, c, g, d, a)$, that is discovered when we try to relabel $g$ via the edge $(d, g)$, does not contain $v$, because $c$ and $d$ belong to the same sub-tree. The cycle discovered when $d$ relabels $e$ contains $v$, because $d$ and $e$ belong to different subtrees. So we will have a variable for keeping to which tree a vertex belongs.

Another important thing is to try to discover the cycle $v, a, d, e, b, v$, before the cycle $v, a, d, h, e, b, v$. This means that we need to check for edges inside the layer $i$, before we try edges between layer $i$ vertices and layer $i + 1$ vertices.

The $fp(w)$ variable keeps the child of $v$ that is an ancestor of $w$. If $u$ relabels $w$, this gives a cycle that includes the root $v$, only if $pf(u) \neq pf(w)$. The set $L_i$ is the set of vertices at distance $i$ from $v$.

The algorithm

**Algorithm 1** *Cycle*

1. Let $L_0 \leftarrow \{v\}$, $L_1 \leftarrow N(s)$ (all the neighbors of $s$).
2. Let $fp(w) \leftarrow w$, for all $w \in N(v)$, and $i \leftarrow 1$.
3. While $L_i$ is not empty Do:

5
(a) If there is an edge between two vertices $a$ and $b$ in $L_i$, so that $fp(a) \neq fp(b)$, then Return(The smallest cycle containing $v$ has length $2i + 1$).

(b) Else, let $L_{i+1}$ be all the unlabeled vertices that are neighbors of at least one $L_i$ vertex.

- If there is a vertex $u \in L_{i+1}$ that has two neighbors $a, b \in L_i$, with $fp(a) \neq fp(b)$, then Return(The smallest cycle containing $v$ is of length $2i + 2$.)
- Else, for every $u \in L_{i+1}$, chose a vertex $p(u) \in L_i$ that is a neighbor of $u$, and $fp(u) \leftarrow fp(p(u))$

4. Return(The graph contains no cycles containing $v$).

We can find the actual minimum cycle containing $v$ (namely, one of them) tracking back the $fp()$ values from the rediscovered vertex. The correctness and time complexity is left to you. For the time complexity: its the same as BFS.