Quiz 1

- Do not open this quiz booklet until directed to do so. Read all the instructions on this page.
- When the quiz begins, write your name on every page of this quiz booklet.
- You have 75 minutes to earn 180 points. **However, 150 points is viewed as full mark. Anything above that are bonus points.** Do not spend too much time on any one problem. Read them all first, and attack them in the order that allows you to make the most progress.
- **This is a closed-book exam**. No calculators or programmable devices are permitted. No cell phones or other communications devices are permitted.
- Write your solutions **using non-erasable pens** in the space provided. If you need more space, write on the back of the sheet containing the problem. Pages may be separated for grading.
- Do not waste time and paper rederiving facts that we have studied. Simply cite them.
- When writing an algorithm, a **clear** description in English will suffice. Pseudo-code is also okay but not required.
- **Pay close attention to the instructions for each problem**. Depending on the problem, partial credit may be awarded for incomplete answers.
- Turn in your exam paper to your designated AI after the exam, and sign your name on the sign-in sheet.

Problem	Parts	Points	Grade	Grader
Name	0	5		
1	9	81		
2	3	32		
3	5	32		
4	3	30		
Total	–	180		

Name: _____

Problem 1. Single Choice Problems [81 points] (9 parts)

For each of the following questions, choose the unique best answer and write the corresponding letter in the bracket before the problem. There is no need to justify the answers. Each problem is worth 9 points.

(a)( ) In the Interval Scheduling Problem, the goal is to find a set of maximum number of compatible intervals. In class we saw the following greedy strategy. First, sort the intervals via some natural order. Then, start with an empty set; for each interval in the order, put the interval in the set whenever it is compatible with previously chosen intervals in the sets.

Such greedy strategy always produces optimal solutions if the "natural order" is

A. the non-decreasing order on start times (break ties arbitrarily).
B. the non-decreasing order on finish times (break ties arbitrarily).
C. the non-decreasing order on interval lengths (break ties arbitrarily).
D. the non-decreasing order on the number of intervals those intersect with the given interval (break ties arbitrarily).

Solution: B.

(b)( ) In the Scheduling All Intervals Problem, the goal is to partition the input intervals into minimum number of compatible sets. In class we saw the following greedy strategy. First, sort the intervals via some natural order. Then, start with no sets. For each interval in the order, if it can be added to an existing set such that the set is still compatible, add the interval to the set (if more than one such sets exist, choose an arbitrary one); otherwise, introduce a new set and add the interval to the new set.

Such greedy strategy always produces optimal solutions if the "natural order" is

A. the non-decreasing order on start times (break ties arbitrarily).
B. the non-decreasing order on finish times (break ties arbitrarily).
C. the non-increasing order on finish times (break ties arbitrarily).
D. both A and C.

Solution: D. Both A and C are correct – when we reverse the order of time travel, the latest finish time become the earliest start time.

In Parts c), d), e), and f), let $G = (V, E)$ be an undirect graph with edge weights $w : E \to \mathbb{R}$.

(c)( ) If all the edges have distinct weights, then there is a unique minimum spanning tree. If the edge weights are not distinct, then it is possible that there are several minimum spanning trees.

A. This statement is True.
B. This statement is False.

Solution: A. The first part was a HW problem. For the second part, consider cycle with each edge having weight 1. Then removing any edge from the cycle will give a minimum spanning tree.

(d)( ) If $w_{\max}$ denote the maximum weight. If G has more than $|V|-1$ edges and there is a unique edge having the largest weight $w_{\max}$, then this edge cannot be part of any minimum spanning tree.

A. This statement is True.

B. This statement is False.

Solution: B. Consider when G is formed by a cycle of $|V|-1$ vertices and another vertex connected to an arbitrary vertex of C by the edge with the largest weight.

(e)( ) If $w_{\min}$ denote the minimum weight. If there is a unique edge having the smallest weight $w_{\min}$, then this edge must be part of every minimum spanning tree.

A. This statement is True.

B. This statement is False.

Solution: A. Follows from the Cycle Property.

(f)( ) If there is a minimum spanning tree T for G and an edge $e \in E$ that does not belong to T, then there exists a *simple* cycle in G such that $w(e)$ is no less than the weight of every edge in the cycle.

A. This statement is True.

B. This statement is False.

Solution: A.

(g)( ) We have seen in class that Dijkstra's algorithm with a Fibonacci heap solves the Single Source Shortest Path Problem (SSSP) with non-negative edge weights in time $O(n \log n + m)$, where n is the number of vertices and m is the number of edges. However, under certain special assumptions, one can modify the algorithm and solve the SSSP in time $O(n+m)$. Such assumptions can be

A. all edge weights are integer numbers in range $[-50, 50]$.

B. all edge weights are integer numbers in range $[0, 100]$.

C. all edge weights are real numbers in range $[0, 100]$.

D. both A and B.

E. both B and C.

Solution: B.

Choice A is incorrect because Dijkstra's algorithm does not support negative-weighted edges. If Choice C were correct, we could rescale all edge weights to the range of $[0, 100]$ and solve every SSSP Problem in time $O(n + m)$. Choice B is correct because such property guarantees that in the priority queue, whenever we ExtractMin, the new smallest element is at most 100 plus the previous one (if the queue is not empty). Therefore we can implement the priority queue by a list sets, where each Set i contains all the elements with key value equal to i. We keep a pointer to the non-empty set with the smallest key value. Whenever ExtractMin makes this set empty, we move the pointer to the next non-empty set. This takes at most 100 moves – which costs constant time.

(h)( ) Boruvka's algorithm runs by iterations. In each iteration, a set of edges are selected for the minimum spanning tree. If we run Boruvka's algorithm on an n-vertex graph for r iterations, the best upper bound on the number of connected components in the graph consisting of the selected edges is

A. $n - r$.
B. $n - r - 1$.
C. n/r.
D. $n/2^r$.

Solution: D.

(i)( ) A fast implementation of priority queue will be most helpful to reduce the time complexity of one of the following algorithms for the minimum spanning tree problem.

A. Boruvka's algorithm.
B. Prim's algorithm.
C. The reverse-delete algorithm.

Solution: B. Prim's algorithm uses a priority queue.

Problem 2. Stable marriages [32 points] (3 parts)

Consider the following input instance consisting of 4 gentlemen (1, 2, 3, and 4) and 4 ladies (A, B, C, and D). Their preferences are listed as follows.

Gentlemen	Ladies
1	B > A > C > D
2	A > C > B > D
3	C > B > A > D
4	B > D > C > A

Ladies	Gentlemen
A	3 > 4 > 1 > 2
B	2 > 3 > 4 > 1
C	4 > 1 > 2 > 3
D	2 > 1 > 3 > 4

(a) [6 points] Is following matching stable? Why?

1 – D 2 – C 3 – B 4 – A

Solution: No, this is not a stable matching. (1, C) is an unstable pair (and (4, C) is the other unstable pair).

(b) [14 points] Run the propose-and-reject algorithm (with gentlemen proposing) for the input instance above by hand, and report the stable matching returned by the algorithm. You may also report the key steps during the execution of the algorithm, to secure partial credits.

Solution:

1. 1 proposes to B, 1 and B pair up.
2. 2 proposes to A, 2 and A pair up.
3. 3 proposes to C, 3 amd C pair up.
4. 4 proposes to B; B dumps 1; 4 and B pair up; 1 is single again.
5. 1 proposes to A; A dumps 2; 1 and A pair up; 2 is single again.
6. 2 proposes to C; C dumps 3; 2 and C pair up; 3 is single again.
7. 3 proposes to B; B dumps 4; 3 and B pair up; 4 is single again.
8. 4 proposes to D; 4 and D pair up
9. We get the following pairings 1 - A; 2 - C; 3 - B; 4 - D .

(c) [12 points] **[Warning: this is a harder problem!]** In the propose-and-reject algorithm with gentlemen proposing, is it possible for an input instance with n gentlemen and n ladies that more than one gentleman get their last choices (i.e. the last ladies in their preference lists)? Prove your answer.

Solution: No, it is not possible. Let us consider the first gentleman (namely X) who is matched with the last lady (namely A) in his preference list. When he proposes to A, X has been rejected by all other ladies, who are in relationships at that time. Then A accepts X (otherwise X would be unmatched and the algorithm would be wrong), this leads to all ladies matched, and therefore all gentlemen are matched. To summarize, the algorithm will immediately terminate when the first gentleman is matched with the last lady in his preference list, so there will not be a second gentleman matched in this way.

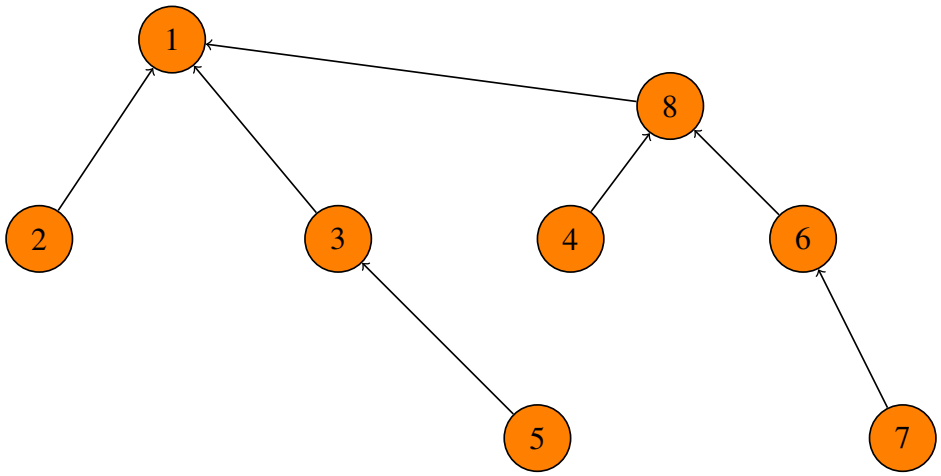
Problem 3. The Union-Find data structure [32 points] (5 parts) Recall that in the Union-Find data structure, $union(u, v)$ links root u to root v (v being the parent) if tree u has less nodes than tree v, and links v to u otherwise. $find(u)$ is implemented with the path compression technique.

For Parts c), d), and e), you need to use $\Theta(\cdot)$ notation to make your answer. I.e. your answer has to be in the form of $\Theta(f(n))$ where f is a function of n. You don't need to provide any proofs for your answer in these parts.

(a) [8 points] Suppose we have 8 elements: $1, 2, 3, \ldots, 8$. Initially, each element is in its own (distinct) set. Please state a sequence of $union(\cdot, \cdot)$ operations so that the resulting tree/forest has depth at least 3. Also you need to draw the tree/forest after the execution of your sequence of union operations.

Solution: The following steps will work.

1. $union(1, 2)$;
2. $union(3, 5)$;
3. $union(8, 4)$;
4. $union(6, 7)$;
5. $union(1, 3)$;
6. $union(8, 6)$;
7. $union(1, 8)$.



(b) [9 points] Suppose we have $n = 2^k$ elements: $1, 2, 3, \ldots, n$ (where $k \in \mathbb{N}$). Initially, each element is in its own (distinct) set. State an $O(n)$-time algorithm to generate a sequence of $union(\cdot, \cdot)$ operations so that the resulting tree/forest has depth at least k. You don't have to prove the correctness/runtime of the algorithm.

Solution: First divide the 2^k elements into two sets of 2^{k-1} elements. Recursively build 2 trees on these two sets, with depth $k - 1$. Then we union the root of these 2 trees, getting a tree with depth k.

Another possible solution is to start with n separate nodes, and every time choose 2 trees with the same size, and union their roots.

(c) [5 points] What is the worst-case time complexity for $find(u)$ after the execution of the sequence of union operations obtained in Part b) of this problem?

Solution: $\Theta(\log n)$.

(d) [5 points] Again, assume we are in the scenario when the sequence of operations in Part b) are executed. What is the time complexity of the following loop?

$$\textbf{for } u \leftarrow 1 \textbf{ to } n \textbf{ do } find(u);$$

Solution: $\Theta(n)$.

For any node u, $find(u)$ is called directly by the for loop once; and may be called recursively by $find(v)$ for some other v's several times. The number of the first type of calls is $\Theta(n)$. We also need to bound the number of the second type of calls by $O(n)$.

Observe that only the first time when $find(u)$ is called, there is a possibility of going into a long recursion. After the first time, $find(u)$ will always call $find(r)$ (where r is the root node) and directly return the root, because of path compression. Therefore, the total number of recursive calls is also at most $O(n)$.

(e) [5 points] What is the amortized time complexity of each $find(u)$ operation in Part d) (in that specific scenario)?

Solution: $\Theta(1)$.

Problem 4. Finding path with maximum bandwidth [30 points] (3 parts) An undirected graph $G = (V, E)$ with non-negative weights $w : E \to \mathbb{R}^{\geq 0}$ can be used to denote a computer network where nodes are computers, and an edge $e = (u, v)$ with weight $w(e)$ represents a communication link between u and v with bandwidth $w(e)$.

Two computers x and y without a direct link can also communicate via a path:

$$x(= z_0), z_1, z_2, \ldots, z_{k-1}, y(= z_k),$$

where there are direct links between adjacent nodes. The communication bandwidth of the path is the minimum bandwidth among all the links used in the path –

$$\min_{i \in \{0,1,2,\ldots,k-1\}} \{w(z_i, z_{i+1})\}.$$

(a) [12 points] Describe a polynomial-time algorithm that finds a path with maximum bandwidth between two computer nodes x and y (where x and y are given as input). You do not have to prove the correctness and running time of your algorithm in this part.

Solution:

Assume this graph is connected ($m \geq n - 1$). We can simply modify Dijkstra's algorithm to solve this problem. We use $\mathrm{LastNode}(u)$ to record the node right before node u in the path from x to u with maximum bandwidth. Here is the algorithm.

Algorithm 1 Problem 4

Input: $G = (V, E), w : E \to \mathbb{R}^{\geq 0}, x \in V, y \in V$
for all $u \in V$ **do**
  $\text{AppBand}(u) \leftarrow +\infty$
  $\text{LastNode}(u) \leftarrow \text{Null}$
end for
$S = \{x\}$
Initialize priority queue $Q(V, \text{AppBand})$
while $y \notin S$ **do**
  $u \leftarrow \text{ExtractMax}(Q)$
  $S \leftarrow S \cup \{u\}$
  for all $v \in \text{neighbors}(u)$ and $v \notin S$ **do**
    if $\min\{\text{AppBand}(u), w((u, v))\} > \text{AppBand}(u)$ **then**
      $\text{ChangeKey}(u, \min\{\text{AppBand}(u), w((u, v))\})$
      $\text{LastNode}(v) \leftarrow u$
    end if
  end for
end while
$\text{path} \leftarrow \emptyset$
$u \leftarrow y$
while $u \neq$ Null **do**
  $\text{path} \leftarrow \{u\} \cup \text{path}$
  $u \leftarrow \text{LastNode}(u)$
end while
Output: path

(b) [12 points] Prove the correctness of the algorithm you described in Part a).

Proof: Let $\mathrm{MaxBand}(u)$ denote the actual maximum bandwidth of all of the paths from x to u. We will use mathematical induction on the size of S to prove that for every node $u \in S$, $\mathrm{MaxBand}(u) = \mathrm{AppBand}(u)$.

The case $|S| = 1$ is easy, since then we have $S = x$ and $\mathrm{MaxBand}(x) = \mathrm{AppBand}(x) = +\infty$. Suppose the claim holds when $|S| = k$ for some value of $k \geq 1$; We now grow S to size $k + 1$ by adding the node v. Let $u = \mathrm{LastNode}(v)$. According to this algorithm, we know that path $x \sim u - v$ has bandwidth $\mathrm{AppBand}(v)$. Suppose there is another path $x \sim u' - v' \sim v$ (v' is the first node this path leaves S) with bandwidth more than $\mathrm{AppBand}(v)$. According to the definition of bandwidth, path $x \sim u' - v'$ must have bandwidth more than $\mathrm{AppBand}(v)$. Since u' is already in S, when we were adding node u', $\mathrm{AppBand}(v')$ would be updated to more than $\mathrm{AppBand}(v)$. Thus, we will have $\mathrm{AppBand}(v') > \mathrm{AppBand}(v)$ when we are adding node v, which is a contradiction according to the selection of node v. Therefore, this claim also holds when $|S| = k + 1$. And the proof is complete.

(c) [6 points] Let $n = |V|$ be the number of nodes in the graph and $m = |E|$ be the number of edges in the graph. Does the algorithm you described in Part a) run in time $O(m \log n)$? If yes, prove it; otherwise state how to improve the running time of the algorithm to $O(m \log n)$ and prove its running time.

Proof:

Suppose the input graph $G = (V, E)$ is in linked list representation. We first leave out the operations on the priority queue Q and analyze them together later. Then, for step 2 to step 5, running time is $O(|V|) = O(n)$. For step 6, running time is $O(1)$. For each iteration, we add a new node to S. Since there are totally n nodes, thus there are at most n iterations. For step 11 to step 16, basically, each node is at most visited twice. So totally, running time for step 11 to step 16 is $O(|E|) = O(m)$. Therefore, running time for step 8 to step 17 is $O(m + n)$. Next, for step 18 to step 24, running time is $O(n)$. To sum up, the overall running time for above steps is $O(m + n)$.

Next, we are going to analyze the running time for operations on priority queue. This depends on what data structure we will use to implement priority queue. Using the heap-based priority implementation, initialization (step 7) requires $O(n)$ time and each priority queue operation can be made to run in $O(\log n)$ time. Since there are at most m $\mathrm{ExtractMax}$ operations (step 9) and $2m$ $\mathrm{ChangeKey}$ operations (step 13), thus the overall time for these operations is $O(n) + O(m \log n) = O(m \log n)$.

And, the total running time for this implementation would be $O(m+n)+O(m \log n) = O(m \log n)$.

If we utilize Fibonacci heap to implement this priority queue, overall time for n $\mathrm{ExtractMin}$ operations and $2m$ $\mathrm{ChangeKey}$ operations would be $O(n \log n + m)$. And the total running time for this implementation would be $O(m+n)+O(n \log n + m) = O(m + n \log n)$ which is better than $O(m \log n)$ noting that $m \geq n - 1$.