Practice Final

- Do not open this exam until you are directed to do so. Read all the instructions first.
- When the exam begins, write your name on every page of this exam booklet.
- The exam contains 8 multi-part problems. You have 180 minutes to earn 160 points.
- This exam booklet contains 14 pages, including this one. Two extra sheets of scratch paper are attached. Please detach them before turning in your exam.
- This exam is closed book. You may use two handwritten A4 or $8\frac{1}{2}'' \times 11''$ crib sheets. No calculators or programmable devices are permitted.
- Show your work, as partial credit will be given. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it. Be neat.
- Good luck!

Problem	Points	Grade	Initials
1	12		
2	18		
3	27		
4	20		
5	16		
6	21		
7	26		
8	20		
Total	160		

Name: Solutions

Circle the name of your recitation instructor:

Problem -1. Recurrences [12 points]

For each of the following recurrences, do the following:

- Give the solution in $\Theta(\cdot)$ notation.
- Name a method that can be used to solve the recurrence. **(Do not give a proof.)**
- Mention a recursive algorithm we've seen in class whose running time is described by that recurrence.

(a) $T(n) = T(n/2) + \Theta(1)$

T(n) = $\Theta\left(\boxed{\log n}\right)$

Method: Master Method

Algorithm: Binary search

(b) $T(n) = T(n/5) + T(7n/10) + \Theta(n)$

T(n) = $\Theta\left(\boxed{n}\right)$

Method: Substitution

Algorithm: SELECT

(c) $T(n) = 7T(n/2) + \Theta(n^2)$

T(n) = $\Theta\left(\boxed{\Theta(n^{\lg 7})}\right)$

Method: by Master Method

Algorithm: Strassen's matrix-multiplication algorithm.

Problem -2. Design Decisions [18 points]

The use of algorithms often requires choices. What is efficient in one context may be suboptimal in another. For each of the following pairs, give *one* reason, circumstance, or application for which Choice 1 would be preferable to Choice 2, and vice-versa. Be succinct.

EXAMPLE: (1) insertion sort (2) merge sort

(1) Insert sort is preferrable on small arrays, or when space is a limited resource.
(2) Merge sort is preferable on large arrays, because of its asymptotically optimal $O(n \log n)$ running time.

(a) (1) randomized quicksort (2) bucket sort

Solution: (1) Randomized quicksort gives better worst-case running time when no distribution is known on the input.
(2) Bucket sort is better (linear-time) when the distribution on input elements is known to be uniform, or when randomness is not available.

(b) (1) randomized select (2) worst-case $O(n)$-time select

Solution: (1) Randomized select has smaller constant factors, so it is better suited for daily use when hard, worst-case guarantees on running time may not be needed.
(2) Linear-time select is preferred when a hard upper limit on the running time is necessary (as in a real-time application), or when randomness is not available.

(c) (1) red-black tree (2) hash table

Solution: (1) Red-black trees are useful for "approximate" searches, augmentation, and guaranteed worst-case running times.
(2) Hash tables support constant time (in expectation) insertions and deletions, and can be designed with guaranteed $O(1)$-time operations when the data set is known in advance.

Problem -3. Short Answer [27 points]

Give *brief*, but complete, answers to the following questions.

(a) Suppose you are given an unsorted array A of n integers, some of which may be duplicates. Explain how you could "uniquify" the array (that is, output another array containing each unique element of A exactly once) in $O(n)$ expected time.

Solution: We use universal hashing to solve this problem. Create a hash table of $2n$ elements, and for each element x in A, search for x in the table and insert it only if the search fails. Then walk down the slots of the table and output every element. The searches and insertions each take $O(1)$ expected time, and walking down the table takes $O(n)$ time, for a total expected runtime of $O(n)$.

(b) Given a list of distinct real numbers $z_0, z_1, \ldots, z_{n-1}$, show how to find the coefficients of the degree-n polynomial $P(x)$ that evaluates to zero only at $z_0, z_1, \ldots, z_{n-1}$. Your procedure should run in time $O(n \log^2 n)$. (*Hint:* The polynomial $P(x)$ has a zero at z_j if and only if $P(x)$ is a multiple of $(x - z_j)$.)

Solution: We multiply all the polynomials $(x - z_j)$ together, in the following divide-and-conquer way: recursively multiply the first $n/2$ linear terms, then recursively multiply the remaining $n/2$ linear terms, then combine the two results by multiplying them together using FFT. The running time is expressed by the recurrence $T(n) = 2T(n/2) + \Theta(n \log n)$. By the Master Method, this solves to $T(n) = \Theta(n \log^2 n)$.

(c) Consider a generalization of the max-flow problem, in which the network G may have many sources and many sinks. Explain how to reduce this problem to the conventional max-flow problem. Specifically, describe how you would convert the generalized network G to a conventional network G', and how you would translate a maximum flow in G' back to a maximum flow in G.

Solution: First, make a copy of G. Then add a new "supersource" vertex s, and a new "supersink" vertex t. Connect s to every old source with infinite-capacity edges, and connect every old sink to t with infinite-capacity edges. This graph is G'. Then run a conventional max-flow algorithm on G' with s as the source and t as the sink.
The flow produced is a max flow in G, discounting the edges between s and the old sources, and between the old sinks to t. This is because it is a valid flow, and the min-cut in G' is the same as in G (because only infinite-capacity edges were added).

Problem -4. True or False, and Justify [20 points]

Circle **T** or **F** for each of the following statements, and briefly explain why. The better your argument, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

(a) T F If a problem L is in NP and L reduces to $3SAT$ ($L \leq_P 3SAT$), then L is NP-complete. (You may assume $P \neq NP$.)

Solution: False. Every problem in P reduces to $3SAT$, and if $P \neq NP$ then those problems are not NP-complete. The statement *would* be true if the reduction went the other way (if $3SAT$ reduced to L).

(b) T F If a graph has negative-weight edges, there exists a constant C such that adding C to every edge weight and running Dijkstra's algorithm produces shortest paths under the original edge weights.

Solution: False. Adding the same value to each edge "distorts" the shortest paths, because each path becomes longer in proportion to the number of its edges. For concreteness, suppose the shortest path between s and t was 3 "hops," and was 1 unit shorter than the edge directly from s to t. Furthermore, if there was an edge somewhere in the graph with weight -1, we would need $C \geq 1$ for Dijkstra's algorithm to be correct. Adding C to each edge causes the 1-hop path from s to t to be the shortest.

(c) T F It is possible to compute the convolution of two vectors, each with n entries, in $O(n \lg n)$ time.

Solution: True. Use polynomial multiplication via FFT.

(d) T F The following procedure produces a minimum spanning tree of n given points in the plane.

- Sort the points by x-coordinate. (Assume that all x-coordinates are distinct.)
- Connect each point to its closest neighbor among all points with smaller x coordinate.
  That is, if $p_1, p_2, \ldots, p_n$ denotes the sorted order of points, then for $2 \leq i \leq n$, connect point p_i to its closest neighbor among $p_1, p_2, \ldots, p_{i-1}$.

Solution: False. Here is a counterexample: take the points $(0, 0)$, $(10, 100)$, and $(11, 0)$ in the xy-plane. Then in the first step, the edge between $(0, 0)$ and $(10, 100)$ is chosen; however, this is the longest of the three edges between points. We know that an MST cannot contain the heaviest edge in a graph, therefore this algorithm is incorrect.

Problem -5. Average Path Lengths in DAGs [16 points]

Instead of shortest paths in a graph, you might be interested in the *average* length of all paths from a vertex s to a vertex t. In order for this to make sense, we assume that the graph $G = (V, E)$ is directed and acyclic. Let $w(u, v)$ denote the weight of edge (u, v). It suffices to compute the total length of all paths from s to t, and the number of such paths. This problem is well-suited to a dynamic programming approach.

(a) [8 points] Define *count*$[x]$ to be the number of distinct paths from x to t. Define *sum*$[x]$ to be the sum of the lengths of all paths from x to t. Give recurrences for *count*$[x]$ and *sum*$[x]$ in terms of the neighbors of x and the edge weights, and give the bases cases as well.

Solution: The paths from x to t can be partitioned based on the neighbor y of x visited first in the path. There are *count*$[y]$ distinct paths to take from that point onward. The corresponding paths from x are each $w(x, y)$ longer. This yields the recurrences:

$$\begin{aligned} \mathit{count}[x] &= \sum_{y\,:\,(x,y)\in E} \mathit{count}[y] \\ \mathit{sum}[x] &= \sum_{y\,:\,(x,y)\in E} (\mathit{count}[y]\cdot w(x,y) + \mathit{sum}[y]). \end{aligned}$$

The base cases are *count*$[t] = 1$ and *sum*$[t] = 0$.

(b) [8 points] Describe a dynamic-programming algorithm to solve the stated problem, and analyze its running time. (*Hint:* In what order should you consider the vertices, so that when considering vertex x, the values of *count*$[y]$ and *sum*$[y]$ have already been computed for all neighbors y of x?)

Solution: We first compute a topological ordering on the vertices, relabelling them $1, \ldots, |V|$, so that all edges $(i, j) \in E$ have $i < j$. Then we build tables for *count* and *sum*, filling them in from V down to 1. Values in both tables are set to 0, until vertex t is encountered, at which point we set the table entries according to the base cases. We then use the recurrences to fill in the remaining entries. This is possible because when computing the entries for vertex x, all neighbors y of x are greater than x, and have had their entries computed already. After filling in the tables, we return *sum*$[s]/$*count*$[s]$ as the average.

The runtime analysis is as follows: topological sort requires $\Theta(V + E)$ time. While filling in the tables, each vertex and edge causes a constant number of math operations to be done, for $\Theta(V + E)$ time overall.

Problem -6. Amortized 2-3-4 Trees [21 points]

In this problem, we will analyze the amortized number of modifications made to a 2-3-4 tree during an INSERT operation. For the purpose of this problem, assume that DELETE operations do not occur.

(a) [3 points] Give a tight asymptotic bound on the number of nodes created or modified during one call to INSERT, in the worst case.

Solution: $\Theta(\log n)$ nodes could be changed, because the height of the tree is $\Theta(\log n)$ and each node in the insertion path could be full and need to be split.

(b) [6 points] Suppose we insert an element into a 2-3-4 tree, and the INSERT algorithm splits k nodes. Give an *exact* (not big-O) upper bound on the number of nodes in the tree that are created or modified in this case.

Solution: Each split creates 2 new nodes (by splitting one), and modifies the parent of the old node (by promoting one key). In addition, the leaf node that gets the inserted key is modified (whether or not there are any splits). This yields a total of at most $3k + 1$ modified nodes (alternatively, we could say $\max(3k, 1)$).

(c) [6 points] Let T be a 2-3-4 tree, and define a potential function

$$\phi(T) = 3 \times (\text{number of “full” nodes in } T)$$

(a node is full if it stores 3 keys).

If a call to INSERT splits k nodes, how does $\phi(T)$ change as a result of this call?

Solution: Each node that is split was full before the split, and is not full afterward. No other full nodes are modified; however, the key that is promoted in the final split may fill a node. Therefore the number of full nodes decreases by $k - 1$, so $\phi(T)$ decreases by at least $3(k - 1)$. (Note: all of these statements remain true when $k = 0$ or $k = 1$; in these cases ϕ “decreases” by a small, non-positive amount.)

(d) [6 points] Prove that the amortized number of nodes created or modified per INSERT is $O(1)$.

Solution: First, we note that $\phi(T) = 0$ on newly-initialized B-tree, and that $\phi(T) \geq 0$ by definition. Therefore, the amortized number of modified nodes per INSERT is the actual number of modified nodes, plus the change in $\phi(T)$. This is at most $3k + 1 - (3k - 3) = 4 = O(1)$.

Problem -7. Maximum Bottleneck Path [26 points]

In the ***maximum-bottleneck-path*** problem, you are given a graph G with edge weights, and two vertices s and t, and your goal is to find a path from s to t whose minimum edge weight is maximized. In other words, you want to find a path from s to t in which *no* edge is light.

(a) [8 points] Suppose that all edges have nonnegative weights. How would you modify a shortest-path algorithm that we covered in lecture to solve the maximum-bottleneck-path problem?

Solution: We would like to use Dijkstra's algorithm, where $d[v]$ keeps a *lower* bound on the *maximum* bottleneck weight of a path from s to v. Therefore, in the initialization step, we set $d[v] \leftarrow -\infty$ for all v, and $d[s] \leftarrow \infty$. In the relaxation step, we change $+$ to $\min$ and change $>$ to $<$. In Dijkstra's algorithm, use a max-heap instead of a min-heap.

(b) [4 points] Does your solution change if the edges have negative weights? What if there are negative-weight cycles?

Solution: Neither of these are a problem. The minimum-weight edge of a path is not changed by going around a cycle several times. Because we are only using $\min$ instead of addition, the relative values of the edges are all that matter, not whether they are positive or negative.

(c) [6 points] Suppose we do not need a path which maximizes the minimum edge weight, but we only need a path in which every edge has at least a certain weight. Describe an $O(V + E)$-time algorithm for finding a path from s to t in which every edge has least a given minimum weight $w_{\min}$. (Such an algorithm would have been useful in the movie *Speed*, in which every road traversed had to have a speed limit of at least 50 mph.)

Solution: We can find a path using breadth-first or depth-first search, while simply deleting (or ignoring) edges with weight smaller than $w_{\min}$.

(d) [8 points] Describe how you can make $O(\lg E)$ calls to the algorithm in part **(c)** to solve the maximum-bottleneck-path problem in $O((V + E) \lg E)$ time.

Solution: First, sort all the edges by weight (there are at most $|E|$ unique weights). The desired weight must be a particular edge weight, so we binary search for the largest value of $w_{\min}$ that maintains a path from s to t (using the algorithm from the previous part to detect whether such a path exists). Specifically, start with $w_{\min}$ set to the median edge weight, then try the $E/4$th or $3E/4$th largest edge weight, etc.

Sorting the edge weights requires $O(E \log E)$ time. The binary search requires $O(\log E)$ iterations of an $O(V + E)$-time subroutine, so the total running time is $O((V + E) \log E)$.

Problem -8. Cliquependent Graphs [20 points]

Given a graph $G = (V, E)$ and nonnegative integers c, s, we say that G is (c, s)-***cliquependent*** if *both* of the following are true:

- there exists a subset $C \subseteq V$ such that $|C| = c$ and, for all distinct $i, j \in C$, $(i, j) \in E$, and
- there exists a subset $S \subseteq V$ such that $|S| = s$ and, for all distinct $i, j \in S$, $(i, j) \notin E$.

Given a graph G and a pair (c, s), the ***cliquependence decision problem*** is to determine whether G is (c, s)-cliquependent.

(a) [3 points] Define the set CLIQUEPENDENT which contains all "yes" instances to the cliquependence decision problem.

Solution:

$$\text{CLIQUEPENDENT} = \{\langle G, c, s\rangle \ : \ G \text{ is } (c, s)\text{-cliquependent}\}$$

(b) [6 points] Show that CLIQUEPENDENT $\in \mathrm{NP}$.

Solution: A witness for an instance is a subset C and a subset S which have the properties listed above (C is a clique of size c, and S is an independent set of size s). The verification algorithm checks that there is an edge between every pair of vertices in C, and that there are no edges between any pair of vertices in S, and accepts the witness only if all the conditions are met. Conversely, if the verifier accepts, then C and S have the properties describe above, and G is (c, s)-cliquependent.

The witness is clearly polynomial-sized in the size of the instance, and the verification algorithm runs in polynomial time.

(c) [7 points] Show that CLIQUEPENDENT is NP-complete. (*Hint:* Reduce from *either* CLIQUE or INDEPENDENT-SET.

Solution: We can reduce from CLIQUE: given an instance $x = \langle G, k\rangle$ of CLIQUE ("is there a clique of size k in G?"), the reduction outputs an instance $f(x) = \langle G, k, 0\rangle$ of CLIQUEPENDENT. Trivially, G has an independent set of size 0, so $f(x) \in$ CLIQUEPENDENT if and only if G has a clique of size k, i.e. if and only if $x \in$ CLIQUE. This reduction is obviously polynomial-time, and because CLIQUE is NP-complete, so is CLIQUEPENDENT.

We can also reduce from INDEPENDENT-SET in a similar way: on input instance $x = \langle G, s\rangle$ of INDEPENDENT-SET, the reduction outputs $f(x) = \langle G, 0, s\rangle$. Because G has a clique of size 0, $x \in$ INDEPENDENT-SET $\iff f(x) \in$ CLIQUEPENDENT.

(d) [4 points] Suppose an $O(n^{100})$-time algorithm were found for the cliquependence decision problem. What would be the implications, if any, on the "P $\stackrel{?}{=}$ NP" question?

Solution: If such a procedure exists, then P $=$ NP with certainty. This is because any language in NP can be reduced to the cliquependence problem, then the algorithm in question would correctly answer "yes" or "no." Therefore any problem in NP could be solved in polynomial time, implying P $=$ NP.

SCRATCH PAPER — Please detach this page before handing in your exam.

SCRATCH PAPER — Please detach this page before handing in your exam.