15-451 Algorithms, Fall 2000

Final Exam **Name:** ______________________

There are 8 problems, for a total of 180 points. You have 180 minutes. One sheet of notes is allowed. No textbook. No calculators.

Problem	1	2	3	4	5	6	7	8	total
Score Out of	 20	 20	 32	 22	 32	 10	 24	 20	 180

---

1. Short-answer questions. (20 pts)

(a) Merging 10 sorted lists, of n elements each, takes time: (circle the fastest possible using a comparison-based algorithm)

$$\Theta(n) \qquad \Theta(n \log n) \qquad \Theta(n^2) \qquad \Theta(2^n)$$

(b) Merging n sorted lists, of 10 elements each, takes time: (circle the fastest possible using a comparison-based algorithm)

$$\Theta(n) \qquad \Theta(n \log n) \qquad \Theta(n^2) \qquad \Theta(2^n)$$

(c) Determining if an n-vertex m-edge graph (represented as an adjacency list data structure) has an odd-length cycle can be done in time (circle the fastest possible):

$$O(n) \qquad O(n+m) \qquad O(m \log n) \qquad O(mn) \qquad O(n^2) \qquad O(m^2)$$

(d) Ford-Fulkerson takes at most F iterations, where F is the value of the maximum flow. Each iteration involves finding a path from S to T in the residual graph, which takes time $O(m)$. Why isn't Ford-Fulkerson in general viewed as a "polynomial time" algorithm?

(e) Suppose we use the Ford-Fulkerson algorithm to solve the perfect matching problem on a $2n$-node bipartite graph with m edges. Is Ford-Fulkerson polynomial-time in this case, and what is its running time?

2. Fractional SAT. (20 pts)

One heuristic for determining if a CNF formula is satisfiable is to solve the *fractional version* of the problem. We allow variables x_i to take on fractional values with $0 \leq x_i \leq 1$, we define $\bar{x}_i = 1 - x_i$, and we replace the clause $(a \vee b \vee c)$ with the inequality $a + b + c \geq 1$. We then plug these constraints into a Linear Programming box to see if there exists a feasible solution. For each statement below, say if it is True or False, and if false give a counterexample.

(a) If the fractional version has *no* solution, then the original formula was *not* satisfiable.

(b) If the fractional version *has* a solution then the original formula *was* satisfiable.

3. Short-answer number-theory questions. (32 pts)

(a) What is $3^{-1} \bmod 11$?

(b) What is $540^{-1} \bmod 541$?

(c) List all square roots of 1 mod 15.

(d) What is $\phi(77)$?

(e) What is $4^{61} \bmod 77$?

In order to prove that an interactive proof is zero-knowledge, it's often necessary to prove that two different ways of randomly generating an object are the same (they select the object from the same distribution). For each of the following parts, state "SAME" if the two methods of generating an object are the same in this sense, or "NOT SAME" if they are not necessarily the same.

(f) n is fixed.

(1) Pick a random element in Z_n^*

(2) Pick a random element in Z_n^*, then square it mod n.

(g) n is fixed, d is an element of Z_n^*.

(1) Pick a random element in Z_n^*

(2) Pick a random element in Z_n^*, and multiply it by d and take the result mod n.

(h) n is fixed, d is an element of Z_n^*.

(1) Pick a random element in Z_n^*

(2) Pick a random element in Z_n^*, and add it to d and take the result mod n.

4. Edit distance. (22 pts)

Given two strings S and T, consider the problem of converting S into T using the fewest number of operations. The operations we allow are removing a character (at a cost of 1) or adding a character (at a cost of 1). So, dist(ABC,BCA) = 2. For a string S of length n, define first(S) to be the first character of S, and rest(S) to be the remaining $n-1$ characters of S. It's not hard to show that if first(S)=first(T), then dist(S,T) = dist(rest(S), rest(T)).

(a) Give a counterexample to the following false statement: if first(S)$\neq$first(T) then dist(S,T) = 1 + dist(rest(S),rest(T)).

(b) The true statement is that if first(S)$\neq$first(T) then

$$\text{dist}(S,T) = 1 + \min(\text{dist}(S, \text{rest}(T)), \text{dist}(\text{rest}(S), T))$$

Use this fact to give an $O(n^2)$-time algorithm to determine the edit distance between two strings of length n.

5. Rent-or-buy. (32 pts)

Consider the rent-or-buy problem in the simple case that the purchase cost is twice the rental cost. For concreteness, assume it costs \$50 to rent and \$100 to buy. The optimal deterministic algorithm (as discussed in class) is: "rent the first time, buy the second time" for a competitive ratio of $3/2$. A generic *randomized* algorithm would look like this: "the first time, buy with probability p_1. The second time, if we don't already own it, then buy with probability p_2, and so on."

Your job is to solve for these probabilities $p_1, p_2, \ldots$ in order to achieve the best possible *randomized* competitive ratio. You may assume (and this is not too hard to prove) that the only two cases you need to worry about are (1) that you end up only using the item once, or (2) that you end up using the item infinitely many times. Let E_1 be the expected cost of your on-line algorithm in case (1) and E_2 be the expected cost in case (2). Let C_1 be the optimal cost in hindsight in case (1) and C_2 be the optimal cost in hindsight in case (2). So, the competitive ratio of your algorithm is $\max(E_1/C_1, E_2/C_2)$.

(a) What are C_1 and C_2?

(b) What should p_2 be (assuming that $p_1 \neq 1$)? Give a short explanation.

(c) Fixing p_2 to the value computed in (b), express E_1 and E_2 as a function of p_1.

(d) Now, solve for p_1 to minimize the competitive ratio of your algorithm. Also, say what the competitive ratio is.

(e) Consider the algorithm that sets $p_1 = 1/2, p_2 = 1/2, p_3 = 1/2, \ldots$. What is its competitive ratio?

6. Edge-disjoint paths. (10 pts)

Given a graph G (undirected, unweighted) and a collection of pairs $(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$, the *edge-disjoint paths problem* is to find a path from s_1 to t_1, a path from s_2 to t_2, and so on, so that no edge in the graph is used more than once. It turns out that this problem is NP-complete. However, if we don't care which start node s_i is connected to which terminal node t_j (so long as each s_i is connected to some t_j and each t_j is connected to some s_i) then the problem can be solved in polynomial time. Explain why by reducing this to something we know how to solve efficiently.

7. Data structures. (24 pts)

Suppose we want to store a set of numbers with the following operations:

insert(x) Inserts the number x into the set.

ismember(x) Returns True if x is in the set, else False.

successor(x) Returns the next largest number after x in the set.

findrank(x) Returns the rank of x. (The rank is a number k such that there are $k-1$ elements smaller than x in the set.

For each data structure below, list which of the above operations that data structure would perform *poorly* on. Poorly means that the operation could take $\Omega(n)$ time.

(a) A sorted array

(b) A hash table

(c) A balanced search tree (like a B-tree)

(d) A heap

We can augment a balanced search tree data structure so that *all* the above operations run in $O(\log n)$ time by storing some extra information in each node.

What information should be stored?

How would you update that information upon performing an insert?

8. Amortized analysis. (20 pts)

Program designer and almost-president-elect Al Goreithm wants to implement a stack as an array. In class we talked about doubling the size of the array whenever it becomes full. However, in order to be more efficient on space, the VP decides instead to multiply the size by $3/2$.

Assume that increasing the size of the array from n to $3n/2$ costs n. So, the *actual cost* of a push is 1 if the array was not full, or $1 + n$ if the array was full and had size n.

(a) What is the *amortized cost* of a push? Explain your answer.

(b) Suppose we want to make sure the array never becomes less than $2/3$ full. One possible way to do this is to shrink the array from size n to size $2n/3$ whenever we pop from an array that is $2/3$-full. Does this yield an amortized cost of $O(1)$ per operation? Explain.

(c) Another approach is to reduce the size from n to $5n/6$ whenever we pop from an array that is $2/3$-full. Does this yield an amortized cost of $O(1)$ per operation? Explain.