Final Assignment

Handed Out: Thursday, May 7 1998
Due Date: **Monday, May 18 1998, 5:00pm prompt**

COLLABORATION POLICY

You may refer to the CLR text, any handouts from this class, or notes you have taken during the class, but you should not use any other outside materials. Also, you are not allowed to discuss the problems with any other people. Feel free to contact the TAs or the instructor for questions regarding the statement of the problems; however they can not give you any direct help in solving the problems. Of course you may certainly ask them for clarifications regarding any topic that we have covered in the class.

PRINCETON UNIVERSITY HONOR CODE

Please sign below, and staple this cover sheet to your solutions.

"I have completed this final assignment in accordance with the above policy."

SIGN YOUR NAME ABOVE THIS LINE

PRINT YOUR NAME ABOVE THIS LINE

(for grader's use only)

Problem	Points	Grade
1	25	
2	25	
3	25	
4	25	
Total	100	

Problem 1. (25 points)

Binary search of a sorted array takes logarithmic search time, but the time to insert or delete elements is linear in the size of the array. We have seen many methods to better balance the costs of these operations, and in this problem we analyze one possible approach.

To represent a set of n elements, we will make use of k different arrays $A_0, A_1, \ldots, A_k$ for $k = \lceil \lg(n+1) \rceil$, where each array A_i has space for storing 2^i elements. We will keep each array either full or empty. If the binary representation of n is $\langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle$, then by filling each array A_i for which $n_i = 1$, we will have room for exactly n items. For those arrays which are filled, we want to maintain that the items within that array are in sorted order, however we make no assumptions about how items in one array relate to items in another array.

a. (4 points)
Describe how to perform the SEARCH operation for this data structure in $O(\lg^2 n)$ worst-case time. (justify your answer).

b. (8 points)
Describe how to insert a new element into this data structure in $O(\lg n)$ amortized time. (justify your answer).

c. (4 points)
Assume we wish to implement the DELETE operation (where we assume, as always, that the operation is given some pointer which identifies the current location of the element to be deleted). Argue that if we insist on using a structure where all of the arrays are either empty or full, there will always be a sequence of k operations which require $\Omega(kn)$ time, and thus $\Omega(n)$ amortized time.

d. (9 points)
If we allow you to relax the restriction that all arrays are either full or empty, show how to implement DELETE in $O(1)$ amortized time, while still maintaining the previous time bounds for SEARCH and INSERT. Make sure that you justify not only the analysis of DELETE, but also why the previous bounds for the other operations are still valid.

Problem 2. (25 points)

Consider the following model for speech recognition. We are given a directed graph $G = (V, E)$, where each edge $(u, v) \in E$ is labeled with a sound $\sigma(u, v)$ from a finite set Σ of sounds. The labeled graph is a formal model for a person speaking a restricted language. Each path in the graph starting from a distinguished vertex $v_0 \in V$ corresponds to a possible sequence of sounds produced by the model. The label of a directed path is defined to be the concatenation of the labels of the edges on that path.

a. (15 points)
Describe an efficient algorithm that, given an edge-labeled graph G with distinguished vertex v_0 and a sequence $s = \langle \sigma_1, \sigma_2, \ldots, \sigma_k \rangle$ of characters from Σ, returns a path in G that begins at v_0 and has s as its label, if any such path exists. Otherwise, the algorithm should return NO-SUCH-PATH. Analyze the running time of your algorithm.

Now, suppose that ever edge $(u, v) \in E$ has also been given an associated nonnegative probability $p(u, v)$ of traversing the edge (u, v) from vertex u and producing the corresponding sound. The sum of the probabilities of the edges leaving any vertex equals 1. The probability of a path is defined to be the product of the probabilities of its edges. We can view the probability of a path beginning at v_0 as the probability that a "random walk" beginning at v_0 will follow the specified path, where the choice of which edge to take at vertex u is made probabilistically according to the probabilities of the available edges leaving u.

b. (10 points)
Extend your answer to part (a) so that if a path is returned, it is a *most probable path* staring at v_0 and having label s. Analyze the running time of your algorithm.

Problem 3. (25 points)

On HW9, we saw that a scaling algorithm could be used to get an approximation scheme for the Knapsack problem. Here, we will see another algorithm which does not guarantee as good of an approximation, but is much, much simpler. For a formal definition of the problem and notations, please refer to HW9. We consider the following three algorithms:

Algorithm1 — arrange the items in decreasing order of value per pound ($\frac{v_i}{w_i}$). Now greedily fill your bag based on this order, placing each item in your bag if it fits in the remaining available space. (See Chapter 17.2)

Algorithm2 — Simply take the single item with the highest overall value v_i. (recall, we assume that any individual item fits in the bag when alone)

Algorithm3 — Run Algorithm1. Separately run Algorithm2. Use whichever solution is the better one.

a. (5 points)
Show that Algorithm1 may return an arbitrarily bad approximation. That is, for any constant c, prove that some instance $\mathcal{I}$ exists such that,
$$\text{Value}_{\text{opt}}(\mathcal{I}) \geq c \cdot \text{Value}_{\text{Algorithm1}}(\mathcal{I}).$$

b. (5 points)
Show that Algorithm2 may return an arbitrarily bad approximation. That is, for any constant c, prove that some instance $\mathcal{I}$ exists such that,
$$\text{Value}_{\text{opt}}(\mathcal{I}) \geq c \cdot \text{Value}_{\text{Algorithm2}}(\mathcal{I}).$$

c. (10 points)
Show that Algorithm3 returns a 2-approximation. That is, prove that for any instance $\mathcal{I}$,
$$\text{Value}_{\text{opt}}(\mathcal{I}) \leq 2 \cdot \text{Value}_{\text{Algorithm3}}(\mathcal{I}).$$

(*Hint:* Consider the first time that an item does not fit during Algorithm1. Even if we stopped right then and took the better solution between this and Algorithm2, we could already prove the desired result.

Think about the amount of value already taken, and think about the amount of wasted space and what the optimal fractional solution would be (see Chapter 17.2). This should get you a lower bound on the optimal integer solution, and then you have to argue why Algorithm3's solution is at least half of that lower bound.)

d. (5 points)
Show that Algorithm3 can not guarantee to do better than a 2-approximation. That is, for any constant $\epsilon > 0$, prove that an instance $\mathcal{I}$ exists such that,
$$\text{Value}_{\text{opt}}(\mathcal{I}) \geq (2 - \epsilon) \cdot \text{Value}_{\text{Algorithm3}}(\mathcal{I}).$$

Problem 4. (25 points)

In class, we discussed the *edge-coloring* problem. Given an undirected graph $G(V, E)$, the goal is to color each edge of the graph, with the restriction that two edges cannot have the same color if they meet at a vertex. If Δ denotes the maximum vertex degree throughout the graph, it is clear that any solution will need to use at least Δ colors (think about the vertex with Δ edges). At the same time it was proven that there is a polynomial algorithm which can edge-color any graph using at most $\Delta + 1$ colors, and yet it is NP-hard to decide whether the best solution uses Δ or $\Delta + 1$ colors.

For this assignment, we consider the *online* edge-coloring problem. The input is a graph $G(V, E)$ with maximum degree Δ, however you do not get to see the graph all at once. Instead, you start with the empty graph on the vertices V, and you get a list of all edges of E, ordered arbitrarily. The rule is that you see the first edge on the list and you must immediately pick a color for that edge. Then you see the second edge on the list, and you must immediately pick a color for that edge. This continues until you reach the end of the edge list.

a. (10 points)
Give a deterministic online algorithm which produces a legal edge-coloring, using at most $2\Delta - 1$ colors for any graph $G(V, E)$ and any order of the edge list. Prove your claim.

We want to show that this is the best possible deterministic online algorithm. Namely we want to show that for any deterministic algorithm A and any value Δ, an adversary can create a graph $G(V, E)$ with maximum degree Δ, such that algorithm A uses $2\Delta - 1$ colors.

Our adversary starts by presenting the algorithm with a forest of β "stars", where each star consists of a center vertex, with an edge to $\Delta - 1$ other vertices. At this point, the adversary looks to see how the algorithm colored each of the stars, and it tries to find Δ different stars, each of which uses the same group of $\Delta - 1$ colors. It then uses one new vertex v and creates an edge between v and each of the centers of these special Δ stars.

b. (5 points)
Assume, for contradiction, that A uses at most $2\Delta - 2$ different colors. How many "different looking" stars can possibly exist? Argue that you can choose a value of β so that the algorithm will be forced to create Δ identical looking stars.

c. (10 points)
Argue that the final graph $G(V, E)$ does indeed have maximum degree equal to Δ, and that algorithm A will be forced to use at least $2\Delta - 1$ distinct colors.