CS 373: Combinatorial Algorithms, Spring 1999

http://www-courses.cs.uiuc.edu/~cs373

Homework 2 (due Thu. Feb. 18, 1999 by noon)

Name:	
Net ID:	Alias:

Everyone must do the problems marked ▶. Problems marked ▷ are for 1-unit grad students and others who want extra credit. (There's no such thing as "partial extra credit"!) Unmarked problems are extra practice problems for your benefit, which will not be graded. Think of them as potential exam questions.

Hard problems are marked with a star; the bigger the star, the harder the problem.

Note: When a question asks you to "give/describe/present an algorithm", you need to do four things to receive full credit:

1. Design the most efficient algorithm possible. Significant partial credit will be given for less efficient algorithms, as long as they are still correct, well-presented, and correctly analyzed.
2. Describe your algorithm succinctly, using structured English/pseudocode. We don't want full-fledged compilable source code, but plain English exposition is usually not enough. Follow the examples given in the textbooks, lectures, homeworks, and handouts.
3. Justify the correctness of your algorithm, including termination if that is not obvious.
4. Analyze the time and space complexity of your algorithm.

Undergrad/.75U Grad/1U Grad Problems

▶1. Faster Longest Increasing Subsequence (LIS)
Give an $O(n \log n)$ algorithm to find the longest increasing subsequence of a sequence of numbers. Hint: In the dynamic programming solution, you don't really have to look back at all previous items.

▶2. SELECT(A, k)
Say that a binary search tree is *augmented* if every node v also stores $|v|$, the size of its subtree.

(a) Show that a rotation in an augmented binary tree can be performed in constant time.

(b) Describe an algorithm SCAPEGOATSELECT(k) that selects the kth smallest item in an augmented scapegoat tree in $O(\log n)$ *worst-case* time. (The scapegoat trees presented in class were already augmented.)

(c) Describe an algorithm SPLAYSELECT(k) that selects the kth smallest item in an augmented splay tree in $O(\log n)$ *amortized* time.

(d) Describe an algorithm TREAPSELECT(k) that selects the kth smallest item in an augmented treap in $O(\log n)$ *expected* time.

►3. Scapegoat trees

(a) Prove that only one tree gets rebalanced at any insertion.

(b) Prove that $I(v) = 0$ in every node of a perfectly balanced tree ($I(v) = max(0, |\hat{v}| - |\check{v}|)$, where $\hat{v}$ is the child of greater height and $\check{v}$ the child of lesser height, $|v|$ is the number of nodes in subtree v, and perfectly balanced means each subtree has as close to half the leaves as possible and is perfectly balanced itself.

*(c) Show that you can rebuild a fully balanced binary tree in $O(n)$ time using only $O(1)$ additional memory.

►4. Memory Management
Suppose we can insert or delete an element into a hash table in constant time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

- After an insertion, if the table is more than 3/4 full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
- After a deletion, if the table is less than 1/4 full, we we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

Show that for any sequence of insertions and deletions, the amortized time per operation is still a constant. Do not use the potential method (it makes it much more difficult).

Only 1U Grad Problems

▷1. Detecting overlap

(a) You are given a list of ranges represented by min and max (e.g. [1,3], [4,5], [4,9], [6,8], [7,10]). Give an $O(n \log n)$-time algorithm that decides whether or not a set of ranges contains a pair that overlaps. You need not report all intersections. If a range completely covers another, they are overlapping, even if the boundaries do not intersect.

(b) You are given a list of rectangles represented by min and max x- and y- coordinates. Give an $O(n \log n)$-time algorithm that decides whether or not a set of rectangles contains a pair that overlaps (with the same qualifications as above). Hint: sweep a vertical line from left to right, performing some processing whenever an end-point is encountered. Use a balanced search tree to maintain any extra info you might need.

Practice Problems

1. Amortization

   (a) Modify the binary double-counter (see class notes Feb. 2) to support a new operation Sign, which determines whether the number being stored is positive, negative, or zero, in constant time. The amortized time to increment or decrement the counter should still be a constant.

   [Hint: Suppose p is the number of significant bits in P, and n is the number of significant bits in N. For example, if $P = 17 = 10001_2$ and $N = 0$, then $p = 5$ and $n = 0$. Then $p - n$ always has the same sign as $P - N$. Assume you can update p and n in $O(1)$ time.]

   *(b) Do the same but now you can't assume that p and n can be updated in $O(1)$ time.

*2. Amortization
Suppose instead of powers of two, we represent integers as the sum of Fibonacci numbers. In other words, instead of an array of bits, we keep an array of "fits", where the ith least significant fit indicates whether the sum includes the ith Fibonacci number F_i. For example, the fit string 101110 represents the number $F_6 + F_4 + F_3 + F_2 = 8 + 3 + 2 + 1 = 14$. Describe algorithms to increment and decrement a fit string in constant amortized time. [Hint: Most numbers can be represented by more than one fit string. This is *not* the same representation as on Homework 0.]

3. Rotations

   (a) Show that it is possible to transform any n-node binary search tree into any other n-node binary search tree using at most $2n - 2$ rotations.

   *(b) Use fewer than $2n - 2$ rotations. Nobody knows how few rotations are required in the worst case. There is an algorithm that can transform any tree to any other in at most $2n - 6$ rotations, and there are pairs of trees that are $2n - 10$ rotations apart. These are the best bounds known.

4. Fibonacci Heaps: SECONDMIN
We want to find the second smallest of a set efficiently.

   (a) Implement SECONDMIN by using a Fibonacci heap as a black box. Remember to justify its correctness and running time.

   *(b) Modify the Fibonacci Heap data structure to implement SECONDMIN in constant time.

5. Give an efficient implementation of the operation **Fib-Heap-Change-Key(**H, x, k**)**, which changes the key of a node x in a Fibonacci heap H to the value k. The changes you make to Fibonacci heap data structure to support your implementation should not affect the amortized running time of any other Fibonacci heap operations. Analyze the amortized running time of your implementation for cases in which k is greater than, less than, or equal to $key[x]$.