Homework #1: Introduction, Asymptotics,Recurrences, Amortization
Due Date: Tuesday, 28 January 2003

Guidelines

Please make sure you adhere to the policies on collaboration and academic honesty as outlined in Handout #1.

A note about problem numbers in CLRS

There is a somewhat confusing numbering system for problems from the textbook. The book has what it terms "Exercises" which are at the end of each section, and "Problems" which are at the very end of each chapter. For example, Exercise 3.1-1 is at the end of Section 3.1 on page 50, whereas Problem 3-1 is at the end of Chapter 3, on page 57.

Reading

Read Ch. 1–4 and Ch. 17 of CLRS, as well as the lecture notes regarding the maximum subarray problem.

Practice

These exercises are purely for your own practice. You should not turn them in, and you are free to discuss them fully with others.

- Do CLRS 3.1-1, 3.1-2.
- Do CLRS 3.1-4.
- Do CLRS 3.1-6.
- Do CLRS 4.2-4, 4.2-5.
- Do CLRS 4.3-1.

Problems

Problem A "You may discuss ideas with other students."

We discussed five algorithms for the maximum subarray problem, based on Chapter 8 of *Programming Pearls* by Jon Bentley. Our goal for this problem is to run experiments to gather data on the efficiency of the algorithms. Fortunately, the author provides source code in C for the five algorithms (a link can be found from the lecture notes). You may rely on that source code if you wish, though you may need to make some alterations. You may also choose to develop your own similar code in another language if your prefer.

The experiment is as follows. An array is created containing n random values in the range [-1,1]. Then one of the algorithms for finding the maximum subarray can be timed. We want you to fill in as much of the following table as possible. For each column, fill in as many times as you can reasonably gather, stopping when the running times become "unreasonable" (e.g., longer than 5–10 minutes?)

	Algorithm									
	1		2		2b		3		4	
n	time	$\frac{\text{time}}{n^3}$	time	$\frac{\text{time}}{n^2}$	time	$\frac{\text{time}}{n^2}$	time	$\frac{\text{time}}{n \log n}$	time	$\frac{\text{time}}{n}$
...										
2^{10}										
2^{12}										
2^{14}										
2^{16}										
2^{18}										
2^{20}										
2^{22}										
...										

Problem B "Work entirely on your own."

Rank the following functions by order of growth, *i.e.*, find an arrangement $g_1, g_2, \ldots, g_{30}$ of the functions satisfying $g_1 = \Omega(g_2), g_2 = \Omega(g_3), \ldots, g_{29} = \Omega(g_{30})$. Partition your list into equivalence classes such that $f(n)$ and $g(n)$ are in the same class iff $f(n) = \Theta(g(n))$.

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(3/2)^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\ln\ln n$	$\lg^* n$	$n \cdot 2^n$	$n^{\lg\lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2\lg n}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

[Note: We do not require formal proof for this problem, simply the ordering and the equivalence classes.]

Problem C "Work entirely on your own."

Let $f(n)$ and $g(n)$ be functions such that $f(n) = \Omega(1)$ and $g(n) = \Omega(1)$. Prove or disprove each of the following conjectures.

i. $f(n) = O(g(n))$ implies $g(n) = O(f(n))$.

ii. $f(n) + g(n) = \Theta(\min(f(n), g(n)))$.

iii. $f(n) = O(g(n))$ implies $\lg(f(n)) = O(\lg(g(n)))$, where $\lg(g(n)) > 0$ and $f(n) \geq 1$ for all sufficiently large n.

iv. $f(n) = O(g(n))$ implies $2^{f(n)} = O(2^{g(n)})$.

v. $f(n) = O((f(n))^2)$.

vi. $f(n) = O(g(n))$ implies $g(n) = \Omega(f(n))$.

vii. $f(n) = \Theta(f(n/2))$.

viii. $f(n) + g(n) = \Theta(f(n))$, where $g(n) = o(f(n))$.

[Your proofs must be formal, although for false conjectures a specific counterexample constitutes a valid proof.]

Problem D "You may discuss ideas with other students."

Amortized weight-balanced trees

Consider an ordinary binary search tree augmented by adding to each node x the field $size[x]$ giving the number of keys stored in the subtree rooted at x. Let α be a constant in the range $1/2 \leq \alpha < 1$. We say that a given node x is **α-balanced** if

$size[left[x]] \leq \alpha \cdot size[x]$

and

$size[right[x]] \leq \alpha \cdot size[x]$.

The tree as a whole is **α-balanced** if every node in the tree is α-balanced. The following amortized approach to maintaining weight-balanced trees was suggested by G. Varghese.

a. A 1/2-balanced tree is, in a sense, as balanced as it can be. Given a node x in an arbitrary binary search tree, show how to rebuild the subtree rooted at x so that it becomes 1/2-balanced. Your algorithm should run in time $\Theta(size[x])$, and it can use $O(size[x])$ auxiliary storage.

b. Show that performing a search in an n-node α-balanced binary search tree takes $O(\lg n)$ worst-case time.

For the remainder of this problem, assume that the constant α is strictly greater than 1/2. Suppose that `Insert` and `Delete` are implemented as usual for an n-node binary search tree, except that after every such operation, if any node in the tree is no longer α-balanced, then the subtree rotted at the highest such node in the tree is "rebuilt" so that it becomes 1/2-balanced.

We shall analyze this rebuilding scheme using the potential method. For a node x in a binary search tree T, we define

$$\Delta(x) = |size[left[x]] - size[right[x]]|,$$

and we define the potential of T as

$$\Phi(T) = c \sum_{x \in T : \Delta(x) \geq 2} \Delta(x),$$

where c is a sufficiently large constant that depends on α.

c. Argue that any binary search tree has nonnegative potential and that a 1/2-balanced tree has potential 0.

d. Suppose that m units of potential can pay for rebuilding an m-node subtree. How large must c be in terms of α in order for it to take $O(1)$ amortized time to rebuild a subtree that is not α-balanced?

e. Show that inserting a node into or deleting a node from an n-node α-balanced tree costs $O(\lg n)$ amortized time.

Problem E **(EXTRA CREDIT)** "You may discuss ideas with other students."

Finding the missing integer

An array $A[1..n]$ contains all the integers from 0 to n except one. It would be easy to determine the missing integer in $O(n)$ time by using an auxiliary array $B[0..n]$ to record which numbers appear in A. In this problem, however, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is "fetch the jth bit of $A[i]$," which takes constant time.

Show that if we use only this operation, we can still determine the missing integer in $O(n)$ time.

[To make the analysis cleaner, we will charge your algorithm only for the number of "bit-fetch" operations. The input to the algorithm consists of an array of n numbers, each of which is comprised of $\lfloor \lg n \rfloor + 1$ bits. Give an algorithm which can find the missing number by fetching only $O(n)$ bits.]