Introduction to Algorithms: 6.006
Massachusetts Institute of Technology
Professors Sivan Toledo and Alan Edelman

February 17rd, 2009
Handout 3

Problem Set 2

This problem set is divided into two parts: Part A problems are programming tasks, and Part B problems are theory questions.

Part A questions are due **Tuesday, March 3rd** at **11:59PM**.

Part B questions are due **Thursday, March 5th** at **11:59PM**.

Solutions should be turned in through the course website in PDF form using LaTeX or scanned handwritten solutions.

A template for writing up solutions in LaTeX is available on the course website.

Remember, your goal is to communicate. Full credit will be given only to the correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Exercises are for extra practice and should not be turned in.

Exercises:

- CLRS 11.2-1 (page 228)
- CLRS 11.2-2 (page 229)
- CLRS 11.3-1 (page 236)
- CLRS 11.3-3 (page 236)
- Prove that red-black trees are balanced, i.e., if a red-black tree contains n nodes, then its height is $O(\log n)$. Red-black trees are binary search trees satisfying the following properties:

  When a node does not have a left (or right) child, we say it has a NIL pointer instead.

  1. Each node is augmented with a bit signifying whether the node is red or black.
  2. If a node is red, then both of its children are black.
  3. The paths from the root to any NIL contain the same number of black nodes.

Part A: Due Tuesday, March 3rd

1. **(50 points)** Longest Common Substring

   Humans have 23 pairs of chromosomes, while other primates like chimpanzees have 24 pairs. Biologists claim that human chromosome #2 is a fusion of two primate chromosomes that they call 2a and 2b. We wish to verify this claim by locating long nucleotide chains shared between the human and primate chromosomes.

   We define the *longest common substring* of two strings to be the longest contiguous string that is a substring of both strings e.g. the longest common substring of DEADBEEF and EA7BEEF is BEEF.[1] If there is a tie for longest common substring, we just want to find one of them.

   Download `ps2-dna.zip` from the class website. *The problem set FAQ section of the course web site contains additional hints for this problem.*

   (a) **(2 points)**

   Ben Bitdiddle wrote `substring1.py`. What is the asymptotic running time of his code? Assume $|s| = |t| = n$.

   (b) **(2 points)**

   Alyssa P Hacker realized that by only comparing substrings of the same length, and by saving substrings in a hash table (in this case, a Python set), she could vastly speed up Ben's code.

   Alyssa wrote `substring2.py`. What is the asymptotic running time of her code?

   (c) **(12 points)** Recall binary search from Problem Set 1. Using binary search on the length of the string, implement an $O(n^2 \log n)$ solution. You should be able to copy Alyssa's `k_substring` code without changing it, and just rewrite the outer loop `longest_substring`.

   Check that your code is faster than `substring2.py` for `chr2_first_10000` and `chr2a_first_10000`.

   Put your solution in `substring3.py`, and submit it to the class website.

   (d) **(30 points)**

   Rabin-Karp string searching is traditionally used to search for a particular substring in a large string. This is done by first hashing the substring, and then using a rolling hash to quickly compute the hashes of all the substrings of the same length in the large string.

   For this problem, we have two large strings, so we can use a rolling hash on both of them. Using this method, implement an $O(n \log n)$ solution for

[1] Do not confuse this with the *longest common subsequence*, in which the characters do not need to be contiguous. The longest common subsequence of DEADBEEF and EA7BEEF is EABEEF.

`longest_substring`. You should be able to copy over your outer loop `longest_substring` from part (c) without changing it, and just rewrite `k_substring`.

Your code should work given any two Python strings (see `test-substring.py` for examples). The comparison should be case-sensitive. We recommend using the `ord` function to convert a character to its ascii value.

Check that your code is faster than `substring3.py` for `chr2_first_10000` and `chr2a_first_10000`.

Put your solution in `substring4.py`, and submit it to the class website.

Remember to thoroughly comment your code, including an explanation of any parameters chosen for the hash function, and what you do about collisions.

(e) **(4 points)**

The human chromosome 2 and the chimp chromosomes 2a and 2b are quite large (over 100,000,000 nucleotides each) so we took the first and last million nucleotides of each chromosome and put them in separate files.

`chr2_first_1000000` contains the first million nucleotides of human chromosome 2, and `chr2a_first_1000000` contains the first million nucleotides of chimpanzee chromosome 2a. Note: these files contain both uppercase and lowercase letters that are used by biologists to distinguish between parts of the chromosomes called introns and extrons.

Run `substring4.py` on the following DNA pairs, and submit the lengths of the substrings.

Warning: This part may take a while depending on your implementation of the Rabin-Karp rolling hash. (Leave more than an hour for this part):

`chr2_first_1000000` and `chr2a_first_1000000`
`chr2_first_1000000` and `chr2b_first_1000000`
`chr2_last_1000000` and `chr2a_last_1000000`
`chr2_last_1000000` and `chr2b_last_1000000`

If your code works, and biologists are correct, then the first million codons of chr2 and chr2a should have much longer substrings in common than the first million codons of chr2 and chr2b. The opposite should be true for the last million codons.

(f) **Optional:** Make your code run in $O(n \log k)$ time, where k is the length of the longest common substring.

Part B: Due Thursday, March 5th

1. **(15 points)** Building a Balanced Search Tree from a Sorted List

   You are given a sorted Python list containing n numbers.

   (a) **(5 points)** Show how to construct a binary search tree containing the same numbers. The tree should be roughly balanced (its height should be $O(\log n)$) and the running time of your algorithm should be $O(n)$.

   (b) **(5 points)** Argue that your algorithm returns a tree of height $O(\log n)$. *Note:* It is probably easier to prove an absolute bound (such as $1 + \log n$ or $2 \log n$) than to use asymptotic notation in the argument.

   (c) **(5 points)** Argue that the algorithm runs in $O(n)$ time (here it is hard to avoid the asymptotic notation, so use asymptotic notation).

2. **(15 points)** Collision resolution

   For parts (a) through (c), assume simple uniform hashing.

   (a) **(3 points)** Consider a hash table with m slots that uses chaining for collision resolution. The table is initially empty. What is the probability that, after three keys are inserted, there is a chain of size 3?

   (b) **(4 points)** Consider a hash table with m slots that uses open addressing with linear probing. The table is initially empty. A key k_1 is inserted into the table, followed by key k_2. What is the probability that inserting key k_3 requires three probes?

   (c) **(4 points)** Suppose you have a hash table where the load-factor α is related to the number n of elements in the table by the following formula:

   $$\alpha = 1 - \frac{1}{\log n}.$$

   If you resolve collisions by chaining, what is the expected time for an unsuccessful search in terms of n?

   (d) **(4 points)** Using the same formula relating α and n from part (c), if you resolve collisions by open-addressing, give a good upper bound on the expected time for an unsuccessful search in terms of n. For this part, assume Uniform Hashing.

3. **(20 points)** Counting Landings in a Time Interval

   In this problem we will augment a binary search tree to improve the running time of a specialized query. Storing additional data in every node of the tree (and maintaining this data when the tree is modified) speeds up the query. In all the parts of this problem, T is a binary search tree that stored n integers.

(a) **(7 points)** Describe an algorithm `countInRange` to compute the number of nodes with values in the range $[a, b]$, where a and b are integer values that are stored in T (your algorithm can assume that a and b are in the tree). The algorithm should run in $O(k + h)$ time where k is the number of nodes with values in the range $[a, b]$ and h is the height of T.

(b) **(3 points)** Describe a strategy to augment the tree. What additional information do you propose to store at every node of the tree so that `countInRange` can be implemented in $O(h)$ time?

(c) **(5 points)** Using your augmented binary search tree, show how to count the number of nodes in the range $[a, b]$ in $O(h)$ time.

(d) **(5 points)** Given the augmented binary search tree, show how you can modify the `insert` operation and how to modify rotation operations so that they correctly maintain the extra information stored at each node.