Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — Feb 21, 2007
Professors Srini Devadas and Erik Demaine — Handout 4

Problem Set 2

This problem set is due **Thursday March 6** 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:

  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 leaf contain the same number of black nodes.

---

1. **(12 points)** `select` in Binary Search Trees

   Implement `select` in `bstselect.py`. `select` takes an index, and returns the element at that index, as if the BST were an array. `select` is essentially the inverse of `rank`, which took a key and returned the number of elements smaller than or equal to that key. The index for `select` should be 1-based (not 0-based like Python often uses).

   Download `ps2-bst.zip`. Read `test-bst.py` to clarify how `select` should work. Put your code in `bstselect.py` until `test-bst.py` works. Be sure to comment your code, explaining your algorithm.

   Submit `bstselect.py` to the class website.

2. **(10 points)** Amortization

   You are given an m-bit binary counter, where the rightmost bit is the "1's" digit, the next bit is the "2's" digit, the next bit is the "4's" digit, and so on, up to the "2^{m-1}'s" digit. The function `increment` adds 1 to the counter, carrying when appropriate.

   Assuming that the counter starts at 0, prove that increment takes $O(1)$ amortized time. In other words, show that after n operations, the total amount of time spent is $O(n)$. For simplicity, assume that the only operation that takes any time is flipping a bit in the counter.

3. **(12 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) **(3 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) **(3 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) **(3 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.

4. **(26 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.

[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.

Download `ps2-dna.zip` from the class website.

(a) **(1 point)**

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

(b) **(1 point)**

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) **(6 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) **(16 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 `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). 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_100000` and `chr2a_first_100000`.

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) **(2 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 (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.