Problem Set 2

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

Part A questions are due **Tuesday, March 2nd** at **11:59PM**.

Part B questions are due **Thursday, March 4th** at **11:59PM**.

Solutions should be turned in through the course website. Your solution to Part A should be in PDF form using LaTeX or scanned handwritten solutions. Your solution to Part B should be two valid Python files which runs from the command line, together with one PDF file containing your solutions to part (a), (b), (e) and the optional part (f).

Templates for writing up solutions in LaTeX are 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.

Part A: Due Tuesday, March 2nd

1. **(14 points)** Building a Balanced Search Tree from a Sorted List

   You are given a sorted Python list containing n distinct numbers.

   (a) **(4 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. **(18 points)** Range Queries for a Balanced Search Tree

   We use balanced binary search trees for keeping a directory of MIT students. We assume that the names of students have bounded constant length so that we can compare two different names in $O(1)$ time. Let n denotes the number of students. Say we have a binary search tree with students' last names as the keys with lexicographic dictionary ordering. In this problem we are going to use the balanced search tree to answer some range queries of the students' last names.

For example, if the tree contains 5 names $\{$ABC, ABD, ADA, ADB, ADC$\}$, then all the 5 names are (inclusively) between ABC and ADC. There are two names, namely ABC and ABD, start with the prefix x=AB.

(a) **(6 points)** Given two strings a and b with $a < b$, give an algorithm that returns all the nodes whose key values are (inclusively) between a and b. If the total number of such nodes is k, then the running time of your algorithm should be $O(k + \log n)$.

(b) **(6 points)** Give an algorithm that outputs a list of all the nodes whose keys starts with a given prefix x in $O(k + \log n)$ time, where k is the number of nodes in the list.

(c) **(6 points)** Give an algorithm to count the number of nodes whose keys start with a given prefix x in $O(\log n)$ time, independent of the number of such nodes. You are allowed to augment the binary search tree nodes.

3. **(18 points)** Collision Resolution

Assume simple uniform hashing in the entire problem.

(a) **(6 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 four keys are inserted, there is a chain of size 4?

(b) **(6 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) **(6 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 open addressing, what is the expected time for an unsuccessful search in terms of n?

Part B: Due Thursday, March 4th

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-source.zip` from the class website. Use the template file ps2B-template.tex provided on the course website to put your solutions to part (a), (b), (e) and the optional part (f) in a PDF file (or you may scan your handwritten solutions and submit it as a PDF file).

(a) **(2 points)**

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

(b) **(2 points)**

Alice 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 Bob's code.

Alice 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 Alice'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 `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`.

[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.

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.