50:198:213 Data Structures
Rutgers University–Camden
Professor Desmond Lun

Problem Set 2

Solutions should be turned in through the course Canvas site in PDF form or scanned handwritten solutions.
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.

---

1. **(16 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 Sakai site.

2. **(16 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. **(16 points)** Collision resolution

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

   (a) **(4 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.

4. **(12 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 Sakai site.

(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) **(8 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 Sakai site.

[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.