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, Tuesday, October 7th** at **11:59PM**.
Part B questions are due **Thursday, Thursday, October 9th** 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.

Part A: Due Tuesday, October 7th

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.

(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 `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`.

[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 (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, October 9th

1. **(25 points)** Augmented BST : `max-gap`

Continuing with the airline reservation problem from class, suppose that we no longer allow airlines to choose their desired takeoff times. Instead, we determine the largest time interval between existing departures and assign the flight a time within that slot.

Conceptually, this is equivalent to finding the maximum gap between consecutive integers in a sorted list. For example:

$$[0, 60, 630, 855, 1140, 1440]$$

represents a day with reservations at 1:00 AM, 10:30 AM, 2:15 PM, and 7 PM. A new reservation request would then be inserted in the range $(60, 630)$ since it has a gap of 570 minutes, which is the largest interval between flights.

Here, the list elements represent existing flight departure reservations in minutes since midnight. Also, notice that we constrain the list on both ends with placeholder "flights" for the beginning and end of the day (1440 is the total number of minutes in a day); these are inserted before any flight and are never deleted.

To maintain this list of numbers and support the operations `max-gap`, `search`, and `insert` we will use an augmented BST; that is, we will use a binary search tree in which every node will contain the flight time, plus some other information to support the `max-gap` queries.

(a) **(7 points)** Describe how to augment the tree. What additional information should be stored in every node?

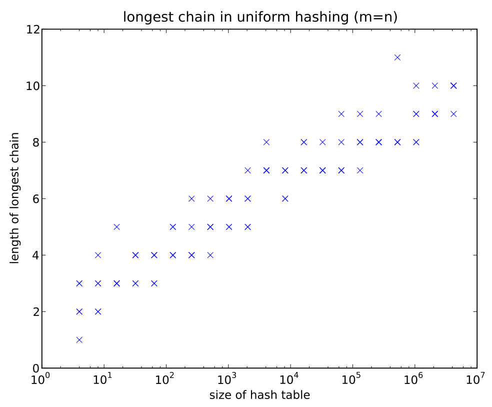
(b) **(9 points)** Given the augmented binary search tree, describe how to efficiently answer `max-gap` queries. `max-gap` returns the range of the maximum gap; so in the example above the returned value would be $(60, 630)$ not 570. What is the asymptotic worst-case running time of `max-gap`? State how you break ties.

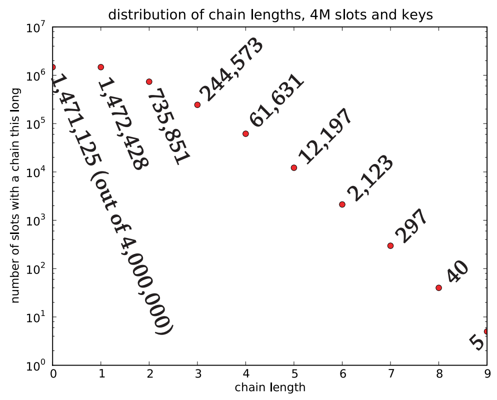
(c) **(9 points)** Explain how to extend the `insert` operation so that the tree remains correctly augmented. You can assume that `insert` is given a flight time in minutes since midnight and starts by inserting a this new flight into the binary search tree. Show how to add the augmented information to the new tree node, and how to update the augmented information in other nodes (if necessary). Analyze the running time of `insert`.

(d) **(optional)** The above method doesn't describe how we choose a time within the appropriate interval. Describe how sequential requests would be assigned if the mechanism were to insert (i) at the halfway point of the interval (ii) 5 minutes from the start of the interval (you may assume that the interval is always larger than 5 minutes).

2. **(25 points)** Length of Chains in Uniform Hashing

In class we only discussed the expected length of chains. For example, if we hash n items into n slots and assume that the hashing is uniform, then the expected length of every chain is just 1. But as the two graphs below show, for large n we are very likely to get longer chains; the question is how long. In this problem we shall estimate how long the chains get. We use an approach that is not completely rigorous but which gives useful insights; the same results can also be proved rigorously. (The Python code that produced these graphs is on the course's web site.)





(a) **(5 points)** Argue that when n keys are inserted to a hash table with m slots, and assuming uniform hashing, then the probability Q_k that exactly k keys hash to one particular slot is

$$Q_k = \binom{n}{k}\left(1-\frac{1}{m}\right)^{n-k}\left(\frac{1}{m}\right)^{k} .$$

(The expression $\binom{n}{k}$, pronounced "n choose k", is the number of ways to choose k items out of n; its value is $\frac{n!}{k!(n-k)!}$.)

(b) **(5 points)** The statistical distribution of Q_k is called the *binomial distribution*. One way to estimate Q_k is to relate the binomial distribution to another statistical distribution called the *Poisson distribution*, for which the probability is given by

$$P_k(\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

where λ is a parameter of the distribution. It turns out that for large m and n,

$$Q_k(m,n) \approx P_k\left(\frac{n}{m}\right) .$$

In this problem, we will not try to prove or use sharp bounds on this approximation, but will use it as if it was an equality; this is not rigorous, but it does give good intuition and good approximations for large m and n.
Use Strirling's approximation for $n!$ (Equation 3.17 in the textbook) and the limit $\lim_{n\to\infty}(1+\frac{x}{n})^n = e^x$ (Equation 3.13 in the textbook) to derive the approximation result for the case $m = n$.

(c) **(5 points)** Use indicator random variables (Section 5.2 in the textbook) to estimate the expected number of empty slots, slots with one key, and slots with two keys for the case $m = n$. Hint: for each of these analyses, you will need an indicator random variable for each slot, as well as one random variable that is the sum of the indicators; use the linearity of expectation.

(d) **(5 points)** Prove that the chain length $\hat{k}$ for which the expected number of chains of that length is close to 1 is $\hat{k} = \Theta(\lg n / \lg\lg n)$, assuming again $n = m$. Hint: write an equation using your approximation for $E[Y_{\hat{k}}]$; then take a logarithm of both sides and use the fact that $\lg(n!) = \Theta(n \lg n)$ (Equation 3.18 in the textbook); finally, use the definition of Θ to complete the proof.

(e) **(5 points)** Does your result agree with the experimental results that are shown in the graph above? Your analysis relies on approximations, but they are pretty good for the problem sizes in the graphs.