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)

  Solution: Each pair has a $\frac{1}{m}$ chance of colliding, and there are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs. Multiplying, we get the expected number of collisions: $\frac{n(n-1)}{2m}$

- CLRS 11.2-2 (page 229)

  Solution: Left for the reader.

- CLRS 11.3-1 (page 236)

  Solution: Hash the key. Walk down the list comparing against the hashes. Only compare the actual strings if the hashes match.

- 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.

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.

   Solution: The optimal solution takes $O(h)$ time, where h is the height of the tree. Assuming balanced binary trees, like the AVL tree described in Lecture 4, $h = O(\log n)$, so the solution will take $O(\log n)$ time.

   The key observation is to augment the BST so that each node keeps track of the size of its subtree (1+ the number of descendants). We have provided the algorithm and a sample implementation for doing this in Recitation 4. The code supplied with the problem set also provides an implementation of BSTs augmented to keep track of subtree size, in `bstsize.py`.

   The algorithm works as follows. Let `root` be the root of the tree, and let i be the rank of the node that we are trying to select. The root's rank is $r = 1 +$ `bstsize.size(root.left)`, because all the nodes in the root's left subtree are smaller than the root, and all the nodes in the right subtree are larger than the root (by definition of the BST).

   (a) $i = r$: the root is the node we are looking for, so we can return the root. We are done.

   (b) $i < r$: the node we are looking for belongs to the root's left subtree, so we can reduce the problem to finding the i^{th} node in the root's left subtree.

   (c) $i > r$: the node we are looking for belongs to the root's right subtree, so we can reduce the problem to finding the $(i - r)^{\text{th}}$ node in the root's right subtree.

   Every time the problem is reduced to a smaller one, we go one level deeper in the tree. Therefore, the number of steps taken until we reach a leaf is $O(h)$.

   There are also several sub-optimal ways to implement `bstselect`.

   (a) Enumerating the first i nodes; the first node is obtained by calling `minimum`, then successor nodes are obtained via calls to `successor`. This method takes $O(i)$ time. Since i can be any number from 1 to n, the worst-case running time is $O(n)$.

(b) Enumerating nodes until the i^{th} is reached; this follows the plan of the previous method, except it uses *bstrank.rank* to determine the rank of each enumerated node. This adds a $O(\log n)$ factor to the enumeration time, for a total time of $O(n \log(n))$

Partial credit was awarded to suboptimal solutions that were explained well.

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.

Solution:

CLRS gives extensive coverage of this problem (with 3 different solutions) in Sections 17.1, 17.2, and 17.3.

A brief solution is obtained by analyzing the total cost (number of bit flips) for n `increment` operations. The key observation is that the "1's" digit is flipped every time, the "2's" digit is flipped once every 2 `increment` operations, and so on. Generally, the "2^r's" digit is flipped once every 2^r `increment` operations.

The total number of bit flips for an m bit counter is obtained by summing the number of increments over all m bits. As explained above, bit r is flipped once every 2^r increments, so it is flipped $\lfloor \frac{n}{2^i} \rfloor$ times over n `increment` operations. The summation over all m bits is $\lfloor n \rfloor + \lfloor \frac{n}{2} \rfloor + \lfloor \frac{n}{2^2} \rfloor + \cdots + \lfloor \frac{n}{2^{\lg n}} \rfloor$.

An upper bound for the summation above is the geometric sum $n + \frac{n}{2} + \frac{n}{2^2} + \cdots + \frac{n}{2^{\lg n}}$, which is, in turn, bounded by the infinite geometric sum $n\big(\sum_1^\infty \frac{1}{2^i}\big) = 2 \cdot n$

In conclusion, n `increment` operations take $\leq 2 \cdot n = O(n)$ bit flips. This means that the ammortized cost per `increment` is $\frac{O(n)}{n} = O(1)$.

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?

Solution: Let h be the hash function used. A chain of size 3 can only be created if the hash function maps all 3 keys to the same slot in the hash table.

The corresponding equation is $h(k_1) = h(k_2) = h(k_3)$. So the probability of having a size 3 cluster is:

$$\begin{aligned}
\Pr[h(k_1) = h(k_2) = h(k_3)] &= \Pr[h(k_1) = h(k_2) \vee h(k_2) = h(k_3)] && (1)\\
&= \Pr[h(k_2) = h(k_1)] \cdot \Pr[h(k_3) = h(k_1) \mid h(k_2) = h(k_1)] && (2)\\
&= \Pr[h(k_2) = h(k_1)] \cdot \Pr[h(k_3) = h(k_1)] && (3)\\
&= \frac{1}{m} \cdot \frac{1}{m} && (4)\\
&= \frac{1}{m^2} && (5)
\end{aligned}$$

The first step breaks down the equation. The second step follows from the definition of conditional probability (taught in 6.041 and 6.042). The third and fourth steps take advantage of the fact that, under simple uniform hashing, a key is equally likely to hash to any slot in the hash table, and this probability is independent of the other keys in the hash table. The rest is algebra.

(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?

Solution: Inserting k_3 requires 3 probes iff the first 2 probes hit slots that are taken, and the 3rd probe finds an empty slot. The table contains 2 keys when k_3 is inserted, so the first 2 probes must hit the slots holding those two keys. If this happens, the 3rd probe is guaranteed to find an empty slot (there are no other keys in the table).

We are analyzing linear probing, so insertions probe consecutive slots. The first 2 probes in k_3's insertion must hit the slots holding k_1 and k_2, so k_1 and k_2 must be stored in consecutive slots, and k_3 must initially hash to the first of those 2 slots.

In order for k_1 and k_2 to be stored in consecutive slots, one of the following must happen:

i. k_2 is stored in the slot preceding k_1. This happens if $h'(k_1) = h'(k_2) + 1 \bmod m$. Since we're assuming h' obeys simple uniform hashing, the probability for this to happen is $\frac{1}{m}$ ($h'(k_2)$ must point to a specific slot).

ii. k_2 is stored in the slot following k_1. This happens if $h'(k_2) = h'(k_1)+1 \bmod m$, or if $h'(k_2) = h'(k_1)$ (in this case, linear probing dictates that the following slot will be probed next). Each of these two events has probability $\frac{1}{m}$, for the same reasons explained above. The events are mutually exclusive, so the probability of either of them happening is $\frac{1}{m} + \frac{1}{m} = \frac{2}{m}$

The cases leading to k_1 and k_2 being stored in consecutive slots are mutually exclusive, so the probability of this happening can be computed as $\frac{1}{m} + \frac{2}{m} = \frac{3}{m}$.

k_3's insertion will use 3 probes only if $h'(k_3)$ is the first of the two consecutive slots. Since we're assuming simple universal hashing, the probabiliy of this event is $\frac{1}{m}$, and the event is independent with any events related to other keys.

Therefore, the probability of 3 probes on k_3's insertion is $\frac{3}{m} \cdot \frac{1}{m} = \frac{3}{m^2}$.

(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?

Solution: We are analyzing a hash table which resolves collisions by chaning, has a load factor of α, and uses a hash function that exhibits the properties of simple uniform hashing. The notes for Lecture 5, as well as Theorem 11.1 in CLRS, indicate that the expected time taken by an unsuccessful search in this scenario is $T(\alpha) = O(1 + \alpha)$. Plugging in $\alpha(n) = 1 - \frac{1}{\log n}$, we obtain:

$$\begin{aligned} T(n) &= T(\alpha(n)) \\ &= O(1 + \alpha(n)) \\ &= O(1 + 1 - \frac{1}{\log n}) \\ &= O(2 - \frac{1}{\log n}) \\ &= O(1) \end{aligned}$$

The last result is obtained by observing that $0 < \frac{1}{\log n} \leq 1$ for $n \geq 2$.

(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.

Solution: We are analyzing a hash table which uses open addressing, has a load factor of α, and uses hash function exhibits the properties of Uniform Hashing. The notes for Lecture 7, as well as Theorem 11.6 in CLRS, indicate that the expected time taken by an unsuccessful search in this scenario is $T(\alpha) = O(\frac{1}{1-\alpha})$.

Plugging in $\alpha(n) = 1 - \frac{1}{\log n}$, we obtain:

$$\begin{aligned} T(n) &= T(\alpha(n)) \\ &= O\Big(\frac{1}{1-\alpha(n)}\Big) \\ &= O\Big(\frac{1}{1-(1-\frac{1}{\log n})}\Big) \\ &= O\Big(\frac{1}{\frac{1}{\log n}}\Big) \\ &= O(\log n) \end{aligned}$$

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.

   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 $|s| = |t| = n$.

   Solution: $\Theta(n^5)$. There are four nested for loops, which each account for a factor of n, and inside the for loops, we use the slice operator (to take substrings), which takes $\Theta(n)$ time. Also, even if the slice operator didn't take $\Theta(n)$ time, the comparison (`==`) operator also takes $\Theta(n)$ time.

   (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?

[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.

Solution: $\Theta(n^3)$. `k_substring` gets called n times. Inside `k_substring`, each for loop runs n times, and each iteration of the loop takes $\Theta(n)$ time (because of both the slice operator, and the hashing to add to a set).

(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.

Solution: Only code needed submission.

(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.

Solution: Again, only code needed submission.

Common mistakes:

- Putting all substrings of length k into a set or dictionary. This takes $O(nk)$ time and space. Instead, if you want a pointer back to a substring, you can use the start-index of the substring.
- Using too small of prime for Rabin-Karp. The prime needed to be around n to make a manageable number of collisions. Anything smaller led to too much time wasted checking the actual substrings during 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):

Solution:

`chr2_first_1000000` and `chr2a_first_1000000`	836
`chr2_first_1000000` and `chr2b_first_1000000`	152
`chr2_last_1000000` and `chr2a_last_1000000`	247
`chr2_last_1000000` and `chr2b_last_1000000`	961

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.