Introduction to Algorithms: 6.006
Massachusetts Institute of Technology
Erik Demaine, Ronald L. Rivest, and Nir Shavit

October 1, 2013
Handout # 3

Problem Set 3

Kevin Peng
Collaborators: Genghis Chau, Jodie Chen, Anubhav Jain, Martin Ma, Xubo Sun

---

This problem set is due **Tuesday, October 22** at **11:59PM**.

This solution template should be turned in through our submission site. [1]

For written questions, full credit will be given only to correct solutions that are described clearly *and concisely*.

---

[1] Register an account, if you haven't done so. Then go to Homework, Problem Set 3, and upload your files.

Problem 3-1. [16 points] **Hash Functions and Data Sequences**

This problem should be submitted using `pset3_1_solution_template.py`.

Problem 3-2. [24 points] **Open Addressing**

(a) [5 points] Ben is supposed to return the list of students to the professor, but he accidentally spills coffee on it, rendering it illegible. To cover his incompetence, Ben wants to hide his mistake by recreating the list of student numbers in order; however, he only remembers two facts about the order in which the student numbers were inserted.

- 98 was the first student number to be inserted.
- 35 was inserted before 14.

Given the completed hash table, in what order did Ben insert the student numbers?

We are given that 98 comes first. We know that 14 comes before 54, which comes before 24 because they have the same residue modulo 10, and in the table, 14 comes before 54, which is before 24 in the table. We are given that 35 comes before 14, so it must also come before 54 and 24. Our order so far is 98, 35, 14, 54, 24. We know that 55 comes after 54 since 14 and 35 must have already hashed to slots 4 and 5, and 54 comes before 55 in the table, so it must have been inserted before 55 by linear probing. We know that 17 must come after 55 since 55 occupies slot 7, which means slot 7 was already filled when we tried to hash student 17. Lastly we know that 24 must come after all the other numbers because it was unable to hash to any of the slots from 4 to 9, so all of those slots must have been filled already. Thus, the order is 98, 35, 14, 54, 55, 17, 24.

(b) [5 points] Student 98 leaves Ben's section, so Ben deletes that record from the hash table. The next day, Ben sees Student 15 yawning during the lecture, so naturally he decides to give Student 15 a 0 in class participation. However, Ben can't remember if Student 15 is in his section, so he decides to look up Student 15 in his hashtable. Which cells does Ben need to inspect to determine if the student number 15 is in the table?

First, Ben inspects slot 5 since $15\%10 = 5$. He finds that slot 5 does not contain student 15. He then proceeds by linear probing to slots 6 and 7, which also do not contain student 15. He proceeds to slot 8 which has a deleted flag, so he proceeds on. He then continues to slots 8, 9, 0. He looks at slot 1 and sees that it's empty, so he sees that 15 is not in the table.

(c) [5 points] Student 98 rejoins Ben's section, so Ben reverts back to his original hash table found above. Having just learned about several sorting algorithms, Ben wants to "sort" the student numbers of the hash table to end up with the hash table below.

0	14
1	17
2	
3	
4	24
5	35
6	54
7	55
8	98
9	

Assuming that duplicate student numbers cannot simulataneously be stored in the hash table, give a length-8 sequence of insertion and deletion operations that would transform the original table into the one shown above.

1. Ben can first delete student 17 from the table. Slot 9 is now empty.
2. Ben can then insert student 15 into the table. 15 is now in slot 9.
3. Ben can then insert student 17 into the table. 17 is now in slot 1.
4. Ben can then delete student 24 from the table. Slot 0 is now empty.
5. Ben can then delete student 14 from the table. Slot 4 is now empty.
6. Ben can then insert student 24 in the table. 24 is now in slot 4.
7. Ben can then insert student 14 in the table. 14 is now in slot 0.
8. Ben can then delete student 15 from the table. Slot 9 is now empty.

(d) [9 points]

Ben hypothesizes that he can start from an arbitrary hash table of size k with student numbers $[a_0, a_1, \ldots, a_{k-1}]$ and end up with student numbers $[b_0, b_1, \ldots, b_{k-1}]$ via a finite sequence of insertion and deletion operations:

Note that each a_i or b_i is allowed to be empty. Ben believes that such a transforming sequence of operations exists regardless of the hash function, assuming linear probing and assuming that duplicate student numbers cannot simultaneously exist in the hash table.

Is Ben's hypothesis correct? If so, demonstrate a procedure to accomplish this. If not, give a counterexample.

Ben's hypothesis is correct. He can first fill in the whole table. Then, for each nonempty b_i in the desired table, he can delete the element that is currently in the slot that the b_i should be in and immediately replace it with b_i (since the rest of the table will be full except for that single slot, linear probing will force it into that slot). After that, he can delete b_i for all the slots where b_i is empty.

Problem 3-3. [35 points] **Finding "Nemo"**

(a) [5 points] Determine the running time of computing $H(\cdot)$ for a string of length k under the word RAM model for computation. Assuming that you have precomputed $a^{k-1} \mod m$, determine the running time for computing $H(\cdot)$ for every substring of length k in T, using the rolling hash method. Your answers should be functions of n, k, and d (or some subset of these variables).

The running time for computing $H(\cdot)$ for a string of length k is $\Theta(n)$. We can do this by starting from the last digit, computing $a^0 = 1$ and multiplying it by c_k. Then, we compute $a^1 = a$, multiply it by c_{k-1} and add it to the two numbers together. We keep proceeding until we reach the first digit. Each of these steps will take constant time because multiplication and addition are assumed to take constant time. When we compute a^i, we use our value of a^{i-1} multiplied by a, so the exponentiation will also take constant time per iteration. We repeat these steps a total of $n-k$ times, so the running time for all the iterations is $\Theta(n-k)$. Combining this with the first computation, we get that $\Theta(n+k)$. In this problem, since we know that $n \geq k$, so we can say this is $\Theta(n)$ as well.

(b) [5 points] For an input string of length k, you must compute $f(k)$ variants, for some function f. This takes time $\Theta(f(k))$. You must then compute the hashes of each of these variants. With some care, this too can be done in time $\Theta(f(k))$ after computing $H(S)$.

Give an expression for f, argue that you can compute hashes of all $f(k)$ variants in time $\Theta(f(k))$ after computing $H(S)$, and give the overall asymptotic running time of this sequence of operations.

First, we can express $f(k)$ as $g(k) + h(k)$ where $g(k)$ is the number of variants that come from transpositions and $h(k)$ is the number of the variants that come from one-character substitutions. Thus $g(k) = k - 1$, since we can transpose each of the first $k-1$ characters with the character following it. $h(k) = k(d-1)$ since for each character, there are $d-1$ other characters that we can substitute it with. Thus, $f(k) = k - 1 + k(d-1) = kd - 1$.

We can compute the hashes of all $f(k)$ variants in $\Theta(f(k))$ time after computing $H(S)$, assuming we have all of the values of a^i (or $a^i \mod m$) for $i \in [0, k]$, which we calculated in part (a). It will take us constant time to calculate all of the hash values of the transpositions since once we have $H(S)$, the only difference is that two constants are swapped, so we can subtract $c_i a^{k-i}$, subtract $c_{i+1} a^{k-(i+1)}$, add $c_i a^{k-(i+1)}$ and add $c_{i+1} a^{k-i}$. This consists of a constant number of multiplications, subtractions, and additions, which each take constant time. Thus, we can calculate all of the transpositions in $c_d(k-1)$ time. For the substitutions, we effectively do the same thing, except we only need to perform a constant number of additions, subtractions, and multiplications for one digit. Thus, we can compute the substitutions in $c_e(k(d-1))$ time. Adding these two together, we can compute all of the variants in $c_e(k(d-1)) + c_d(k-1) = \Theta(f(k))$ time.

The sequence of operations up to this point takes $\Theta(kd) + \Theta(n+k) = \Theta(n+k+kd) = \Theta(n+kd)$ since $k \leq n$.

(c) [5 points] After computing hashes of all length-k substrings of T and of all variants of S, you must examine substrings of T and variants of S that hash to the same value and test for actual equality. Using the simple uniform hashing assumption and assuming that testing for equality takes time $\Theta(k)$, compute the expected running time in the worst case of this operation and argue that your answer is correct. Your answer should be a function of n, k, d, and m (or some subset of these variables).

Finally, compute the overall running time of approximate string search for a string of length k in a document of length n. Asymptotically, what value of m must you choose so that this running time is linear in n?

The expected running time in the worst case of this operation is $O(\frac{k^2dn}{m})$. We expect, under the simple uniform hashing assumption, that there will be at most a $\frac{kd}{m}$ chance that two keys will have the same hash value (since there are $kd - 1 + 1$ elements that we insert into the table for S and its variants). When two keys have the same hash value, we must compare the two strings, which takes $\Theta(k)$ time. Two keys have at least a $1 - \frac{kd}{m}$ chance of having different hash values. When they have different hash values, we know they're not equal in constant time. We need to do this for $n - k + 1$ windows. Thus, the expected running time is $O(k\frac{kd}{m} + 1 - \frac{kd}{m})(n - k + 1) = O(\frac{k^2dn}{m})$.

If we want the overall running time to be linear in n, we need $m = \Theta(dk^2)$. We can assume $n \geq k$ (if it weren't, we already know that the result should be `None`). Our overall running time is $\Theta(n + k) + \Theta(kd) + O(\frac{k^2dn}{m})$. If $m = \Theta(k^2d)$, then the third part of the sum breaks down to $O(n)$. We can also assume that d is constant. Thus, the whole algorithm will run in $O(n)$ time.

(d) [20 points]

This problem should be submitted using `pset3_1_solution_template.py`.

Problem 3-4. [25 points] **Multiple Multiplications**

(a) [10 points] Your implementation for `naive_multiply` should be in `pset3_4_solution_template.py`

(b) [10 points] Your implementation for `karatsuba_multiply` should be in `pset3_4_solution_template.py`

(c) [5 points] You finished writing your Karatsuba algorithm, but the Python Bee's testing system has broken down! In order to show that your new algorithm is superior to your old one (and Ben Bitdiddle's), run `naive_multiply` and `karatsuba_multiply` on various input values and measure the runtimes of the two functions. Show that the runtime of `karatsuba_multiply` grows at an asymptotically slower rate than `naive_multiply`.

Just like in Problem 1-3b in Problem Set 1, assume that each algorithm has a running time of the form

$$T(n) = r \cdot n^s \tag{1}$$

and use the estimators

$$s \approx \lg_2 \frac{T(n)}{T(n/2)}, \qquad r \approx \frac{T(n)}{n^s}, \tag{2}$$

where n is the largest n for which you measure the running time. You should choose n large enough so that the algorithm takes a non trivial amount of time to run on your machine, and so that the running times are significantly different. You may want to take the average of a large number of runtimes to get the most accurate results. You can use the Python module `time`, or other similar functionality, to help you measure time.

Karatsuba Multiply:
When $n = 32800$, Karatsuba multiply takes 54.3092210293 seconds to run.
When $n = 16400$, Karatsuba multiply takes 18.0952749634 seconds to run.
When $n = 8200$, Karatsuba multiply takes 6.25127199173 seconds to run.
When $n = 4100$, Karatsuba multiply takes 1.90001797676 seconds to run.
When $n = 2050$, Karatsuba multiply takes 0.559615850449 seconds to run.

$$s \approx \lg_2 \frac{T(n)}{T(n/2)} = \lg_2 3.0012929 = 1.58558414$$

$$r = \frac{T(n)}{n^s} = 3.7547033 \times 10^{-6}$$

Naive Multiply:
When $n = 1768$, Naive multiply takes 58.7721760273 seconds to run.
When $n = 884$, Naive multiply takes 10.74081087112 seconds to run.
When $n = 442$, Naive multiply takes 1.84538403156 seconds to run.
When $n = 221$, Naive multiply takes 0.325037088394 seconds to run.

$$s \approx \lg_2 \frac{T(n)}{T(n/2)} = \lg_2 5.471857930283 = 2.45203077300$$

$$r = \frac{T(n)}{n^s} = 6.4009632344 \times 10^{-7}$$