Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — Thursday, October 4
Instructors: Zachary Abel, Erik Demaine, Jason Ku — Problem Set 5

Problem Set 5

All parts are due Thursday, October 11 at 11PM.

Name: Kevin Jiang

Collaborators: Suraj Srinivasan

Problem 5-1.

(a) Given an unordered list of n items, we take each value, k, and add $g - k$ to a hash table if it doesn't already exist and $2k \neq g$ (if $2k = g$, then return True). If $g - k$ is already in the hashtable, then we return True since elements k and $g - k$ both exist in the table. Since we are only doing constant time comparisons and look ups (due to hash table property) for at most n items, we expect an $O(n)$ runtime.

(b) Suppose we have two pointers, a and b, that initially point to the first and last elements in the sorted item list A, respectively. Compare the sum g with $s = A[a] + A[b]$. If $g > s$, then compare g and $s = A[a] + A[b-1]$. If $g < s$, then compare g and $s = A[a+1] + A[b]$. If $g = s$, then return True. We continue these comparisons until the first pointer a becomes strictly larger than the second pointer b, at which point we return False. Since we are decreasing the size of the problem from n to $n - 1$ each time we compare, this algorithm has a worst-case run time of $O(n)$.

(c) Let $S = [a_1, a_2, \cdots, a_n]$ be the unordered list of n items. Now let $S' = [a_1, a_2, \cdots, a_k]$ be the unordered list of k elements where $a_j \leq g$ for $1 \leq j \leq k$. Creating S' takes $O(n)$ time since we are iterating over each element in S and comparing the value to g.

We apply radix sort on S', which takes $O(n + b)$ time for each counting sort, where b is an arbitrary base. Since we are applying counting sort to each digit d of a_j, our total run time becomes $O((n + b)d)$. Note that d is at most $\log_b g$ since $\max(S') \leq g$. Thus, we have $O((n + b)d) = O((n + b)\log_b g)$. If we let $b = n$, we get

$$O((n + b)\log_b g) = O((n + n)\log_n g) = O(cn) = O(n),$$

since we are given that $\log_n g = O(1)$. Therefore, radix sort on S' has a run time of $O(n)$.

Finally, we apply part (b) to solve the problem, which we know has a run time of $O(n)$. Since all of our steps have a run time of $O(n)$, this algorithm has a worst-case time of $O(n)$.

Problem 5-2.

We can use a min heap with a hash table keeping track of the location of each item x. To insert an item x, we add it as a leaf to the min heap and min heapify up as necessary, which we know is correct and has $O(\log n)$ amortized time from lecture. To return an item with the smallest key, we just return the first element of the min heap, which is correct and has $O(1)$ worst case time look up from lecture. To remove an item with the smallest key, we switch the absolute min with a leaf, pop it off, and min heapify down, which is correct and has $O(\log n)$ worst case time from lecture. To return item x having identifier ID, we can just look up the ID since we hashed the items x into a hash table of ID numbers. If the ID exists, we return the corresponding x; otherwise, we return None. In either case, the algorithm is correct and has an expected runtime of $O(1)$ since we are utilizing the constant time look up of a hash table. To change the key of item x having identifier ID to k, we locate item x via hash table look up, change the key to k, and then min heapify up/down depending on the position, which is correct and has worst case runtime of $O(\log n)$ since we are only min heapifying.

Problem 5-3.

(a) First, we sort the weets by length. Iterate through every weet and hash each length to a value from 0 to 60. Sorting each weet length takes $O(1)$ time, so sorting all the weets by length is $O(n)$ time.

Next, iterate through the hash table of weet lengths and sort each chain alphabetically via radix sort. We apply radix sort by converting the letters a, b, c, d, etc. to 1,2,3,4, etc. Since we are radix sorting all of our weets, the sorting run time is $O((n+26)\cdot 60)$ since each counting sort is base 26, and we are applying counting sort at most 60 times. Thus, sorting each length alphabetically takes $O(n)$ time.

This algorithm is correct since sorting by length by hashing and alphabetically by radix sort are stable sorting algorithms. Thus, we are maintaining previous orders when sorting. Since both sorts are $O(n)$ time, our overall run time is $O(n)$. This is the most efficient algorithm since the length of the weets is constant, making radix sort the most efficient sorting algorithm for this problem.

(b) If we use radix sort, then the run time is $O(wn)$, where w is the number of digits of the largest number of likes in a weet. Since we are told that the average number of likes per weet is much much greater than the number of tweets, then we can assume that $w = \log(k)$ is not $O(1)$, where k is the largest number of likes. Thus, radix sort will be best case $O(n \log k)$. Using a comparison-based sorting algorithm like heap sort ($O(n \log n)$ time) will have a better worst case run time than radix sort, but it will use the least space possible ($O(1)$). Therefore, heap sort is the best since it runs in the next best run time $O(n \log n)$ and uses constant space.

(c) Since we want the m most-liked weet, we use a max heap. We will be inserting n weets regardless of what sorting algorithm we use, so the most efficient sorting algorithm will be the one that returns the max in the minimum time and has the least insert time. Max heap and AVL trees have the fastest find max(es) time of $O(\log n)$ and fast inserts of $O(\log n)$. However, max heap has a space complexity of n rather than $5n$ for AVL tree. Therefore, using a max heap to sort the m most-liked weets is the most efficient since it has the best find max time, best insert time, and least space.

(d) We are given k sorted weet lists that are very easy to compare and wish to combine these k lists into a single sorted list of n weets. Therefore, we can just apply the merge step of the merge sort algorithm. More specifically, we combine two lists together by continuously comparing the first elements of the lists until the two lists are combined and sorted. The total number of merges is the number of times we can compare two lists out of k lists, which is $O(\log k)$. Since we are making at most n comparisons to create the overall sorted merged list, the run time of this algorithm is $O(n \log k)$.

Note that we have to use a comparison-based sorting algorithm since the timestamps are easy to compare but difficult to parse. The run times of heap sort and AVL sort are both $O(n \log n)$ since we have to add each weet to a binary tree and return the maxes/mins. Therefore, using the merge step in merge sort is the most efficient since $O(n \log k) < O(n \log n)$.

Problem 5-4.

(a) First, we look for unique anagram sets: (stop, spot, tops, pots), (altering, integral, triangle). For the first anagram set, there are $\binom{4}{2} = 6$ pairs; the second set has $\binom{3}{2} = 3$ pairs. Thus, there are $6 + 3 = 9$ distinct anagram pairs.

(b) Create two direct access arrays of length 26. For each string, we iterate over the letters and add one to the counter of the respective letter in the DAA (where $a = 1$, $b = 2$, etc.) via hashing. This takes $O(2(k + 26)) = O(k)$ time for both strings. Next, we iterate over the keys 1 to 26 and compare their respective counters. If all 26 counters are equal, then the two strings are anagrams. This comparison step takes $O(26)$ time. Therefore, the total run time of checking whether or not two strings are anagrams is $O(k) + O(26) = O(k)$.

(c) For each of the n strings of at most k length, we hash the word into a number by multiplying the letters together using the first 26 primes as the conversion factor. For example, if the word is "bad", then the hashed number would be $3 * 2 * 7 = 42$. Create a dictionary that keeps track of the hashed values. In addition, create a set that keeps track of the unique words already used. If the word is not in the set, then we add the hashed number to our dictionary. If the hashed number is not already in our dictionary, then we create a new dictionary entry with the hashed number as the key and 1 as the value. Otherwise, we add 1 to the value associated with the hashed value key. If the word is already in the set, then we skip it. To return the number of anagrams, we loop through each dictionary key, and if the value, v, is greater than 1, then add $\binom{v}{2}$ to the total; return the total.

This algorithm is correct since we are keeping track of unique anagrams and each word is uniquely hashed to a value. The algorithm has a worst case run time of $O(nk)$ since we are creating a hashed value by iterating through at most k letters for n words. We iterate through the $O(n)$ dictionary entries, but the $O(nk)$ term dominates. Moreover, we guarantee no collisions in the hashing process due to the property of unique prime factorization.

(d) Submit your implementation to `alg.mit.edu/PS5`