Problem Set 5

All parts are due on October 11, 2018 at 11PM. Please write your solutions in the LaTeX and Python templates provided. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct. Solutions should be submitted on the course website, and any code should be submitted for automated checking on `alg.mit.edu`.

Problem 5-1. [20 points] **Musical Pairs**

Your friend gave you a gift card to the AlgoRhythms music store for your birthday! The gift card value is exactly g dollars, and you want to spend the entire value without spending additional money. You have an unordered list of the n items the store sells, including the price of each in integer dollars. For each of the following scenarios, describe an algorithm satisfying the requested running time, to determine whether there exist **two** (not necessarily distinct) items from the catalog whose prices sum to **exactly** g dollars.

(a) [7 points] Describe an **expected** $O(n)$-time algorithm.

Solution: First insert each item into a hash table, keyed on price. Then, iterate through the list of items again, and for each item i with price p_i, check if $g - p_i$ exists in the hash table. If it does, then item i and this item from the hash table have prices that add to exactly g.

In total, the $2n$ hash table inserts and lookups take expected time $O(n)$.

Rubric:

- 5 points for correct description / justification of algorithm
- 2 points for correct run-time analysis
- Partial credit: 3 points for using hash table

(b) [7 points] For this part only, assume the item list is provided to you in sorted order by price. Describe a **worst-case** $O(n)$-time algorithm. **Hint:** Decrease and conquer!

Solution: Consider the two endpoints of our array, with prices p_0 and p_{n-1}, respectively. If $p_0 + p_{n-1} < g$, then item 0 is too small to appear in any pair with price adding to g, because $p_0 + p_i \leq p_0 + p_{n-1} < g$ for any $0 \leq i \leq n-1$. So we can safely ignore item 0, and decrease (and conquer!) into the subarray of items 1 through $n-1$. Similarly, if $p_0 + p_{n-1} > g$ then p_{n-1} is too big, so we can safely recurse on the subarray of items 0 through $n-2$. In the last case, $p_0 + p_n = g$, and we've found our pair.

The list shrinks by one item in $O(1)$ time per recursive call, so there is $O(n)$ work done across all $O(n)$ recursive calls.

Rubric:

- 5 points for correct description / justification of algorithm
- 2 points for correct run-time analysis
- Partial credit may be awarded

(c) [6 points] For this part only, assume that g satisfies $\log_n g = O(1)$. Describe a **worst-case** $O(n)$-time algorithm. **Hint:** Use part (b).

Solution: Let $c = \log_n g = O(1)$, meaning $g \leq n^c$. We may eliminate all items with price greater than g (in $O(n)$ time), so all remaining prices are at most $g \leq n^c$. Then these numbers can be sorted with radix sort, using at most $c + 1$ rounds of counting sort, taking $O((c + 1)n) = O(n)$ time. Then the algorithm in part (b) finishes the search in $O(n)$ time.

Rubric:

- 4 points for correct description / justification of algorithm
- 2 points for correct run-time analysis
- Partial credit: 3 points for using radix sort

Problem 5-2. [15 points] **Changeable-Priority Queue**

A minimum priority queue stores a set of keyed items supporting item insertion and operations involving the minimum key. In a few weeks, in shortest-path algorithms, we will want to support **changing** the key of an item stored in a priority queue. In order to uniquely identify each item, each item `x` will store a **unique integer identifier** `x.id` in addition to its key `x.key` (keys may not be unique). Design a data structure that supports the following operations in the requested time bounds:

`insert(x)`	Insert item `x`	$O(\log n)$
`find_min()`	Return an item with smallest key	$O(1)$
`delete_min()`	Remove an item with smallest key	$O(\log n)$
`find_id(id)`	Return item `x` having identifier `id`, or `None`	$O(1)$
`change_key(id, k)`	Change the key of item `x` having identifier `id` to `k`	$O(\log n)$

You may assume that `change_key(id, k)` is only called when an item with the query identifier exists, i.e. `id == x.id` for some stored item x. Each time bound may be worst-case, amortized, and/or expected, but you should clearly state which operations are subject to which restrictions within your implementation.

Solution: Keep a min heap, M, that organizes items by key. Additionally maintain a hash table, T, that maps each id to the index of the corresponding item in M: specifically, `M[T[id]]` is the item with identifier `id`. This means `find_id` requires one hash table lookup and one array lookup, taking expected $O(1)$ time. The `find_min` operation is even easier: just return `M[0]` in worst-case constant time.

The challenge is to keep these **cross-linked** data structures up to date with each other during dynamic operations. To insert an item x, we first add it to the end of array M (in amortized $O(1)$ time) and grow T by assigning `T[x.id]` to this new index (in amortized, expected $O(1)$ time).

Every time M swaps two entries x and y during `min_heapify_up`, say with indices i and j, we also update hash table T by swapping the values of `T[x.id], T[y.id]` from `i, j` to `j, i` (in expected $O(1)$ time per swap). This maintains the invariant that T maps ids to their correct positions in M, and as there are at most $O(\log n)$ swaps, the total runtime is amortized, expected $O(\log n)$.

The steps are analogous for `delete_min`: run M's corresponding operation as usual, and any time two items are swapped in M, swap the corresponding ids in T. Likewise, at the end when M's array pops its last item, remove the corresponding id from T. As above, this takes amortized, expected $O(\log n)$ time.

The `change_key` operation is similar. Lookup the item with `T[id]`, assign its new key, and perform `min_heapify_up` or `min_heapify_down` as needed, updating T along the way. M and T do not change size, so no amortized operations are needed. This is expected $O(\log n)$ time.

Rubric:

- 2 points for each operation implemented within the run time.
- 1 point per operation for proving its runtime.

Problem 5-3. [20 points] **Social Sorting**

Solve each of the following sorting problems by choosing an algorithm or data structure that best fits the application, and justify your choice. **Don't forget this! Your justification will be worth more points than your choice.** You may choose any algorithm or data structure we have covered in this class, or you may modify them as necessary to fit the scenario. If you find that multiple solutions could be appropriate for a scenario, identify their pros and cons, and choose the one that best suits the application. State and justify any assumptions you make. "Best" should be evaluated primarily by asymptotic running time, and secondly in terms of stability, space, and implementation.

Rubric:

- 1 point for statement of a choice of a sorting algorithm
- 3 points for reasonable justification of their choice
- Partial credit may be awarded
- 1 point for correct running time that is same or better than solutions

(a) [5 points] **Witter**, a competitor of Twitter that was also founded in 2006, prides itself as a "nanoblogging" platform: users can weet **weets** that are limited to 60 characters each. Given a list of all **60**-character weets weeted in '**06**, tell Witter how to sort them, first by weet length and then alphabetically.

Solution: We're assuming weets may be shorter than 60 characters, and that there are only $O(1)$ unique characters (e.g., ASCII has 256 characters, and Unicode has room for at most 1,114,112 characters). We can thus use tuple sort, first by length and then

by each of the 60-character slots. Use stable counting sort in each of the 61 rounds, taking $O(n)$ time and space.

(b) [5 points] Witter users love to **like** weets. They love liking so much in fact that the average number of likes per weet is much much more than the number of weets weeted in a year. Tell Witter how to quickly sort the **60**-character weets from '**06** by their number of likes.

Solution: Number of likes is unbounded, so any $O(n \log n)$ optimal comparison sort will do. Merge sort or heap sort can be accepted. Merge sort can be argued for speed, heap sort can be argued for space.

(c) [5 points] Witter's front page maintains and displays the m **most-liked** weets of the day. Witter needs their listing to stay current as new weets are weeted and liked. Tell Witter how to keep their most-liked weet listing sorted throughout the day.

Solution: An AVL tree is the only data structure we've discussed that can maintain dynamic order operations in sub-linear time, so it is the clear choice; but if all weets have the same number of likes, finding any particular weet in the tree can be slow. So store weets in an AVL Tree keyed by likes, cross-linked with a hash table mapping each weet to its node in the tree. When a new weet is weeted, add it to the tree and hash table, and when an existing weet is liked, find the weet in the hash table, update its likes, and then maintain the tree. Maximum 4 points if only an AVL tree is used.

(d) [5 points] A small, dedicated group of k MIT friends are avid weeters who have been weeting **60**-character weets on Witter, consistently since '**06**. Given a list of each friend's weets in chronological order—a total of $n \gg k$ weets combined— tell the friends how to combine their lists into one chronological timeline containing all of their weets. Note: Witter timestamps are encoded in a confusing proprietary format that are easy to compare but otherwise difficult to parse. **Hint:** aim for $O(n \log k)$.

Solution: First solution: Use merge sort! In each round you merge sorted timelines in pairs and therefore cut the number of timelines in half, so there are $O(\log k)$ rounds. Each round takes $O(n)$ time, making $O(n \log k)$ time total.

Second solution: Use a technique halfway between heapsort and mergesort. The heap stores the minimum element from each of the k sorted streams. Each item in the heap remembers which stream it came from, so when `remove_min` is called, the next item from the same stream can be inserted. This performs a k-way merge operation in $n \log k$ time.

Problem 5-4. [45 points] **Anagram Pairs**

Two strings whose spellings are permutations of each other are known as **anagrams** of each other; for example `(indicatory, dictionary)` or `(brush, shrub)` are anagrams. For this problem, strings will be written in lowercase using only the English letters a–z.

(a) [5 points] Show that the sentence `'stop altering the integral spot on the tops of those triangle spot pots'` has nine distinct **pairs** of strings that are anagrams of each other, and explain your computation. Note that tuples `('stop','tops')` and `('tops','stop')` represent the same anagram pair, while `('spot','spot')` is not a valid anagram pair.

Solution: There are two groups of anagrams, `('stop', 'pots', 'tops', 'spot')` and `('altering', 'integral', 'triangle')`. There are $\binom{4}{2} = 6$ anagram pairs in the first group, and $\binom{3}{2} = 3$ pairs in the second group, so 9 overall.

Rubric:

- 5 points for a correct justification (full credit for listing out all pairs)
- Partial credit may be awarded

(b) [5 points] Given two strings s_1 and s_2, each containing no more than k letters, describe how to use direct access arrays to determine whether s_1 and s_2 are anagrams of each other in worst-case $O(k)$ time.

Solution: We'll build a frequency table for s_1: a length-26 array A where entries $A[0], A[1], \ldots, A[25]$ count the number of 'a's, 'b's, ..., 'z's in string s_1, respectively. To do this, initialize A's entries to 0, and for each character of s_1, treat A as a direct access array and increment the appropriate counter. This takes $O(k)$ time. If we create a similar frequency table B for string s_2, then s_1 and s_2 are anagrams if and only if A and B carry the same values at every index (i.e., have the same letter frequencies) but s_1 and s_2 are not identically the same string. Comparing the 26 entries of A and B takes $O(1)$ time, and comparing the $\leq k$ characters of s_1 and s_2 takes time $O(k)$.

Alternatively, use counting sort to rearrange s_1 into a string t_1 whose letters are sorted order, and likewise rearrange s_2 into a sorted string t_2. Then s_1 and s_2 are anagrams if $s_1 \neq s_2$ but $t_1 = t_2$.

Rubric:

- 3 points for a correct algorithm
- 2 points for runtime analysis

(c) [10 points] Given a list of n strings, each of length at most k, describe an algorithm to compute the total number of anagram pairs in the list (as in part (a), the list may contain duplicate strings). Your algorithm should run in $O(nk)$ time (specify whether your running time is expected or worst-case). [1]

[1] **Note**: You may assume without proof that a string of k characters can be hashed in $O(k)$ time, via a hash function chosen randomly from a universal hash family. (If you're curious, CLRS Exercise 11.3-6 implies one method, also referenced on Wikipedia.) You can solve this problem without using this fact, but feel free to use it if you wish.

Solution: First solution: Tuple Sort. To remove duplicate strings, start with tuple sort to sort the n strings in alphabetical order. The k rounds of counting sort require $O(nk)$ time total. Repeated strings will now be next to each other, so by comparing adjacent strings in the sorted list, we can make a new list of the **distinct** strings.

Next, compute the frequency table for every string (in $O(nk)$ time) and tuple-sort these frequency tables. The 26 rounds of counting sort take $O(n)$ time total. All strings with the same frequency table are now together, so by reading this list in order we can compute all unique frequency tables, T_i, and how many times each one appears, m_i. The answer is then the sum of $\binom{m_i}{2}$. In all, we needed $O(nk)$ time, worst-case.

Second Solution: Hashing. The universal hash family discussed in class defined by $(ax + b \bmod p) \bmod m$ was only designed for $O(1)$-sized keys, because arithmetic on k-word integers gets hairy for non-constant k. So we need a different strategy for hashing length-k strings efficiently. Thankfully there do exist universal hash families that take $O(k)$ time to compute the hash of a length-k string (see footnote in problem statement).

Use one of these hash families to insert all n strings into a hash table, eliminating repeats along the way. This takes $O(nk)$ time to compute the hashes, and each insert uses $O(1)$ expected string comparisons and therefore $O(k)$ work, totaling to $O(nk)$ expected work. The keys now form the list of distinct strings.

Compute the frequency tables for these distinct strings as above ($O(nk)$ time), and insert these frequency tables into a second hash table. These keys have size $O(1)$, so our standard hashing techniques apply, taking $O(n)$ expected time for the n inserts. This hash table allows us compute the dintinct frequency tables T_i and their number of occurrences m_i as above. Overall, this takes expected $O(nk)$ time.

Rubric:

- 7 points for a correct algorithm
- 3 points for runtime analysis

(d) [25 points] Implement your function `count_anagrams` in the code template provided. The input will be a tuple of (possibly repeated) lowercase strings, such as

```
('at', 'least','do', 'not', 'steal', 'the', 'slate', 'tesla'),
```

and your function should return the number of anagram pairs (in this example, six). You can download a code template containing some test cases from the website. Built-in Python functions `ord('a') == 97` and `chr(97) == 'a'` may be used, but for this problem, **you may NOT use Python's built-in sort functionality** (the code checker will remove `list.sort` and `sorted` from Python prior to running your code). Submit your code online at `alg.mit.edu`.

Solution:

```
def frequency_table(word):
    table = [0] * 26
    for c in word:
        table[ord(c) - 97] += 1
    return tuple(table)

def count_anagrams(words):
    count = {}
    unique = {}
    for word in words:
        if word not in unique:
            unique[word] = True
            table = frequency_table(word)
            if table in count:
                count[table] += 1
            else:
                count[table] = 1
    total = 0
    for table in count:
        total += (count[table] - 1) * count[table] // 2
    return total
```