# 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.

**(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!

**(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).

**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.

**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.

**(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.

**(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.

**(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.

**(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 $O(n \log k)$.

**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.

**(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.

**(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]

**(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`.

```python
def count_anagrams(strings):
    "Return the number of pairs of anagrams in a tuple of strings"
    ##################
    # YOUR CODE HERE #
    ##################
    return 0
```

[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.