Problem Set 5

All parts are due on March 16, 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 `py.mit.edu/6.006`.

Problem 5-1. [35 points] **Dynamic Hash Browns**

(a) [5 points] Insert integer keys $(1, 6, 2, 4, 7, 8, 3)$ in order into a hash table using hash function $h(k) = (11 * k + 5) \mod 7$. Collisions should be resolved via chaining, where collisions are stored at the end of a chain. Draw a picture of the hash table after all keys have been inserted.

Solution:

```
 0 1 2 3 4 5 6
+-+-+-+-+-+-+-+
|4|6|.|3| |7|2|
+-+-+|+-+-+-+-+
     V
    +-+
    |1|
    +-+
    |.|
    +|+
     V
    +-+
    |8|
    +-+
    | |
    +-+
```

Rubric:

- 5 points for drawing hash table
- -1 point per incorrect insertion

(b) [10 points] Given an array A containing n integers, describe an expected $O(n^2)$ time algorithm to determine whether two pairs of distinct integers from A have the same sum, i.e. there exists $\{a, b, c, d\} \subseteq A$ where $a + b = c + d$.

Solution: This problem is a bit tricky. The main approach will be to insert all $O(n^2)$ possible pair sums into a hash table. If an insertion ever finds that a pair sum already

exists in the hash table, then we've found two pairs that sum to the same value. However, this approach only works if the pairs found do not share an array element in common. For example, input $(a, b, c) = (1, 2, 1)$, allows two pairs $(a + b) = (b + c)$, but b is common between them, so do not represent two pairs of **distinct** integers. This cannot occur when the integers in A are distinct. If students provide only the above solution without dealing with repeated array elements, we will give significant partial credit, but limit to maximum 7 points.

There are multiple ways to get around this problem; we outline one approach below. Pre-process to find duplicates in the input by inserting each array element into a hash table, but for each integer value, store a list of array elements having the same value. Each insertion can be performed in expected amortized constant time. There are few possibilities on the structure of this duplicates hash table:

- If the lists associated with two distinct integer values each contain two or more array elements, we will can construct a satisfying quadruple by choosing two elements from each, so return True.
- Alternatively, exactly one integer value list contains two or more array elements.
  - If the list contains four or more array elements, four of them comprise a satisfying quadruple, so return True.
  - Otherwise, at most three array elements have the same value. Run the initial algorithm as described above. Now, if a pair sum collision occurs, either the collision represents a satisfying quadruple, or if the array elements overlap, it is because they are of the form (a, b) and (a', b). If $a' + b = c + d$ for some other pair (c, d) for which $c \neq a$ and $d \neq a$, then $a + b$ also equals $c + d$, and we can safely ignore pair (a', b).

The pre-processing step to find duplicates takes expected $O(n)$ time to discover which case above holds, while expected $O(n^2)$ will be used in the last case, leading to an expected $O(n^2)$ time algorithm.

```
1  def match_pair_sums(A):
2      values = {}
3      for i in range(len(A)):
4          if A[i] not in values:
5              values[A[i]] = [i]
6          else:
7              values[A[i]].append(i)
8      first = True
9      for v in values:
10         # check for one value with four or more array elements
11         if 4 <= length(v):
12             return True
13         # check for two values with more than one array element
14         if 2 <= length(v):
15             if first:
16                 first = False:
```

```
            else:
                return True
    # otherwise, pair sum collisions can be ignored
    sum_to_pair = {}
    for i in range(1, len(A)):
        for j in range(i):
            s, pair = A[i] + A[j], (i, j)
            if s in sum_to_pair:
                k, l = sum_to_pair[s]
                if i == k or i == l or j == k or j == l:
                    # pair indices collide, ignore it!
                    continue
                # pairs correspond to distinct array elements
                print('Array indices %s and %s both sum to %s!' %
                    (pair, sum_to_pair[s], s))
                return True
            sum_to_pair[s] = pair
    return False
```

Rubric:

- 7 points for a correct algorithm description
- 3 points for running time analysis
- Max 7 points if distinct array elements not checked

(c) [10 points] A dynamic array efficiently implements an **addressable stack** interface, supporting worst-case constant time array indexing, and insertion and removal of items at the back of the array (the largest index) in amortized constant time. But the same operations will take linear time when performed at the front of the dynamic array (at index zero), as items must be shifted down. Design a **double-ended addressable queue** data structure to store a sequence of items that supports **worst-case** constant time array indexing to the ith item in the stored sequence, as well as four dynamic operations: insertion and removal of items from both the front and back of the sequence, where each dynamic operation runs in at most **amortized** constant time.

Solution: There are many possible solutions. One solution modifies the two-stack queue implementation described in lecture, where care needs to be taken when popping from an empty stack, or pushing to a full stack. An alternative approach would be to store the queued items in the middle of an array rather than at the front, leaving a linear number of extra slots at both the beginning and end whenever rebuilding occurs, guaranteeing that linear time rebuilding only occurs once every $\Omega(n)$ operations.

For example, whenever reallocating space to store a sequence of n elements, copy them to the middle of a length $m = 3n$ array. To insert or remove an item to the beginning or end of the sequence, add or remove an element at the appropriate end in constant time. If no free slot is exists during an insertion, at least a linear number of insertions must have happened since the last rebuild, so we can afford to rebuild the array. If removing an item brings the ratio n/m of items to array size to below $1/6$,

at least $m/6 = \Omega(n)$ removals must have occurred since the last rebuild, so we can again afford to rebuild the array.

A linear number of operations between expensive linear time rebuilds ensures that each dynamic operation takes at most amortized $O(1)$ time. To support array indexing in constant time, we maintain the index location i of the left-most item in the array and the number of items n stored in the array, both of which can be maintained in worst-case constant time per update. To access the jth item stored in the queue sequence using zero-indexing, confirm that $i + j < n$ and return the item at index $i + j$ of the array container in worst-case constant time.

Rubric:

- 4 points for data structure description
- 4 points for amortized analysis for dynamic operations
- 2 points for constant time indexing
- Partial credit may be awarded

(d) [10 points] Your favorite bookstore, Drytent, recently closed because of a fire. As a result of the fire, their books have been torn to pieces and scattered throughout the store. As you survey the damage, you find a browned scrap of paper upon which is written a passage of p beautifully written and particularly inspiring words, and you are determined to find the book it came from. The owners of Drytent give you access to an online repository containing text from the set B of books they sell, where the text from book $b \in B$ contains w_b words. You may assume that each word contains at most twenty characters. Describe an expected $O(p + \sum_{b \in B} w_b)$ time algorithm to determine which book contains the inspiring passage.

Solution: Compute a rolling hash of characters contained in passage p in $O(p)$ time, since each word contains at most a constant number of characters. For each book $b \in B$, apply Rabin-Karp to check if p is a substring of b in expected $O(w_b)$ time, by comparing each length p sub-string of b from a close rolling hash in amortized expected $O(1)$. Doing this for all books then yields expected running time $O(p + \sum_{b \in B} w_b)$ as desired.

Rubric:

- 7 points for a correct algorithm description
- 3 points for running time analysis
- Partial credit may be awarded

Problem 5-2. [20 points] **Trading Portfolios**

Herkshire Bathaway (HB) conducts business on the stock market, entering a huge number of trades every day. On the other side of each trade is some company which we call a **counterparty** of HB. Each trade that HB makes is assigned a portfolio name, though trades with different counterparties may be assigned the same portfolio, and trades with the same counterparty may be assigned different portfolios, according to an internal policy that cannot be questioned.

Whenever HB enters into a trade, a new line $(+, C, P)$ is appended to a daily log file: the plus sign indicates that a new trade was initiated by HB, the C indicates the counterparty identifier, and the P is the portfolio associated with that trade. However at any time during a day, a counterparty may cancel all trades with HB so far that day. Such a cancellation would be logged using a line such as $(-, C)$. Here, the minus sign indicates that a cancellation took place, with C identifying the counterparty who requested the cancellation. Here is an example of an HB daily log file.

```
1   +   MalWart      horizon
2   +   DomeHepot    emerald
3   +   MalWart      emerald
4   +   KolaKoka     diamond
5   -   MalWart
6   +   KolaKoka     horizon
7   +   FoleWhoods   platinum
8   +   KurgerBing   gold
9   -   DomeHepot
```

At the end of each day, HB needs to determine the set of distinct portfolios associated with trades that are still **active**, i.e. portfolios of trades that were not canceled during the day. For the provided HB daily log, the portfolios associated with active trades at the end of the day are `horizon`, `diamond`, `platinum`, and `gold`.

(a) [10 points] Given an HB daily log containing n entries, describe a worst-case $O(n^2)$ time algorithm to determine the set of distinct portfolios still associated with active trades at the end of the day. You may assume that names of portfolios and counterparties can be compared in constant time.

Solution: Process each line of the log from top to bottom while maintaining a list (dynamic array) storing active portfolio names. For each plus line $(+, C, P)$ scan all $O(n)$ lines below it and check if the line $(-, C)$ appears in $O(1)$ time per line. If $(-, C)$ appears in any subsequent line, the plus line was canceled during the day, so skip it and process the next plus line. If $(-, C)$ does **not** appear in any subsequent line, then the plus line was **not** canceled during the day by definition of active, so we must add it to the active portfolio list. Search through the list of active portfolio names for P in $O(n)$ time; if P does not appear, append P to the end of the dynamic array in amortized $O(1)$ (but worst-case $O(n)$) time. The algorithm processes all $O(n)$ lines, and each line takes at most worst-case $O(n)$ to process, so the algorithm runs in worst-case $O(n^2)$ time.

Rubric:

- 7 points for a correct algorithm description
- 3 points for running time analysis
- Partial credit may be awarded

(b) [10 points] Herkshire Bathaway is not happy with your quadratic solution and announces a round of layoffs. You decide that, for job security, you need to come up with a better algorithm. Describe an algorithm that produces a list of distinct portfolios associated with active trades which runs in expected $O(n)$ time.

Solution: Store a hash table of hash tables. The top level hash table maps each counterparty to a hash table maintaining the set of portfolios associated with the counterparty. To process plus line $(+, C, P)$, add P to the hash table associated with C in expected amortized $O(1)$ time per insertion. To process a minus line $(-, C)$, replace the hash table associated with C with an empty hash table, also in expected constant time. The resulting size of the data structure will be worst-case $O(n)$ because each portfolio in a counterparty hash table uniquely corresponds to a line of the log, and we can enforce that each hash table maintain a constant load factor. To return the list of portfolios at the end of the day, construct a new output hash table to store distinct portfolios. Scan all counterparty hash tables in worst-case $O(n)$ time, adding each portfolio found to the output portfolio hash table in expected amortized $O(1)$ time per insertion. Then return all portfolios contained in the output portfolio hash table via a worst-case linear time scan. Thus in expectation over an appropriate choice of hash function, this algorithm runs in expected $O(n)$ time.

Rubric:

- 7 points for a correct algorithm description
- 3 points for running time analysis
- Partial credit may be awarded

Problem 5-3. [45 points] **Tedious Decryption**

Bob wants to send Alice a secret message of characters via a specially encoded sequence of integers. Each character of Bob's message corresponds to a **satisfying** triple of distinct integers from the sequence satisfying the following property: every permutation of the triple occurs consecutively within the sequence.

For example, if Bob sends the sequence $A = (4, 10, 3, 4, 10, 9, 3, 10, 4, 3, 10, 4, 8, 3)$, then the triple $t = \{3, 4, 10\}$ is satisfying because every permutation of t appears consecutively in the sequence, i.e. $(3, 4, 10)$, $(3, 10, 4)$, $(4, 3, 10)$, $(4, 10, 3)$, $(10, 3, 4)$ and $(10, 4, 3)$ all appear as consecutive sub-sequences of A.

For any satisfying triple, its **initial occurrence** is the smallest index i such that the triple is also satisfying, for the prefix of the input sequence ending at index i. For example, the initial occurrence of t in A is 10.

Bob's secret message will be formed by the characters corresponding to each satisfying triple in the encoded sequence, increasingly ordered by initial occurrence. To convert a satisfying triple (a, b, c) into a character, take the remainder of its sum divided by 27: remainders 0 to 25 correspond to the lower-case letters 'a' to 'z', while remainder 26 corresponds to a space character. For example, the triple $\{3, 4, 10\}$ corresponds to the letter 'r'.

(a) [5 points] Suppose Bob sends Alice the following sequence.

$(10, 13, 9, 10, 13, 5, 2, 13, 5, 10, 9, 13, 10, 9, 13, 2, 5, 13, 2, 67, 23, 1, 2, 10, 1, 2, 1, 10, 2, 1)$

Write down the list of satisfying triples contained in the sequence, as well as their associated characters, ordered by initial occurrence. What is the secret message?

Solution: The list of satisfying triples is below, yielding the secret message 'fun'.

Triple	Remainder	Letter	Initial Occurrence
(13,10,9)	5	f	13
(5,13,2)	20	u	18
(10,2,1)	13	n	29

Rubric:

- 1 point per correct triple (3)
- 2 points for correctly decoded message

(b) [10 points] Bob has started to send longer messages, and Alice is tired of decoding his messages by hand. Design an expected $O(n)$ time algorithm to decode a sequence of n integers from Bob, and output his secret message.

Solution: Scan each consecutive triple from Bob's sequence, and use a hash table to keep track of the triples seen so far. When processing a triple of integers (a, b, c), check whether a, b and c are unique. If they are not unique, then (a, b, c) cannot be a satisfying triple. It they are unique, check whether (a, b, c) is stored in the hash table.

If so, we've seen this triple before, so this triple will not complete an initial occurrence of a satisfying triple. Otherwise, if (a, b, c) is not in the hash table, add the newly seen ordered triple to the hash table. Then check whether the five permutations of the triple, $(a, c, b), (b, a, c), (b, c, a), (c, a, b), (c, b, a)$, all exist in the hash table. If all five permutations exist in the hash table, this means that we have previously encountered all the other permutations, followed by the first occurrence of (a, b, c). This is the definition of a satisfying triple, so append the triple's decoded character to an output string. Each hash table insert and find operation takes expected $O(1)$ time; each triple requires at most six finds and one insertion, so can be processed in expected $O(1)$ time. Thus, to process all $O(n)$ consecutive triples of Bob's sequence can be done in expected $O(n)$ time.

Rubric:

- 7 points for a correct algorithm description
- 3 points for running time analysis
- Partial credit may be awarded

(c) [30 points] Write the Python function `decode_message(sequence)` that implements your decoding algorithm and returns the secret message. You may assume that all elements in the input list are non-negative integers. You can download a code template containing some test cases from the website. The template code contains a helper method `convert_to_char(k)`, which converts an integer k to a character according to the procedure described above. Submit your code online at `py.mit.edu/6.006`.

Solution:

```
def convert_to_char(k):
    mod = k % 27
    if mod == 26:
        return ' '
    return chr(97 + mod)

def decode_message(sequence):
    triples = set()      # set is a hash table storing only keys
    out = ''             # (dict also stores a value with each key)
    for i in range(len(sequence) - 2):      # loop through O(n) triples
        a, b, c = sequence[i:i + 3]         # O(1) slice
        if a != b and a != c and b != c:    # check unique
            if (a, b, c) not in triples:    # check new
                triples.add((a,b,c))                # add new
                if (    (a, c, b) in triples and    # check permutations
                        (b, a, c) in triples and
                        (b, c, a) in triples and
                        (c, a, b) in triples and
                        (c, b, a) in triples):
                    out += convert_to_char(a + b + c)   # add character
    return out
```