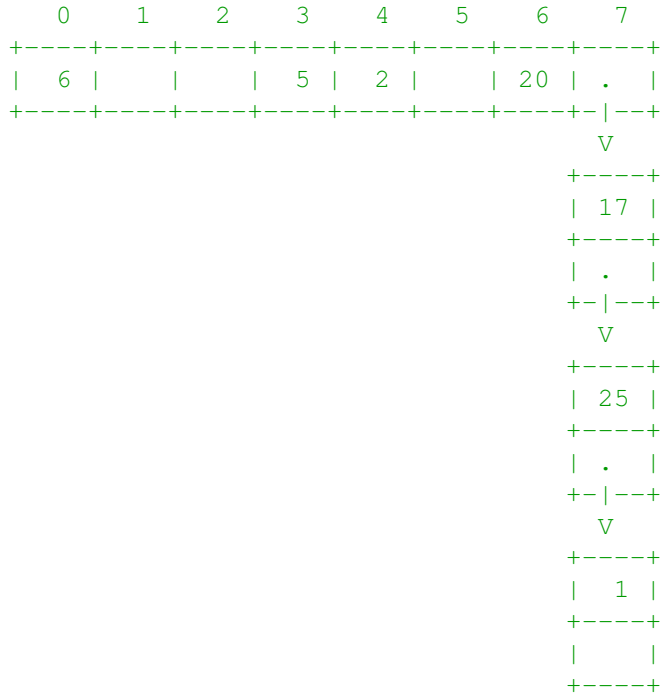
Problem Set 4

All parts are due on March 8, 2019 at 6PM. 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 4-1. [8 points] **Hash Practice**

Insert integer keys $(2, 17, 6, 20, 5, 25, 1)$ in order into a hash table using hash function $h(k) = 13k + 2 \bmod 8$. 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. What would happen if you tried to insert the key 25 into the hash table again?

Solution:



$h(25) = 7$ so we would try to insert it into the chain at slot 7. However, 25 already exists in the chain, so we do not add it to the chain again because hash tables do not support duplicate keys. Specifics of how chains are drawn should be flexible; it is most important that student hashes keys with the same hash to the same chain.

Rubric:

- 1 point per correct insertion (7)
- 1 point for explanation of repeated key

Problem 4-2. [8 points] **Hashing Insecurities**

In this problem, we call a hash family $\mathcal{H}$ **insecure** if there exists a pair of distinct keys (k_1, k_2) for which $h(k_1) = h(k_2)$ for all $h \in \mathcal{H}$.

(a) [2 points] Can a universal hash family ever be insecure? Explain.

Solution: A hash family $\mathcal{H}$ is universal if for any pair of keys k_1, k_2, the probability over $h \in \mathcal{H}$ that $h(k_1) = h(k_2)$ is at most $1/m$. We claim if $\mathcal{H}$ is universal and $m > 1$, it is not insecure. Suppose for contradiction, it is insecure and there exists two keys for which $h(k_1) = h(k_2)$ for all $h \in \mathcal{H}$. Then the probability that $h(k_1) = h(k_2)$ is 1 which is larger than $1/m$, so $\mathcal{H}$ is not universal which is a contradiction.

Rubric:

- 2 points for a correct argument
- Partial credit may be awarded

Prove that the following families of hash functions (mapping keys from $\{0, \ldots, u-1\}$ to the range $\{0, \ldots, m-1\}$) are each insecure.

(b) [2 points] $\mathcal{H}_b = \{h(k) = (k + a) \bmod m \mid a \in \{1, \ldots, u\}\}$

Solution: Choose any $k_1 \equiv k_2 \bmod m$, for example $k_2 = k_1 + m$. Then confirm $h(k_1) = (k_1 + a) \bmod m = (k_1 \bmod m) + (a \bmod m) = (k_2 \bmod m) + (a \bmod m) = (k_2 + a) \bmod m = h(k_2)$ as desired.

(c) [2 points] $\mathcal{H}_c = \{h(k) = ak \bmod m \mid a \in \{1, \ldots, u\}\}$

Solution: Choose any $k_1 \equiv k_2 \bmod m$, for example $k_2 = k_1 + m$. Then confirm $h(k_1) = ak_1 \bmod m = (a \bmod m)(k_1 \bmod m) = (a \bmod m)(k_2 \bmod m) = ak_2 \bmod m = h(k_2)$ as desired.

(d) [2 points] $\mathcal{H}_d = \{h(k) = (h_s(\lfloor k/s \rfloor) + h_s(k - s\lfloor k/s \rfloor)) \bmod m \mid h_s \in \mathcal{H}_s\}$ where $s = \sqrt{u}$ is an integer, and $\mathcal{H}_s$ is a universal family of hash functions $h_s : \{0, \ldots, s-1\} \to \{0, \ldots, m-1\}$.

Solution: Given key k_1, let $p = \lfloor k_1/s \rfloor < s$ and $q = k_1 - ps < s$ so that $k_1 = ps + q$, and construct $k_2 = qs + p$. Then confirm that $h(k_1) = h_s(\lfloor (ps+q)/s \rfloor) + h_s(ps + q - s\lfloor (ps+q)/s \rfloor) = h_s(p) + h_s(q) = h_s(q) + h_s(p) = h_s(\lfloor (qs+p)/s \rfloor) + h_s(qs + p - s\lfloor (qs+p)/s \rfloor) = h(k_2)$ as desired. Note that this only works when $k_1 \neq k_2$, i.e., when $p \neq q$.

Rubric: For each (b), (c), (d):

- 1 point for a pair of keys certifying insecurity
- 1 point for justification

Problem 4-3. [8 points] **Changeable Priority Queue**

When we discuss shortest-path algorithms in a few weeks, we will need a priority queue which supports **changing** the key of a stored item. In order to identify items, each item `x` stores a unique integer **identifier** `x.id` in addition to its key `x.key` (keys might not be unique). Describe a data structure that supports the following operations, each in $O(\log n)$ time:

- `insert(x)`: insert item `x`
- `delete_min()`: remove an item with smallest key
- `change_key(id, k)`: change key of stored item `x` with identifier `id` to `k` (if it exists)

Solution:

There are multiple ways to solve this problem, but the main idea is to store items in two data structures: one maintaining dynamic and efficient operations on the item with minimum key (a key data structure), and the other enabling fast look up by identifier (an id data structure) pointing to the item's location in the key data structure. For the former, we can use an AVL tree or a min heap; if a min heap is used, these operations will be amortized $O(\log n)$ running times. For the later, we can use an AVL tree or a hash table; if an AVL tree is used, the operation running times will be worst-case $O(\log n)$ time, while if a hash table is is used, the operation running times will be expected amortized $O(\log n)$ time (either are acceptable for this problem).

Note that this problem can also be solved using a single AVL tree, storing items keyed by identifier, augmenting each AVL node with the identifier of an item from its subtree having minimum key. Such a data structure can support each of the requested operations in $O(\log n)$ time; we leave the full argument as an exercise to the reader. We instead present below a full solution to one of the previously described linked data structures.

We describe an implementation using a min heap storing items keyed by key and a hash table mapping item IDs to an array index of the min heap. We initialize each with zero items in constant time. To implement `insert(x)`, insert x into the min heap in amortized $O(\log n)$ time, making note of its final index location i in the min heap. Then insert $x.id$ into the hash table mapping to index location i in expected amortized $O(1)$ time. In addition, for each item y whose index j changed in the min heap during the insertion, updated its mapping in the hash table $y.id$ to j. There are at most $O(\log n)$ of these items, so these index pointers can be maintained in expected $O(\log n)$ time. To implement `delete_min()`, remove the item with minimum key from the min heap in amortized $O(\log n)$ time, remove its ID from the hash table in amortized expected $O(1)$ time, and then update the hash table with all the item min heap indices that changed during the deletion; as with insertion, there are at most $O(\log n)$ items from the heap whose position may have changed, so this operation can be implemented in expected amortized $O(\log n)$ time.

Lastly, to implement `change_key(id, k)`, lookup the ID in the hash table, to find that item in the min heap in expected $O(1)$. Update the stored item's key to the new key. This update may invalidate the min heap property, but the property can be restored by at most $O(\log n)$ swaps up or down in the heap (as in insert and delete min). Again, for each item which changes index in the

min heap, update its mapped location in the hash table in expected $O(1)$ time. Since there are at most $O(\log n)$ items whose index changes, this operation takes expected $O(\log n)$ time.

Rubric:

- 2 points for description of data structure
- 2 points per operation (description, correctness, running time)
- Partial credit may be awarded

Problem 4-4. [16 points] **Evenly Uneven**

Loapo Ohyeah and her team are competing in a speed skating relay race having n race segments, where each race segment has integer length. Loapo will skate exactly two of the race segments, and she needs to choose which ones to skate. In the interest of pacing, Loapo wants her pair of race segments to be **evenly uneven**: specifically, she wants one of her two race segments to be exactly four times the length of the other.

(a) [8 points] Given the list of race segments in the relay, describe an **expected** $O(n)$-time algorithm to find an evenly uneven pair, or return that no such pair exists.

Solution: Initialize an empty hash table with size $O(n)$. For each of the n race segments, insert its length into the hash table in expected $O(1)$ time. Then, for each race segment with length s, lookup whether $4s$ exists in the hash table. If it does, an evenly uneven pair exists, so return $(s, 4s)$. Otherwise, no pair satisfies the property. This algorithm performs $O(n)$ hash table operations, each in expected $O(1)$ time, so this algorithm runs in expected $O(n)$ time.

Rubric:

- 3 points for description of a correct algorithm (even if inefficient)
- 2 points for correct running time analysis (even if inefficient)
- 3 points if correct algorithm is expected $O(n)$
- Partial credit may be awarded

(b) [8 points] If the length of every race segment is less than n^4, describe a **worst-case** $O(n)$-time algorithm to find an evenly uneven pair, or return that no such pair exists.

Solution: In this part, race segment lengths are bounded in a polynomial range, so we can use radix sort to sort them in worst-case $O(n)$ time into an array A. Then, we can sweep the sorted list to find an evenly uneven pair if it exists. Specifically, initialize indices $i = 0$ and $j = 1$, and repeat the following procedure. If $A[j] = 4A[i]$, then you have found an evenly uneven pair, so return $(A[i], A[j])$. If $i = j$ or $A[j] < 4A[i]$, then $A[i]$ cannot be in an evenly uneven pair with any $A[k]$ for $k \leq j$, so increase j by one. If $A[j] > 4A[i]$, then $A[j]$ cannot be in an evenly uneven pair with any $A[k]$ for $k \leq i$, so increase i by one. If j reaches n, then return not found. This loop maintains the invariant that at the start of each loop, we have confirmed that no pair $(A[k], A[l])$ for $k \leq i$, $l < j$, and $k < l$ is evenly uneven, so it is correct. Since each iteration

of the loop takes $O(1)$ time and increases either i or j by one, and since $i \leq j \leq n$ throughout, this procedure takes at most $O(n)$ time in the worst case.

Rubric:

- 3 points for description of a correct algorithm (even if inefficient)
- 2 points for correct running time analysis (even if inefficient)
- 3 points if correct algorithm is worst-case $O(n)$
- Partial credit may be awarded

Problem 4-5. [10 points] **Doggo Sort**

Kindella Angel has $101 \cdot n$ dalmation pups that she loves very much and would never make coats out of them. However, while most of these puppies are oh so very good pups, Kindella admits that some of them are better than others, and better pups deserve more treats. Help Kindella sort her puppies, so she can decide who gets the most treats.

(a) [5 points] Kindella has decided that the best pups have the most spots. She knows that the total number of spots on all of the puppies is exactly n^{101}. Given a list of the spots on each pup, describe an efficient[1] algorithm to sort these pupperinos by their number of spots.

Solution: The number of spots is polynomial in n, so we can use radix sort to sort the dogs by their spots in $O(n)$ time.

Rubric:

- 2 points for description of a correct algorithm (even if inefficient)
- 1 points for correct running time analysis (even if inefficient)
- 2 points if correct algorithm is worst-case $O(n)$
- Partial credit may be awarded

(b) [5 points] Although spots are very important, Kindella has found that many puppies have the same number of spots. She concludes that the best pups actually have the highest **spot density**: each pupper has an integer representing the surface area of its body, and a pup's spot density is its number of spots divided by its surface area. Given a list containing each pup's spot number and surface area, describe an efficient[1] algorithm for Kindella to sort her puppies by spot density, once and fur all.

Solution: Since we have no bound on surface area, the number of possible keys is not polynomially bounded, so the best we can do is an optimal comparison sort, e.g., merge sort, heap sort, or AVL sort. The spot densities of two dogs can be compared by cross multiplication: a dog with s_1 spots and surface area a_1 has a higher density than a dog with s_2 spots and surface area a_2 if and only if $s_2 a_1 > s_1 a_2$.

Rubric:

[1] By "efficient", we mean that faster correct algorithms will receive more points than slower ones.

- 2 points for description of a correct algorithm (even if inefficient)
- 1 points for correct running time analysis (even if inefficient)
- 2 points if correct algorithm is worst-case $O(n \log n)$
- Partial credit may be awarded
- Max 2 points total if non-integer division is used (finite precision)

Problem 4-6. [50 points] **Po-*k*-er Hands**

Meff Ja is a card shark who enjoys playing card games. He has found an unusual deck of cards, where each of the n cards in the deck is marked with a lowercase letter from the 26-character English alphabet. We represent a deck of cards as a sequence of letters, where the first letter corresponds to the top of the deck. Meff wants to play a game of Po-k-er with you. To begin the game, he deals you a Po-k-er hand of k cards in the following way:

1. The deck D starts in a pile face down in a known order.
2. Meff **cuts** the deck uniformly at random at some location $i \in \{0, \ldots, n-1\}$, i.e., move the top i cards in order to the bottom of the deck.
3. Meff then deals you the top k cards from the top of the cut deck.
4. You **sort** your k cards alphabetically, resulting in your Po-k-er **hand**.

Let $h(D, i, k)$ be the Po-k-er hand resulting from cutting a deck D at location i. Then cutting deck $D =$ `'abcdbc'` at location 2 would result in the deck `'cdbcab'`, which would then yield the Po-4-er hand $h(D, 2, 4) =$ `'bccd'`. From a given starting deck, many hands are possible depending on where the deck is cut. Meff wants to know the **most likely** Po-k-er hand for a given deck. Given that the most likely Po-k-er hand is not necessarily unique, Meff always prefers the lexicographically smallest hand.

(a) [5 points] Given the deck $D =$ `'koperrepokpokeropkre'`, list the set of all possible Po-3-er hands $\{h(D, i, 3) \mid i \in \{0, \ldots, 19\}\}$, along with their probabilities of appearing given a cut location chosen uniformly at random.

Solution:

`(kop, 6/20), (ekr, 3/20), (err, 2/20), (epr, 2/20), (eop, 2/20), (eko, 2/20), (opr, 1/20), (kpr, 1/20), (eor, 1/20)`

Rubric:

- $\lceil n/2 \rceil$ points, where n is the number of correct hand probabilities

(b) [10 points] Describe a data structure that can be built in $O(n)$ time from a deck D of n cards and integer k, after which it can support `same(i, j)`: a constant-time operation which returns True if $h(D, i, k) = h(D, j, k)$ and False otherwise.

Solution: We build a direct access array mapping each index $i \in \{0, \ldots, n-1\}$ to a frequency table of the letters in hand $h(D, i, k)$, specifically a direct access array

A of length 26 where $A[j]$ corresponds to the number of times the $(j+1)$th letter of the English alphabet occurs in the hand. The frequency table of hand $h(D, 0, k)$ can be computed in $O(k)$ time by simply looping through the cards in the hand and adding them to the frequency table. Then given the frequency table of $h(D, i, k)$, we can compute the frequency table of $h(D, i+1, k)$ in constant time by subtracting one from letter $D[i]$ and adding one to letter $D[i+k]$. Building the above hash table then takes $O(k) + nO(1) = O(n)$ time. To support `same(i, j)`, look up indices i and j in the direct access array in constant time. If the corresponding frequency tables are the same, then the hands must also match. We can check if they match in worst-case constant time since each frequency has constant length (i.e., 26), so this operation takes worst-case $O(1)$ time. Students my use a hash table to achieve expected $O(1)$ time.

Rubric:

- 2 points for description of correct data structure (even if inefficient)
- 1 point for correct running time analysis (even if inefficient)
- 2 points for data structure construction in $O(n)$
- 2 points for description of correct `same(i, j)` (even if inefficient)
- 1 point for correct running time analysis (even if inefficient)
- 2 points for correct `same(i, j)` is $O(1)$

(c) [10 points] Given a deck of n cards, describe an $O(n)$-time algorithm to find the most likely Po-k-er hand, breaking ties lexicographically. State whether your algorithm's running time is worst-case, amortized, and/or expected.

Solution: Build the data structure from part (b) in worst-case $O(n)$ time, specifically a direct access array of hand frequency tables. Now, compute the frequency of each hand directly: loop through the direct access array and add each hand frequency table to a hash table T mapping to value 1; if a hand table h already exists in T, increase $T[h]$ by 1. This procedure performs one hash table operation for each of the n hand tables, so it runs in expected $O(n)$ time. Next, find the largest frequency of any hand directly by looping through all hands in T, keeping track of f the largest frequency seen in worst-case $O(n)$ time. Then, construct a list of hand tables with frequency f directly by looping through all hands in T again, appending to the end of a dynamic array A every hand table that has frequency f, also in worst-case $O(n)$ time. The lexicographically first hand will be the one whose hand frequency table is lexicographically last (e.g., `(1,0,...)>(0,1,...)` but `'a...'<'b...'`), so loop through the hand tables and keep track of the lexicographically last hand table t in worst-case $O(n)$ time. Lastly, convert hand table t back into a hand by concatenating k letters in order based on their frequency in worst-case $O(k)$ time, and then return the hand. Then in total, this procedure runs in expected $O(n)$ time.

Rubric:

- 4 points for description of a correct algorithm (even if inefficient)

- 2 points for correct running time analysis (even if inefficient)
- 4 points for correct algorithm is expected $O(n)$
- Partial credit may be awarded

(d) [25 points] Write a Python function `most_likely_hand(D, k)` that implements your algorithm. Built-in Python functions `chr` and `ord` may be useful. You can download a code template containing some test cases from the website. Submit your code online at `alg.mit.edu`.

Solution:

```
def most_likely_hand(D, k):
    ALPHA = 'abcdefghijklmnopqrstuvwxyz'
    INDEX = {'a':  0, 'b':  1, 'c':  2, 'd':  3, 'e':  4, 'f':  5, 'g': 6,
             'h':  7, 'i':  8, 'j':  9, 'k': 10, 'l': 11, 'm': 12, 'n': 13,
             'o': 14, 'p': 15, 'q': 16, 'r': 17, 's': 18, 't': 19, 'u': 20,
             'v': 21, 'w': 22, 'x': 23, 'y': 24, 'z': 25}
    D2 = D + D[:k - 1]
    hand_char_freq = [0] * 26     # first hand letter frequency table
    for c in D2[:k]:
        hand_char_freq[INDEX[c]] += 1
    hand = tuple(hand_char_freq)
    frequency = {hand: 1}
    most_likely = hand            # letter frequency table of most likely so far
    high_freq = 0
    for i in range(len(D) - 1):   # compute other hand letter frequency tables
        hand_char_freq[INDEX[D2[i]]]     -= 1
        hand_char_freq[INDEX[D2[i + k]]] += 1
        hand = tuple(hand_char_freq)            # new hand
        if hand not in frequency:
            frequency[hand] = 0
        frequency[hand] += 1                    # count new hand
        if frequency[hand] > high_freq:         # higher frequency found
            most_likely = hand
            high_freq = frequency[hand]
        elif frequency[hand] == high_freq:      # same frequency found
            for i in range(len(hand)):
                if hand[i] < most_likely[i]:    # higher lexicographically
                    break
                elif hand[i] > most_likely[i]:  # lower  lexicographically
                    most_likely = hand
    return ''.join([ALPHA[i] * most_likely[i] for i in range(26)])
```