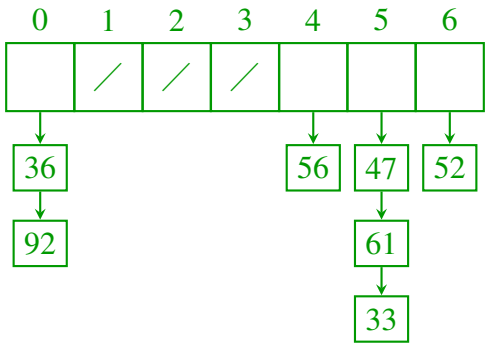
Problem Set 3

All parts are due on February 28, 2020 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 3-1. [5 points] **Hash Practice**

(a) [2 points] Insert integer keys `A = [47, 61, 36, 52, 56, 33, 92]` in order into a hash table of size 7 using the hash function $h(k) = (10k + 4) \bmod 7$. Each slot of the hash table stores a linked list of the keys hashing to that slot, with later insertions being appended to the end of the list. Draw a picture of the hash table after all keys have been inserted.

Solution:



Rubric: $\lfloor n/4 \rfloor$ points for n correct insertions

(b) [3 points] Suppose the hash function were instead $h(k) = ((10k + 4) \bmod c) \bmod 7$ for some positive integer c. Find the smallest value of c such that no collisions occur when inserting the keys from `A`.

Solution:

By the pigeonhole principle, at least one collision occurs for $c < 7$, so check collisions manually for sequentially larger c. If $c = 7$ then 47, 61, and 33 all hash to 5. If $c = 8$ then $36, 52, 56, 92$ all hash to 4. If $c = 9$ then 47 and 56 hash to 6. If $c = 10$ then everything hashes to 4. If $c = 11$ then 47 and 36 hash to 1. If $c = 12$ then 56 and 92 hash to 0. If $c = 13$ then there are no collisions.

k	47	61	36	52	56	33	92
$10k+4$	474	614	364	524	564	334	924
$10k+4 \bmod 7 \bmod 7$	**5**	**5**	0	6	4	**5**	0
$10k+4 \bmod 8 \bmod 7$	2	6	**4**	**4**	**4**	6	**4**
$10k+4 \bmod 9 \bmod 7$	**6**	2	4	2	**6**	1	**6**
$10k+4 \bmod 10 \bmod 7$	**4**	**4**	**4**	**4**	**4**	**4**	**4**
$10k+4 \bmod 11 \bmod 7$	**1**	2	**1**	0	3	4	0
$10k+4 \bmod 12 \bmod 7$	6	2	4	1	**0**	3	**0**
$10k+4 \bmod 13 \bmod 7$	6	3	0	4	5	2	1

The table above was generated using this python code:

```
A = [47, 61, 36, 52, 56, 33, 92]
for c in range(7, 100):
    hashes = [((10 * k + 4) % c) % 7 for k in A]
    print('\t'.join([str(h) for h in hashes]))
    if len(set(hashes)) == 7:
        break
```

Rubric:

- 3 point for $c = 13$
- Partial credit may be awarded if there is work shown of a correct approach that does not yield the correct solution.

Problem 3-2. [15 points] **Dorm Hashing**

MIT wants to assign $2n$ new students to n rooms, numbered 0 to $n-1$, in Pseudorandom Hall. Each MIT student will have an ID: a postive integer less than u, with $u \gg 2n$. No two students can have the same ID, but new students are allowed to choose their own IDs after the start of term.

MIT wants to find students quickly given their IDs, so will assign students to rooms by hashing their IDs to a room number. So as not to appear biased, MIT will publish a family $\mathcal{H}$ of hash functions online before the start of term (before new students choose their IDs), and then after students choose IDs, MIT will choose a rooming hash function uniformly at random from $\mathcal{H}$.

New MIT freshmen Rony Stark and Tiri Williams want to be roommates. For each hash family below, show that either:

- Rony and Tiri can choose IDs k_1 and k_2 so as to guarantee that they'll be roommates, or
- prove that no such choice is possible and compute the highest probability they could possibly achieve of being roommates.

(a) [5 points] $\mathcal{H} = \{h_{ab}(k) = (ak + b) \bmod n \mid a, b \in \{0, \ldots, n-1\} \text{ and } a \neq 0\}$

Solution: Rony and Tiri can choose any two IDs k_1, k_2 such that $k_1 \equiv k_2 \bmod n$, which are not difficult to find. For example, Rony could choose $k_1 = 3$ and Tiri could choose $k_2 = 2n + 3$. Then, $ak_1 + b \equiv ak_2 + b \bmod n$ for all a, b, so Rony and Tiri are guaranteed to hash to the same room.

(b) [5 points] $\mathcal{H} = \left\{h_a(k) = \left(\left\lfloor \frac{kn}{u} \right\rfloor + a\right) \bmod n \mid a \in \{0, \ldots, u-1\}\right\}$

Solution: Since $u \gg n$, the quantity $\lfloor \frac{kn}{u} \rfloor$ will be the same for most adjacent values of k. For example, Rony could choose $k_1 = 1$ and Tiri could choose $k_2 = 2$, and this quantity would be 0 for both of them. Adding the constant a and taking the result $\bmod n$ would not affect whether the values are the same since $\lfloor \frac{kn}{u} \rfloor$ is always an integer between 0 and $n - 1$, so if Rony's and Tiri's IDs have the same hash for any function in this family, they will have the same hash for *all* functions in this family.

(c) [5 points] $\mathcal{H} = \{h_{ab}(k) = ((ak + b) \bmod p) \bmod n \mid a, b \in \{0, \ldots, p-1\} \text{ and } a \neq 0\}$ for fixed prime $p > u$ (this is the universal hash family from Lecture 4)

Solution: For any two keys, the probability that they collide given a random function from a universal hash family is at most $\frac{1}{m}$ where m is the number of possible hash outputs. Hence, in this case $\frac{1}{n}$ is the highest probability that Rony and Tiri can achieve, and cannot guarantee that they will be roommates.

Rubric:

- For (a) and (b)
  - 3 points for a correct choice of keys
  - 2 points showing that chosen keys collide
- For (c)
  - 3 points for citing or deriving bound on probability for universal hash family
  - 2 points for putting bound in terms of n and equating with probability of roommates

Problem 3-3. [20 points] **The Cold is Not Bothersome Anyway**

Ice cores are long cylindrical plugs drilled out of deep glaciers, which are accumulations of snow piled on top of each other and compressed into ice. Scientists can divide an ice core into distinct slices, each representing one year of deposits. For each of the following scenerios, describe an **efficient**[1] algorithm to sort n slices collected from multiple ice cores. Justify your answers.

(a) [5 points] Every ice core is given a unique ***core identifier*** for bookkeeping, which is a string of exactly $16\lceil\log_4(\sqrt{n})\rceil$ ASCII characters.[2] Sort the slices by core identifier.

Solution: Each string is stored in memory as a contiguous sequence of at most $16\lceil\log_4(\sqrt{n})\rceil \times 8 = O(\log n)$ bits. In the word-RAM, these bits can then be interpreted as an integer stored in a constant number of machine words (since $w \geq \lg n$), upper bounded by $2^{16\lceil\log_4(\sqrt{n})\rceil\times 8} < n^{33}$, so we can sort them in $\Theta(n + n\log_n n^{33}) = \Theta(n)$ time via radix sort.

(b) [5 points] The deepest ice cores in the database are up to 800,000 years old. Sort the slices by their ***age***: the integer number of years since the slice was formed.

Solution: The ages form a constant-bounded range $[0, 8\cdot 10^5]$, so we can use counting sort to order them in ascending order in worst-case $\Theta(8\cdot 10^5 + n) = \Theta(n)$ time. (radix sort may also be used)

(c) [5 points] Variation in the amount of snowfall each year will cause a glacier to accumulate at different rates over time. Sort the slices by ***thickness***, a rational number of centimeters of the form m/n^3 between 0 and 4, where m is an integer.

Solution: Multiplying by n^3, these are integers m in a polynomially-bounded range $[0, 4n^3]$, so sort them using Radix Sort in worst-case $\Theta(n + n\log_n n^3) = \Theta(n)$ time.

(d) [5 points] Elna of Northendelle has discovered that water has ***memory***, but is unable to quantify the memory of a given slice. Luckily, given two slices, she can distinguish which has more memory in $O(1)$ time using her "two-finger algorithm" (touching the slices with her two index fingers). Sort the slices by memory.

Solution: The only way to discern order information from the slices is via comparisons, so we choose merge sort which runs in $\Theta(n\log n)$ time, which is optimal in the comparison model.

Rubric:

- 1 points for a correct choice of algorithm
- 1 points if chosen algorithm is efficient
- 3 points for a correct justification
- Partial credit may be awarded

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

[2] You may assume a string of k ASCII characters is a pointer to a contiguous sequence of k bytes in memory, where each byte stores an integer from 0 to 127 inclusive representing an ASCII character.
`https://en.wikipedia.org/wiki/ASCII`

Problem 3-4. [20 points] **Pushing Paper**

Farryl Dilbin is a forklift operator at Munder Difflin paper company's central warehouse. She needs to ship exactly r reams of paper to a customer. In the warehouse are n boxes of paper, each one foot in width, lined up side-by-side covering an n-foot wall. Each box contains a known positive integer number of reams, where **no two boxes contain the same number of reams**. Let $B = (b_0, \ldots b_{n-1})$ be the number of reams per box, where box i located i feet from the left end of the wall contains b_i reams of paper, where $b_i \neq b_j$ for all $i \neq j$. To minimize her effort, Pharryl wants to know whether there is a ***close*** pair (b_i, b_j) of boxes, meaning that $|i - j| < n/10$, that will ***fulfill*** order r, meaning that $b_i + b_j = r$.

(a) [10 points] Given B and r, describe an expected $O(n)$-time algorithm to determine whether B contains a close pair that fulfills order r.

Solution: It suffices to check for each b_i whether $r - b_i = b_j$ for some $b_j \in B$, and then checking whether $|i - j| < n/10$. Since each box has a unique number of reams, if there is a match with b_i it is a unique b_j. Naively, we could perform this check by comparing $r - b_i$ against all $b_j \in B - \{b_i\}$, which would take $O(n)$ time for each b_i, leading to $O(n^2)$ running time. We can speed up this algorithm by first storing the elements of B in a hash table H along with their index, e.g., (b_i, i), so that looking up each $r - b_i$ can be done quickly. For each $b_i \in B$, insert b_i into H mapped to i in expected amortized $O(1)$ time. Now all unique values that occur in B appear in H, so for each b_i, check whether $r - b_i$ appears in H in expected $O(1)$ time. Then, if it does, H will return a j for which we can test for closeness with i in $O(1)$ time. Building the hash table and then checking for matches each take expected $O(n)$ time, so this algorithm runs in expected $O(n)$ time. This brute force algorithm is correct because we check every b_i for its only possible fulfilling partner, and check directly whether it is close.

Rubric:

- 3 points for a description of a correct algorithm
- 2 points for analysis of correctness
- 2 points for analysis of running time
- 3 points if correct algorithm is efficient
- Partial credit may be awarded

(b) [10 points] Now suppose that $r < n^2$. Describe a worst-case $O(n)$-time algorithm to determine whether B contains a close pair that fulfills order r.

Solution: Replace each b_i in B with the tuple (b_i, i), to keep track of the index of the box in the original order. We do not know whether every b_i is polynomially bounded in n; but we do know that r is. If some $b_i \geq r$, it can certainly not be part of a pair from B that fulfills order r. So perform a linear scan of B and remove all (b_i, i) for which $b_i \geq r$, to construct set B'. Now the ream count integers b_i in B' are each upper bounded by $O(n^2)$, so we can sort the tuples in B' by

their ream counts b_i in worst-case $O(n + n \log_n n^2) = O(n)$ time using radix-sort, and store the output in an array A.

Now we can sweep the sorted list using a two-finger algorithm similar to the merge step in merge sort to find a pair that sums to r, if such a pair exists. Specifically, initialize indices $i = 0$ and $j = |A| - 1$, and repeat the following procedure until $i = j$. If $A[i] = (b_k, k)$, let $A[i].b = b_k$ and $A[i].x = k$. There are three cases:

- $A[i].b + A[j].b = r$: a pair that fulfills the order has been found.
  Check whether $|A[i].x - A[j].x| < n/10$ and return True if so; or
- $A[i].b + A[j].b < r$: $A[i].b$ cannot be part of a pair that fulfills the order with any $A[k].b$ for $k \in \{i + 1, \ldots, j\}$, so increase i; or
- $A[i].b + A[j].b > r$: $A[j].b$ cannot be part of a pair that fulfills the order with any $A[k]$ for $k \in \{i, \ldots, j - 1\}$, so decrease j.

This loop maintains the invariant that at the start of each loop, we have confirmed that no pair $(A[k].b, A[\ell].b)$ is close and fulfills the order, for all $k \leq i \leq j \leq \ell$, so if we reach the end without returning a valid pair, the algorithm will correctly conclude that there is none. Since each iteration of the loop takes $O(1)$ time and decreases $j - i$ decrease by one, and $j - i = |B'| - 1$ starts positive and ends when $j - i < 0$, this procedure takes at most $O(n)$ time in the worst case.

Rubric:

- 3 points for a description of a correct algorithm
- 2 points for analysis of correctness
- 2 points for analysis of running time
- 3 points if correct algorithm is efficient
- Partial credit may be awarded

Problem 3-5. [40 points] **Anagram Archaeology**

String A is an ***anagram*** of another string B if A is a permutation of the letters in B; for example, `(indicatory, dictionary)` and `(brush, shrub)` are pairs of words that are anagrams of each other. In this problem, all strings will be ASCII strings containing only the lowercase English letters `a` to `z`.

Given two strings A and B, the ***anagram substring count*** of B in A is the number of contiguous substrings of A that are anagrams of B. For example, if $A =$ `'esleastealaslatet'` and $B =$ `'tesla'`, then, of the 13 contiguous substrings in A of length $|B| = 5$, exactly 3 of them are anagrams of B, namely `('least', 'steal', 'slate')`, so the anagram substring count of B in A is 3.

(a) [12 points] Given string A and a positive integer k, describe a data structure that can be built in $O(|A|)$ time, which will then support a single operation: given a different string B with $|B| = k$, return the anagram substring count of B in A in $O(k)$ time.

Solution: For this data structure, we need a way to find how many substrings of A are anagrams of an input B in a running time that does not depend on $|A|$. The idea will be to construct and store a constant-sized cannonicalization of each string in a hash table, where anagrams of the string will have the same cannonicalization; in particular, we will construct a **frequency table** with 26 entries, one for each lowercase English letter, where each entry stores the number of occurances of that letter in the string. Two strings will have the same frequency table if and only if they are anagrams of each other.

Let $S = (S_0, \ldots, S_{|A|-k})$ be the $|A|-k+1$ contiguous length-k substrings of A, where substring S_i starts at character $A[i]$. Constructing a frequency table naïvely for each $S_i \in S$ would take $O(|A|k)$ time. However, after computing the frequency table of S_0 naïvely in $O(k)$ time, we can construct the frequncy table forS_{i+1} from the frequency table for S_i in $O(1)$ time by subtracting the letter at $A[i]$ from the frequency table of S_i, and adding in the letter at $A[i+k]$. In this way, we can compute the constant-sized frequency table for each $S_i \in S$ in $O(k) + (|A| - k)O(1) = O(|A|)$ time. Then insert each of these frequency tables into a hash table H, mapped to the number of $S_i \in S$ having that frequency table in expected $O(|A|)$ time (each frequency is at most n, so each frequency table can be thought of as a $26\lceil \lg n \rceil$-sized integer, which fits within a constant number of machine words, that can be used as a hash key).

Then given our data structure H, we can support the requested operation by first computing the frequency table f of the input string B naïvely in $O(k)$ time, and then looking it up in H in $O(1)$ expected time, for a total of expected $O(k)$ time. Since $H(f)$ stores the number of $S_i \in S$ with frequency table f, if f is in H, the stored value will be the anagram substring count of B in A. Otherwise, if f is not in H, f is not an anagram of any substring of A, so return zero.

Rubric:

- 4 points for a description of a correct data structure
- 2 points for analysis of correctness
- 2 points for analysis of running time
- 4 points if correct algorithm is efficient
- Partial credit may be awarded

(b) [3 points] Given string T and an array of n length-k strings $\mathcal{S} = (S_0, \ldots, S_{n-1})$ satisfying $0 < k < |T|$, describe an $O(|T| + nk)$-time algorithm to return an array $A = (a_0, \ldots, a_{n-1})$ for which a_i is the anagram substring count of S_i in T for all $i \in \{0, \ldots, n-1\}$.

Solution: Construct the data structure in part (a) substituting string T for A, and using k in $O(|T|)$ expected time. Then for each $S_i \in S$, we can perform the operation supported by the data structure in expected $O(k)$ time, storing their outputs in an array A, for a total of expected $O(|T| + nk)$ time. This algorithm is correct based on the corrctness of the data structure from (a).

Rubric:

- 2 points for a description of a correct algorithm
- 1 point for analysis of running time and correctness
- Partial credit may be awarded

(c) [25 points] Write a Python function `count_anagram_substrings(T, S)` that implements your algorithm from part (b). Note the built-in Python function `ord(c)` returns the ASCII integer corresponding to ASCII character `c` in $O(1)$ time. You can download a code template containing some test cases from the website. Submit your code online at `alg.mit.edu`.

Solution:

```
ORD_A = ord('a')
def lower_ord(c):           # map a lowercase letter to range(26)
    return ord(c) - ORD_A

def count_anagram_substrings(T, S):
    m, n, k = len(T), len(S), len(S[0])
    D = {}                  # map from freq tables to occurances
    F = [0] * 26            # initial freq table
    for i in range(m):      # compute T freq tables
        F[lower_ord(T[i])] += 1          # add character
        if i > k - 1:
            F[lower_ord(T[i - k])] -= 1  # remove character
        if i >= k - 1:
            key = tuple(F)
            if key in D:                 # increment occurance
                D[key] += 1
            else:                        # add freq table to map
                D[key] = 1
    A = [0] * n             # compute anagram substring counts
    for i in range(n):
        F = [0] * 26
        for c in S[i]:      # compute S_i freq table
            F[lower_ord(c)] += 1
        key = tuple(F)
        if key in D:        # check in dictionary
            A[i] = D[key]
    return tuple(A)
```