Problem Set 3

All parts are due on September 27, 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 3-1. [5 points] **Hash It Out**

(a) [2 points] Insert integer keys `A = [67, 13, 49, 24, 40, 33, 58]` in order into a hash table of size 9 using the hash function $h(k) = (11k + 4) \bmod 9$. 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    7    8
+----+----+----+----+----+----+----+----+----+
|    |    |    | 67 |    |    |    | 24 |    |
+----+----+----+----+----+----+----+----+----+
                 \/                  \/
               +----+              +----+
               | 13 |              | 33 |
               +----+              +----+
                 \/
               +----+
               | 49 |
               +----+
                 \/
               +----+
               | 40 |
               +----+
                 \/
               +----+
               | 58 |
               +----+
```

Rubric: $\lfloor n/3 \rfloor$ points for n correct insertions

(b) [1 points] What is the maximum chain length of this hash table?

Solution: 5

Rubric: 1 point for a correct chain length

(c) [2 points] Suppose the hash function were instead $h(k) = ((11k + 4) \bmod c) \bmod 9$ 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 $c \leq 7$, so check collisions manually for sequentially larger c. For $c = 8$, 24 and 40 collide. $c = 9$ is the same as part (a). For $c = 10$, 13 and 33 collide. For $c = 11$, all keys collide. For $c = 12$, 67 and 40 collide. For $c = 13$, no collision occurs as desired, hashing the keys to the table `[67, 49, 40, 33, 13, 58,--,--,24]`. Alternatively, you could write a simple program to do the search for you:

```python
A = [67, 13, 49, 24, 40, 33, 58]
for c in range(1, 100):                       # check first 100
    H = [[] for _ in range(9)]                # init hash table
    for k in A:
        h = ((11*k + 4) % c) % 9              # compute hash
        H[h].append(k)                        # increment chain counter
    max_chain = max([len(L) for L in H])      # compute max chain length
    print("%2d: %s  %s" % (c, max_chain, H))
    if max_chain is 1:  break                 # stop if no collisions
```

Rubric:

- 1 point for $c = 13$
- 1 point for a correct case analysis (manual or by program)

Problem 3-2. [10 points] **Hash Sequence**

Hash tables are not only useful for implementing Set operations; they can be used to implement Sequences as well! (Recall the Set and Sequence interfaces were defined in Lecture and Recitation 02.) Given a hash table, describe how to use it as a black-box[1] (using only its Set interface operations) to implement the Sequence interface such that:

- `build(A)` runs in expected $O(n)$ time,
- `get_at` and `set_at` each run in expected $O(1)$ time,
- `insert_at` and `delete_at` each run in expected $O(n)$ time, and
- the four dynamic first/last operations each run in amortized expected $O(1)$ time.

Solution: To use a hash table H to implement the Sequence operations, store each Sequence item x in an object b with key $b.key$ and value $b.val = x$, and we will store these keyed objects in the hash table. We also maintain the lowest key s stored in the hash table, to maintain invariant that the n stored objects have keys $(s, \ldots, s + n - 1)$, where the i^{th} item in the Sequence is stored in the object with key $s + i$.

To implement `build(A)`, for each item x_i in $A = (x_0, \ldots, x_{n-1})$ construct its keyed object b, initially with key $b.key = i$, in worst-case $O(1)$ time; then insert it into the hash table using Set `insert(b)` in expected $O(1)$ time, for an expected total of $O(n)$ time. Initializing $s = 0$ ensures the invariant is satisfied.

[1]By black-box, we mean you should not modify the inner workings of the data structure or algorithm.

To implement `get_at(i)`, return the value of the stored object with key $s+i$ using Set `find(s + i)` in expected $O(1)$ time, which is correct by the invariant. Similarly, to implement `set_at(i, x)`, find the object with key $s+i$ using `find(s + i)` and change its value to x, also in expected $O(1)$ time.

To implement `insert_at(i, x)`, for each j from $s+n-1$ down to $s+i$, remove the object b with key j using `delete(j)` in expected $O(1)$ time, change its key to $j+1$ in worst-case $O(1)$ time, and then insert the object with `insert(b)` in expected $O(1)$ time. Then, construct a keyed object b' with value x and key $s+i$, and insert with `insert(b')` in expected $O(1)$ time, for an expected total of $O(n)$ time. This operation restores the invariant for each affected item.

Similarly, to implement `delete_at(i)`, remove the object b' stored at $s+i$ with `delete(s + i)` in expected $O(1)$ time; then for each j from $s+i+1$ to $s+n-1$, remove the object b with key j using `delete(j)` in expected $O(1)$ time, change its key to $j-1$ in worst-case $O(1)$ time, and then insert the object with `insert(b)` in expected $O(1)$ time. Then return the value of object b', for an expected total of $O(n)$ time. This operation returns the correct value by the invariant, and restores the invariant for each affected item.

To implement `insert_last(x)` or `delete_last()`, simply reduce to `insert_at(s + n)` or `delete_at(s + n - 1)` in expected $O(1)$ time since no objects need to be shifted.

To implement `insert_first(x)`, construct a keyed object b with value x and key $s-1$ and insert it with `insert(b)` in expected $O(1)$ time. Then setting the stored value of s to $s-1$ restores the invariant. Similarly for `delete_first()`, remove the object with key s using `delete(s)` in expected $O(1)$ time, and return the value of the object. Then setting the stored value of s to $s+1$ restores the invariant.

Rubric:

- 1 point for a correct implementation of each operation (9 total)
- 1 point for augmentation of information from which you can compute a correct offset

Problem 3-3. [15 points] **Critter sort**

Ashley Getem collects and trains Pocket Critters to fight other Pocket Critters in battle. She has collected n Critters in total, and she keeps track of a variety of statistics for each Critter C_i. Describe **efficient**[2] algorithms to sort Ashley's Critters based on each of the following keys:

(a) [3 points] Species ID: an integer x_i between $-n$ and n (negative IDs are grumpy)

Solution: These integers are in a linearly bounded range, but are not positive. So take worst-case $O(n)$ time to add n to each critter's ID so that $x_i \leq 2n = u$ for all i, sort them using counting sort in worst-case $O(n+2n) = O(n)$ time, and then subtract n from each ID, again in worst-case $O(n)$ time.

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

(b) [4 points] Name: a unique string s_i containing at most $10\lceil \lg n \rceil$ English letters

Solution: Let's assume that each string is stored sequentially in a contiguous chunk of memory, in an encoding such that the numerical representation of each character is bounded above by some constant number k, and the numerical representation of one character c_i is smaller than that of another character c_j if c_i comes before c_j in the English alphabet. Then each string can be thought of as an integer between 0 and $u = k^{10\lceil \lg n \rceil} = O(n^{10})$, stored in a constant number of machine words, so can be sorted using radix sort in worst-case $O(n + n \log_n n^{10}) = O(n)$.

Alternatively, if each character in the each string s_i is stored in its own machine word, then the input is has size $O(n \log n)$. For each string, compute its $O(n^{10})$ numerical representation by direct computation in $O(\log n)$ arithmetic computations (which can each be performed in $O(1)$ time, since each intermediate representation fits into at most 10 machine words). Then sort using radix-sort as above. Computing the numerical representations of the strings takes $O(n \log n)$ time, which is linear in the size of the input.

(c) [3 points] Number of fights fought: a positive integer f_i under n^2

Solution: These integers are in a polynomially bounded range $u = n^2$, so sort them using radix-sort in worst-case $O(n + n \log_n n^2) = O(n)$ time.

(d) [5 points] Win fraction: ratio w_i/f_i where $w_i \leq f_i$ is the number of fights won

Solution: Non-integer division may yield a number that requires an infinite number of digits to represent, so we cannot compute these numbers directly. Solutions attempting to compute and compare such numbers without accounting for precision should not be awarded any points. We present two solutions here.

The first solution uses an optimal comparison sorting algorithm like merge sort to sort the win fractions in worst-case $O(n \log n)$ time. Two win ratios w_1/f_1 and w_2/f_2 can be compared via cross multiplication, since $w_1/f_1 < w_2/f_2$ if and only if $w_1 f_2 < w_2 f_1$. This solution done correctly is worth 4/5 points.

The second solution is more tricky. The idea will be to scale the ratios sufficiently such that when they are not equal, their integer parts also not equal. First, for each win number, compute $w'_i = w_i \cdot n^4$ in $O(1)$ time. Then compute $p_i = \lfloor w'_i/f_i \rfloor$ in $O(1)$ time via integer division, where $w'_i = p_i \cdot f_i + q_i$ for $q_i = w'_i \bmod f_i$. Then, since each p_i is a positive integer bounded by $O(n^6)$, we can sort by p_i in worst-case $O(n + n \log_n n^6) = O(n)$ time via radix sort.

Now we must prove that sorting by p_i is equivalent to sorting by w_i/f_i. It suffices to prove that $w_i/f_i - w_j/f_j > 0$ is true if and only if $p_i - p_j > 0$ is true. Without loss of generality, assume that $d_w = w_i/f_i - w_j/f_j > 0$. Since

$$d_w n^4 = w'_i/f_i - w'_j/f_j = (p_i + q_i/f_i) - (p_j + q_j/f_j) = (p_i - p_j) + (q_i/f_i - q_j/f_j),$$

it suffice to show that $p_i - p_j = d_w n^4 - q_w > 0$ where $q_w = q_i/f_i - q_j/f_j$. First, q_w is maximized when:

$$q_w = \frac{n^2 - 2}{n^2 - 1} - \frac{0}{1} < 1,$$

while $d_w n^4$ is minimized when:

$$d_w n^4 = \left(\frac{1}{n^2-2} - \frac{1}{n^2-1}\right) n^4 = \frac{n^4}{n^4 - 3n^2 + 2} > 1,$$

so $d_w n^4 - q_w > 0$ as desired.

Rubric:

- Full points for justification of a correct and efficient sorting algorithm
- Partial credit may be awarded

Problem 3-4. [20 points] **College Construction**

MIT has employed Gank Frehry to build a new wing of the Stata Center to house the new College of Computing. MIT wants the new building be as tall as possible, but Cambridge zoning laws limit all new buildings to be no higher than positive integer height h. As an innovative architect, Frehry has decided to build the new wing by stacking two giant aluminum cubes on top of each other, into which rooms will be carved. However, Frehry's supplier of aluminum cubes can only make cubes with a limited set of positive integer side lengths $S = \{s_0, \ldots, s_{n-1}\}$. Help Frehry purchase cubes for the new building.

(a) [10 points] Assuming the input S fits within $\Theta(n)$ machine words, describe an **expected** $O(n)$-time algorithm to determine whether there exist a pair of side lengths in S that exactly sum to h.

Solution: It suffices to check for each s_i whether $(h - s_i) \in S$. Naively, we could perform this check by comparing $h - s_i$ against all $s_j \in S - \{s_i\}$, which would take $O(n)$ time for each s_i, leading to $O(n^2)$ running time. We can speed up this algorithm by first storing the elements of S in a hash table H so that looking up each $h - s_i$ can be done quickly. For each $s_i \in S$, insert s_i into H in expected $O(1)$ time. Now all unique values that occur in S appear in H, so for each s_i, check whether $h - s_i \neq s_i$ appears in H in expected $O(1)$ time. Building the hash table and then checking for matches each take expected $O(n)$ time, so this algorithm runs in $O(n)$ time. This brute force algorithm is correct because each s_i satisfies $s_i + k_i = h$ for exactly one integer k_i, and we check all possible (s_i, k_i).

Rubric:

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

(b) [10 points] Unfortunately for Frehry, there is no pair of side lengths in S that sum exactly to h. Assuming that $h = 600n^6$, describe a **worst-case** $O(n)$-time algorithm to return a pair of side lengths in S whose sum is closest to h without going over.

Solution: We do not know whether all $s_i \in S$ are polynomially bounded in h; but we do know that h is. If some $s_i \geq h$, it can certainly not be part of a pair of positive side lengths from S that sum to under h. So first perform a linear scan of S and remove all $s_i \geq h$ to construct set S'. Now the integers in S' are each upper bounded by $O(n^6)$, so we can sort them in worst-case $O(n + n \log_n n^6)$ 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 with the largest sum at most h, if such a pair exists. Specifically, initialize indices $i = 0$ and $j = |S'| - 1$, and repeat the following procedure, keeping track of the largest sum t found so far initialized to zero. If $A[i] + A[j] \leq h$, then if $t < A[i] + A[j]$, you have found a better pair, so set $t = A[i] + A[j]$; regardless $A[k] + A[j] < t$ for all $k \leq i$, so increase i by one. Otherwise if $A[i] + A[j] > h$, then $A[i] + A[\ell] > h$ for all $\ell \geq j$, so decrease j by one. If $j < i$, then return False. This loop maintains the invariant that at the start of each loop, we have confirmed that $A[k] + A[\ell] \leq t$ holds true for all $k \leq i \leq j \leq \ell$ for which $A[k] + A[\ell] \leq h$, so the algorithm is correct. Lastly, we can check the special case that A includes the value $h/2$ in $O(\log n)$ time via binary search. Since each iteration of the loop takes $O(1)$ time and decreases $j - i$ decrease by one, and $j - i = |S'| - 1$ starts positive and ends when $j - i < 0$, this procedure takes at most $O(n)$ time in the worst case.

Rubric:

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

Problem 3-5. [50 points] **Copy Detection**

Plagiarism can be difficult to detect, especially when a copied text has been modified from its original form. It can be easier to detect plagiarism when a copied text appears without changes inside another text. Given a document string D and a query string Q, the **copy detection problem** asks whether Q appears as a contiguous substring of D. (For this problem, we assume that a string D is a sequence $(d_0, \dots, d_{|D|-1})$ of $|D|$ **ASCII characters**[3].) As an example, if $D =$ `'algorithm'`, the copy detection problem would answer True if $Q =$ `'rith'`, but answer False if $Q =$ `'log'`. A simple brute force algorithm solves the copy detection problem in $O(|D||Q|)$ time: there are

[3]An ASCII character is represented by an integer between 0 and 127 inclusive. The built-in Python function `ord(c)` returns the ASCII integer corresponding to ASCII character `c`. `https://en.wikipedia.org/wiki/ASCII`

$|D| - |Q| + 1$ contiguous substrings of D that have length $|Q|$:

$$\mathcal{D} = \{D_i = (d_i, \dots, d_{|Q|+i}) \mid i \in \{0, \dots, |D| - |Q|\}\},$$

so for each substring $D_i \in \mathcal{D}$, check whether $D_i = Q$, character-by-character in $O(|Q|)$ time. If we could determine whether $D_i = Q$ in $O(1)$ time instead of $O(|Q|)$ time, then the above algorithm would run much faster, in $O(|D|)$ time. Let's try to accomplish this goal using hashing!

(a) [3 points] Given a string $S = (s_0, \dots, s_{|S|-1})$, let's think of it as the $|S|$-digit base-128 number $R(S) = \sum_{i=0}^{|S|-1} s_i \cdot 128^{(|S|-1-i)}$, where s_0 is the most significant digit. Note that $R(S_1) = R(S_2)$ if and only if $S_1 = S_2$. Describe an algorithm to compute $R(S)$ using $O(|S|)$ arithmetic operations, i.e., `(-, +, *, //, %)`. For parts (a), (b), and (c), assume that one arithmetic operation can operate on integers of arbitrary size.

Solution: Let's generalize $R(S)$ to $T(S, k) = \sum_{i=0}^{k} s_i \cdot 128^{(k-i)}$. Then $T(S, 0) = s_0$, $T(S, |S| - 1) = R(S)$, and $T(S, k) = T(S, k - 1) \cdot 128 + s_k$. So to compute $R(S)$, compute each $T(S, k)$ iteratively from $T(S, k - 1)$ for k from 1 to $|S| - 1$ using 2 arithmetic operations each, for a total of $(|S| - 1) \cdot 2 = O(|S|)$ operations. (An alternative correct solution precomputes the powers 128^i from 0 to $|S| - 1$ and then evaluates the sum directly.)

Rubric:

- 2 points for a description of a correct algorithm
- 1 points for analysis of operation count
- Partial credit may be awarded

(b) [3 points] Given $R(D_i)$, i.e., the numerical representation of the i^{th} contiguous length-$|Q|$ substring of D for $i < |D| - |Q|$, and the value $f = 128^{|Q|}$, describe an algorithm to compute $R(D_{i+1})$ using $O(1)$ arithmetic operations.

Solution: Observe that $R(D_i) = d_i \cdot 128^{|Q|-1} + \sum_{j=1}^{|Q|-1} d_{i+j} \cdot 128^{(|Q|-1-j)}$ and $R(D_{i+1}) = d_{i+|Q|} + \sum_{j=0}^{|Q|-2} d_{i+1+j} \cdot 128^{(|Q|-1-j)} = d_{i+|Q|} + 128 \sum_{j=1}^{|Q|-1} d_{i+j} \cdot 128^{(|Q|-1-j)}$. So:

$$R(D_{i+1}) = (128 \cdot R(D_i) - f \cdot d_i) + d_{i+|Q|},$$

So we can compute $R(D_{i+1})$ from $R(D_i)$ and f using just $4 = O(1)$ arithmetic operations, as desired.

Rubric:

- 2 points for a description of a correct algorithm
- 1 points for analysis of operation count
- Partial credit may be awarded

(c) [8 points] Describe an algorithm to solve the copy detection problem that uses at most $O(|D|)$ arithmetic operations.

Solution: First compute f naively using $O(|Q|)$ multiplications. Compute $R(Q)$ and $R(D_0)$, each using $O(|Q|)$ arithmetic operations via part (a). Then repeat the following algorithm for i starting at 0, terminating when i is $|D| - |D|$: check whether $R(Q)$ equals $R(D_i)$ by evaluating whether $R(Q) - R(D_i) = 0$ using a single arithmetic operation. If they are equal, then return True, as $R(Q) = R(D_i)$ if and only if $Q = D_i$. Otherwise, compute $R(D_{i+1})$ from $R(D_i)$ and f using $O(1)$ arithmetic operations via part (b) and increase i by one. This brute force algorithm uses $O(|Q|) + (|D| - |Q|) \cdot O(1) = O(|D|)$ arithmetic operations and is correct because it checks whether $R(Q) = R(D_i)$ for every $D_i \in \mathcal{D}$.

Rubric:

- 4 points for a description of a correct algorithm
- 2 points for analysis of correctness
- 2 points for analysis of operation count
- Partial credit may be awarded

The above algorithms pose a problem: if string $|S|$ is large, then $R(|S|)$ is roughly $\Omega(128^{|S|})$, likely much too large to fit in a single register on your CPU, so we won't be able to perform arithmetic operations on such numbers in $O(1)$ time. Instead of computing $R(|S|)$ directly, let's instead hash it down to a smaller range, specifically $R'(S) = R(S) \bmod p$, where p is a randomly chosen large prime number that fits into a machine word so it can be operated on in a CPU register in $O(1)$ time. If $Q = D_i$, then certainly $R'(Q) = R'(D_i)$, and we will be able to identify the match quickly! However, sometimes $R'(Q) = R'(D_i)$ when $Q \neq D_i$, and we will get a false match. Assume that p can be chosen sufficiently randomly such that false matches are unlikely to occur, i.e., the probability that $R'(Q) = R'(D_i)$ when $Q \neq D_i$ is $\leq \frac{1}{|Q|}$, for any $D_i \in \mathcal{D}$.

(d) [3 points] Given $R'(D_i)$ and the value $f' = (128^{|Q|}) \bmod p$, describe an $O(1)$-time algorithm to compute $R'(D_{i+1})$.

Solution: This part is the same as (b) except that the computation should be taken modulo p:

$$R'(D_{i+1}) = R(D_{i+1}) \bmod p = (128 \cdot R'(D_i) - f' \cdot d_i) + d_{i+|Q|} \bmod p.$$

Since the numbers in this computation always fit within a constant number of machine words, so each of the constant number of arithmetic operations can be done in worst-case $O(1)$ time.

Rubric:

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

(e) [8 points] Describe an expected $O(|D|)$-time algorithm to solve the copy detection problem.

Solution: This part is the same as part (c) except that we compute and compare $R'(Q)$ with $R'(D_i)$ values instead of $R(Q)$ and $R(D_i)$. We can compute f' in $O(|Q|)$ time by performing multiplications modulo p, and we can similarly compute $R(D_0)$ via the computations in (a) modulo p in $O(|Q|)$ time. When comparing $R'(Q)$ to $R'(D_i)$, if $R'(Q) \neq R'(D_i)$, we know that $Q \neq D_i$ and we can continue. However, when $R'(Q) = R'(D_i)$, it is possible that $Q \neq D_i$, so and we spend $O(|Q|)$ time to compare each character of Q against each character of D_i. If the two match, then return True, otherwise, continue checking D_{i+1}. The time spent checking each D_i is $O(1)$ when $R'(Q) \neq R'(D_i)$, and $O(|Q|)$ when $R'(Q) = R'(D_i)$. However, by the assumption provided, this only occurs with probability $1/|Q|$ when $Q \neq D_i$, so the expected time to check any particular D_i is $O(1)$. Thus this algorithm runs in expected $O(|D|)$ time.

Rubric:

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

(f) [25 points] Write a Python function `detect_copy` that implements your algorithm. Instead of choosing p randomly, please use the Mersenne prime $p = 2^{31} - 1$. You can download a code template containing some test cases from the website. Submit your code online at `alg.mit.edu`.

Solution:

```python
def detect_copy(D, Q):
    p = 2**31 - 1
    d, q = len(D), len(Q)
    Rd, Rq = 0, 0
    f = 1
    for i in range(q):
        Rd = (Rd * 128 + ord(D[i])) % p
        Rq = (Rq * 128 + ord(Q[i])) % p
        f = (f * 128) % p
    for i in range(d - q - 1):
        if Rd == Rq:
            j = 0
            while D[i + j] == Q[j]:
                j += 1
                if j == q:
                    return True
        Rd = (128*Rd - f*ord(D[i]) + ord(D[i + q])) % p
    return False
```

Rubric:

- This part is automatically graded at `alg.mit.edu`.