Problem Session 3

Problem 3-1. Hash It Out

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 |
               +----+
```

Problem 3-2. 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,

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

- `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.

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.

Problem 3-3. 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:

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

(a) 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 $x_i \le 2n = u$ 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.

(b) 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 (e.g., 26 for efficient English letter encoding, or 256 for byte encoding), where 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 $u = k^{10\lceil \lg n \rceil} = O(n^{10 \lg k}) = n^{O(1)}$, stored in a constant number of machine words, so can be sorted using radix sort in worst-case $O(n + n \log_n n^{O(1)}) = O(n)$ time.

Alternatively, if each character in the each string s_i is stored in its own machine word, then the input has size $\Theta(n \log n)$. For each string, compute its $n^{O(1)})$ 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) 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) Win fraction: ratio w_i/f_i where $w_i \le f_i$ 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 suffices 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.

Problem 3-4. 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 $S = \{s_0, \dots, s_{n-1}\}$. Help Frehry purchase cubes for the new building.

(a) 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 $h - s_i$ appears in H in expected $O(1)$ time. (If the supplier can build only one of each block size, we can also check that $h - s_i \neq s_i$.) 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).

(b) 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 n; 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 $j < i$ (or $j = i$ and we want distinct s_i, s_j), then return False. This loop maintains the invariant that at the start of each loop, we have confirmed that $A[k] + A[\ell] \geq t$ for all $k \leq i \leq j \leq \ell$ for which $A[k] + A[\ell] \leq h$, so the algorithm is correct. 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.

Problem 3-5. 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 $P(D, i, k)$ 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 $P(D, 2, 4) =$ `'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) 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 $P(D, i, k) = P(D, j, k)$ and False otherwise.

Solution: We build a direct access array mapping each index $i \in \{0, \dots, n-1\}$ to a frequency table of the letters in hand $P(D, i, k)$, 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 $P(D, 0, k)$ 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 $P(D, i, k)$, we can compute the frequency table of $P(D, i+1, k)$ 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.

(b) 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 (a) 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.

We can reduce to **worst-case** $O(n)$ time using radix sort instead of a hash table to count the frequencies of hand tables. Namely, we apply tuple/radix sort to the data structure from part (a). Each hand frequency table consists of 26 numbers between 0 and n, so we can treat them as a base-$(n+1)$ integer of 26 digits. Sorting by each digit from least to most significant, we put the hand frequency tables into lexically increasing order. Now a single scan through the array, at each step checking whether the hand frequency table matches the previous, lets us compute the frequency of each table. A scan of these occurrence frequencies lets us find the maximum frequency f, and another scan of the array lets us find the lexicographically last hand with frequency f.