Problem Set 3

Name: Gabriel Chiong

Collaborators: None

Problem 3-1.

(a) The slots of the hash table are labelled from $0, \ldots, 1$ vertically on the left-most side, with chained elements linked by an arrow.

$[0] \longrightarrow 36 \longrightarrow 92$
$[1]$
$[2]$
$[3]$
$[4] \longrightarrow 56$
$[5] \longrightarrow 47 \longrightarrow 61 \longrightarrow 33$
$[6] \longrightarrow 52$

(b) We know the c must be at least 7, otherwise there will be at least one collision by the Pigeonhole Principle (since there are 7 elements in total). Therefore, trying integers one-by-one from $c \geq 7$ eventually yields a solution of $c = 13$.

Problem 3-2.

(a) To hash into the same room, we choose k_1, k_2 such that $k_1 \equiv k_2 \pmod{n}$. For example, if we choose $k_1 = 3$ and $k_2 = 2n + 3$, we obtain $h(k_1) = (3a + b) \pmod{n} = h(k_2)$.

(b) Since $u \gg n$, most adjacent k's will cause their respective $h(k)$'s to be rounded down to the same integer. For example, for $k_1 = 1, k_2 = 2$, we have $h(k_1) = a = h(k_2)$. The $+a$ and $\mod n$ would not have a significant impact on the outcome of the hash function since $\lfloor \frac{kn}{u} \rfloor$ will always be an integer from $0, \ldots, n-1$. In fact, they will have the same value regardless of the hash function chosen in $\mathcal{H}$.

(c) From the lecture on hashing, the probability that two key hashes collide given a uniformly random selection of $h \in \mathcal{H}$ from a universal hash family is at most $\frac{1}{n}$, with n being the number of slots that can be hashed to. In this case, it cannot be guaranteed that they will hash to the same room.

Problem 3-3.

(a) Each identifier can be stored in memory as a contiguous sequence of $16 \lceil \log_4(\sqrt{n} \rceil \times 8$ bits. Therefore, we can interpret these strings as a number between $0, \ldots, 2^{16\lceil \log_4(\sqrt{n}) \rceil \times 8}$, where $2^{16\lceil \log_4(\sqrt{n}) \rceil \times 8} = O(\sqrt{n}^{16(\log_4 2) \times 8}) = O(n^{\frac{1}{2} \cdot 16 \frac{1}{2} \cdot 8}) = O(n^{32}) = O(n^{O(1)})$. Therefore, using radix sort would allow the sorting to be done in worst-case $O(n + n \log_n n^{O(1)}) = O(n)$ time.

(b) Effectively, each age is an integer, so we can use counting sort with a worst-case time complexity of $O(n + u) = O(n + 800,000) = O(n)$.

(c) Multiply each integer by n^3 to obtain integers in the range of $[0, 4n^3]$. Then apply radix sort in worst-case $O(n + n \log_n n^3) = O(n)$ time.

(d) Since this requires comparing two slices, any comparison sort is lower bounded by $\Omega(n \log n)$. Therefore, we can use merge sort for a worst-case time complexity of $\Theta(n \log n)$.

Problem 3-4.

(a) To determine a close pair that fulfils r in expected $O(n)$ time, we first iterate through the boxes B and hash each box, b_i as the key, and its index, i as the corresponding value. Let us call the resulting hash table H. We then iterate over B a second time, each time checking H if the value $r - b_i$ exists in H. If it does exist (let us call this point b_j), check if if the index value, j stored by the key $r - b_i$ in H is less than $n/10$ from i. If this holds, return our close pair. Otherwise, continue with the loop.

This algorithm makes two passes over B, each time doing expected constant work (arithmetic and hash table operations), therefore the overall worst-case time complexity is expected $O(n)$ time.

(b) Make a first pass over all boxes and cast out boxes which have a value greater or equal to n^2. This can be done in $O(n)$ time. Next, initialize an empty array A, and make a second pass over the new B array. At each loop iteration, append the tuple (b_i, i) to A. This can be also done in $O(n)$ time ($O(1)$ to initialize an empty array and $O(n)$ to make the second iteration).

Since each b_i is a positive integer, we can apply radix sort to array A, using the first element of the tuple, b_i as the key. We can then sort A by the number of reams in each box in $O(n + n \log_n n^2) = O(n)$ worst-case time.

Now use the "two-finger" algorithm with $i = 0$ and $j = |A| - 1$ on A. At each iteration, check the following cases:

- If $b_i + b_j = r$, check if $|i - j| < n/10$. If this holds, return true. Otherwise, continue with the algorithm by decrementing j (arbitrarily, it could have been increasing i too).
- If $b_i + b_j < r$, increase i since the boxes are ordered by their number of reams in increasing order.
- If $b_i + b_j > r$, decrease j since the boxes are ordered by their number of reams in decreasing order.

The "two-finger" algorithm touches each element at most once, therefore it runs in worst-case $O(n)$ time. If we terminate the loop with $i >= j$, return false - we did not find a close pair that fulfilled order r. Since the overall algorithm only makes a constant number of iterations over arrays A and B (both with length at most n), the whole algorithm runs in worst-case $O(n)$ time.

Problem 3-5.

(a) Initialize a hash table H which will map a frequency table for a particular substring in A to a count, the number of anagrams for a particular sequence of characters in A (they will have the same frequency table since they have the exact same number and type of characters). The frequency table can be implemented as a 26-tuple, an entry for every lower-case letter of the English alphabet.

To fill H, we iterate over A in two distinct parts. For the first k elements, compute their frequency table directly and insert it into H once the k-th iteration of the loop has been reached. For the next $|A| - k$ elements, use a sliding window technique to insert a new frequency table at every iteration of the loop, with the difference being removing (or decrementing) $A[i]$, and inserting (or incrementing) $A[i + k]$.

The time complexity of filling H is the sum of $O(k)$ for the first k elements, $(|A| - k) \times O(1)$ for the remaining $|A| - k$ elements (each element in the loop doing $O(1)$ work). Therefore, the overall time complexity is $O(|A|)$.

Note that since the frequency of any character is at most n, each frequency table can be thought of as a $26 \lceil \log n \rceil$ sized integer, which fits into a constant number of machine words. Therefore, it can be used as a hash key.

To check the count of B's anagrams in $O(k)$ time, simply compute B's frequency table in $O(k)$ time, and check if B's frequency table exists in H. If it exists, return the corresponding count, otherwise, return 0.

(b) Initialize result array A and the data structure from Part (a) in $O(T)$ time, with parameters T and k. Let us call this data structure H. Then iterate through $\mathcal{S}$. For each $S_i \in \mathcal{S}$, use $O(k)$ time to compute the frequency table of S_i (let us call this frequency table F_i), and $O(1)$ time to append the result of $H[F_i]$, the count of anagrams in T, to the result array A.

The loop runs for n iterations, therefore the time complexity for generating A is $O(nk)$. The total time complexity for the algorithm is the sum of the time to generate the data structure H and the loop over $\mathcal{S}$. This is $O(T + nk)$ as required.

(c) Submit your implementation to `alg.mit.edu`.