Problem Set 2

Problem 2-1. [0 points] **Instructions**

All parts are due on March 4, 2020 at 10PM. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct. Solutions should be submitted on Gradescope, and any code should be submitted for automated checking on `https://alg.mit.edu`.

Please abide by the collaboration policy as stated in the course information handout. Note that the first problem is an *Individual* problem and collaboration on this problem is not allowed.

Problem 2-2. [20 points] **Non-Collaborative Problem: Hashing**

Chase wishes to create a hash table to store his n credit card numbers. Each credit card number is composed of 6 digits, so there are 10^6 possible credit card numbers. He will solve collisions via chaining.

(1) [5 points] Which of the following hash table sizes would be most appropriate for this application?

1. A constant (i.e. not dependent on n), approximately 100
2. A constant (i.e. not dependent on n), approximately 10^6
3. $O(n)$
4. $O(n^2)$
5. $O(n^6)$

Solution: The answer is $O(n)$. The size of a hash function should be linear in the number of values we wish to store; this space efficiency is one of the benefits over a direct access array.

(2) [5 points] Chase chooses to use the hash function $f(x) = (17x + 2) \mod 97$,[1] and he uses it to insert the following credit card numbers into his hash table:

$$492585, 512054, 509422, 389142, 320943.$$

After inserting these card numbers into his hash table, how many collisions are there?

Solution: **One.** Computing the hashes, we get $34, 43, 16, 16,$ and 74, respectively. Hence, there is a single collision (in bucket 16).

(3) [5 points] He continues by inserting

$$911625, 130959, 533131.$$

What is the length of the longest chain in the hash table following these insertions?

Solution: We use the hash function again to compute hashes of $34, 58,$ and 34. We see that both 911625 and 533131 hash to the same value that 492585 did, resulting in a longest chain of length **three**.

(4) [5 points] Chase forgot which credit card numbers he had, and wants to check whether X is one of his credit card numbers. He determines that X is *not* one of his credit card numbers, and notices that checking whether X was in the hash table took the longest it could have possibly taken, given the current state of his hash table. Which of the following could have been X?

1. 492585
2. 581771

[1] In practice, this hash function would not be used because it lacks some probabilistic properties. You don't need to worry about this, as it is beyond the scope of 6.006.

3. 901100
4. 320944
5. 145034

Solution: The value of X could have been 145034. This value also hashes to 34, and so searching for membership in this hash table would require traversing the entire chain of length 3. The answer could not have been 492585, since this is actually in the hash table, and we are told X was not.

Problem 2-3. [20 points] **Individual problem 2**

Ben Bitdiddle is trying to schedule a Zoom call with all of his friends tomorrow. Everyone has given him intervals of time during which they are not available. However, Ben Bitdiddle would like to simplify this information before finding a time that works for everyone. In particular, if one interval is entirely included within another, he wants to get rid of it because it doesn't tell him anything new. Suppose that Ben gets n total inclusive intervals (e.g. $[1, 2]$; $[1, 3]$; $[2, 3]$); can you help him remove all the intervals that are included within another?

(1) [5 points] Under what condition is one interval $[a, b]$ entirely contained within another interval $[c, d]$?

Solution: When $c \leq a$ and $b \leq d$.

Rubric:

- 5 points for correct upper and lower bounds.
- Partial credit may be awarded.

(2) [15 points] Design an $O(n \log n)$ solution.

Solution: Sort the intervals by their left endpoints with ties broken by the right endpoint (increasing order). Notice that an interval $[x_i, y_i]$ can only potentially be included in intervals that precede it in the sorted order. Indeed, only when $j < i$ can we have $x_j \leq x_i$. Scan the list from left to right keeping track, for every i, of the maximum right endpoint y_i^+ seen so far. That is, $y_i^+ = \max\{y_j | j < i\}$. Then, the i-th interval $[x_i, y_i]$ is not included in any other interval if and only if $y_i > y_i^+$. Indeed, if the condition holds, then no interval $[x, y]$ has both $x < x_i$ and $y > y_i$ and, if the condition does not hold, there is an interval $[x_j, y_j]$ for $j < i$ and with $y_j = y_i^+ \geq y_i$ so $[x_j, y_j]$ contains $[x_i, y_i]$.

Correctness: Has been argued in the paragraph above.

Time Analysis: Sorting takes $O(n \log n)$ time. The linear scan takes $O(n)$ for a total of $O(n \log n)$ time.

Rubric:

- 5 points for sorting the intervals (with correct ordering).
- 5 points for a linear scan.
- 5 points for correctly tracking the maximum right endpoint.
- Partial credit may be awarded.

Problem 2-4. [15 points] **Lazy Sorting**

Tim would like to be maximally lazy (like any good computer scientist) when sorting arrays, by finding the smallest slice he'd need to sort for the entire array to be sorted. That is, for an array L, he wishes to find indices (i, j) such the array

$$L[: i] + \text{sorted}(L[i : j+1]) + L[j+1 :]$$

is sorted. For example, if the original array is

$$[1, 3, 4, 14, 6, 5, 7, 10, 17, 20],$$

the algorithm (using zero-indexing) should return $(3, 7)$, which corresponds to the subarray $[14, 6, 5, 7, 10]$. Note that the rest of the elements are already in their final sorted location. Find an $O(n)$ time algorithm for this task, and briefly justify correctness and runtime.

Solution:

We first find the left index. The right index can be found in a similar fashion.

Note that, in what follows, we use Python's indexing notation.

To find the left index, we do as follows. First we scan the list from left to right until we find the first index i such that $L[i] < L[i-1]$. Then clearly the left index must be before i. The problem is that, even though $L[: i]$ is sorted, there may be elements to the right of i that are smaller than some elements to the left of i. So we find the minimum element m in the list $L[i :]$. Finally, we go back, starting from i, until we find the first entry $L[i']$ smaller than m. Then the sought after left index is $i' + 1$.

Correctness: Follows from noticing that the segment $L[: i'+1]$ is sorted and also that all elements in $L[i'+1 :]$ are larger than $L[i']$.

Time analysis: The first scan takes $O(n)$. Finding the minimum m also takes $O(n)$. Finally, scanning back to find i' takes also $O(n)$. Finding the right index takes an equal amount of time. All in all, the algorithm works in $O(n)$ time.

Rubric:

- 10 points for a correct algorithm.
- 3 points for justifying correctness.
- 2 points for justifying runtime.
- Partial credit may be awarded.

Problem 2-5. [15 points]

Given an unsorted list L of n integers, find the greatest difference D between two consecutive elements in the list when sorted. For example, if $L = [10, 8, 15, 9]$, then $D = 5$ (the sorted list is $[8, 9, 10, 15]$ and the biggest difference occurs between 10 and 15).

(1) [3 points] Suppose that the list of integers has a maximum value of `max` and a minimum value of `min`. Find a lower bound l on the desired "maximum difference".
Hint: think of some examples. Suppose you have a sequence of 5 numbers where the minimum is 1 and the maximum is 10. Can the maximum difference be 1?

Solution: $l = \lfloor \frac{max-min}{n} \rfloor$. If the maximum difference of consecutive elelments is strictly less than l, then we cannot get from `min` to `max` in n steps.

Rubric:

- 3 points for correct lower bound.

(2) [12 points] Construct an $O(n)$ solution to the problem. Briefly justify correctness and runtime.
Hint: consider grouping elements that are less than l apart from each other into buckets.

Solution: By Part (1), we know that the greatest difference D must be at least l. In a sense, that means that any two numbers that are closer than l, can be regraded as one. This motivates the idea of bucketing numbers so that numbers in a bucket are closer than l. So we divide the interval $[\min, \max]$ into n sub-intervals (buckets), each of length l. Note that a number $x \in [\min, \max]$ falls into bucket $\lfloor \frac{x-\min}{l} \rfloor$ and that there are $\frac{\max - \min}{l} = O(n)$ buckets altogether. Throw out any empty bucket.

For each bucket b calculate its minimum (b^-) and maximum values (b^+). Let's look at consecutive numbers of the original list L which are good candidates to be the further apart. First, in a bucket b with only 2 elements, we could have $b^+ - b^- = l$ (if the numbers are at the endpoints of the bucket), so, in this case, the two numbers in the bucket are a possible candidate. Also, between adjacent buckets b_j and b_{j+1}, we have the pair b_j^+ and b_{j+1}^- as a possible candidate. So we need to find the maximum distance between these linear number of candidates.

Correctness: Follows from the fact that the desired distance has to be between the $\max$ and $\min$ of consecutive non-empty buckets (or from within a bucket with only two numbers).

Time analysis: Placing the elements into buckets takes $O(1)$ per element for a total of $O(n)$. Computing the $\min$ and $\max$ in each bucket takes time linear in the size of the bucket so, in total, takes $O(n)$ time. Finally, finding the maximum distance between $\max$ and $\min$ of consecutive non-empty buckets in linear in the number of buckets or $O(n)$. We have a constant number of steps, each taking $O(n)$ time for a grand total of

$O(n)$.

Rubric:

- 4 points for correct buckets (number and ranges).
- 4 points for correctly tracking and updating the min/max of each bucket (in linear time).
- 4 points for correctly calculating the answer (with a linear scan).
- Partial credit may be awarded.

Problem 2-6. [30 points] **TubeYou**

TubeYou is rolling out a new system called ContentID to solve their copyright issues. For a new video uploaded to the platform, TubeYou wants to find if the video has used copyrighted music. Since TubeYou gets a billion hours uploaded every single day, they want to do this task very efficiently. TubeYou has a library of songs that it wants to check newly uploaded videos against.

Specifically, given a string T representing a video transcription, and a string S representing some song lyrics, we say that there is a match if there exists a contiguous substring of T that exactly matches S. As an example, if $T =$ `'baby shark doo doo doo'`, then T has a match with the strings $S =$ `'shark'` or $S =$ `'aby'`, but not with the string $S =$ `'grandpa'`.

For this problem, we assume that all our strings are made up of ASCII characters, which map each character to a number between 0 and 127 inclusive. The Python function `ord(c)` gives us the ASCII value of character `c`.

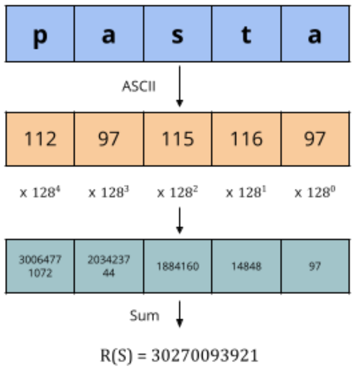
We can solve this problem using a simple brute force algorithm in $O(|T||S|)$ time: there are $|T| - |S| + 1$ contiguous substrings of T that have length $|S|$:

$$\mathcal{T} = \{T_i = (t_i, \ldots, t_{|S|+i-1}) \mid i \in \{0, \ldots, |T| - |S|\}\},$$

So for each substring $T_i \in \mathcal{T}$, we can check whether $T_i = S$, character-by character in $O(|S|)$ time. If we could determine whether $T_i = S$ in $O(1)$ time instead of $O(|S|)$ time, then the above algorithm would run much faster, in $O(|T|)$ time. We can use hashing to our advantage!

(1) [2 points] We start by creating a hash function for our strings. Given some string S, we want to create a function $R(S)$ that turns S into a number, such that $R(S_1) = R(S_2)$ if and only if $S_1 = S_2$.

Since we're using ASCII characters, we can actually just represent our string as $|S|$-digit base-128 numbers. Say there are 8 characters in our string, and the characters of S are $(s_0, s_1, s_2 \ldots)$. We can compute $R(S) = 128^7 \times s_0 + 128^6 \times s_1 + 128^5 \times s_2 \ldots$



In this case s_0 is known as the most-significant digit, which means that it gets assigned the highest power of 128.

Describe an algorithm to compute $R(S)$ using $O(|S|)$ arithmetic operations, i.e., `(-, +, *, //, %)`. Note that computing an exponent here is the expensive part, for instance 128^8 will take 8 operations. For parts (2), (3), and (4), assume that one arithmetic operation can operate on integers of arbitrary size.

Solution: The entire problem is based on the **Rabin-Karp** fingerprint algorithm.

The general idea is to reuse the powers of 128 from the last iteration. 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 pre-computes the powers 128^i from 0 to $|S|-1$ and then evaluates the sum directly.)

Rubric:

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

(2) [2 points] Now we have a way to turn a string into a number using $R(S)$. Consider T_i, which is the i^{th} substring of our video transcript of size $|S|$. Given, $R(T_i)$ describe an algorithm to compute $R(T_{i+1})$ using $O(1)$ arithmetic operations.

You can assume you have access to a value $f = 128^{|S|}$ precomputed.

Solution: Observe that $R(T_i) = t_i \cdot 128^{|T|-1} + \sum_{j=1}^{|T|-1} t_{i+j} \cdot 128^{(|T|-1-j)}$ and $R(T_{i+1}) = t_{i+|T|} + \sum_{j=0}^{|T|-2} t_{i+1+j} \cdot 128^{(|T|-1-j)} = t_{i+|T|} + 128 \sum_{j=1}^{|T|-1} t_{i+j} \cdot 128^{(|T|-1-j)}$. So:

$$R(T_{i+1}) = (128 \cdot R(T_i) - f \cdot t_i) + t_{i+|T|},$$

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

Rubric:

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

(3) [4 points] Describe an algorithm to check if a song lyrics snippet S appears inside the video transcript T using at most $O(|T|)$ arithmetic operations.

Solution: First compute f naively using $O(|S|)$ multiplications. Compute $R(S)$ and $R(T_0)$, each using $O(|S|)$ arithmetic operations via part (a). Then repeat the following algorithm for i starting at 0, terminating when i is $|T|-|S|$: check whether $R(S)$ equals $R(T_i)$ by evaluating whether $R(S) - R(T_i) = 0$ using a single arithmetic operation. If they are equal, then return True, as $R(S) = R(T_i)$ if and only if $S = T_i$. Otherwise, compute $R(T_{i+1})$ from $R(T_i)$ and f using $O(1)$ arithmetic operations via part (b) and increase i by one. This brute force algorithm uses $O(|S|) + (|T| -$

$|S|) \cdot O(1) = O(|T|)$ arithmetic operations and is correct because it checks whether $R(S) = R(T_i)$ for every $T_i \in \mathcal{T}$.

Rubric:

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

The above algorithms pose a problem: if string $|S|$ is large, then $R(|S|)$ can be $\Omega(128^{|S|})$, 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 $S = T_i$, then certainly $R'(S) = R'(T_i)$, and we will be able to identify the match quickly! However, sometimes $R'(S) = R'(T_i)$ when $S \neq T_i$, 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 $R'(S) = R'(T_i)$ when $S \neq T_i$ is at most $\frac{1}{p} \leq \frac{1}{|S|}$, for any $T_i \in \mathcal{T}$ (we assume that $|S| \leq p$).

(4) [2 points] Given $R'(T_i)$ and the value $f' = (128^{|S|}) \bmod p$, describe an $O(1)$-time algorithm to compute $R'(T_{i+1})$.

Solution: This part is the same as (2) except that the computation should be taken modulo p:

$$R'(T_{i+1}) = R(T_{i+1}) \bmod p = (128 \cdot R'(T_i) - f' \cdot t_i) + t_{i+|S|} \bmod p.$$

The key insight here is that multiplication and addition are true under modulo, and it doesn't matter if we use $R'(T_i)$ or $R(T_i)$ since $R'(T_i) = R(T_i) \bmod p$.

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:

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

(5) [5 points] Describe an *average* $O(|T|)$-time algorithm to solve the same problem in (3) using our new R'. Note that now we may have collisions where $A \neq B$ even if $R'(A) = R'(B)$.

Solution: This part is the same as part (3) except that we compute and compare $R'(S)$ with $R'(T_i)$ values instead of $R(S)$ and $R(T_i)$. We can compute f' in $O(|S|)$ time by performing multiplications modulo p, and we can similarly compute $R(T_0)$ via the computations in (a) modulo p in $O(|S|)$ time. When comparing $R'(S)$ to $R'(T_i)$, if $R'(S) \neq R'(T_i)$, we know that $S \neq T_i$ and we can continue. However, when $R'(S) = R'(T_i)$, it is possible that $S \neq T_i$, so and we spend $O(|S|)$ time to compare each character of S against each character of T_i. If the two match, then return True, otherwise, continue checking T_{i+1}. The time spent checking each T_i is $O(1)$ when $R'(S) \neq R'(T_i)$, and $O(|S|)$ when $R'(S) = R'(T_i)$. However, by the assumption provided, this only occurs with probability $1/|S|$ when $S \neq T_i$, so the expected time to check any particular T_i is $O(1)$. Thus this algorithm runs in average $O(|T|)$ time.

Rubric:

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

(6) [15 points] Implement `copyright_match` that implements your above algorithm. As your prime p, please use the provided Mersenne prime $p = 2^{31} - 1$. Submit your code on `https://alg.mit.edu/spring21/PS2`

Please submit a screenshot of your report that includes your `alg.mit.edu` student ID (top right corner) and the percentage of tests passed.

Solution:

Rubric:

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