CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 9

For: March 17–March 21

Problem 1. Quadratic Probing

Quadratic probing is another open-addressing scheme very similar to linear probing. Recall that a linear probing implementation searches the next bucket on a collision.

We can also express linear probing with the following pseudocode (on insertion of element x):

```
for i in 0..m:
  if buckets[hash(x) + i % m] is empty:
    insert x into this bucket
    break
```

Quadratic probing follows a very similar idea. We can express it as follows:

```
for i in 0..m:
  // increment by squares instead
  if buckets[hash(x) + i * i % m] is empty:
    insert x into this bucket
    break
```

(a) Consider a hash table with size 7 with hash function `h(x) = x % 7`. We insert the following elements in the order given: 5, 12, 19, 26, 2. What does the final hash table look like?

Solution: [26,X,19,2,X,5,12]

(b) Continuing from the above question, we now delete the following elements in the order given: 12, 5. What does the final hash table look like?

Solution: [26,X,19,2,X,5 (deleted),12 (deleted)]

(c) Can you construct a case where quadratic probing fails to insert an element despite the table not being full?

Solution: Consider the case when table capacity = 3, buckets 0 and 1 filled but 2 unfilled. Insertion of x where $\text{hash}(x) = 0$ would fail.

Problem 2. Implementing Union/Intersection of Sets

You are given 2 finite sets, A and B. How can you efficiently find the intersection and union of the two sets?

Solution: **Intersection:** The general idea for intersection is to hash all the items for one of the two sets, e.g. A. Then go through all the items of $b \in B$, and check whether b is also in A by using the hashtable. The runtime in expectation is $O(|A| + |B|)$.
Union: On the other hand, for the union, we just want to insert all the elements of A into the table. Then insert all the elements of B into the table. The insertion algorithm now needs to handle the possibility of duplicate keys being inserted.
In particular, for hashing with chaining, this means that when inserting elements from B, we need to run through the entire chain to make sure we don't "double insert" the same element. For hashing with linear probing, there is not a big difference, since during insertions as you probe the entire run, if you see that the key has been inserted, you can just return thereafter.
After we insert elements from A and B, run through the table again to output the elements. Again, the runtime in expectation is $O(|A| + |B|)$.

Problem 3. You're given an array of n integers (possibly negative), and an value k. Decide if there is a contiguous sub-array whose average value is k.

E.g. Given array $[1, 3, 2, 5, 7, 20]$, and $k = 6$. Then the answer is yes, because $[5, 7]$ has average value 6.

What is a straightfoward solution that solves this problem in $O(n^2)$ time? What is a solution that solves this in expected $O(n)$ time?

Solution: The straightforward solution is to try every pair of starting and ending points as a sub-array. Compute the average value and check if it is equal to k. This takes $O(n^3)$ time (or $O(n^2)$ with a quick optimization).
For the less straightforward solution, let's think of the following:

1. Subtract every element of the array by k, call this o_arr.
2. Compute the cumulative-sum array, call this c_arr.

Using our example in the question above, the arrays will look like:

1. $arr = [1, 3, 2, 5, 7, 20]$
2. $o\_arr = [-5, -3, -4, -1, 1, 14]$
3. $c\_arr = [-5, -8, -12, -13, -12, 2]$

Think what about what $c\_arr[i] - c\_arr[j]$ represents:

$$\begin{aligned} c\_arr[i] - c\_arr[j] &= \sum_{r=0}^{i} o\_arr[r] - \sum_{r=0}^{j} o\_arr[r] \\ &= \sum_{r=0}^{i} (arr[r] - k) - \sum_{r=0}^{j} (arr[r] - k) \\ &= \sum_{r=j+1}^{i} (arr[r] - k) \end{aligned}$$

So if we find $i > j$ such that $c\_arr[i] - c\_arr[j] = 0$, then

$$\sum_{r=j+1}^{i} (arr[r] - k) = 0$$

$$\sum_{r=j+1}^{i} (arr[r]) - (i - (j+1) + 1)k = 0$$

$$\sum_{r=j+1}^{i} (arr[r]) - (i - j)k = 0$$

$$\sum_{r=j+1}^{i} (arr[r]) = (i - j)k$$

$$\frac{\sum_{r=j+1}^{i} (arr[r])}{(i - j)} = k$$

i.e. notice if we find 2 distinct positions in the prefix sum array that have the same value, then we know that corresponding subarray of the original input array has mean k.
So, when we process the cumulative sum array, to find repeat values, use a hashtable!

Problem 4. (Priority queue)

There are situations where, given a data set containing n unique elements, we want to know the top k highest-valued elements. A possible solution is to store all n elements first, sort the data set in $O(n \log n)$, then report the right-most k elements. This works, but we can do better.

(a) Design a data structure that supports the following operation better than $O(n \log n)$:

- `getKLargest()`: returns the top k highest-valued elements in the data set.

(b) Instead of having a static data set, you could have the data streaming in. However, your data structure must still be ready to answer queries for the top k elements efficiently. Expand or modify your data structure to support the following two operations better:

- `insertNext(x)`: adds a new item x into the data set in $O(\log k)$ time.
- `getKLargest()`: returns the current top k highest-valued elements in the data set in $O(k)$ time.

For example, if the data set contains {1, 13, 7, 9, 8, 4} initially and we want to know the top 3 highest value elements, calling `getKLargest()` should return the values {13, 9, 8}.

Suppose we then add the number 11 into the data set by calling `insertNext(11)`. The data set now contains {1, 13, 7, 9, 8, 4, 11} and calling `getKLargest()` should return {13, 11, 9}.

Note: we do not need to have to return the elements in sorted order.

Solution: For part (a), we can quick-select the k^{th} largest element in expected $O(n)$ time. During this process, the collection gets partitioned and all elements strictly larger than the k^{th} largest element will be on the right of the pivot. These elements together with the pivot are exactly what we want.

For part (b), two key observations are:

1. **the bottom $(n-k)$ lower-valued elements are irrelevant to our query**, we only need to keep the top k elements.

Make a min-heap. Keep inserting elements as long as our heap is of size $< k$. If we were to insert a new element x into a heap of size k, check x against the smallest of the k elements we have stored. If x is smaller, discard x. If x is larger, insert x into the heap, and extract-min from the heap. This way the heap stores the k largest elements.

Problem 5. Stack 2 Queue

Do you know that we actually can implement a queue using two stacks? But is it really efficient?

(a) Design an algorithm to `push/enqueue` and `pop/dequeue` an element from the queue using two stacks (and nothing else).

Solution: Let's call the two stacks as S_1 and S_2.
Push

- **Push(x)** - Push x onto stack S_1.
- **Pop(x)** - If S_2 is not empty, pop from S_2. If S_2 is empty, repeatedly pop from S_1 and push into S_2.

Intuitively, if S_2 is empty, we are just "reversing" S_1 by popping it into S_2. So we can turn LIFO into FIFO.

(b) Determine the *worst case* and *amortized* runtime for each operation. Recall that if push was amortised to a cost $O(f(n))$, pop was amortised to a cost of $O(g(n))$, then after a series of t pushes and s pops, the sum total cost of the entire series of operations is at most $O(t \cdot f(n)) + O(s \cdot g(n))$.

Solution: The worst case for the push operation is $O(1)$. The worst case for the pop/dequeue operation is $O(n)$, where n is the number of inserted elements.
However, the amortized cost is much smaller than that. We will prove that the amortized cost for each operation is $O(1)$ using *Accounting Method*. Whenever we push a new element, we will deposit \$2 to the bank (think of this as paying for time in advance). When we transfer an element from S_1 to S_2 or popping an element from S_2, we will pay \$1.
Note that when pop is called and S_2 is empty, we have at least $\$2k$ deposited in the bank, where k is the number of inserted elements in S_1. This is enough money to pay for the transferring cost that takes $O(k)$ time. Note that the remaining money $\$2k - \$k = \$k$ can be used to pay for when the element is popped from S_2.
So any sequence of t inserts and pops costs $O(t)$ time, not $O(t^2)$ time.

Problem 6. Min Queue

Implement a queue (FIFO) that supports the following operations:

- `push/enqueue` - pushes/enqueues a value x
- `pop/dequeue` - pops/dequeues a value x
- `getMin` - returns the minimum value currently stored in the queue

Do this so that any sequence of t operations runs in $O(t)$ time. (I.e. the sum total cost of t operations is $O(t)$)

Solution: One solution that doesn't work is to just use a single queue, and when `getMin` is called, run through the entire array. If we had to recompute the minimum value every time after a push or a pop, this costs $O(n^2)$ for n operations.
There are a few solutions for this, including using a linked list with some "skip" pointers that you might find. However, the simplest solution actually uses two stacks.
Think first about how to create a stack that supports:

- `push` - pushes/enqueues a value x
- `pop` - pops/dequeues a value x
- `getMin` - returns the minimum value currently stored in the stack

The way to do this would be to store pairs of values instead of just values alone.
The idea is that we will store pairs (x, m). Where m is the minimum value of the stack. This way, the implementation of `getMin` is to simply peek the top pair of the stack, obtain (x, m), and output m.
To insert x into an empty stack, insert (x, x). To insert into a non-empty stack, run `getMin` to get m. push $(x, \min(m, x))$ onto the stack.
To pop, just remove the pair itself.
So how do we do this for a queue? Use the solution to the previous question!