CS2040S: Data Structures and Algorithms

Recitation 7

Goals:

- How can hash tables be used efficiently to solve problems?
- Understand variants of hashing

Note: In this recitation, you may assume that hashing takes $O(1)$.

Problem 1. Putting the Sum in DimSum

You have an unsorted array A containing n integers where each element represents an *item price* on the menu at *Dim-Sum Dollars*—your favorite dim sum restaurant, and its index representing the *item number* on the menu. Having x dollars on hand in cash and wanting to avoid loose change, you seek to purchase two items such that their price sums up to *exactly* x.

For example, given the following menu:

\$30	\$8	\$15	\$18	\$23	\$20	\$25	\$1
#1	#2	#3	#4	#5	#6	#7	#8

If your cash on hand is $x = \$33$, then there are two possible item pairs whose values sum up to that amount, namely pairing #2 (\$8) with #7 (\$25) and pairing #3 (\$15) with #4 (\$18).

Your task is therefore to propose algorithms which will find you *any* such winning pairs of dim sum items on the menu.

Problem 1.a. Come up with an algorithm which finds a valid pair of item *numbers* if it exists. For instance in the given example above, either $(\#2, \#7)$ or $(\#3, \#4)$ will be a valid pair. your solution must only incur $O(1)$ space. What is its time complexity?

Problem 1.b. Now come up with an algorithm which finds a valid pair of item *prices* if it exists. For instance in the given example above, either $(\$8, \$25)$ or $(\$15, \$18)$ will be a valid pair. Your solution must only incur $O(\log n)$ extra space. What is its time complexity?

How might you then modify your solution to return a valid pair of item *numbers* instead and what are the additional overheads (if any) due to your modification?

Problem 1.c. Finally, come up with the *fastest* algorithm which finds a valid pair of item *numbers* if it exists. For instance in the given example above, either $(\#2, \#7)$ or $(\#3, \#4)$ will be a valid pair. Your solution may now incur $\Theta(n)$ extra space. What is its time complexity?

How might you modify your solution to cater for duplicate prices?

Depending on your solution proposed, you might have introduced a subtle caveat. *Hint:* What happens when target sum is an even number?

Problem 1.d. How might you minimally modify the previous 3 algorithms such that instead of just finding one valid pair, *all valid pairs* will be returned by the algorithm?

Problem 1.e. What if instead of two, now you want to find *three* elements a, b, c in the array that sum to x. Can you generalize the previous solutions to solve this variant?

Did you know? This is the classic 3SUM problem! It turns out that there exists many important problems which either reduces to this form or entails a subproblem that does. There is actually a lot of ongoing research trying to understand the best possible solution for 3SUM!

Problem 2. Cuckoo Hashing

This week, we are going to learn a new hashing called Cuckoo Hashing. The basic idea behind Cuckoo Hashing is that we use to tables A and B of size $m = 2n$ to store n items. For table A, we use hash function f, and for table B, we use hash function g. To insert an item, we proceed as follows, executing insert(x, A, f):

```
insert(item x, Table T, hashfunction h)
      slot = h(x)
      z = T[slot]
      T[slot] = x
      if (z != null)
            if (T == A)
                  insert(z, B, g)
            else if (T == B)
                  insert(z, A, f)
```

An item can be searched as follows.

```
search(x)
      if (A[f(x)] == x) then return A[f(x)];
      if (B[g(x)] == x) then return B[g(x)];
      else return NOT_IN_TABLE;
```

Problem 2.a. Inserting an item can fall into three different cases which are given below. For each case, given an example and analyse the running time.

1. All nestless keys are distinct.
2. Nestless keys are not distinct. But, the insertion process terminates.
3. Nestless keys are not distinct. But, the insertion process does not terminate.

Problem 2.b. For case 3, we need to do rehashing and resizing to insert the remaining item. Assume we maintain hashtables of size at least r where $r >= (1 + \epsilon)n$. Here ϵ is a constant. Find the time needed to insert n items into the tables using Cuckoo hashing.

Problem 2.c. What are some advantages of cuckoo hashing over other hashing schemes?