CSE 332: Data Abstractions

Section 1: `WorkLists` Solutions

`WorkList` ADT

`add(`**work**`)`	Notifies the worklist that it must handle **work**
`peek()`	Returns the next item to work on
`next()`	Removes and returns the next item to work on
`hasWork()`	Returns true if there's any work left and false otherwise

0. Interview Question: The Missing Number

Suppose `nums` is a `WorkList` of size n which contains each number between 1 and n exactly once. A user calls `nums.next()` but forgets to save the value. Recover the value that was removed. Can you do it if two values are removed? What are the time and space complexity of your solution?

Solution:

We know that the total sum S of all of the numbers in `nums` before the user calls `nums.next()` is equal to $S = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. We also know that after the user calls `nums.next()`, the total sum of the remaining numbers R must be $S = R + a$, where a is the missing number. We then find the missing number by solving for $a = S - R$. In code, this looks something like:

```
function findRemoved(nums)
   sum = n * (n + 1) / 2
   while (nums.hasWork())
      sum = sum − nums.next()
   return sum
```

If two numbers are missing, we can use a similar approach, and start with a modified equation $S_1 = R_1 + a + b$, where S_1 is the sum, R_1 is the sum of the remaining numbers in the worklist, and a and b are our unknowns. However, we cannot solve for two unknowns with one equation – we need a second. This equation could be anything, but we'll use the sum of the *squares* of each number such that $S_2 = R_2 + a^2 + b^2$ where:

$$S_2 = \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{2}, \qquad R_2 = \sum_{i \in \texttt{nums}} i^2 = \text{use a while loop}$$

For simplicity, let $d_1 = S_1 - R_1 = a + b$ and $d_2 = S_2 - R_2 = a^2 + b^2$. We can then solve for a and b algebraically:

$$d_2 = a^2 + (d_1 - a)^2 \qquad \text{substitute}$$

$$0 = 2a^2 - 2d_1 a + (d_1^2 - d_2) \qquad \text{expand and rearrange}$$

$$a = \frac{2d_1 \pm \sqrt{4d_1^2 - 8(d_1^2 - d_2)}}{4} = \frac{d_1 \pm \sqrt{2d_2 - d_1^2}}{2} \qquad \text{use the quadratic formula; simplify}$$

The two possible values of a therefore must be the two missing values a and b. We can express this in code:

```
1  findTwoRemoved(nums)
2     s1 = n * (n + 1) / 2
3     s2 = n * (n + 1) * (2n + 1) / 6
4     while (nums.hasWork())
5        i = nums.next()
6        s1 = s1 − i
7        s2 = s2 − i * i
8     return (s1 ± √(2s2 − s1^2))/2
```

Note that at the end of the while loop in line 8, we have $\texttt{s1} = a + b = d_1$ and $\texttt{s2} = a^2 + b^2 = d_2$.
Both solutions will run in $\Theta(n)$ time and will use $\Theta(1)$ extra space.

1. Choosing The Data Structures

Choose data structures and algorithms to solve the following problems:

(a) Call all the phone numbers with a particular area code in someone's phone book.

What is the time complexity of your solution? What is the space complexity?

Solution:

One way to solve this would be using a `HashMap` where the keys are the area codes and the values is a list of corresponding phone numbers. We will need to parse the phone number to get the first three numbers.

Another way to solve this is by using a `Trie`. We would use the entire phone number as the "route" and insert all numbers into the trie. Then, to find all the phone numbers to call, we would use the area code to partially travel down the `Trie`, then visit all children nodes to find up the phone numbers to print.

If we compare these two approaches, both will have the same runtime efficiency, but the `Trie` will be more space-efficient in the average case.

If we let n be the total number of phone numbers and e be the expected number of phone numbers per area code, we can find that it takes $\Theta(n)$ time to build either the `HashMap` or the `Trie`. Likewise, given some area code, it takes $\Theta(e)$ time to visit and call each phone number.

(Initially, it may seem like the `Trie` would be slower due to the traversals. However, recall that the depth of the trie is always equal to the length of a phone number, which is a constant value.)

The reason why the `Trie` turns out to be more space-efficient on average is because the `Trie` is capable of storing near-duplicate phone numbers in less space then the `HashMap`. If we have the phone numbers `123-456-7890`, `123-456-7891`, and `123-456-7892`, the map must store each number individually whereas the `Trie` is able to combine them together and only branch for the very last number.

That said, in the absolute worst case where we try and insert every single possible 10-digit permutation of numbers into either data structure, both the `HashMap` and the `Trie` will end up taking up the same amount of space. However, this is an unlikely scenario, given how phone numbers are typically structured.

(b) Text on nine keys (T9)'s objective is to make it easier to type text messages with 9 keys. It allows words to be entered by a single keypress for each letter in which several letters are associated with each key. It combines the groups of letters on each phone key with a fast-access dictionary of words. It looks up in the dictionary all words corresponding to the sequence of keypresses and orders them by frequency of use. So for example, the input '2665' could be the words `{book, cook, cool}`. Describe how you would implement a T9 dictionary for a mobile phone.

Solution:

One way to implement this would be by using a `Trie`. The routes (branches) are represented by the digits and the node's values are collection of words. So if you typed in 2, 6, 6, 5, you would choose the child representing 2, then 6, then 6, then 5, traveling four layers deep into the `Trie`.

Then, that child node's value would contain a collection of all dictionary words corresponding to this particular sequence of numbers.

To populate the `Trie`, you would iterate through each word in the dictionary, and first convert the word into the appropriate sequence of numbers.

Then, you would use that sequence as the key or "route" to traverse the `Trie` and add the word.