Section 01: Solutions

1. CSE 143 review

These section problems are meant to serve as review of intro programming concepts like those taught in 142/143. Use these problems to review any concepts and data types you feel unsure about, and feel free to come back to them anytime for practice. It is normal to be a little out of practice and to struggle with many of these problems, but it is not something that should not prevent you from succeeding in this course so remember to review early!

1.1. Reference semantics

(a) What does this program print out? (Note: you should probably try and draw diagrams to help you figure out the answer rather then doing it in your head!)

```
import java.util.Arrays;

public class Mystery {
    public static void main(String[] args) {
        int x = 1;
        int[] a = new int[2];
        mystery1(x, a);
        System.out.println(x + " " + Arrays.toString(a));

        x -= 1;
        a[1] = a.length;
        mystery1(x, a);
        System.out.println(x + " " + Arrays.toString(a));

        mystery2(x, a);
        System.out.println(x + " " + Arrays.toString(a));
    }

    public static void mystery1(int x, int[] list) {
        list[x] += 1;
        x += 1;
        System.out.println(x + " " + Arrays.toString(list));
    }

    public static void mystery2(int x, int[] list) {
        list = new int[]{x, x};
        System.out.println(x + " " + Arrays.toString(list));
    }
}
```

Solution:

Remember that Java "passes by value" for primitive types (like `int`s) and "passes a reference (aka pointer)" for Object types. That means that if you change an `int` inside a function call, it will not be altered at all in the original method. On the other hand, if you mutate an object (i.e. change its fields) in a function call, the changes will be reflected in the original method. Unless you change what the reference is pointing to (by doing an assignment with the "=" operator), any changes after reassignment will not be reflected in the caller.

You can find a line-by-line walkthrough of the code using memory diagrams in the Section slideshow. This is the final output of the program:

```
2 [0, 1]
1 [0, 1]
1 [1, 2]
0 [1, 2]
0 [0, 0]
0 [1, 2]
```

(b) What is the output of this other program?

```
public class Mystery2 {
    public static void main(String[] args) {
        Point p = new Point(11, 22);
        System.out.println(p);

        int n = 5;
        mystery(p, n);
        System.out.println(p);

        p.x = p.y;
        mystery(p, n);
        System.out.println(p);

        Point p2 = new Point(100, 200);
        p = p2;
        mystery(p2, n);
        System.out.println(p + " :: " + p2);
    }

    public static void mystery(Point p, int n) {
        n = 0;
        p.x = p.x + 33;
        System.out.println(p.x + ", " + p.y + " " + n);
    }

    public static class Point {
        public int x;
        public int y;

        public Point(int x, int y) {
            this.x = x;
            this.y = y;
        }

        public String toString() {
            return "(" + this.x + ", " + this.y + ")";
        }
    }
}
```

Solution:

```
(11, 22)
44, 22 0
(44, 22)
55, 22 0
(55, 22)
133, 200 0
(133, 200) :: (133, 200)
```

1.2. Being an implementor of ArrayLists

Note: in this section, you are acting as an *implementor* of the `MyArrayList` data structure. Assume you are adding to the following `MyArrayList` class:

```
public class MyArrayList<T> {
    private T[] data;
    private int size;

    // constructors and other methods ommitted for space
}
```

For simplicity, assume the list does not contain any null items.

(a) Write a method `removeFront` that takes an integer `k` as a parameter and removes the first `k` values. For example, if a variable called `list` stores this sequence of values:

`[8, 17, 9, 24, 42, 3, 8]`

If we call `list.removeFront(4)`, the list should now store:

`[42, 3, 8]`

Assume that the parameter value passed is between 0 and the size of the list inclusive.

Solution:

```
public void removeFront(int k) {
    for (int i = 0; i < this.size - k; i++) {
        this.data[i] = this.data[i + k];
    }
    this.size -= k;
}
```

Note that we haven't actually deleted the last few elements of `data`. We will have to ensure all of our other methods do not try to access any elements of data beyond index `size - 1`.

(b) Write a method `removeBack` that takes an integer `k` as a parameter and removes the **last** `k` values. For example, if a variable called `list` stores this sequence of values:

`[8, 17, 9, 24, 42, 3, 8]`

If we call `list.removeBack(4)`, the list should now store:

`[8, 17, 9]`

Assume that the parameter value passed is between 0 and the size of the list inclusive.

Solution:

```
public void removeBack(int k) {
    this.size -= k;
}
```

(c) Write a method `removeAll` that takes in an item of type `T` and removes all occurrences of that value from the list.

For example, if the variable named `list` stores the following values:

`["a", "b", "c", "d", "a", "d", "d", "e", "f", "d"]`

If we call `list.removeAll("d")`, the list should now store:

`["a", "b", "c", "a", "e", "f"]`

Assume you have previously implemented a method called `remove` that takes an index as a parameter and removes the value at the given index.

Solution:

```
public void removeAll(T value) {
    for (int i = 0; i < this.size; i++) {
        if (this.data[i].equals(value)) {
            this.remove(i);
            i -= 1;
        }
    }
}
```

(d) Write a method `stretch` that takes an integer `k` as a parameter and that increases a list by a factor of `k` by taking each element in the original list and replacing it with `k` copies of that element. For example, if a variable called `list` stores this squence of values:

```
[18.2, 7.5, 4.2, 24.9]
```

If we call `list.stretch(3)`, the list should now store:

```
[18.2, 18.2, 18.2, 7.5, 7.5, 7.5, 4.2, 4.2, 4.2, 24.9, 24.9, 24.9]
```

Make sure to implement the logic to resize the array as necessary.

Solution:

```
public void stretch(int k) {
    this.ensureCapacity(k * this.size);
    for (int i = this.size - 1; i >= 0; i-=1) {
        for (int j = 0; j < k; j++) {
            this.data[i * k + j] = this.data[i];
        }
    }
    this.size *= k;
}

private void ensureCapacity(int newSize) {
    if (this.data.length < newSize) {
        T[] newData = (T[]) new Object[newSize];
        for (int i = 0; i < this.size; i++) {
            newData[i] = this.data[i];
        }
        this.data = newData;
    }
}
```

1.3. Being an implementor of LinkedLists

Note: in this section, you are acting as an *implementor* of the `MyLinkedList` data structure. Assume you are adding to the following `MyLinkedList` class:

```
public class MyLinkedList<T> {
    private Node<T> front;
    private static class Node<T> {
        public final T data;
        public Node<T> next;

        // constructors omitted for space
    }
}
```

Do not create any new nodes (unless you are trying to implement `stretch`). The focus of these problems is to practice manipulating the links of list nodes. For simplicity, assume the list does not contain any null items.

(a) Try implementing the methods described in questions 1.2.a-1.2.d for `MyLinkedList`.

Solution:

```
public void removeFront(int k){
    Node<T> curr = this.front;
    Node<T> prev = null;
    for (int i = 0; i < k; i++) {
        prev = curr;
        curr = curr.next;
        prev.next = null;
    }
    this.front = curr;
}
```

The `prev` pointer is not strictly necessary here – the code would remove the elements without it. We set `prev.next` to `null` to make it easier for the garbage collector to identify which nodes are no longer in use, and can be deleted. In a language like C or C++, you would need the `prev` pointer to deallocate the memory yourself, and you'll see other examples in this class where a pointer like that is helpful.

```
public void removeBack(int k){
    removeBackHelper(k, this.front);
}

private int removeBackHelper(int k, Node<T> curr){
    //base case, we're at the end of the list.
    if(curr.next == null){
        return k;
    }
    //recursive case: recurse to the end of the list.
    int leftToRemove = removeBackHelper(k, curr.next);

    //delete the element after curr, if it should be deleted.
    if(leftToRemove > 0){
        curr.next = null;
        leftToRemove -= 1;
    }

    //tell element before us how many things are left to delete.
    return leftToRemove;
}
```

```java
public void removeAll(T toDelete) {
    while(toDelete.equals(this.front)){
        remove(0);
    }
    int i = 0;
    Node<T> curr = this.front;
    Node<T> prev = null;
    while (curr != null) {
        if (curr.data.equals(toDelete)) {
            remove(i);
            curr = prev.next;
        } else {
            i++;
            prev = curr;
            curr = curr.next;
        }
    }
}

public void stretch(int k) {
    Node<T> curr = this.front;
    while (curr != null) {
        for(int i = 0; i < k-1; i++){
            curr.next = new Node<T>(curr.data, curr.next);
            curr = curr.next;
        }
        curr = curr.next;
    }
}
```

(b) Write a method `switchPairs` that switches the order of each pair of elements: your method should switch the first two values, then the next two, and so forth. For example, if a variable called `list` stores this sequence of values:

`[3, 7, 4, 9, 8, 12]`

If we call `list.switchPairs()`, the list should now store:

`[7, 3, 9, 4, 12, 8]`

If there are an odd number of values, the final element is not moved.

Solution:

An iterative solution is shown (although a recursive solution is cleaner, that task is left to the reader):

```
public void switchPairs() {
    if (this.front != null && this.front.next != null) {
        Node<T> current = this.front.next;
        this.front.next = current.next;
        current.next = this.front;
        this.front = current;
        current = current.next;
        while (current.next != null && current.next.next != null) {
            Node<T> temp = current.next.next;
            current.next.next = temp.next;
            temp.next = current.next;
            current.next = temp;
            current = temp.next;
        }
    }
}
```

(c) Write a method `reverse` that reverses the order of elements in a linked list. For example, if a variable called `list` stores this sequence of values:

`["a", "b", "c", "d", "e"]`

If we call `list.reverse()`, the list should now store:

`["e", "d", "c", "b", "a"]`

Solution:

```
public void reverse() {
    Node<T> current = this.front;
    Node<T> previous = null;
    while (current != null) {
        Node<T> nextNode = current.next;
        current.next = previous;
        previous = current;
        current = nextNode;
    }
    this.front = previous;
}
```

(d) Write a method `transferFrom` that accepts two `MyLinkedList` as parameters and moves values from the second list to the first list (and empties the second list). For example, suppose the two lists store these sequences of values:

```
list1: [8, 17, 2, 4]
list2: [1, 2, 3]
```

The call of `transferFrom(list1, list2)`, should leave the lists as follows:

```
list1: [8, 17, 2, 4, 1, 2, 3]
list2: []
```

Note that order matters: doing `transferFrom(list2, list1)` will NOT produce the same output as above. Either list may be empty. Assume, however, that the given lists will not be null.

Solution:

```
public void transferFrom(MyLinkedList<T> list1, MyLinkedList<T> list2) {
    if (list1.front == null) {
        list1.front = list2.front;
    } else {
        Node<T> current = list1.front;
        while (current.next != null) {
            current = current.next;
        }
        current.next = list2.front;
    }
    list2.front = null;
}
```

1.4. Being a client of Stacks and Queues

For the following problems, you will practice being the *client* of a data structure and two ADTs. Assume that you are given a class named `DoubleLinkedList` that implements both the Stack and Queue interfaces.

For simplicity, assume the stacks and queues you are given will never contain any null elements.

(a) Write a static method `copyStack` that accepts an `Stack<T>` as a parameter and returns a new stack containing exactly the same elements as the original. Your method should leave the original stack unchanged after the method is over. You may use one `Queue<T>` as auxiliary storage.

Solution:

```
public Stack<T> copyStack(Stack<T> stack) {
    Queue<T> queue = new LinkedList<T>();
    Stack<T> newStack = new Stack<T>();
    while (!stack.isEmpty()) {
        queue.add(stack.pop());
    }

    //stacks reverse the order of their elements,
    //so we need to pass the elements in a second time.
    while (!queue.isEmpty()) {
        stack.push(queue.remove());
    }
    while (!stack.isEmpty()) {
        queue.add(stack.pop());
    }

    while (!queue.isEmpty()) {
        T element = queue.remove();
        newStack.push(element);
        stack.push(element);
    }
        return newStack;
}
```

(b) Write a static method named `rearrange` that accepts an `Queue<Integer>` and rearranges the values so that all of the even values appear before the odd values and otherwise preserves the original order of the list. For example, suppose a queue called `q` stores this sequence of values:

`front [3, 5, 4, 17, 6, 83, 1, 84, 16, 37] back`

If we call `q.rearrange()`, the queue should now store:

`front [4, 6, 84, 16, 3, 5, 17, 83, 1, 37] back`

evens: 4, 6, 84, 16 — odds: 3, 5, 17, 83, 1, 37

Note that all of the evens are at the front, the odds are in the back, and that the order of the evens and the odds are the same as the original list. You may use one `Stack<Integer>` as auxiliary storage.

Solution:

```
public static void rearrange(Queue<Integer> q) {
    Stack<Integer> stack = new Stack<Integer>();
    int oldSize = queue.size();
    for (int i = 0; i < oldSize; i++) {
        int n = queue.remove();
        if (n % 2 == 0) {
```

```
            queue.add(n);
        } else {
            stack.push(n);
        }
    }
    int evenCount = queue.size();
    while (!stack.isEmpty()) {
        queue.add(stack.pop());
    }
    for (int i = 0; i < evenCount; i++) {
        queue.add(q.remove());
    }
    for (int i = 0; i < oldSize - evenCount; i++) {
        s.push(q.remove());
    }
    while (!s.isEmpty()) {
        q.add(s.pop());
    }
}
```

(c) Write a static method named `isPalindrome` that accepts a `Queue<T>` as a parameter and returns `true` if those values form a palindrome and `false` otherwise. For example, suppose a queue of ints called `q` stores this sequence of values:

`front [3, 5, 4, 17, 6, 6, 17, 4, 5, 3] back`

Then calling `isPalindrome` would return true, because the queue is exactly the same forwards and backwards (the definition of a palindromic object).

The queue must remain in its original state once the method is over. Assume that the empty queue is a palindrome. You may use one `Stack<T>` as auxiliary storage.

Solution:

```
public boolean isPalindrome(Queue<T> queue) {
    Stack<T> stack = new Stack<T>();
    for (int i = 0; i < queue.size(); i++) {
        int n = queue.remove();
        queue.add(n);
        stack.push(n);
    }
    boolean ok = true;
    for (int i = 0; i < queue.size(); i++) {
        int n1 = queue.remove();
        int n2 = stack.pop();
        if (!n1.equals(n2)) {
            ok = false;
        }
        queue.add(n1);
    }
    return ok;
}
```

2. Food For Thought – Design decisions: ArrayLists vs. Linked Lists

Look back at the code you wrote in section 1.2 and 1.3. For each of the methods you wrote for both ArrayLists and Linked Lists, determine the big-$\mathcal{O}$ running time for both versions of the code. In all of the following problems, let n be the length of the list.

(a) `removeFront(3)`

Solution:

ArrayLists: the for-loop does $n-3$ iterations (each taking constant time), we ignore lower order terms (like "-3") and constant factors (like the exact number of operations per for-loop iteration) so the running time is $\mathcal{O}(n)$.

Linked Lists: the for-loop does 3 iterations (each taking constant time), so the running time is $\mathcal{O}(1)$.

(b) `removeFront(n-3)` **Solution:**

ArrayLists: the for-loop does 3 iterations (each taking constant time), we ignore constant factors (like the exact number of operations per for-loop iteration) so the running time is $\mathcal{O}(1)$.

Linked Lists: the for-loop does $n-3$ iterations (each taking constant time), so the running time is $\mathcal{O}(n)$.

(c) `removeBack(4)`

Solution:

ArrayLists: we do one (constant time) operation: $\mathcal{O}(1)$.

Linked Lists: We will talk in a few weeks about how to use recurrences to carefully find the big-$\mathcal{O}$ of this code. For now, notice that we do a constant amount of work at each node (make another recursive call, possibly change a pointer, and return a number), this will lead to a running time of $\mathcal{O}(n)$.

(d) Suppose in your application you'll call `removeFront(3)` repeatedly, and it's critical that this be as efficient as possible. Which implementation should you choose? What if it's critical `removeFront(n - 3)` is efficient instead? `removeBack`?

Solution:

From our previous analysis, `removeFront(3)` takes $\mathcal{O}(n)$ time for ArrayLists, but $\mathcal{O}(1)$ time for Linked Lists, so we should choose Linked Lists here.

On the other hand `removeFront(n-3)` will take $\mathcal{O}(1)$ time for ArrayLists, but $\mathcal{O}(n)$ time for Linked Lists, so we should choose Array Lists.

For `removeBack`, Array Lists are faster regardless of how many items are to be removed.

Notice that depending on what you care about, the better implementation isn't always the same. Sometimes it isn't even clear from just a big-$\mathcal{O}$ running time analysis which implementation you should prefer. You might need to experiment with both or think about other aspects you care about beyond running time (like memory usage or ease of implementation). Learning how to think about these decisions and justify them is one of the goals of this course.

Challenge Problems:
Determine the big-$\mathcal{O}$ running time for both versions of the code. In all of the following problems, let n be the length of the list.

(a) `removeAll`. For this question, you'll need to think about how `remove` is likely to be implemented.

Solution:

ArrayLists: We'll assume the `remove` method shifts all elements after the removed one (very similarly to how `removeFront` does. In this case, it takes $\mathcal{O}(n-i)$ time to `remove` the element at index i. Our for-loop considers all possible values of i (from 0 to $n-1$), so in the **worst case** where we remove all elements, we get a running time like:

$$\sum_{i=0}^{n-1} n-i = \sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i = n(n-1) - \frac{n(n-1)}{2} = \frac{n(n-1)}{2}$$

which is $\mathcal{O}\left(n^2\right)$. See the Math Review on the webpage to review the summation rules we used.

Linked Lists: We'll assume the `remove` method starts at the head, traverses to element i, and then removes it by rearranging pointers. The analysis is similar to ArrayLists – the while loop has n iterations, where i goes from 0 to $n-1$, leading to the same $\mathcal{O}\left(n^2\right)$ time.

(b) `stretch(k)`. For the purposes of this analysis, assume that the ArrayList will need to resize. In doing this problem treat both n and k as variables.

Solution:

ArrayList: `ensureCapacity` creates a new array of size nk, and then has a for-loop over n elements (each iteration doing constant work). In Java, creating a new array of size s takes $\mathcal{O}(s)$ time, so the time for this method is $\mathcal{O}(nk+n)$ which simplifies to $\mathcal{O}(nk)$ (since $k \geq 1$). Stretching has nested for-loops, the first with n iterations, the second with k. The inner-most loop does constant work per iteration. Because the loops are nested, there will be nk iterations. This leads to a running time of $\mathcal{O}(nk)$ for the loops. The two parts of the code run in $\mathcal{O}(nk)$ time, which gives a final time of $\mathcal{O}(nk)$.

Linked List: The while-loop runs for n iterations, with the inner for-loop running for k iterations each time. So the inner-most operations are executed nk times. Since all of those operations are constant time, the final running time is $\mathcal{O}(nk)$.

3. Selecting ADTs and data structures

For each of the following scenarios, choose:

(a) An ADT: `Stack` or `Queue`

(b) A data structure: `array list` or `linked list with front` or `linked list with front and back`

Justify your choice.

(a) You're designing a tool that checks code to verify all opening brackets, braces, parenthesis, etc... have closing counterparts.

Solution:

We'd use the `Stack` ADT, because we want to match the *most recent* bracket we've seen first.

Since `Stacks` push and pop on the same end, there is no reason to use an implementation with two pointers. (We don't need access to the "back" ever.)

Asymptotically (i.e. in big-$\mathcal{O}$ terms), there is no difference between the `LinkedList` with a `front` pointer and the `Array` implementation. In practice, the `Array` implementation is almost certain to be faster due to how computer caches work. Later in the quarter we will use the term *spatial locality* to explain this behavior.

(b) Disneyland has hired you to find a way to improve the processing efficiency of their long lines at attractions. There is no way to forecast how long the lines will be.

Solution:

We'd use the `Queue` ADT here, because we're dealing with…a line.

The important thing to note here is that if we try to use the implementation of a `LinkedList` with *only a* `front` *pointer*, either *add* or *next* will be very slow. That is clearly not a good choice.

Arguably, the `LinkedList` implementation with both pointers is better than the array implementation because we will never have to resize it.

(c) A sandwich shop wants to serve customers in the order that they arrived, but also wants to look ahead to know what people have ordered and how many times to maximize efficiency in the kitchen.

Solution:

This is still clearly the `Queue` ADT, but it's unclear that any of these implementations are a good choice!

One of the cool things about data structures is that if only one isn't good enough, you can use *two*. If we only care about the "normal queue features", then we would probably use the `LinkedList` implementation with one pointer. However, we can **ALSO** simultaneously use a *Map* to store the "number of times a food item appears in the queue".

4. Challenge Problems – Adapting ADTs and data structures

Choose appropriate ADTs, data structures, and algorithms to solve the following problems. You may use any ADT and data structure you can think of, including ones covered in CSE 143. Feel free to be creative!

For your reference, between CSE 143 and 373 we've covered the following ADTs: Lists, Stacks, Queues, Sets, Maps, PriorityQueues. We've also discussed the following data structures: array lists, linked lists, trees, hash maps.

(a) We want to call all the phone numbers with a particular area code in someone's phone book.

Describe how you would implement this. What is the time complexity of your solution? 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 kind of data structure called a `Trie`. A trie is a kind of data structure based around the concept of a *tree*. However, unlike standard binary trees, tries have:

- A variable number of children (instead of just two)
- Have "labels" associated with each branch of the tree

Usually, we store the children of each node in a Map: the keys are the labels, the values are references to children node.

What we can do here is to make each node store exactly 10 children, labeling each branch from 0 to 9. We then take the first digit in the phone number and pick the branch with that label. From that node, we take the second digit and pick that branch.

If there are any nodes missing during this process, we create them as we go.

We repeat, using the entire phone number as the "route" to traverse the trie. Once we reach the end of the phone number, we stop, and set that node's data field to `true` to indicate that "route" corresponds to a valid phone number.

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 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) Long long ago, before smartphones were a thing, people who wanted to enter text using phones needed some way of entering arbitrary text using just 9 keys (the digits 1 through 9).

One such system is called "Text on nine keys" (T9). It associates 3 or 4 letters per each digit and lets you type words using just a single keypress per letter. To do this, it takes the sequence of digits entered and looks up all words corresponding to that sequence of keypresses within a fast-access dictionary of words and orders them by frequency of use.

For example, if the user types in '2665', the output could be the words `[book, cook, cool]`. Describe how you would implement a T9 dictionary for a mobile phone.

(For reference, the number '2' is associated with the letters 'abc', the number '3' is associated with 'def', etc... The number '9' is associated with 'wxyz'. The numbers '1' and '0' are used for other purposes.)

Describe how you would implement this. What is the time complexity of your solution? The space complexity?

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.

(c) One refinement we could make to our T9 system is to train it so it "gains familiarity" with the words and phrases the current user likes to commonly use. So, if a particular user uses the word "cool" more frequently than the word "book", eventually the T9 system, given the input '2665', will learn to return `[cool, book, cook]` instead of `[book, cook, cool]`.

Describe how you would implement this. What is the time complexity of your solution? The space complexity?

Solution:

One possible way of doing this would be to store, at each node, how frequently each word was accessed and use that information to sort the list of results.

To help provide reasonable results, we could provide "pre-trained" weights and adjust them over time as users use their phones more and more frequently.

The main issue we run into this solution is that eventually, our word counts will overflow (if we increment the counts very frequently). If our word pattern habits start changing, the T9 system will also be relatively slow to update, especially if some counts are very high and some are very low.

One way we could work around this problem is to keep track of the counts for just the past n messages or n days. This would solve the overflow and "slow update" problem – the challenge now would be to efficiently keep track of and manage this constantly changing data structure.