# Section 2. ADTs, Big-O, and Intro to Recursion

*Section materials curated by Neel Kishnani, drawing upon materials from previous quarters.*

This week's section exercises explore ADT and Big-O problems. By the end of this week, you'll be well-versed in all kinds of ADTs, and you'll be able to think critically about program runtime. Fun times!

Remember that every week we will also be releasing a Qt Creator project containing starter code and testing infrastructure for that week's section problems. When a problem name is followed by the name of a `.cpp` file, that means you can practice writing the code for that problem in the named file of the Qt Creator project. Here is the zip of the section starter code:

Starter code

## 1) Friend List (`friendlist.cpp`)

*Topic: Maps*

Write a function named **`friendList`** that takes in a file name, reads friend relationships from the file, and writes them to a **`Map`**. **`friendList`** should return the populated **`Map`**. Friendships are bi-directional, so if Abby is friends with Barney, Barney is friends with Abby. The file contains one friend relationship per line, with names separated by a single space. You do not have to worry about malformed entries.

If an input file named **`buddies.txt`** looked like this:

```
Barney Abby
Abby Clyde
```

Then the call of **`friendList("buddies.txt")`** should return a resulting **`map`** that looks like this:

**`{"Abby":{"Barney", "Clyde"}, "Barney":{"Abby"}, "Clyde":{"Abby"}}`**

Here is the function prototype you should implement:

**`Map<string, Vector<string> > friendList(String filename)`**

Solution

```
Map<string, Vector<string> > friendList(string filename) {
    ifstream in;
    Vector<string> lines;

    if (openFile(in, filepath)) {
        readEntireFile(in, lines);
    }

    Map<string, Vector<string> > friends;
    for (string line: lines) {
        Vector<string> people = stringSplit(line, " ");
        string s1 = people[0];
        string s2 = people[1];
        friends[s1] += s2;
        friends[s2] += s1;
    }
    return friends;
}
```

## 2) Twice (`twice.cpp`)

*Topic: Sets*

Write a function named **`twice`** that takes a vector of integers and returns a set containing all the numbers in the vector that appear exactly twice.

Example: passing `{1, 3, 1, 4, 3, 7, -2, 0, 7, -2, -2, 1}` returns `{3, 7}`.

Bonus: do the same thing, but you are not allowed to declare any kind of data structure other than sets.

Solution

```
// solution
Set<int> twice(Vector<int>& v) {
    Map<int, int> counts;
    for (int i : v) {
        counts[i]++;
    }
    Set<int> twice;
    for (int i : counts) {
        if (counts[i] == 2) {
            twice += i;
        }
    }
    return twice;
}

// bonus
Set<int> twice(Vector<int>& v) {
    Set<int> once;
    Set<int> twice;
    Set<int> more;
    for (int i : v) {
        if (once.contains(i)) {
            once.remove(i);
            twice.add(i);
        } else if (twice.contains(i)) {
            twice.remove(i);
            more.add(i);
        } else if (!more.contains(i)) {
            once.add(i);
        }
    }
    return twice;
}
```

## 3) Reversing a Map (`reverse.cpp`)

*Topic: Nested data structures*

Write a function

**`Map<int, Set<string>> reverseMap(Map<string, int>& map)`**

that, given a **`Map<string, int>`** that associates string values with integers, produces a **`Map<int, Set<string>>`** that's essentially the reverse mapping, associating each integer value with the set of strings that map to it. (This is an old job interview question from 2010.)

Solution

Here's one possible implementation.

```
Map<int, Set<string>> reverseMap(Map<string, int>& map) {
    Map<int, Set<string>> result;
    for (string oldKey : map) {
        if (!result.containsKey(map[oldKey])) {
            result[map[oldKey]] = {};
        }
        result[map[oldKey]].add(oldKey);
    }
    return result;
}
```

## 4) Oh No, Big-O, Too Slow

*Topics: Big-O, code analysis*

Give a tight bound of the nearest runtime complexity class for each of the following code fragments in Big-Oh notation, in terms of the variable N.

Code Snippet A

```
int sum = 0;
for (int i = 1; i <= N + 2; i++) {
    sum++;
}
for (int j = 1; j <= N * 2; j++) {
    sum++;
}
cout << sum << endl;
```

Code Snippet B

```
int sum = 0;
for (int i = 1; i <= N - 5; i++) {
    for (int j = 1; j <= N - 5; j += 2) {
        sum++;
    }
}
cout << sum << endl;
```

Code Snippet C

```
int sum = 0;
for (int i = 0; i < 1000000; i++) {
    for (int j = 1; j <= i; j++) {
        sum += N;
    }
    for (int j = 1; j <= i; j++) {
        sum += N;
    }
    for (int j = 1; j <= i; j++) {
        sum += N;
    }
}
cout << sum << endl;
```

Solution

```
Code Snippet A has a runtime complexity of O(N).
Code Snippet B has a runtime complexity of O(N^2).
Code Snippet C has a runtime complexity of O(1).
```

## 5) More Big O

Below are five functions. Determine the big-O runtime of each of those pieces of code, in terms of the variable **n**.

```
void function1(int n) {
    for (int i = 0; i < n; i++) {
        cout << '*' << endl;
    }
}

void function2(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << '*' << endl;
        }
    }
}

void function3(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            cout << '*' << endl;
        }
    }
}

void function4(int n) {
    for (int i = 1; i <= n; i *= 2) {
        cout << '*' << endl;
    }
}
```

Finally, what is the big-O runtime of this function in terms of **n**, the number of elements in **v**?

```
int squigglebah(Vector<int>& v) {
    int result = 0;
    for (int i = 0; i < v.size(); i++) {
        Vector<int> values = v.subList(0, i);
        for (int j = 0; j < values.size(); j++) {
            result += values[j];
        }
    }
    return result;
}
```

Solution

1. The runtime of this code is **O(n)**: We print out a single star, which takes time **O(1)**, a total of **n** times.
2. The runtime of this code is **O(n^2)**. The inner loop does **O(n)** work, and it runs **O(n)** times for a net total of **O(n^2)** work.
3. This one also does **O(n^2)** work. To see this, note that the first iteration of the inner loop runs for **n - 1** iterations, the next for **n - 2** iterations, then **n - 3** iterations,

etc. Adding all this work up across all iterations gives `(n - 1) + (n - 2) + … + 3 + 2 + 1 + 0 = O(n^2)`.

4. This one runs in time `O(log n)`. To see why this is, note that after **k** iterations of the inner loop, the value of **i** is equal to **2^k**. The loop stops running when **2^k** exceeds **n**. If we set **2^k = n**, we see that the loop must stop running after **k = log₂ n** steps.

   Another intuition for this one: the value of i doubles on each iteration, and you can only double `O(log n)` times before you overtake the value **n**.

For the final function, Let's follow the useful maxim of "when in doubt, work inside out!"" The innermost for loop (the one counting with **j**) does work proportional to the size of the values list, and the values list has size equal to **i** on each iteration. Therefore, we can simplify this code down to something that looks like this:

```
int squigglebah(Vector<int>& v) {
    int result = 0;
    for (int i = 0; i < v.size(); i++) {
        Vector<int> values = v.subList(0, i);
        do O(i) work;
    }
    return result;
}
```

Now, how much work does it take to create the values vector? We're copying a total of **i** elements from **v**, and so the work done will be proportional to **i**. That gives us this:

```
int squigglebah(Vector<int>& v) {
    int result = 0;
    for (int i = 0; i < v.size(); i++) {
        do O(i) work;
        do O(i) work;
    }
    return result;
}
```

Remember that doing `O(i)` work twice takes time `O(i)`, since big-O ignores constant factors. We're now left with this:

```
int squigglebah(Vector<int>& v) {
    int result = 0;
    for (int i = 0; i < v.size(); i++) {
        do O(i) work;
    }
    return result;
}
```

This is the same pattern as function2 in the previous problem, and it works out to `O(n^2)` total time.

## 6) Recursion Mystery Part 1

*Topics: recursive function calls, return value tracing*

Code Snippet A

```
int recursionMystery(int x, int y) {
    if (x < y) {
        return x;
    } else {
        return recursionMystery(x - y, y);
    }
}
```

For each call to the above recursive function, indicate what value is returned by the function call.

```
Call                              Return value

recursionMystery(6, 13);          ______________
recursionMystery(14, 10);         ______________
recursionMystery(37, 12);         ______________
```

Solution

```
6

4

1
```

## 7) Recursion Mystery Part 2

*Topics: recursive function calls, output tracing*

```
void recursionMystery2(int x, int y) {
    if (y == 1) {
        cout << x;
    } else {
        cout << (x * y) << ", ";
        recursionMystery2(x, y - 1);
        cout << ", " << (x * y);
    }
}
```

For each call to the above recursive function, write the output that would be produced, as it would appear on the console.

```
Call                              Output

recursionMystery2(4, 1);    ___________________________________
recursionMystery2(4, 2);    ___________________________________
recursionMystery2(8, 2);    ___________________________________
recursionMystery2(4, 3);    ___________________________________
recursionMystery2(3, 4);    ___________________________________
```

Solution

```
4

8, 4, 8

16, 8, 16

12, 8, 4, 8, 12

12, 9, 6, 3, 6, 9, 12
```

## 8) Recursion Tracing

*Topics: Recursion, strings, recursion tracing*

Below is a recursive function to reverse a string.

```
string reverseOf(string s) {
    if (s.empty()) {
        return "";
    } else {
        return reverseOf(s.substr(1)) + s[0];
    }
}
```

Trace through the execution of **`reverseOf("stop")`** along the lines of what we did in lecture, showing recursive call information for each call that's made and how the final value gets computed.

Solution

Our initial call to **`reverseOf("stop")`** fires off a call to **`reverseOf("top")`**. This call fires off a call to **`reverseOf("op")`**. This in turn calls **`reverseOf("p")`**. This in turn calls **`reverseOf("")`**. This triggers the base case and returns the empty string. (Notice that the reverse of the empty string "" is indeed the empty string ""). We now append **p** to return **"p"**. We now append **o** to return **"po"**. We append **t** to return **"pot"**. And finally we append s to return **"pots"** back to whoever called us. Yay!

## 9) Sum of Squares (**squares.cpp**)

*Topics: Recursion*

Write a recursive function named **sumOfSquares** that takes an int **n** and returns the sum of squares from **1** to **n**. For example, **sumOfSquares(3)**should return 1^2 + 2^2 + 3^2 = 14. If **n** is negative, you should report an error to that effect.

Solution

```
int sumOfSquares(int n) {
    if (n < 0) {
        error("Value of provided n was negative");
    } else if (n == 0) {
        return 0;
    } else {
        return n * n + sumOfSquares(n-1);
    }
}
```