Section 3. Recursion

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

This week's section exercises continue our exploration of recursion to tackle even more challenging and interesting problems. In particular, many of this week's section problems center around recursive backtracking, a very powerful and versatile problem-solving technique.

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) 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
```

2) 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!

---

3) Random Shuffling (**`shuffle.cpp`**)

How might the computer shuffle a deck of cards? This problem is a bit more complex than it might seem, and while it's easy to come up with algorithms that randomize the order of the cards, only a few algorithms will do so in a way that ends up generating a uniformly-random reordering of the cards.

One simple algorithm for shuffling a deck of cards is based on the following idea:

- Choose a random card from the deck and remove it.
- Shuffle the rest of the deck.
- Place the randomly-chosen card on top of the deck. Assuming that we choose the card that we put on top uniformly at random from the deck, this ends up producing a random shuffle of the deck.

Write a function

```
string randomShuffle(string input)
```

that accepts as input a string, then returns a random permutation of the elements of the string using the above algorithm. Your algorithm should be recursive and not use any loops (**`for`**, **`while`**, etc.).

The header file **`"random.h"`** includes a function

```
int randomInteger(int low, int high);
```

that takes as input a pair of integers low and high, then returns an integer greater than or equal to low and less than or equal to high. Feel free to use that here.

Interesting note: This shuffling algorithm is a variant of the Fisher-Yates Shuffle. For more information on why it works correctly, take CS109!

Solution

Here is one possible solution:

```
string randomShuffle(string input) {
    /* Base case: There is only one possible permutation of a string
    * with no characters in it.
    */
    if (input.empty()) {
        return input;
    } else {
        /* Choose a random index in the string. */
        int i = randomInteger(0, input.length() - 1);
        /* Pull that character to the front, then permute the rest of
        * the string.
        */
        return input[i] + randomShuffle(input.substr(0, i) +
input.substr(i + 1));
    }
}
```

This function is based on the recursive observation that there is only one possible random shuffle of the empty string (namely, itself), and then using the algorithm specified in the handout for the recursive step.

Here is another possible solution (using a modified function header) which shows how to implement this function using references and no return value. Shoutout to one of our awesome SLs, Rachel Gardner, for this alternate solution!

```
void randomShuffle(string &input) {
    if (input == "") return;
    int rand = randomInteger(0, input.length() - 1);
    char chosen = input[rand];
    input.erase(rand, 1);
    randomShuffle(input);
    input = chosen + input;
}
```

---

4) Zig Zag (`zigzag.cpp`)

Topics: Recursion, printing output to console

Write a recursive function named **`zigzag`** that returns a string of **n** characters as follows. The middle character (or middle two characters if **n** is even) is an asterisk (*). All characters before the asterisks are `'<'`. All characters after are `'>'`. Report an error if **n** is not positive.

Call	Output
zigzag(1)	*
zigzag(4)	<**>
zigzag(9)	<<<<*>>>>

Solution

```
string zigzag(int n) {
    if (n < 1) {
        error("The value of n was negative");
    } else if (n == 1) {
        return "*";
    } else if (n == 2) {
        return "**";
    } else {
        return "<" + zigzag(n-2) + ">";
    }
}
```

5) Double Stack (**double.cpp**)

Topics: Recursion, Stacks

Write a recursive function named **doubleStack** that takes a reference to a stack of ints and replaces each integer with two copies of that integer. For example, if **s** stores **{1, 2, 3}**, then **doubleStack(s)** changes it to **{1, 1, 2, 2, 3, 3}**.

Solution

```
void doubleStack(Stack<int>& s)
{
    if (!s.isEmpty()) {
        int n = s.pop();
        doubleStack(s);
        s.push(n);
        s.push(n);
    }
}
```

6) String Subsequences (**subsequence.cpp**)

Topics: Recursion, verifying properties

Write a recursive function named **isSubsequence** that takes two strings and returns **true** if the second string is a subsequence of the first string. A string is a subsequence of another if it contains the same letters in the same order, but not necessarily consecutively. You can assume both strings are already lower-cased.

```
Call                                  Output
isSubsequence("computer", "core")     false
isSubsequence("computer", "cope")     true
isSubsequence("computer", "computer") true
```

Solution

```
bool isSubsequence(string big, string small)
{
    if (small.empty()) {
        return true;
    } else if (big.empty()) {
        return false;
    } else {
        if (big[0] == small[0]) {
            return isSubsequence(big.substr(1), small.substr(1));
        } else {
            return isSubsequence(big.substr(1), small);
        }
    }
}
```

7) Recursion and Time Complexity and Exponents, (Big-)Oh My

Topics: recursion, time complexity, big O, algorithm comparison

Below is a simple function that computes the value of m^n when n is a nonnegative integer:

```
int raiseToPower(int m, int n) {
    int result = 1;
    for (int i = 0; i < n; i++) {
        result *= m;
    }
    return result;
}
```

1) What is the big-O complexity of the above function, written in terms of m and n? You can assume that it takes time **O(1)** to multiply two numbers.

2) If it takes 1 microsecond (µs) to compute **raiseToPower(100, 100)**, approximately how long will it take to compute **raiseToPower(200, 10000)**?

Below is a recursive function that computes the value of **m^n** when **n** is a nonnegative integer:

```
int raiseToPower(int m, int n) {
    if (n == 0) return 1;
    return m * raiseToPower(m, n - 1);
}
```

3) What is the big-O complexity of the above function, written in terms of **m** and **n**? You can assume that it takes time **O(1)** to multiply two numbers.

4) If it takes 1µs to compute **raiseToPower(100, 100)**, approximately how long will it take to compute **raiseToPower(200, 10000)**?

Here's an alternative recursive function for computing **m^n** that uses a technique called exponentiation by squaring. The idea is to modify the recursive step as follows:

- If **n** is an even number, then we can write **n** as **n = 2k**. Then **m^n = m^(2k) = (m^k)^2**.
- If **n** is an odd number, then we can write **n** as **n = 2k + 1**. Then **m^n = m^(2k+1) = m * m^(2k) = m * (m^k)^2**.

Based on this observation, we can write this recursive function:

```
int raiseToPower(int m, int n) {
    if (n == 0) {
        return 1;
    } else if (n % 2 == 0) {
        int halfPower = raiseToPower(m, n / 2);
        return halfPower * halfPower;
    } else {
        int halfPower = raiseToPower(m, n / 2);
        return m * halfPower * halfPower;
    }
}
```

5) What is the big-O complexity of the above function, written in terms of **m** and **n**? You can assume that it takes time **O(1)** to multiply two numbers.

6) If it takes 1µs to compute **raiseToPower(100, 100)**, approximately how long will it take to compute **raiseToPower(200, 10000)**?

Solution

1. This function runs in time **O(n)**. It runs the loop **n** times, at each step doing **O(1)** work. There is no dependence on **m** in the runtime.

2. We know that this code runs in time **O(n)**, so it scales roughly linearly with the size of **n**. Therefore, if it took 1µs to compute a value when **n = 100**, it will take roughly 100 times longer when we plug in **n = 10000**. As a result, we'd expect this code would take about 100µs to complete.

3. If we trace through the recursion, we'll see that we make a total of `n` recursive calls, each of which is only doing `O(1)` work. Adding up all the work done by these recursive calls gives us a total of `O(n)` work, as before.

4. As before, this should take about 100µs.

5. Notice that each recursive call does `O(1)` work (there are no loops anywhere here), then calls itself on a problem that's half as big as the original one. This means that only `O(log n)` recursive calls will happen (remember that repeatedly dividing by two is the hallmark of a logarithm), so the total work done here is `O(log n)`.

6. We know that the runtime when n = 100 is roughly 1µs. Notice that 100^2 = 10,000, so we're essentially asking for the runtime of this function when we square the size of the input. Also notice that via properties of logarithms that `log n^2 = 2 log n`. Therefore, since we know the runtime grows roughly logarithmically and we've squared the value of `n`, this should take about twice as long as before, roughly 2µs.