Section 4. Recursive Backtracking

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

<=”” style="box-sizing: border-box; position: relative; width: 733.333px; padding-right: 15px; padding-left: 15px; flex: 0 0 83.3333%; max-width: 83.3333%;">
This week’s section exercises explore the ins and outs of content from weeks 4 and 5, focusing on delving more deeply into recursive backtracking and its applications, plus a bit of Big-O!

Each 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) Weights and Balances

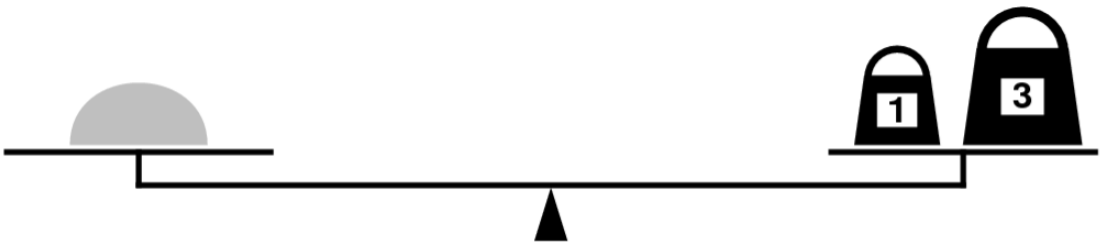
Thank you to Eric Roberts for this problem

Topics: Recursion, Combinations, Backtracking

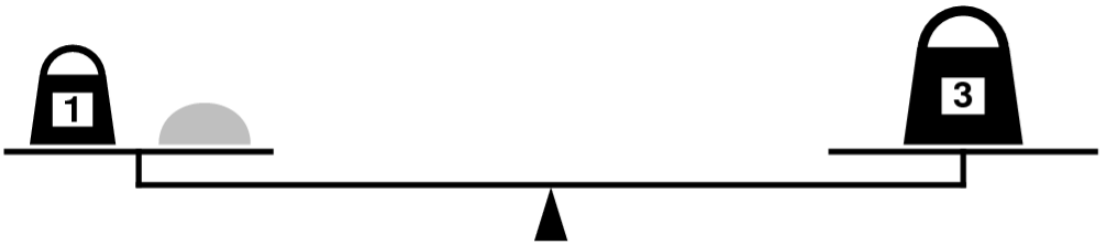
> I am the only child of parents who weighed, measured, and priced everything; for whom what could not be weighed, measured, and priced had no existence.
>
> **—Charles Dickens, Little Dorrit, 1857**

In Dickens’s time, merchants measured many commodities using weights and a two-pan balance – a practice that continues in many parts of the world today. If you are using a limited set of weights, however, you can only measure certain quantities accurately.

For example, suppose that you have only two weights: a 1-ounce weight and a 3-ounce weight. With these you can easily measure out 4 ounces, as shown:



It's more interesting to discover that you can also measure out 2 ounces by shifting the 1-ounce weight to the other side, as follows:



Write a recursive function,

```
bool isMeasurable(int target, const Vector<int>& weights);
```

that determines whether it is possible to measure out the desired target amount with a given set of weights, which is stored in the vector weights.

As an example, the function call

```
isMeasurable(2, { 1, 3 })
```

should return **true** because it is possible to measure out two ounces using the sample weight set as illustrated in the preceding diagram. On the other hand, calling

```
isMeasurable(5, { 1, 3 })
```

should return **false** because it is impossible to use the 1- and 3-ounce weights to add up to 5 ounces. However, the call

```
isMeasurable(6, { 1, 3, 7 })
```

should return **true**: you can measure the six-ounce weight by placing it and the one-ounce weight on one side of the scale and the seven-ounce weight on the other.

Here's a function question to ponder: let's say that you get to choose *n* weights. Which ones would you pick to give yourself the best range of weights that you'd be capable of measuring?

Solution

---

2) CHeMoWIZrDy

Topics: Recursion, Backtracking, Sets

Some words in the English language can be spelled out using just element symbols from the Periodic Table. For example, "began" can be spelled out as BeGaN (beryllium, gallium, nitrogen), and "feline" can be spelled out as FeLiNe (iron, lithium, neon). Not all words have this property, though; the word "interesting" cannot be made out of element letters, nor can the word "chemistry" (though, interestingly, the word "physics" can be made as PHYSICS (phosphorous, hydrogen, yttrium, sulfur, iodine, carbon, sulfur).

Write a function

```
bool isElementSpellable(const string& text, const Set<string>&
symbols);
```

that accepts as input a string and a **Set<string>** containing all element symbols (stored with the proper capitalization), then returns whether that string can be written using only element symbols. Once you've gotten that function working, modify the function so that it has this signature:

```
bool isElementSpellable(const string& text, const Set<string>&
symbols, string& result);
```

This function should behave as before, except that if it turns out that it is possible to spell the input string just using element symbols, the variable result is overwritten with one possible way of doing so.

Here's a final variation to consider, which is much more challenging than the previous one but would be a great way to practice your recursive problem-solving. (As in, do this problem only if you have a good amount of time; it's challenging!) As mentioned above, not all strings can be written using element symbols. The title of this problem is supposed to be "Chemowizardry," but that just isn't quite spellable using element symbols, so we compromised on "Chemowizrdy," cutting out two letters. Write a function:

```
string closestApproximationTo(const string& text, const Set<string>&
symbols);
```

that takes as input a string, then returns the longest subsequence of the input string that can be spelled out using element symbols, capitalized appropriately. For example, given the input "Chemowizardry," the function should return "CheMoWIZrDy."

Solution

---

3) Barnstorming Brainstorming

Topics: Recursion, Backtracking, Permutations

You're campaigning for office and it's down to the very last week before the election. A last-minute tour of swing states/districts/areas can have a huge impact on your final vote totals, so you decide to see whether it's possible to visit all of them in a short amount of time. As as simplifying assumption for this problem, let's assume that each of your campaign stops is represented as a **`GPoint`**, which represents a point in space. You can access the x and y coordinates of a **`GPoint`**, which are **`double`**s, by using the syntax **`pt.x()`** and **`pt.y()`**. Further, let's assume the travel time between two points is equal to their Euclidean (straight line) distance. Write a function

```
bool canVisitAllSites(const Vector<GPoint>& sites, double
travelTimeAvailable);
```

that takes as input a list of all the sites you'd like to visit and an amount of free time available to you and returns whether it's possible to visit all those sites in the allotted time (assume you've already factored in the cost of speaking at each site and that you're just concerned about the travel time.) You can start wherever you'd like. Once you've gotten that working, update your function so that it has this signature:

```
bool canVisitAllSites(const Vector<GPoint>& sites, double
travelTimeAvailable, Vector<GPoint>& result);
```

This function works as before, except that if it's possible to visit all the sites, it fills in the parameter result with the list of the cities in the order you should visit them. Then think about whether memoization would be appropriate here and, if so, update your code to use it.

Solution

---

4) Pattern Matching

Topics: Recursion, Backtracking, Strings

One of the concepts you'll probably run into if you continue on as a programmer (or take CS103!) is the ***regular expression***, a way of representing a pattern to match as a string. Regular expressions make it easy to write code to search for complicated patterns in text and break them apart, and a lot of our starter files include them to parse test case files. This problem addresses a simplified version of regular expression matching.

Let's imagine that you have a ***pattern string*** that consists of letters, plus the special characters star (*), dot (.), and question-mark (?). The star symbol means "any string of zero or more characters," the dot means "any individual character," and the question-mark means "nothing, or any character." Here are some examples:

- The pattern `a*` means "match the letter a, then match any number of characters," so it essentially means "match anything beginning with an a." As a result, `a*` would match apple, apply, and apoplexy, but not Amicus (it's case-sensitive), banana (contains an a, but doesn't start with one), or moose (which isn't even close).
- The pattern `*a*` means "match any number of characters, then an a, then any number of characters," so it essentially means "match any string containing an a." Therefore, the pattern `*a*` would match ramadan, diwali, shavuot, and advent but not the strings eid, sukkot, lent, or holi.
- The pattern `th...` means "match th, then math any three characters," so it matches five-letter words starting with th. For example, this would match there and third, but not the or other.
- The pattern `colo?r` means "match colo, then optionally match another character, then match r," so it would match color and colour (as well as coloxr), but not colors or colours.

Your task is to write a function

```
bool matches(const string& text, const string& pattern);
```

that takes as input a string and a pattern, then returns whether that string matches the pattern.

Once you're done, ask yourself whether memoization would make this function any faster, and, if so, update this function to use memoization.

Solution

---

5) Advocating for Exponents

Topics: Big-O (and a lil' recursion)

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. ii. If it takes 1µs 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);
}
```

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

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;
    }
}
```

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

Solution

---

6) Revisiting Reversals

Topics: Recursion, Big-O

In one of our earlier lectures, we wrote this function to reverse a string:

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

Let n be the length of the input string. What is the big-O complexity of the above function? You may find the following facts useful:

- The runtime of the string::substr function is O(k), where k is the length of the string returned.
- The runtime of concatenating two strings is O(k), where k is the length of the string returned.
- The runtime of comparing two strings is O(k), where k is the length of the shorter of the two strings being compared.
- The runtime of making a copy of a string is O(k), where k is the length of the string.
- The runtime of choosing a single character out of a string is O(1).

Now, let's suppose you change that function so that it takes its argument by const reference, as shown here:

```
string reverseOf(const string& str) {
    if (str == "") {
        return str;
```

```
    } else {
        return reverseOf(str.substr(1)) + str[0];
    }
}
```

Now, what's the big-O time complexity of this function? Do you think it would be faster than before?

Here's a completely different way of reversing a string:

```
string reverseOf(const string& str) {
    if (str.length() <= 1) {
        return str;
    } else {
        return reverseOf(str.substr(str.length() / 2)) +
                    reverseOf(str.substr(0, str.length() / 2));
    }
}
```

Talk with your fellow sectionees about how this function works. What does it do? Why is it correct? Then, once you've got that sorted out, think about how efficient it is. What's the big-O time complexity of this function, assuming that string::length runs in time O(1)?