

Section Handout 4

Problem One: Weights and Balances (Courtesy of Eric Roberts)

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?

Problem Two: CHeMoWIZrDy

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.”

Problem Three: Barnstorming Brainstorming

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 a simplifying assumption for this problem, let’s assume that each of your campaign stops is represented as a `GPoint` and that 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.

Problem Four: Pattern Matching

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 match 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.

Problem Five: Cell Towers Redux

In one of our earlier lectures, we considered the following problem. You're given a list of cities along a highway and the populations of each of those cities. You'd like to build cell towers to provide coverage to as many of those cities as possible, but there's a catch: if you build a cell tower in one city, you can't build a cell tower in any city adjacent to it. Given this restriction, what's the greatest number of people you can provide coverage to? Here's the code that we wrote from Way Back When:

```
/* Given a list of cities, returns the maximum number of people we can provide
 * cell service to given that no two adjacent cities can have coverage.
 */
int bestCoverageFor(const Vector<int>& cities) {
    /* Base case: If there are no cities, we can cover a whopping zero people. */
    if (cities.isEmpty()) {
        return 0;
    } else {
        /* Option 1: Exclude the first city. In that case, we want to
         * maximize the coverage to everyone else.
         */
        int without = bestCoverageFor(cities.subList(1, cities.size() - 1));

        /* Option 2: Include the first city. Then we can't cover the second, and
         * we want to maximize coverage to all the remaining cities.
         */
        int with = cities[0]; // Coverage from the first city.
        if (cities.size() >= 2) {
            with += bestCoverageFor(cities.subList(2, cities.size() - 2));
        }

        /* One of those two options is better. Return whichever one that is! */
        return max(with, without);
    }
}
```

To start things off, using your knowledge of exhaustive recursion and recursive backtracking, discuss in a group how this function works. Does it make a bit more sense now?

Right now, this function makes a *lot* of copies of vectors due to the use of the `subList` function. However, all of those vectors are just suffixes of the original vector given as input. As a result, it's possible to rewrite this function to have this signature:

```
int bestCoverageFor(const Vector<int>& populations, int index);
```

This function should return the best coverage you could provide to people in the given towns *at or after the specified index*. Rewrite this function so that it has this signature and never makes any calls to `subList`.

Next, discuss whether this function would be amenable to memoization. Would that help at all? If so, rewrite it to use memoization. If not, explain why not.

Finally, talk about the efficiency of the solution you ended up with from a big-O perspective. For reference, the code presented here runs in time $O(n \cdot \varphi^n)$, where φ is the *golden ratio*, roughly 1.61.

Problem Six: Advocating for Exponents

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;
}
```

- i. 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.
- ii. If it takes $1\mu\text{s}$ 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);
}
```

- iii. 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.
- iv. If it takes $1\mu\text{s}$ 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 as $n = 2k$. Then $m^n = m^{2k} = (m^k)^2$.
- If n is an odd number, then we can write $n = 2k + 1$. Then $m^n = m^{2k+1} = m \cdot (m^k)^2$.

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;
    }
}
```

- v. 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.
- vi. If it takes $1\mu\text{s}$ to compute `raiseToPower(100, 100)`, approximately how long will it take to compute `raiseToPower(200, 10000)`?

Problem Seven: Revisiting Reversals

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:

1. The runtime of the `string::substr` function is $O(k)$, where k is the length of the string returned.
2. The runtime of concatenating two strings is $O(k)$, where k is the length of the string returned.
3. The runtime of comparing two strings is $O(k)$, where k is the length of the shorter of the two strings being compared.
4. The runtime of making a copy of a string is $O(k)$, where k is the length of the string.
5. 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 $O(1)$?