

## Section Solutions 4

---

### Problem One: CHeMoWIZrDy

Here's one possible implementation of the `isElementSpellable` function:

```
/**
 * Given a word and an element symbol, returns whether the word starts with that
 * particular element symbol.
 *
 * @param word The word in question
 * @param symbol The symbol in question.
 * @return Whether the word starts with that element symbol.
 */
bool startsWithElement(const string& word, const string& symbol) {
    return startsWith(toLowerCase(word), toLowerCase(symbol));
}

/**
 * Given a word and a set containing all the element symbols in the Periodic
 * Table, returns whether it's possible to spell that word using just element
 * symbols.
 *
 * @param text The word
 * @param symbols The element symbols in the Periodic Table.
 * @return Whether that text can be spelled out.
 */
bool isElementSpellable(const string& word, const Set<string>& symbols) {
    /* Base case: the empty string can be spelled out by simply using no strings
     * from the list of symbols.
     */
    if (word == "") {
        return true;
    }
    /* Recursive case: try each element symbol to see whether any of them match
     * the first characters of the input string. We could alternatively rely on
     * the fact that all element symbols are between 1 and 3 characters long, but
     * just in case that changes we won't assume that here. :-))
     */
    else {
        for (string symbol: symbols) {
            if (startsWithSymbol(word, symbol) &&
                isElementSpellable(word.substr(symbol.length()), symbols)) {
                return true;
            }
        }
        /* If none of those options work, there is no way to spell this word using
         * element symbols.
         */
        return false;
    }
}
```

We can modify this code to report the way in which the string would match by making a slight modification to the recursive step to accumulate element symbols together as we unwind back up.

```
bool isElementSpellable(const string& word, const Set<string>& symbols,
                        string& result) {
    /* Base case: the empty string can be spelled out by simply using no strings
     * from the list of symbols.
     */
    if (word == "") {
        result = ""; // This is the proper way to spell this word.
        return true;
    }
    /* Recursive case: try each element symbol to see whether any of them match
     * the first characters of the input string. We could alternatively rely on
     * the fact that all element symbols are between 1 and 3 characters long, but
     * just in case that changes we won't assume that here. :-)
     */
    else {
        for (string symbol: symbols) {
            if (startsWithSymbol(word, symbol)) {
                /* See if we can spell what's left. */
                if (isElementSpellable(word.substr(symbol.length()),
                                        symbols, result)) {
                    /* Because we could, we know that result is now filled in with
                     * how to spell the rest of the word (that's what the function
                     * says it will do!). We just need to prepend the element
                     * symbol we used.
                     */
                    result = symbol + result;
                    return true;
                }
            }
        }
        /* If none of those options work, there is no way to spell this word using
         * element symbols.
         */
        return false;
    }
}
```

This final version of the code is a bit more nuanced. The idea is the following. Focus on the very first letter of the word. That letter will either (1) be deleted and not take part in the overall word, (2) be included purely by itself (for example, C for carbon or Y for Yttrium), or (3) be included as the first letter of a two-letter symbol (for example, C as part of Cs for cesium). We'll try all three options and see what we find as a result.

```
string closestApproximationTo(const string& word, const Set<string>& symbols) {
    /* Base case: the empty string can be spelled out by simply using no strings
     * from the list of symbols, and that's the best way to approximate it.
     */
    if (word == "") {
        return "";
    }

    /* One option that's always valid is to just drop the first letter and
     * maximize what's left.
     */
    string bestOption = closestApproximationTo(word.substr(1), symbols);

    /* First letter may be an element symbol. If so, one option is to use it by
     * itself and then maximize what's left.
     */
    string element = word.substr(0, 1);
    if (symbols.contains(toUpperCase(element))) {
        string justFirst = toUpperCase(element) +
            closestApproximationTo(word.substr(1), symbols);
        if (justFirst.length() > bestOption.length()) {
            bestOption = justFirst;
        }
    }

    /* Try pairing this with all follow-up letters. For each follow-up letter,
     * if we pair with that letter we have to delete all the letters between the
     * first letter and that letter from consideration, then maximize what
     * remains of what's left.
     */
    for (int i = 1; i < word.length(); i++) {
        string thisElement = element + toLowerCase(word.substr(i, 1));
        if (symbols.contains(thisElement)) {
            string theseTwo = thisElement +
                closestApproximationTo(word.substr(i+1), symbols);
            if (theseTwo.length() > bestOption.length()) {
                bestOption = theseTwo;
            }
        }
    }
    return bestOption;
}
```

The final question was about memoization – is it going to help us here? The answer is yes! Consider the string `chair`. We're going to end up recursively trying to process `hair` in two ways: first, by matching the `c` in `chair` (Carbon); second, by dropping off the first letter. Memoizing the result prevents us from doing a ton of unnecessary work, so it would be great to add that in here. Here's what that looks like:

```
string closestApproximationMemo(const string& word, const Set<string>& symbols,
                               Map<string, string>& memo) {
    if (word == "") {
        return "";
    }

    /* New base case: If we already know this word, we're done! */
    if (memo.containsKey(word)) return memo[word];

    /* Core logic here is essentially unmodified. */
    string bestOption = closestApproximationMemo(word.substr(1), symbols, memo);

    string element = word.substr(0, 1);
    if (symbols.contains(toUpperCase(element))) {
        string justFirst = toUpperCase(element) +
            closestApproximationMemo(word.substr(1), symbols,
                                    memo);
        if (justFirst.length() > bestOption.length()) {
            bestOption = justFirst;
        }
    }

    for (int i = 1; i < word.length(); i++) {
        string thisElement = element + toLowerCase(word.substr(i, 1));
        if (symbols.contains(thisElement)) {
            string theseTwo = thisElement +
                closestApproximationMemo(word.substr(i+1), symbols,
                                        memo);
            if (theseTwo.length() > bestOption.length()) {
                bestOption = theseTwo;
            }
        }
    }

    /* Write down the answer. */
    memo[word] = bestOption;
    return bestOption;
}

string closestApproximationTo(const string& word, const Set<string>& symbols) {
    Map<string, string> memo;
    return closestApproximationMemo(word, symbols, memo);
}
```

This can still be sped up by avoiding making copies of the string at each step and adapting the technique from the Cell Towers Redux problem. It's a great exercise to do this!

## Problem Two: Barnstorming Brainstorming

This problem essentially boils down to generating all permutations of the sites and seeing whether any of them fit in the specified timeframe. The intuition we'll use in writing up this solution is similar to the one we used to generate permutations in class – we'll look at all options for the next place to go, consider what would happen if we visited any of them, and see if any of those options lead to success.

One catch here is that in order to measure distances we need to remember where we just were, since we have to measure distances based on where we used to be. That in itself is somewhat interesting because the very first place we visit isn't preceded by anything, so we'll separate that from the rest of the recursion logic.

Here's what that looks like:

```
/**
 * Given a list of sites to visit and a total travel time, plus the location of
 * the last city visited, returns whether it's possible to visit all of those
 * locations in the specified amount of time.
 *
 * @param sites The list of sites left to visit.
 * @param timeAvailable How much time is left.
 * @param last The last place we visited.
 * @return Whether we can visit those sites starting at the given location.
 */
bool canVisitAllSitesRec(const Vector<GPoint>& sites, double timeAvailable,
                        const GPoint& last);

/**
 * Given a Vector, returns a new Vector formed by removing the element at the
 * specified index.
 *
 * @param sites The list of sites.
 * @param index The index in question.
 * @return That vector with that index removed.
 */
Vector<GPoint> removeAt(Vector<GPoint> sites, int index);

bool canVisitAllSites(const Vector<GPoint>& sites, double timeAvailable) {
    /* If there aren't any sites, we can always visit them all! */
    if (sites.isEmpty()) return true;

    /* Try all possible starting points and see if any of them work. */
    for (int i = 0; i < sites.size(); i++) {
        if (canVisitAllSitesRec(removeAt(sites, i), timeAvailable, sites[i])) {
            return true;
        }
    }
    return false;
}

/* continued on the next page */
```

```

/**
 * Returns the Euclidean distance between two points.
 *
 * @param one The first point.
 * @param two The second point.
 * @return The distance between them.
 */
double distanceBetween(const GPoint& one, const GPoint& two) {
    double dx = one.getX() - two.getX();
    double dy = one.getY() - two.getY();
    return sqrt(dx * dx + dy * dy);
}

bool canVisitAllSitesRec(const Vector<GPoint>& sites, double timeAvailable,
                        const GPoint& last) {
    /* Base case: If no sites remain, we're done! */
    if (sites.isEmpty()) {
        return true;
    }
    /* Recursive case: see where we go next. */
    else {
        for (int i = 0; i < sites.size(); i++) {
            /* See how long this is going to take. If it's too far, then we
             * can't go there next.
             *
             * We can actually be way more aggressive here due to the triangle
             * inequality: the fastest way to a point is to go straight there.
             * If we can't make it there from here in time, there's no alternate
             * route we could take that would be any better. The only reason we
             * didn't optimize the code this way was because in general you can't
             * make assumptions like this.
             */
            double distance = distanceBetween(last, sites[i]);
            if (distance <= timeAvailable &&
                canVisitAllSitesRec(removeAt(sites, i), timeAvailable - dist,
                                    sites[i])) {
                return true;
            }
        }
        /* Looks like no options worked. Oh well! */
        return false;
    }
}

```

To update this code to not just tell us whether there is a route, but to also say what the route is, we can update the function so that, when it finds a route that works, it adds in the city that we considered at the current level of the recursion. Here's what that looks like, with the helper functions and documentation removed:

```

bool canVisitAllSites(const Vector<GPoint>& sites, double timeAvailable,
                    Vector<GPoint>& result) {
    /* If there aren't any sites, we can always visit them all! */
    if (sites.isEmpty()) {
        result.clear(); // Best option is the empty list.
        return true;
    }

    /* Try all possible starting points and see if any of them work. */
    for (int i = 0; i < sites.size(); i++) {
        if (canVisitAllSitesRec(removeAt(sites, i), timeAvailable, sites[i],
                                   result) {
            /* Prepend the starting city. */
            result.insert(0, sites[i]);
            return true;
        }
    }
    return false;
}

bool canVisitAllSitesRec(const Vector<GPoint>& sites, double timeAvailable,
                        const GPoint& last, Vector<GPoint>& result) {
    /* Base case: If no sites remain, we're done! */
    if (sites.isEmpty()) {
        result.clear(); // Empty list is the correct visit order here.
        return true;
    }
    /* Recursive case: see where we go next. */
    else {
        for (int i = 0; i < sites.size(); i++) {
            double distance = distanceBetween(last, sites[i]);
            if (distance <= timeAvailable &&
                canVisitAllSitesRec(removeAt(sites, i), timeAvailable - dist,
                                   sites[i], result)) {
                /* Result will have been filled in with the best sequence to use
                 * given the remaining cities, so we just need to fill in this
                 * particular city.
                 */
                result.insert(0, sites[i]);
                return true;
            }
        }
        /* Looks like no options worked. Oh well! */
        return false;
    }
}

```

This function would *not* be a good candidate for memoization. It's extremely unlikely that we'd arrive at the same recursive call in two different ways, since that would mean that somehow we visited the same set of cities in two different ways and ended up using exactly the same amount of time to do so.

## Problem Three: Pattern Matching

The recursion here works by recursively consuming both the pattern and the text, but its base case is only for the case where the *pattern* is empty, since an empty pattern only matches the empty string while an empty string can match a nonempty pattern. The solution we've introduced here uses a quick optimization that's worth keeping in your back pocket. Because we always munch from the front of the pattern and text strings, any text or pattern string we encounter later on is going to be a suffix of the original text or pattern. Therefore, rather than making lots of copies of strings by using `string::substr`, we'll just keep track of the index of the next character in each string that we need to process.

```
bool matchesRec(const string& text, int textIndex,
               const string& pattern, int patternIndex) {
    /* Base case: If we've consumed the pattern, confirm we consumed the text. */
    if (patternIndex == pattern.length()) {
        return textIndex == text.length();
    }
    /* Recursive step: there's more pattern to match. See what to do here. */
    /* Case 1: The next pattern character is a letter. */
    else if (isalpha(pattern[patternIndex])) {
        return textIndex != text.length() && // Text isn't empty
               text[textIndex] == pattern[patternIndex] && // That char matches
               matchesRec(text, textIndex + 1, pattern, patternIndex + 1);
    }
    /* Case 2: The next pattern character is a dot. */
    else if (pattern[patternIndex] == '.') {
        return textIndex != text.length() &&
               matchesRec(text, textIndex + 1, pattern, patternIndex + 1);
    }
    /* Case 3: The next pattern character is a ?. */
    else if (pattern[patternIndex] == '?') {
        return matchesRec(text, textIndex, pattern, patternIndex + 1) ||
               (textIndex != text.length() &&
                matchesRec(text, textIndex + 1, pattern, patternIndex + 1));
    }
    /* Case 4: The next pattern character is a star. */
    else if (pattern[patternIndex] == '*') {
        return matchesRec(text, textIndex, pattern, patternIndex + 1) ||
               (textIndex != text.length() &&
                matchesRec(text, textIndex + 1, pattern, patternIndex));
    } else {
        error("Unknown pattern character.");
        return false; // Unreachable, but makes the compiler happy.
    }
}

bool matches(const string& text, const string& pattern) {
    return matchesRec(text, 0, pattern, 0);
}
```

This function is very amenable to memoization, especially given that texts with multiple stars or question marks in them can possibly match the same text in several different ways. Fun fact: many years ago, Keith got this exact question as a job interview question for a technical internship at Facebook and didn't think to use memoization after writing a recursive solution. ☺



We're going to use the handy `SparseGrid` type for our memoization. It's essentially a 2D grid that may have missing entries, which is perfect for memoization where our table is initially empty and then has entries filled in as the recursion progresses.

```
bool matchesRec(const string& text, int textIndex,
                const string& pattern, int patternIndex,
                SparseGrid<bool>& memo) {

    /* Base case: If we've consumed the pattern, confirm we consumed the text. */
    if (patternIndex == pattern.length()) {
        return textIndex == text.length();
    }
    /* Base case: If we've memoized the result, return it. */
    else if (memo.isSet(textIndex, patternIndex)) {
        return memo[textIndex][patternIndex];
    }
    /* Recursive step always has to write the answer down. We'll store that value
     * in a variable that we write at the very end of the function.
     */
    bool answer;

    /* Case 1: The next pattern character is a letter. */
    else if (isalpha(pattern[patternIndex])) {
        answer = textIndex != text.length() &&
            text[textIndex] == pattern[patternIndex] &&
            matchesRec(text, textIndex + 1, pattern, patternIndex + 1, memo);
    }
    /* Case 2: The next pattern character is a dot. */
    else if (pattern[patternIndex] == '.') {
        answer = textIndex != text.length() &&
            matchesRec(text, textIndex + 1, pattern, patternIndex + 1, memo);
    }
    /* Case 3: The next pattern character is a ?. */
    else if (pattern[patternIndex] == '?') {
        answer = matchesRec(text, textIndex, pattern, patternIndex + 1, memo) ||
            (textIndex != text.length() &&
            matchesRec(text, textIndex + 1, pattern, patternIndex + 1, memo));
    }
    /* Case 4: The next pattern character is a star. */
    else if (pattern[patternIndex] == '*') {
        answer = matchesRec(text, textIndex, pattern, patternIndex + 1, memo) ||
            (textIndex != text.length() &&
            matchesRec(text, textIndex + 1, pattern, patternIndex, memo));
    } else {
        error("Unknown pattern character.");
    }
    memo[textIndex][patternIndex] = answer;
    return answer;
}

bool matches(const string& text, const string& pattern) {
    SparseGrid<bool> memo(text.length() + 1, pattern.length() + 1);
    return matchesRec(text, 0, pattern, 0, memo);
}
```

## Problem Four: Advocating for Exponents

```
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.

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.

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

We know that this code runs in time  $O(n)$ , so it scales roughly linearly with the size of  $n$ . Therefore, if it took  $1\mu\text{s}$  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  $100\mu\text{s}$  to complete.

```
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.

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.

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

As before, this should take about  $100\mu\text{s}$ .

```

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.

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  $O(\log n)$ .

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

We know that the runtime when  $n = 100$  is roughly  $1\mu\text{s}$ . 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  $2\mu\text{s}$ .

## Problem Five: Revisiting Reversals

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

Notice that when we call this function with a string of length  $n$ , we do  $O(n)$  work inside the function. That comes from the cost of making the substring of length  $n-1$ , plus the work to concatenate the resulting string with `str[0]`, plus the cost of initializing the argument of the function we called, which takes its argument by value. We then make a recursive call on a problem of size  $n-1$ . The net effect is that, like with insertion and selection sort, we're roughly doing work

$$n + (n-1) + (n-2) + \dots + 2 + 1$$

which works out to  $O(n^2)$  work.

If we rewrite this code so that we have the argument passed in by `const` reference, then we still haven't changed the fact that we're doing  $O(n)$  work inside the body of the function call, so the overall runtime is still going to be  $O(n^2)$ . However, we should expect it to run a bit faster, since we are reducing the total amount of work that we need to do. at each step.

```
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));  
    }  
}
```

Notice that any given function call to `reverseOf` will still do  $O(n)$  work on a string of length  $n$ , since we have to create the two substrings (total length  $n$ ) and concatenate them together to form a longer string of length  $n$ . However, this code is different from the previous part in that there are two recursive calls, not one, and each one is to a subproblem whose size is roughly  $n/2$ .

But we've seen this before! This is just like mergesort, which does linear work at each call and makes two subcalls on problems of size  $n/2$ . That means that this runs in time  $O(n \log n)$ .

## Problem Six: Cell Towers Revisited

Here's what this code would look like if we chose to rewrite it using the strategy suggested:

```
int bestCoverageRec(const Vector<int>& populations, int index) {
    /* Base case 1: If we are out of cities, we can't cover anyone. */
    if (index == populations.size()) {
        return 0;
    }
    /* Base case 2: If there's just one city, cover it. */
    else if (index == populations.size() - 1) {
        return populations[index];
    }
    /* Recursive step: consider our options when we do or do not cover the first
     * tower. If we cover the first tower, we have to give up the second.
     */
    else {
        return max(bestCoverageRec(populations, index + 2) + populations[0],
                   bestCoverageRec(populations, index + 1));
    }
}

int bestCoverageFor(const Vector<int>& populations) {
    return bestCoverageRec(populations, 0);
}
```

Wow, that's a lot cleaner!

This code is just screaming for memoization, since we can get to the same subproblems in all sorts of different ways. For example, the subproblem consisting of all but the first two cities can be reached either by choosing to include the first city or choosing to exclude the first two cities. Here's what that might look like:

```
int bestCoverageRec(const Vector<int>& populations, int index,
                  Map<int, int>& memo) {
    /* Base case 1: If we are out of cities, we can't cover anyone. */
    if (index == populations.size()) {
        return 0;
    }
    /* Base case 2: If there's just one city, cover it. */
    else if (index == populations.size() - 1) {
        return populations[index];
    }
    /* Base case 3: If we memoized the result, just return it. */
    else if (memo.containsKey(index)) {
        return memo[index];
    }
    /* Recursive step: consider our options when we do or do not cover the first
     * tower. If we cover the first tower, we have to give up the second.
     */
    else {
        int with = bestCoverageRec(populations, index + 2, memo) + populations[0];
        int without = bestCoverageRec(populations, index + 1, memo);

        int result = max(with, without);
        memo[index] = result;
        return result;
    }
}

int bestCoverageFor(const Vector<int>& populations) {
    Map<int, int> memo;
    return bestCoverageRec(populations, 0, memo);
}
```

So what's the efficiency of this code? Notice that at each recursive call we do  $O(1)$  work doing a bunch of multiplications, additions, function calls, etc., plus some amount of table reading. We'll end up having recursive calls made for the index ranging from 0 to the number of cities, so there's going to be  $O(n)$  work done, plus the work done to read and write from the Map.

We haven't talked about this yet, but the map has lookup time  $O(\log n)$  for fun reasons we'll discuss in a few weeks. That means that this code runs in time  $O(n \log n)$ . If you used a different structure, like a Vector, though, you can drop this down to  $O(n)$  runtime, a *massive* improvement over the original!