Section Solutions 4

Problem One: CHeMoWIZrDy

```
bool isElementSpellable(string text, Lexicon& symbols) {
   /* Base case: If there is no text at all, it can be spelled with no
    * element symbols at all.
    */
   if (text == "") return true;

   /* Recursive step: See if there is some prefix of the text that can
    * be removed that happens to be an element symbol.  This code uses
    * the fact that all element symbols are at most three characters
    * long.
    */
   for (int i = 1; i <= text.length() && i <= 3; i++) {
      if (symbols.contains(text.substr(0, i)) &&
          isElementSpellable(text.substr(i), symbols)) {
          return true;
      }
   }

   /* If no option works, then the text is not element-spellable. */
   return false;
}
```

Problem Two: Big-O Notation

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 the same amount of time to multiply together any two numbers.

The function has complexity O(n). To see this, note that the inner loop runs exactly n times, each doing a constant amount of work. Therefore, the overall complexity is O(n). This means that there is no dependence on m.

ii. If it takes 1µs to compute `raiseToPower(100, 200)`, about how long will it take to compute `raiseToPower(50, 400)`? Why can't you give an exact value for the runtime?

Since n has doubled from 200 to 400 and the time complexity is O(n), the new runtime should be about twice the runtime as before, so it should take about 2µs.

We can't give an exact value for the runtime because big-O notation ignores lower-order growth terms. These other terms can contribute to the runtime as well for small values of n, and might influence the overall runtime.

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 the same amount of time to multiply together any two numbers.

The runtime is O(n). To see this, note that

- **`raiseToPower(`m, n`)` does O(1) work, then calls `raiseToPower(`m, $n - 1$`)`.**
- **`raiseToPower(`m, $n - 1$`)` does O(1) work, then calls `raiseToPower(`m, $n - 2$`)`.**
- **…**
- **`raiseToPower(`m, 1`)` does O(1) work, then calls `raiseToPower(`m, 0`)`.**
- **`raiseToPower(`m, 0`)` does O(1) work.**

This means that there are a total of $n + 1$ calls, each of which does O(1) work. Therefore, the total work done is O(n).

iv. If it takes 1μs to compute `raiseToPower(100, 200)`, about how long will it take to compute `raiseToPower(50, 400)`?

As before, the runtime will be around 2μs.

It turns out that there is a much faster way to compute m^n when n is a nonnegative integer. 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 $m^n = m^{2k+1} = m \cdot (m^{2k}) = m \cdot (m^k)^2$

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

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

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

v. What is the big-O complexity of the above function, written in terms of m and n? You can assume that it takes the same amount of time to multiply together any two numbers.

The time complexity is O(log n). Note that at each level of the recurrence, n's value goes down by a factor of two. This means that the maximum number of recursive calls can be at most O(log n), since at that point n will have shrunk down to 0 (since we always round down). Each level does only O(1) work, so the total runtime is O(log n).

vi. If it takes 1μs to compute `raiseToPower(100, 100)`, about how long will it take to compute `raiseToPower(50, 10000)`?

Note that log 10000 = log 100^2 = 2 log 100. Therefore, we would expect the second call to `raiseToPower` to take about twice as long as before, giving a runtime of 2μs.

vii. *(Challenge problem, if you have the time)* What happens to the big-O time complexity if you rewrite the function in the following way?

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

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

Notice that this function makes *two* recursive calls at each level. This means that

- **There is one recursive call with *n* at its initial value.**
- **There are two recursive calls with *n* around *n* / 2.**
- **There are four recursive calls with *n* around *n* / 4.**
- **There are eight recursive calls with *n* around *n* / 8.**
- **…**
- **There are 2^k recursive calls with *n* around *n* / 2^k.**

Eventually, this process stops when $k > \log_2 n$. When that happens, the bottom layer will have a total of around *n* total recursive calls (since $2^k > 2^{\log n} = n$). Each recursive call does a total of O(1) work, so the total amount of work done is equal to the total number of recursive calls, which is

$$1 + 2 + 4 + 8 + \ldots + 2^{\log n}$$

This is the sum of a geometric series. It turns out that this is equal to

$$2^{1 + \log n} - 1 = 2 \cdot 2^{\log n} - 1 = 2n - 1$$

So the total runtime is O(*n*).