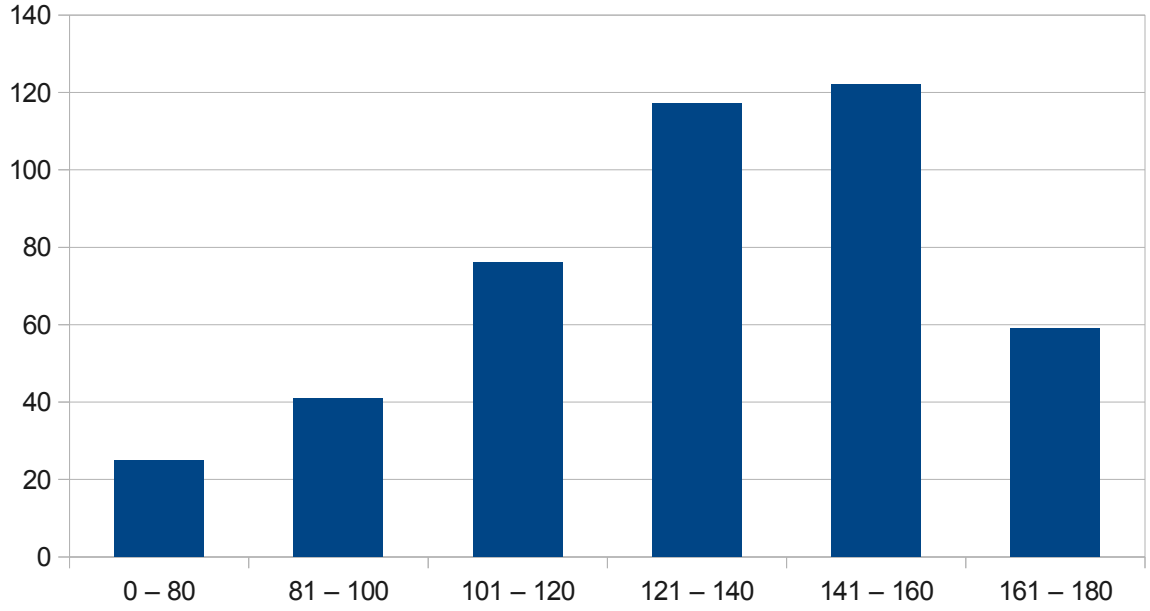
CS106B Midterm Exam #1 Solutions

Below is the score distribution for the first midterm exam:



Overall, the final statistics were as follows:

Mean: 130 / 180 (72.2%)

Median: 135 / 180 (75%)

Standard Deviation: 29 (16%)

We are **not** grading this course using raw point totals and will instead be grading on a (fairly generous) curve. Roughly speaking, the median score corresponds to roughly a B/B+. As always, if you have any comments or questions about the midterm or your grade on the exam, please don't hesitate to drop by office hours with questions! You can also email Dawson or Keith with any questions.

If you think that we made any errors in our grading, please feel free to submit a regrade request to us. Just write a short (one-paragraph or so) description of what you think we graded incorrectly, staple it to the front of your exam, and hand your exam to either Dawson or Keith by Monday, May 20 at 2:15PM.

Regrade requests should probably be for cases where we have made an error while grading the exam (such as marking correct code as incorrect or totaling points incorrectly). We grade based on a criteria and try to apply deductions consistently, so in the interest of fairness we usually don't make local adjustments to point deductions.

Problem One: Pet Matchmaking (25 Points)

```
Set<string> allPetsMatching(Map<string, Set<string> >& adjectiveMap,
                            Vector<string>& requirements) {
    Set<string> result;
    foreach (string pet in adjectiveMap) {
        if (hasAllRequirements(adjectiveMap[pet], requirements)) {
            result += pet;
        }
    }
    return result;
}


bool hasAllRequirements(Set<string>& adjectives, Vector<string>& requirements) {
    foreach (string req in requirements) {
        if (!adjectives.contains(req)) {
            return false;
        }
    }
    return true;
}
```

Most solutions we saw for this problem were excellent. Common mistakes included finding all pets that matched *at least one* adjective rather than *all* of the adjectives, incorrectly iterating over the keys in a map (for example, using a counting **`for`** loop), or confusing operations on the **`Vector`** with operations on the **`Set`**. Incidentally, most solutions don't require a special case to handle the set of zero adjectives, any many solutions we saw had redundant code for this case.

Problem Two: Shrinkable Words Revisited (40 Points)

There are many possible solutions to this problem, some using no recursion at all, some using simple recursion, and some using recursive backtracking. Here is one possible solution:

```
bool isShrinkingSequence(string word, Lexicon& words, Stack<string> path) {
    /* Base Case 1: If the stack is empty, this can't be a shrinking sequence. */
    if (path.isEmpty()) return false;

    /* Base Case 2: If the word parameter isn't a word, the path can't possibly
     * be a shrinking path.
     */
    if (!words.contains(word)) return false;

    /* Get the top of the stack and confirm that it matches the word. */
    string top = path.pop();
    if (top != word) return false;

    /* If the word is just one character, the stack should be empty. */
    if (word.length() == 1) return path.isEmpty();

    /* If the stack is empty, then this can't be a shrinking path because we
     * would have returned in the previous step.
     */
    if (path.isEmpty()) return false;

    /* Confirm that the top word differs from the current word by exactly one
     * letter.  If not, this isn't a shrinking sequence.
     */
    if (!differsByOneLetter(word, path.peek()) return false;

    /* This is a shrinking sequence iff the remainder of the stack is a shrinking
     * sequence for the top of the stack.
     */
    return isShrinkingSequence(path.peek(), words, path);
}

bool differsByOneLetter(string longer, string shorter) {
    for (int i = 0; i < longer.size(); i++) {
        if (longer.substr(0, i) + longer.substr(i + 1) == shorter) {
            return true;
        }
    }
    return false;
}
```

This problem is a lot trickier than it initially appears and there are many different cases to check for. Common mistakes included not checking that the stack wasn't empty before peeking or popping it; forgetting to check that the words in the stack were English words; counting the empty string, which isn't a word, as a shrinkable word; incorrectly checking whether adjacent words in the stack differed by exactly one letter; or forgetting to check that the stack ends on a string of length one.

A subtle but common error in this problem was writing a loop like this one:

```
for (int i = 0; i < path.size(); i++) {
    string top = path.pop();
    /* … */
}
```

This code appears to iterate over the contents of the stack, but has a slight error in it. Specifically, note that on each iteration, the value of `i` increases and the value of `path.size()` decreases, so the total number of iterations is roughly half what it should be.

Problem Three: Balanced Parentheses (45 Points)

```
Vector<string> balancedStringsOfLength(int n) {
    Vector<string> result;
    if (n == 0) {
        result += "";
        return result;
    } else {
        for (int i = 0; i < n; i++) {
            foreach (string one in balancedStringsOfLength(i)) {
                foreach (string two in balancedStringsOfLength(n - i - 1)) {
                    result += "(" + one + ")" + two;
                }
            }
        }
        return result;
    }
}
```

Common mistakes included generating duplicate copies of certain strings of parentheses, generating some (but not all) strings of the proper number of parentheses, forgetting to generate the empty string, or generating strings that were not balanced.

One particular mistake we saw was trying to generate all strings of *n* balanced parentheses by starting with the empty string and repeatedly applying one of the following three operations:

- Append `()`.
- Prepend `()`.
- Surround the string in parentheses.

Unfortunately, this does not generate all strings of balanced parentheses. For example, the string

`(())(())`

cannot be made this way.

As an interesting aside – the number of ways that you can make a string of exactly *n* pairs of balanced parentheses is given by the *n*th *Catalan number*, denoted C_n. The value of C_n is equal to

$$C_n = \frac{(2n)!}{n!(n+1)!}$$

This sequence starts 1, 1, 2, 5, 14, 42, 132, … . These numbers appear in many surprising places!

Problem Four: The Hungry Frog (45 Points)

```
bool canFrogEatFly(int frogPos, int flyPos,
                   Vector<int>& jumpSizes,
                   Set<int>& lilypads) {
    /* Base Case: If the frog is in the water, it can't get to the fly. */
    if (!lilypads.contains(frogPos)) return false;

    /* Base Case: If the frog is at the fly, the frog can eat the fly. */
    if (frogPos == flyPos) return true;

    /* For each possible next jump, try making that jump forwards and backwards
     * and see if either approach works.  Note that this loop doesn't execute if
     * there are no jumps left, so no extra base cases are necessary.
     */
    for (int i = 0; i < jumpSizes.size(); i++) {
        Vector<int> remainder = jumpSizes;
        remainder.remove(i);
        if (canFrogEatFly(frogPos + jumpSizes[i], flyPos, remainder, lilypads) ||
            canFrogEatFly(frogPos - jumpSizes[i], flyPos, remainder, lilypads)) {
            return true;
        }
    }
    return false;
}
```

Common mistakes included not considering all possible orderings of the jumps, only considering jumps in a single direction, returning after only one of the two recursive calls returned, or trying to simulate moving the frog by instead moving the fly position (which does not work without adjusting the positions of the lily-pads.)

Problem Five: Palindrome Efficiency (25 Points)

```
bool isPalindrome(string word) {
    if (word.length() <= 1) return true;
    if (word[0] != word[word.length() - 1]) return false;
    return isPalindrome(word.substr(1, word.length() - 2));
}
```

(i) Analyzing the Efficiency (17 Points)

Let *n* be the length of the string given to **`isPalindrome`** as input. What are the ***best case*** and ***worst-case*** big-O time complexities of the **`isPalindrome`** function? Justify your answer.

In the best case, given a string of length n, the algorithm could return before making a single recursive call if the first and last characters of the string aren't the same. Because it takes time $O(n)$ to copy the string when passing it by value, this means that the total amount of work done by the algorithm is $O(n)$, so the best-case runtime is $O(n)$.

In the worst case, if the string is a palindrome, we start off with a call with a string of size n, then a string of size $n - 2$, then a call with a string of size $n - 4$, etc. Each call does $O(n)$ work, so the total work done is roughly (in the even case)

$$n + (n - 2) + (n - 4) + \ldots + 4 + 2 = O(n^2)$$

and in the odd case

$$n + (n - 2) + (n - 4) + \ldots + 3 + 1 = O(n^2)$$

So the worst-case runtime is $O(n^2)$.

```
bool recIsPalindrome(string& word, int lhs, int rhs) {
    if (rhs <= lhs) return true;
    if (word[lhs] != word[rhs]) return false;
    return recIsPalindrome(word, lhs + 1, rhs - 1);
}

bool isPalindrome(string& word) {
    return recIsPalindrome(word, 0, word.length() - 1);
}
```

(ii) Analyzing the Efficiency (Again) (8 Points)

Let *n* be the length of the string given to the above **`isPalindrome`** as input. What are the ***best case*** and ***worst-case*** big-O time complexities of the **`isPalindrome`** function? Justify your answer.

In the best-case, this algorithm returns false without recursing because the string does not have the same first and last character. In this case, we do only O(1) work setting up the function calls and returning (because the string is passed by value). Therefore, the best-case runtime for the function is **O(1).**

In the worst-case, as before, the function makes recursive calls on strings of size n, n - 2, n - 4, …, 4, 2, 0 (ending in 5, 3, 1 in the odd case). This is a total of O(n) recursive calls. However, each call now does only O(1) work, so the total work done in the worst case is **O(n).**

The most common mistake on this problem was saying that the best-case runtime is O(1) because the function could run on a string of size 0 or size 1. While this is true, remember that best-case runtime refers to best-case runtime *for arbitrary values of n*. In other words, you would need to consider the best-case runtime for strings of arbitrary lengths, not just 0 or 1.

Another common mistake in part (i) was claiming that the best-case runtime was O(1) if given a long string whose first and last characters differ. However, remember that the parameter to the function is passed by value, meaning that it takes time at least O(*n*) just to copy the string when it is passed into the function.

The last class of common mistake we saw was claiming that the worst-case runtime for the first version of the function is O(n^3). This often occurred because there are three O(*n*)'s floating around – O(*n*) to create the substring, O(*n*) to copy the parameter, and O(*n*) function calls. In this case, you don't want to multiply all these values together. You should add the first two (the O(*n*) times for substrings and for passing by value) to get the total runtime within one function call (O(*n*)), then multiply that by O(*n*) calls to get the runtime of O(n^3).