CS106B
Instructor: Cynthia Bailey

CS106B PRACTICE MIDTERM EXAM #2 - SOLUTIONS

---

1. ADTs I

```
Output 1:

Output 2:

Output 3: { 9, 3, 1, 1, 1, 1, 1, 1 }
```

---

2. ADTs II

(a) `Map<string, Stack<string>> storage;`

We use `Map` for quick lookup of each student's `sunet`, to access their collection of submissions. The best collection for their submissions is Stack, because we want to retrieve the *most recent* submission (Last-In, First-Out).

(b)

```
void submitCode(TYPE& storage, const string& sunet, const string& code) {
    storage[sunet].push(code); // will create empty stack if missing key sunet
}
```

(c)

```
string seeMostRecent(const TYPE& storage, const string& sunet) {
    string code;
    if (storage.containsKey(sunet)) { // must guard against creating empty stack by []
        code = storage[sunet].peek();
    }
    return code;
}
```

(d)

```
bool discardMostRecent(TYPE& storage, const string& sunet) {
    if (storage.containsKey(sunet)) { // must guard against creating empty stack by []
        storage[sunet].pop();
        if (storage[sunet].size() == 0) {
            storage.remove(sunet); // remove student from count of students with hw
        }
        return true;
    }
    return false;
}
```

---

3. **Recursion**

```
void makeWords(const Lexicon& english, Set<char> letters, Lexicon& result) {
    string word;
    helper(english, letters, word, result);
}

void helper(const Lexicon& english, Set<char> letters, string word, Lexicon& result) {
    // it turns out that for this problem the order of these two base cases
    // doesn't affect correctness
    if (english.contains(word)) {
        result.add(word);
    }
    if (!english.containsPrefix(word)) {
        return;                 // needed to stop infinite recursion of adding vowels
    }
    for (char c : letters) {
        // only "choose" a letter if it is a vowel or we haven't already used it
        if (isVowel(c) || word.find(c) == string::npos) {
            helper(english, letters, word + c, result);
        }
    }
}

// alternative way of writing choose-unchoose loop
    Set<char> copyLetters = letters; // copy so we don't mess up the for loop
    for (char c : letters) {
        if (!isVowel(c)) {     // "choose": remove consonant from future consideration
            copyLetters.remove(c);
        }
        helper(english, copyLetters, word + c, result);
        if (!isVowel(c)) {
            copyLetters.add(c);    // "unchoose"
        }
    }

// alternative way of writing choose-unchoose loop
    for (char c : letters) {
        if (!isVowel(c)) {
            // returns new set with c removed, doesn't actually modify letters
            // so won't mess up for loop, and we don't need to add it back to "unchoose"
            helper(english, letters - c, word + c, result);
        } else {
            helper(english, letters, word + c, result);
        }
    }
```

```
// this alternative has the bad style issue of being arm's-length recursion, but it
// does work
void helper(const Lexicon& english, Set<char> letters, string word, Lexicon& result) {
    for (char c : letters) {
            // only "choose" a letter if it is a vowel or we haven't already used it
            if (isVowel(c) || word.find(c) == string::npos) {
                if (english.containsPrefix(word + c)) {
                    if (english.contains(word + c)) {
                        result.add(word);
                    }
                    helper(english, letters, word + c, result);
                }
            }
        }
    }
}


// yet another alternative is to have the wrapper function do some prep work of dividing
// the given set of letters into two sets, vowels and consonants, and pass them as
// separate arguments to the helper; then here in the helper, loop over them separately
// one after the other
void helper(const Lexicon& english, Set<char> consonants, Set<char> vowels,
             string word, Lexicon& result) {
    // it turns out that for this problem the order of these two base cases
    // doesn't affect correctness
    if (english.contains(word)) {
        result.add(word);
    }
    if (!english.containsPrefix(word)) {
        return;                 // needed to stop infinite recursion of adding vowels
    }

    for (char c : consonants) {
        helper(english, consonants - c, vowels, word + c, result);
    }
    for (char v : vowels) {
        helper(english, consonants, vowels, word + v, result);
    }
}
```

---

4. **Big-O**

O(N) – first loop is straightforward N, plus second loop is N/2; strip the 3/2 coefficient from the sum
$O(2^N)$ – basically same pattern as Fibonacci
$O(N^2)$ – insert at the beginning is expensive since it causes all existing elements to “scoot over”
O(N) – basically same pattern as factorial