Practice CS106B Midterm Solutions

Here's one possible set of solutions for the midterm questions. Before reading over these solutions, please, please, please work through the problems under semi-realistic conditions (spend three hours, load things up on BlueBook, have a notes sheet, etc.) so that you can get a better sense of what taking a coding exam is like.

This solutions set contains some possible solutions to each of the problems in the practice exam. It's not meant to serve as "the" set of solutions to the problems – there are lots of ways you can go about solving each problem here. In other words, if your solution doesn't match ours, don't take that as a sign that you did something wrong. Feel free to stop by the CLaIR, to ask questions on Piazza, or to ask your section leader for their input!

The solutions we've included here are a lot more polished than what we'd expect most people to turn in on an exam. You'll three hours to complete this whole exam. We have a lot of time to write up these solutions, clean them up, and try to get them into a state where they'd be presentable as references. Don't worry if what you wrote isn't as clean as what we have here, but do try to see if there's anything we did here that would help you improve your own coding habits.

Problem One: Container Classes (8 Points)

A Matter of Context

One way to solve this problem is to keep track of the most-recently-seen word and to use that to help fill in the predecessor map. Here's a solution that uses this strategy:

```
Map<string, Lexicon> predecessorMap(istream& input) {
    TokenScanner scanner(input);

    Map<string, Lexicon> result;
    string lastWord; // Most-recently-read string, initially empty as a sentinel.

    while (scanner.hasMoreTokens()) {
        /* Grab the next token. For consistency, we convert everything to lower-
         * case.
         */
        string token = toLowerCase(scanner.nextToken());

        /* If this isn't a word, we don't care about it. */
        if (!isalpha(token[0])) continue;

        /* Update the predecessor map, unless this is the first word. */
        if (lastWord != "") result[token].add(lastWord);
        lastWord = token;
    }
    return result;
}
```

Why we asked this question: This question was designed to see whether you were comfortable with the Map type's autoinsertion feature, with processing a stream of data one element at a time using `TokenScanners`, and more generally with the sorts of data processing techniques from Assignment 1 and Assignment 2.

Problem Two: Recursive Enumeration (8 Points)

Checking Your Work

There are many ways to approach this problem. Our solution works by keeping track of two groups – the group of people who haven't proofread anything and the group of people who haven't had their sections proofread – and at each point chooses a person and gives them a section to read. We try out all options except for those that directly assign a person their own section.

Here's what this looks like in code:

```
void listAssignmentsRec(const Set<string>& unassignedReaders,
                        const Set<string>& unassignedSections,
                        Map<string, string>& assignments) {
    /* Base Case: If everyone is assigned to read something, print out the
     * reading assignments.
     */
    if (unassignedReaders.isEmpty()) {
        cout << assignments << endl;
    }
    /* Recursive Case: Choose a reader and look at all ways to give them
     * something to read.
     */
    else {
        string reader = unassignedReaders.first();

        /* Loop over all sections that haven't yet been assigned. */
        for (string section: unassignedSections) {
            /* Don't assign someone to read their own section. */
            if (section != reader) {
                /* Explore all options that include giving this person this
                 * section to read.
                 */
                assignments[reader] = section;
                listAssignmentsRec(unassignedReaders - reader,
                                   unassignedSections - section,
                                   assignments);

                /* Undo this commitment. */
                assignments.remove(reader);
            }
        }
    }
}

void listCheckingArrangements(const Set<string>& people) {
    Map<string, string> assignments;
    listAssignmentsRec(people, people, assignments);
}
```

Why we asked this question: We chose this particular problem for a number of reasons. First, we included it because it's closely related to the classic problem of listing permutations, which by this point we hoped you'd have a good handle on. Second, we liked how this problem forced you to think back to the intuition behind how to generate permutations (choose a position and think of all the things that could go into that position), since the problem structure was sufficiently different from the traditional permutations problem as a consequence of having the items stored in an inherently unordered structure. Third, we thought this problem would allow you to demonstrate what you'd learned about using extra arguments to recursive functions in order to keep track of the decisions made so far and decisions left to make; our solution requires two extra arguments to remember unassigned sections and unassigned people separately. Finally, we wanted to include this problem because it has nice mathematical properties; the assignments you're asked to list here are called ***derangements*** and show up frequently in CS theory.

Fun fact: this same algorithm can be used to list off all the ways to do a "Secret Santa" type arrangement where everyone needs to get someone else a gift!

Common mistakes: We saw a number of mistakes on this problem that were focused on small details of the problem rather than the overall structure. A number of solutions correctly attempted to keep track of which sections weren't read and who hadn't read anything, but accidentally looped over the wrong one when trying to determine which section to assign or which person to assign something to. Some solutions did correctly list all the options, but did so by generating all permutations and then filtering out invalid ones at the last second (which was specifically prohibited by the problem statement). Other solutions modified a map or set while iterating over it, which leads to runtime errors.

If you had trouble with this problem: Our advice on how to proceed from here varies a bit based on what specific difficulty you were having.

If you looked at this problem and didn't recognize that you were looking at a permutations-type problem, we would recommend trying to get more practice looking at these sorts of problems and broadly categorizing them into subtypes like "permutation problems," "subset problems," etc. Section Handout 3 and Section Handout 4 have a number of good problems to work through that might be good spots to look for questions like those. It might also be worth revisiting Assignment 3 and Assignment 4 to look at your solutions and make sure you understood which approach to use for each of the different parts of the assignments. In retrospect, did you make the right design decisions when choosing problem-solving strategies? If not, it might actually be worthwhile to revisit those assignments and code them up against knowing what you now know.

If you had trouble avoiding repeated assignments in the course of solving this problem, we'd recommend taking some time to look at the different recursive functions we've written in the past that enumerate options to see how their design prevents duplicates. Why, for example, does the "list all subsets" function we wrote when we first talked about recursion not list any subsets twice? Why does the "list all permutations" function list each option once? See if you can connect this back to recursion trees.

If you weren't sure how to keep track of which people were assigned sections and which sections had been assigned readers, it might be worth practicing working with the decision tree model of recursive problem-solving. At each point in time, your goal is to make some sort of choice about what the next option will be. What information would you need to have access to in order to make that choice? What would the possible choices look like? A little extra practice with this skill can go a long way.

Problem Three: Recursive Optimization (8 Points)

Avoiding Sampling Bias

One way to solve this problem is to focus on a single person. If we choose this person, we'd want to pick, from the people who aren't friends with that person, the greatest number of people that we can. If we exclude this person, we'd like to pick from the remaining people the greatest number of people that we can. One of those two options must be optimal, and it's just a question of seeing which one's better.

```
/**
 * Given a network and a group of permitted people to pick, returns the largest
 * group you can pick subject to the constraint that you only pick people that
 * are in the permitted group.
 *
 * @param network The social network.
 * @param permissible Which people you can pick.
 * @return The largest unbiased group that can be formed from those people.
 */
Set<string> largestGroupInRec(const Map<string, Set<string>>& network,
                              const Set<string>& permissible) {
    /* Base case: If you aren't allowed to pick anyone, the biggest group you
     * can choose has no people in it.
     */
    if (permissible.isEmpty()) return {};

    /* Recursive case: Pick a person and consider what happens if we do or do not
     * include them.
     */
    string next = permissible.first();

    /* If we exclude this person, pick the biggest group we can from the set of
     * everyone except that person.
     */
    auto without = largestGroupInRec(network, permissible - next);

    /* If we include this person, pick the biggest group we can from the set of
     * people who aren't them and aren't one of their friends, then add the
     * initial person back in.
     */
    auto with = largestGroupInRec(network,
                                  permissible - next - network[next]) + next;

    /* Return the better of the two options. */
    return with.size() > without.size()? with : without;
}

Set<string> largestUnbiasedGroupIn(const Map<string, Set<string>>& network) {
    Set<string> everyone;
    for (string person: network) {
        everyone += person;
    }
    return largestGroupInRec(network, everyone);
}
```

Why we asked this question: We included this problem for several different reasons. First, we wanted to include at least one "subset-type" recursion on this exam and figured that this particular problem would be a nice way to do this. Second, we liked how this problem connected back to the assignments: it's closely related to both the Universal Health Care and Disaster Preparation problems from Assignments 3 and 4, respectively. Third, we thought that the recursive optimization here was particularly elegant to code up once you had the right insights. You can solve this either by keeping track of what options are permissible (as we did here) or by which options are impermissible (track which items can't be chosen again), and can code the solution up in a number of different ways. Finally, this problem is related to a famous problem called the ***maximum independent set problem***, which has a rich history in computer science and shows up in all sorts of interesting practical and theoretical contexts.

Common mistakes: We saw a number of solutions that recognized that picking someone precluded any future call from choosing any of that person's friends, but which forgot to *also* mark that the chosen person couldn't be included in future recursive calls (oops!), which is fairly easy to correct but an easy mistake to make. Many solutions attempted to adapt the optimization from the Disaster Preparation problem by looking at a person and their friends and either choosing the person or one of their friends, which is a reasonable approach to take except that it leads to a lot of redundant work being done by the recursion (trace through an example or two and see if you can see why this is!)

If you had trouble with this problem: As with Problem Two, our advice on what the next steps should be if you had trouble with this particular problem depend on which aspect of the problem you were having trouble with.

If, fundamentally, you didn't recognize that this problem was a subset-type problem, then your first step should probably be to just get a little bit more practice working with recursion problems and trying to recognize their structure. You might benefit from revisiting some of the older section problems (Section Handout 3 and Section Handout 4 have a bunch of great recursion problems on them. Without looking at the solutions, try to see if you can predict which of them use a subsets-type recursion, which use a permutations-type recursion, and which use something altogether different. Or go back to Assignment 3 and Assignment 4 and make sure you have a good understanding about which problems fall into which types.

If you had some trouble keeping track of the decisions you'd made so far and how they interacted with the future (that is, making sure you didn't pick the same thing twice), you may want to get a bit more practice converting between the different container types. This problem is a lot easier to solve if you have the right data structure to remember what's already been seen and what hasn't yet been considered. As a general rule, try to find a way to solve the problems you're working on that involves the minimal amount of "fighting with the code," even if that means doing some sort of conversion step beforehand.

Problem Four: Recursive Backtracking (8 Points)

Mmmmm… Pancakes!

The key idea behind this problem is to follow the hint suggested and to try all possible flips at each point in time. If we ever get the stack sorted, great! We're done. If we're out of moves, then we report failure.

```
/**
 * Given a stack of pancakes, reports whether it's sorted.
 *
 * @param pancakes The stack of pancakes.
 * @return Whether it's sorted.
 */
bool isSorted(Stack<double> pancakes) {
    double last = -1; // No pancake has negative size, fortunately!
    while (!pancakes.isEmpty()) {
        if (last > pancakes.peek()) return false;
        last = pancakes.pop();
    }
    return true;
}

/**
 * Given a stack of pancakes and a number of pancakes to flip, returns the stack
 * you'd get by flipping that many pancakes off the top of the stack.
 *
 * @param pancakes The initial stack of pancakes.
 * @param toFlip How many pancakes to flip.
 * @return The resulting stack.
 */
Stack<double> flipTopOf(Stack<double> stack, int toFlip) {
    Queue<double> stash;
    for (int i = 0; i < toFlip; i++) {
        stash.enqueue(stack.pop());
    }
    while (!stash.isEmpty()) {
        stack.push(stash.dequeue());
    }
    return stack;
}

bool canSortStack(Stack<double> pancakes, int numFlips, Vector<int>& flipsMade) {
    /* Base case 1: If the stack is already sorted, we're done! */
    if (isSorted(pancakes)) return true;

    /* Base case 2: If we're out of flips, we fail! */
    if (numFlips == 0) return false;

    /* Otherwise, try all possible flips. We skip the option of flipping just the
     * top pancake, since that doesn't actually accomplish anything.
     */
    for (int i = 2; i <= pancakes.size(); i++) {
        if (canSortStack(flipTopOf(pancakes, i), numFlips - 1, flipsMade)) {
            flipsMade.insert(0, i);
            return true;
        }
    }
    return false;
}
```

Why we asked this question: We included this question as an example of a recursive backtracking problem that didn't easily fit into any of the molds we'd covered so far (subsets, permutations, combinations, and partitions). Our hope was that you'd use the general decision tree framework to determine what options were available at each step, then combine that with your knowledge of recursive backtracking to put together a solution that tried all options in search of one that would get the stack sorted. We also included this question to make sure you were familiar with container types that weren't exercised in the previous parts of the exam. This question requires you to use the `Stack` type, and while it wasn't necessary to do so, we hoped you'd recognize the connection to `Queue`s as well.

Common mistakes: One of the most common mistakes we saw people make on this problem was writing a for loop like this one:

```
for (int i = 0; i < pancakes.size(); i++) {
    // do something that pops an element off the pancakes stack
}
```

This loop won't actually loop over all the pancakes in the stack. If you're not sure why this is, check out Section Handout 2, which explores why this is.

We also saw a number of solutions that had the right base cases, but checked them in the wrong order. This can lead the code to accidentally return false in cases where there is a valid series of flips. Finally, we saw a large number of solutions that produced the final Vector in the reverse of the order in which it should have been produced.

If you had trouble with this problem: There are essentially four parts to this problem: identifying the recursive insight (that at each step you have some number of pancakes that you need to flip), finding the right base cases (the stack being sorted and no flips being left), structuring the backtracking properly (trying out each option and returning failure only if none of them work), and properly manipulating the stack (finding a way to flip the top without messing up the stack from option to option.)

If you had trouble developing the recursive insight – namely, to flip each possible number of pancakes – we recommend getting more practice with the decision tree framework. Although many of the recursive problems we've talked about this quarter nicely fall into clean categories like subsets or permutations, the most general pattern we've explored for these sorts of problems is the decision tree model. If you're still a little shaky on this idea, we'd recommend revisiting some of the recursive functions from earlier in the quarter to make sure that you have a clear sense of how decision trees factor in. This is something you can absolutely pick up with practice. Once everything clicks, problems like these become a lot easier to solve.

The base cases in this problem are definitely trickier than in the other recursive problems on this exam. One technique we'd recommend choosing a good base case is to look at the function you're writing and make sure you understand, very specifically, what the function is supposed to return given its arguments. This function tries to see if it's possible to get something into sorted order under a certain resource constraint (a number of moves). So what happens if you're already at the goal (it's sorted)? Or you're out of resources (you have no flips left?) Look back at some of the other backtracking problems you've worked through this quarter and see if you can ask similar questions like these about them.

Finally, if you were a bit shaky on the stack manipulation logic, we'd recommend working through some extra practice problems on stacks and queues. For example, could you write a function to use stacks and queues to reverse a stack or reverse a queue? How about to turn the bottom half of a stack upside down?

Problem Five: Big-O and Efficiency (8 Points)

Password Security

```
string makeRandomPassword(const Vector<string>& wordList) {
   string result;
   for (int i = 0; i < 4; i++) {
      int wordIndex = randomInteger(0, wordList.size() - 1);
      result += wordList[wordIndex];
   }
   return result;
}
```

i. **(2 Points)** Let *n* denote the number of words in `wordList`. What is the big-O time complexity of the above code as a function of *n*? You can assume that the words are short enough that the cost of concatenating four strings together is O(1) and that a random number can be generated in time O(1). Explain how you arrived at your answer. Your answer should be no more than 50 words long.

The cost of looking at the size of the `Vector` is O(1) and the cost of reading any element it is O(1). We do four reads and call size four times. Each operation takes time O(1). Therefore, this runs in time O(1).

ii. **(2 Points)** Let's suppose that the above code takes 1ms to generate a password when given a word list of length 50,000. Based on your analysis from part (i), how long do you think the above code will take to generate a password when given a word list of length 100,000? Explain how you arrived at your answer. Your answer should be no more than 50 words long.

Since the runtime is O(1), it's independent of the size of the `Vector`. Therefore, we'd expect that this code would take 1ms to complete in this case.

```
string breakPassword(const Vector<string>& wordList) {
  for (int i = 0; i < wordList.size(); i++) {
    for (int j = 0; j < wordList.size(); j++) {
      for (int k = 0; k < wordList.size(); k++) {
        for (int l = 0; l < wordList.size(); l++) {
          string password = wordList[i] + wordList[j] + wordList[k] + wordList[l];
          if (passwordIsCorrect(password)) {
            return password;
          }
        }
      }
    }
  }
}
```

iii. **(2 Points)** What is the *worst-case* big-O time complexity of the above piece of code as a function of *n*, the number of words in the word list? Explain how you arrived at your answer. Your answer should be no more than 50 words long.

This code loops over all possible ways to choose four words, of which there are n^4. We do O(1) work with each of those combinations, so the overall runtime is $O(n^4)$.

iv. **(2 Points)** Imagine that in the worst case it takes 1,000 years to break a four-word password when given a word list of length 50,000. Based on your analysis from part (iii), how long will it take, in the worst case, to break a password when given a word list of length 100,000? Explain how you arrived at your answer. Your answer should be no more than 50 words long.

Since the runtime is $O(n^4)$, doubling the size of the input will increase the runtime by a factor of $2^4 = 16$. Therefore, we'd expect this would take 16,000 years to complete in the worst-case.

Why we asked this question: We chose this question to make sure you were comfortable with three aspects of big-O notation. First: are you familiar with the big-O time complexities of the core data structures we've covered so far (for example, what's the time required to access an element of a `Vector`?) Second: can you analyze a piece of code to determine its big-O time complexity? Finally: can you use that analysis to make quantitative predictions about how long a piece of code will take to complete? We hoped that this particular application to password security would help you get an appreciation for why this analysis is useful.

Common mistakes: A lot of people did this problem under the assumption that indexing into a `Vector` or computing its size takes time $O(n)$ rather than $O(1)$, which led the runtime analysis to be off by a factor of n. It's worth making sure you understand why this isn't the case, since these are important and fundamental operations.

If you had trouble with this problem: Given that this question was designed to assess three different aspects of big-O notation, your next steps if you had trouble with this problem will depend on which of those areas you weren't comfortable with.

If you're still a bit shaky on the big-O runtimes of the different data structures we've covered so far, then your first order of business is to take a few minutes to go and review them. Make sure you understand the runtimes not as "these are the answers because that's what I read somewhere," but rather as "these are the runtimes because I see how these operations are implemented and understand why they are the way they are."

If you're not yet comfortable taking a block of code and analyzing its big-O runtime, the best thing you can do is just get more practice. Pick some (non-recursive!) code you've written for this class and work through it to determine its efficiency. Ask your section leader if you have any questions, or stop by the LaIR to get another section leader to look over things.

Finally, if you had trouble using your analysis to extrapolate the runtimes of the different pieces of code here, we'd recommend picking some of the common runtimes ($O(1)$, $O(\log n)$, $O(n)$, $O(n^2)$, $O(n!)$, etc.) and thinking about how they scale. Look over the tables of runtimes to see if your guesses match the actual numbers, or ask your SL to double-check your math. And look back at Section Handout 4, which has a bunch of questions along these lines.