CS106B
J Zelenski

Handout #25
Feb 11, 2008

Midterm Practice

Midterm Exam: **Tuesday, Feb 19th 7-9pm**
Terman Auditorium

We'd be very appreciative if you could arrange your schedule to attend the regular exam, but if you have an unavoidable conflict, please e-mail the head TA Jason Turner-Maier by noon on Thursday Feb 14th to make arrangements for an
alternate Tuesday time. Late requests will not be honored.

SCPD students: Local SCPD students are asked to attend the regular on-campus exam. This ensures you will be able to ask questions and receive clarifications along with everyone else. We added a link to a campus map on our web site to help you find Terman. Folks outside the Bay Area will take the exam at your local site on Tuesday. Your SCPD site coordinator will administer the exam. Please send an e-mail to Jason by Thursday Feb 14th to initiate making the arrangements to take the exam on-site.

Coverage

The exam covers material from Chapters 1-7 of the reader and client use of CS106 classes (scanner, vector, map, etc.). The format will largely focus on writing code to solve problems, the kind of work you do on the assignments. We will not be especially picky about syntax or other mostly conceptually shallow ideas. Instead, we are looking for thorough and correct understanding of the core programming concepts.

The exam is open-book and open-notes, but no computers allowed. Don't let the "open-book" nature mislead you. There is not enough time to (re-)learn the material during the exam. You should plan to be facile and experienced enough with the material to readily answer the questions, relying on your notes only for the occasional detail.

Writing code on paper in a relatively short time period is not quite the same as working with the compiler and a keyboard. Most students benefit from preparation to adapt their skills to this format. I recommend that you practice until you are comfortable writing solutions in longhand to small problems. You won't have to look far to find lots of practice problems (this handout, section handouts, textbook exercises, lecture examples, etc.) that can be used to sharpen your prowess.

These problems were drawn from previous midterm exams, so they are fairly representative in terms of format, content, and difficulty. I included a few more problems than on a typical exam, figuring you might want the extra practice. To conserve trees, I cut back on answer space. The real exam will have much more room for your answers and scratch work.

1. Write the function `ConvertMacLineEndingsToPC` to solve a perennial CS106B problem: converting Macintosh text to PC. Due to historical accident and lack of standardization, Macs use the single character `'\n'` to denote a new line, whereas PCs use the two-character sequence `"\r\n"`. This function returns a new string where every character `'\n'` in the input string has been converted to the string `"\r\n"`.

```
string ConvertMacLineEndingsToPC(string str)
```

2a. Write the function `Similarity` that computes a measure of similarity between two sorted vectors. The similarity metric is defined as the count of elements the vectors have in common divided by the average size of the vectors. Here are some examples:

v1	v2	Similarity(v1, v2)	Notes
[a, b, c, c, e]	[a, c, c, k, m]	.6	[a, c, c] in common, avg size 5
[a, m]	[w, x, y, z]	0	None in common, avg size 3
[j, k, k]	[k, z]	.4	[k] in common, avg size 2.5

Note that the input vectors are in **sorted order**. You should use this fact to efficiently implement this operation. The function **must run in O(N) time** where N is the length of the input vectors. Hint: consider an approach similar to that used for the merge step of merge sort.

```
double Similarity(Vector<char> & v1, Vector<char> & v2)
```

2b. Write the `MarkovMatch` function that compares two Markov models. Each Markov model is represented as in the Random Writer assignment: a map where each key is a seed and its value is a vector of characters that follow the seed within the input text. The `MarkovMatch` function returns true if `m1` matches `m2`, false otherwise.

Model `m1` is defined to match `m2` if for each seed in `m1` that is also present in `m2`, the associated vectors have similarity of at least 0.7, as reported by the `Similarity` function from part (a). You may assume that all of the vectors in the maps are already in sorted order and that the `Similarity` function has been implemented correctly.

```
bool MarkovMatch(Map<Vector<char> > & m1, Map<Vector<char> > & m2)
```

3. Below you are given the code from lecture for recursively generating and printing the permutations of a string. You are to make the following changes to this function:

- Instead of printing, the function should return the number of permutations that form a valid word (as reported by a `Lexicon` object).
- The function should not count the same word more than once (e.g. if input is "lepap", count the word "apple" only once).
- The function should use the Lexicon member function `containsPrefix` to avoid searching down dead ends.

You can either show your changes by marking up the existing code, or copying a new version with the changes. Just be sure your intentions are clear.

```
void Permute(string soFar, string rest)
{
   if (rest == "") {
      cout << soFar << endl;
   } else {
      for (int i = 0 i < rest.length(); i++) {
         string remaining = rest.substr(0, i) + rest.substr(i+1);
         Permute(soFar + rest[i], remaining);
      }
   }
}
```

4. The game of Boggle is played using sixteen letter cubes. A letter cube can be represented as a string of length 6, one character for each face on the cube. Given a vector of letter cubes and a target word, you want to determine whether it is possible to spell that word using those letter cubes, where each cube can be used at most once in spelling the word.

Examples: given the vector of cubes `{"etaoin", "shrdlu", "qwerty"}`, it is possible to spell the words "as" and "law" but not "weld" or "toe".

Write a recursive function `CanSpell` that takes a target word along with a vector of letter cubes. The function should return `true` if it is possible to spell the word using the cubes in the vector and `false` if otherwise. Assume that the word and letter cubes will only contain lowercase letters.

```
bool CanSpell(string word, Vector<string> & cubes)
```

5. Write the function `ExtractStrand` that extracts a sorted strand from a non-empty queue of integers. The strand is extracted as follows:

- The first element of the queue is removed and becomes the first element of the strand.
- The remaining queue elements are considered in order. Each element that can be added to the strand while preserving sorted order is removed from the queue and added to the strand.

The function returns a new queue containing the sorted strand. The input queue, which was passed by reference, has been modified to remove the extracted numbers, the remaining numbers are preserved in their original order. **Your solution can use queues and primitive variables but no other data structures (no arrays, vectors, etc.).**

Here are some examples:

q	ExtractStrand(q) returns	After call, q
[2, 1, 7, 12, 5, 10, 2]	[2, 7, 12]	[1, 5, 10, 2]
[3, 3, 1, 2, 3, 4, 1]	[3, 3, 3, 4]	[1, 2, 1]
[6, 4, 2, 3, 3]	[6]	[4, 2, 3, 3]

```
Queue<int> ExtractStrand(Queue<int> & q)
```

6. Write the `Contains` function that given two linked lists will determine whether the second list is a subsequence of the first. To be a subsequence, every value of the second

must appear within the first list and in the same order, but there may be additional values interspersed in the first list. A list contains itself; the NULL list is contained in any list.

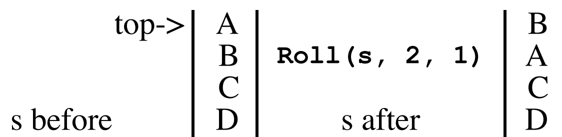
Here are some examples:

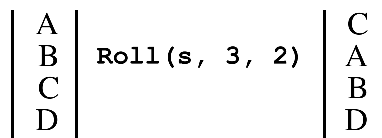
list	sub	Contains(list, sub)
1 -> 4 -> 2 -> 9	1 -> 4	true
1 -> 4 -> 2 -> 9	9 -> 4	false
1 -> 4 -> 2 -> 9	1 -> 9	true
1 -> 4 -> 2 -> 9	1 -> 1 -> 4	false
1 -> 4 -> 2 -> 9	2 -> 9 -> 10	false

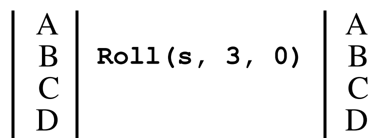
```
struct cellT {
   int val;
   cellT *next;
};

bool Contains(cellT *list, cellT *sub)
```

7. One of the operations in the stack-based printer description language PostScript is **`roll n p`**, which rotates the top n elements of a stack by p positions. The operation works by removing the top n elements, cycling the top element to the last position p times, and then placing those elements back onto the stack. Consider these examples:







You may find it helpful to use temporary storage (additional stacks, queues, vectors, etc.) You may assume **`nElems`** is non-negative and never larger than the stack size and than **`nPos`** is non-negative and less than **`nElems`**.

```
void Roll(Stack<char> &s, int nElems, int nPos)
```

8a. The function **`Winky`** is defined as follows:

```
void Winky(int n)
{
    for (int i = 0; i < n; i++)
        Winky(i);
}
```

Give the computational complexity of **Winky** expressed in big-O notation, where N is the value of the argument **n**, assumed to be a nonnegative integer. Briefly justify your answer.

8b. Consider the following simple program:

```
void Toss(Stack<string> s)
{
   Stack<string> tmp;
   while (!s.isEmpty())
      tmp.push(s.pop());
   s = tmp;
}

int main()
{
   Stack<string> salad;
   salad.push("lettuce");
   salad.push("tomato");
   salad.push("cucumber");
   Toss(salad);
   cout << salad.pop() << endl;
   return 0;
}
```

Trace through the program as written and describe its output.

Now change the prototype of the Toss function to `void Toss(Stack<string> & s)`. Without making any other changes to the rest of the code, will this program still compile? If so, what does this program now output?

8c. You have a group of students that were randomly assigned to dorm rooms. You wish to re-assign them to rooms such that the students are arranged in decreasing height order as you go down the hall. Each student has a lot of personal stuff to move when switching rooms, so you want to minimize the number of moves, however you don't care how far any particular move is. Would you rather use an insertion sort or selection sort algorithm?