CSCI-1200 Data Structures — Fall 2015
Lecture 15 — Problem Solving Techniques, Continued

Review of & Finishing Lecture 14

- General Problem Solving Techniques:
  1. Generating and Evaluating Ideas
  2. Mapping Ideas into Code
  3. Getting the Details Right (Today!)
- Small exercises to practice these techniques
- Problem Solving Strategies / Checksheet (Today!)
- Design Example: Conway's Game of Life (Today!)
- Another Design Example: Inverse Word Search (Today!)

14.3 Getting the Details Right

- Is everything being initialized correctly, including boolean flag variables, accumulation variables, max / min variables?
- Is the logic of your conditionals correct? Check several times and test examples by hand.
- Do you have the bounds on the loops correct? Should you end at n, $n-1$ or $n-2$?
- Tidy up your "notes" to formalize the invariants. Study the code to make sure that your code does in fact have it right. When possible use assertions to test your invariants. (Remember, sometimes checking the invariant is impossible or too costly to be practical.)
- Does it work on the corner cases; e.g., when the answer is on the start or end of the data, when there are repeated values in the data, or when the data set is very small or very large?

14.7 Example: Non-Linear Word Search

- What did we need to think about to **Get the Details Right** when we finished the implementation of the nonlinear word search program? What did we worry about when writing the first draft code (a.k.a. pseudo-code)? When debugging, what test cases should we be sure to try? Let's try to break the code and write down all the "corner cases" we need to test.

```
bool search_from_loc(loc position, const vector<string>& board, const string& word, vector<loc>& path) {

  // start by adding this location to the path
  path.push_back(position);
  // BASE CASE: if the path length matches the word length, we're done!
  if (path.size() == word.size()) return true;

  //  search all the places you can get to in one step
  for (int i = position.row-1; i <= position.row+1; i++) {
    for (int j = position.col-1; j <= position.col+1; j++) {
      // don't walk off the board though!
      if (i < 0 || i >= board.size()) continue;
      if (j < 0 || j >= board[0].size()) continue;
      // don't consider locations already on our path
      if (on_path(loc(i,j),path)) continue;
      // if this letter matches, recurse!
      if (word[path.size()] == board[i][j]) {
        // if we find the remaining substring, we're done!
        if (search_from_loc (loc(i,j),board,word,path))
          return true;
      }
    }
  }
```

```
    // We have failed to find a path from this loc, remove it from the path
    path.pop_back();
    return false;
  }
```

14.9 Problem Solving Strategies / Checksheet

Here is an outline of the major steps to use in solving programming problems:

1. Before getting started: study the requirements, carefully!
2. Get started:
   (a) What major operations are needed and how do they relate to each other as the program flows?
   (b) What important data / information must be represented? How should it be represented? Consider and analyze several alternatives, thinking about the most important operations as you do so.
   (c) Develop a rough sketch of the solution, and write it down. There are advantages to working on paper first. Don't start hacking right away!
3. Review: reread the requirements and examine your design. Are there major pitfalls in your design? Does everything make sense? Revise as needed.
4. Details, level 1:
   (a) What major classes are needed to represent the data / information? What standard library classes can be used entirely or in part? Evaluate these based on efficiency, flexibility and ease of programming.
   (b) Draft the main program, defining variables and writing function prototypes as needed.
   (c) Draft the class interfaces — the member function prototypes.

   These last two steps can be interchanged, depending on whether you feel the classes or the main program flow is the more crucial consideration.
5. Review: reread the requirements and examine your design. Does everything make sense? Revise as needed.
6. Details, level 2:
   (a) Write the details of the classes, including member functions.
   (b) Write the functions called by the main program. Revise the main program as needed.
7. Review: reread the requirements and examine your design. Does everything make sense? Revise as needed.
8. Testing:
   (a) Test your classes and member functions. Do this separately from the rest of your program, if practical. Try to test member functions as you write them.
   (b) Test your major program functions. Write separate "driver programs" for the functions if possible. Use the debugger and well-placed output statements and output functions (to print entire classes or data structures, for example).
   (c) Be sure to test on small examples and boundary conditions.

   The goal of testing is to incrementally figure out what works — line-by-line, class-by-class, function-by-function. When you have incrementally tested everything (and fixed mistakes), the program will work.

Notes

- For larger programs and programs requiring sophisticated classes / functions, these steps may need to be repeated several times over.
- Depending on the problem, some of these steps may be more important than others.
  - For some problems, the data / information representation may be complicated and require you to write several different classes. Once the construction of these classes is working properly, accessing information in the classes may be (relatively) trivial.
  - For other problems, the data / information representation may be straightforward, but what's computed using them may be fairly complicated.
  - Many problems require combinations of both.

15.1 Example: Quicksort

- Quicksort also the partition-exchange sort is another efficient sorting algorithm. Like mergesort, it is a divide and conquer algorithm.
- Quicksort first divides a large array into two smaller sub-arrays, the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.
- The steps are:
  1. Pick an element, called a pivot, from the array.
  2. Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
  3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.
- Here's an example of a quicksort routine. Let's compare it to mergesort.

```
#include <iostream>
#include <vector>
using namespace std;

int quickSort(vector<double>& array, int start, int end);


int partition(vector<double>& array, int start, int end, int& swaps) {
  int mid = (start + end)/2;
  double pivot = array[mid];


  }
}


int quickSort(vector<double>& array, int start, int end) {
  int swaps = 0;
  if(start < end) {
    int pIndex = partition(array, start, end, swaps);

    //after each call one number(the PIVOT) will be at its final position
    swaps += quickSort(array, start, pIndex-1);
    swaps += quickSort(array, pIndex+1, end);
  }

  return swaps;
}


int main() {
  vector<double> pts(7);
  pts[0] = -45.0;  pts[1] =   89.0;  pts[2] =    34.7;  pts[3] =   21.1;
  pts[4] =   5.0;  pts[5] =  -19.0;  pts[6] =  -100.3;

  quickSort(pts, 0, pts.size()-1);

  for (unsigned int i=0; i<pts.size(); ++i)
    cout << i << ": " << pts[i] << endl;
}
```

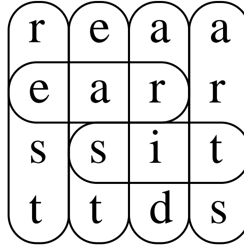
15.2 Another Example: Inverse Word Search

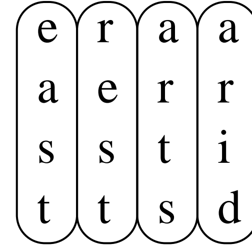
Let's flip the classic word search problem and instead *create the board* that contains the specified words! We'll be given the grid dimensions and the set of words, each of which must appear in the grid, in a straight line. The words

may go forwards, backwards, up, down, or along any diagonal. Each grid cell will be assigned one of the 26 lowercase letters. We may also be given a set of words words that should *not* appear anywhere in the grid. Here's an example:

Input:

```
4 4
+ arts
+ arid
+ east
+ rest
- ear
- at
- sit
```

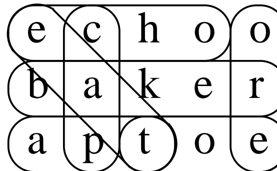




In the middle above, is an *incorrect* solution. Though it contains the 4 required words, it also contains two of the forbidden words. The solution on the right is a fully correct solution. This particular problem has 8 total solutions including rotations and reflections.

Here's another example:

```
5 3
+ echo
+ baker
+ apt
+ toe
+ ore
+ eat
+ cap
```



And a couple more puzzles:

```
3 3
+ ale
+ oat
+ zed
+ old
+ zoo
```

```
7 5
+ avocado
+ magnet
+ cedar
+ robin
+ chaos
+ buffalo
+ trade
+ lad
+ fun
- ace
- coat
```

15.3 Generating Ideas

- If running time & memory are not primary concerns, and the problems are small, what is the simplest strategy to make sure all solutions are found. Can you write a *simple* program that tries *all possibilities*?

- What variables will control the running time & memory use of this program? What is the order notation in terms of these variables for running time & memory use?

- What incremental (baby step) improvements can be made to the naive program? How will the order notation be improved?

15.4 Mapping Ideas to Code

- What are the key steps to solving this problem? How can these steps be organized into functions and flow of control for the main function?

- What information do we need to store? What C++ or STL data types might be helpful? What new classes might we want to implement?

15.5 Getting the Details Right

- What are the simplest test cases we can start with (to make sure the control flow is correct)?

- What are some specific (simple) corner test cases we should write so we won't be surprised when we move to bigger test cases?

- What are the limitations of our approach? Are there certain test cases we won't handle correctly?

- What is the maximum test case that can be handled in a reasonable amount of time? How can we measure the performance of our algorithm & implementation?