CSCI-1200 Computer Science II — Fall 2006
Lecture 14 — Problem Solving Techniques

Review from Lecture 13

- Merge sort
- Non-linear word search

Today's Class

- General problem solving techniques
- Today we will focus on the design and implementation of (relatively) short algorithms
- We'll consider three steps or stages to solving problems:
  1. Generating and Evaluating Ideas
  2. Mapping Ideas into Code
  3. Getting the Details Right

14.1 Generating and Evaluating Ideas

- Most importantly, play with examples! Can you develop a strategy for solving the problem? You should try any strategy on several examples. Is it possible to map this strategy into an algorithm and then code?
- Try solving a simpler version of the problem first and either learn from the exercise or generalize the result.
- Does this problem look like another problem you know how to solve?
- If someone gave you a partial solution, could you extend this to a complete solution?
- What if you split the problem in half and solved each half (recursively) separately?
- Does sorting the data help?
- Can you split the problem into different cases, and handle each case separately?
- Can you discover something fundamental about the problem that makes it easier to solve or makes you able to solve it more efficiently?
- Once you have an idea that you think will work, you should evaluate it: will it indeed work? are there other ways to approach it that might be better / faster? if it doesn't work, why not?

14.2 Mapping Ideas Into Code

- How are you going to represent the data? What is most efficient and what is easiest?
- Can you use classes to organize the data? What data should be stored and manipulated as a unit? What information needs to be stored for each object? What operations (beyond simple accessors) might be helpful?
- How can you divide the problem into units of logic that will become functions? Can you reuse any code you're previously written? Will any of the logic you write now be re-usable?
- Are you going to use recursion or iteration? What information do you need to maintain during the loops or recursive calls and how is it being "carried along"?
- How effective is your solution? Is your solution general? How is the performance? (What is the order notation of the number of operations)? Can you now think of better ideas or approaches?
- Make notes for yourself about the logic of your code as you write it. These will become your *invariants*; that is, what should be true at the beginning and end of each iteration / recursive call.

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 extreme cases; e.g., when the answer is on the extreme end of the data, when there are repeated values in the data, or when the data set is very small or very large?

14.4 Exercises: Practice using these Techniques on Simple Problems

- A perfect number is a number that is the sum of its factors. The first perfect number is 6. Let's write a program that finds all perfect numbers less than some input number n.

```
int main() {
  std::cout << "Enter a number: ";
  int n;
  std::cin >> n;
```

- Given a sequence of n floating point numbers, find the two that are closest in value.

```
int main() {

  float f;
  while (std::cin >> f) {

  }
```

- Now let's write code to remove duplicates from a sequence of numbers:

```
int main() {

  int x;
  while (std::cin >> x) {

  }
```

14.5 Example: Box-Packing

- Homework 6 asks you to write a program to solve simple box-packing problems. I hope everyone has started to **Generate and Evaluate Ideas**.
- Instead of thinking about how to code it up, think about how you would solve it by hand. Would you just randomly try things or would you organize your strategy somehow?

14.6 Example: Merge Sort

- In Lecture 13, I gave you the basic framework for the merge sort algorithm and we finished the implementation of the merge helper function. How did we **Map Ideas Into Code**?
- What invariants can we write down within the `merge_sort` and `merge` functions? Which invariants can we test using assertions? Which ones are too expensive (i.e., will affect the overall performance of the algorithm)?

```
// We split the vector in half, recursively sort each half, and
// merge the two sorted halves into a single sorted interval.
template <class T>
void mergesort(int low, int high, vector<T>& values, vector<T>& scratch) {
  if (low >= high) return;
  int  mid = (low + high) / 2;

  mergesort(low, mid, values, scratch);
  mergesort(mid+1, high, values, scratch);

  merge(low, mid, high, values, scratch);

}


// Non-recursive function to merge two sorted intervals (low..mid & mid+1..high)
// of a vector, using "scratch" as temporary copying space.
template <class T>
void merge(int low, int mid, int high, vector<T>& values, vector<T>& scratch) {
  int i=low, j=mid+1, k=low;

  // while there's still something left in one of the sorted subintervals...
  while (i <= mid && j <= high) {

    // look at the top values, grab the smaller one, store it in the scratch vector
    if (values[i] < values[j]) {
      scratch[k] = values[i]; ++i;
    } else {
      scratch[k] = values[j]; ++j;
    }
    ++k;
  }

  // Copy the remainder of the interval that hasn't been exhausted
  for ( ; i<=mid; ++i, ++k ) scratch[k] = values[i];  // low interval
  for ( ; j<=high; ++j, ++k ) scratch[k] = values[j]; // high interval

  // Copy from scratch back to values
  for ( i=low; i<=high; ++i ) values[i] = scratch[i];

}
```

14.7 Example: Nonlinear 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;
}
```

14.8 Exercise: Maximum Subsequence Sum

- Problem: Given is a sequence of n values, $a_0, \ldots, a_{n-1}$, find the maximum value of $\sum_{i=j}^{k} a_i$ over all possible subsequences $j \ldots k$.
- For example, given the integers: $14, -4, 6, -9, -8, 8, -3, 16, -4, 12, -7, 4$
  The maximum subsequence sum is: $8 + (-3) + 16 + (-4) + 12 = 29$.
- Let's write a first draft of the code, and then talk about how to make it more efficient.

```
int main() {
  std::vector<int> v;
  int x;
  while (std::cin >> x) {
    v.push_back(x);
  }
```

14.9 Problem Solving Strategies

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.

14.10 Design Example: Conway's Game of Life

Let's design a program to simulate Conway's Game of Life. Initially, due to time constraints, we will focus on the main data structures of needed to solve the problem.

Here is an overview of the Game:

- We have an infinite two-dimensional grid of cells, which can grow arbitrarily large in any direction.
- We will simulate the life & death of cells on the grid through a sequence of generations.
- In each generation, each cell is either alive or dead.
- At the start of a generation, a cell that was dead in the previous generation becomes alive if it had exactly 3 live cells among its 8 possible neighbors in the previous generation.
- At the start of a generation, a cell that was alive in the previous generation remains alive if and only if it had either 2 or 3 live cells among its 8 possible neighbors in the previous generation.
  - With fewer than 2 neighbors, it dies of "loneliness".
  - With more than 3 neighbors, it dies of "overcrowding".
- Important note: all births & deaths occur simultaneously in all cells at the start of a generation.
- Other birth / death rules are possible, but these have proven to be a very interesting balance.
- Many online resources are available with simulation applets, patterns, and history. For example:

```
http://www.math.com/students/wonders/life/life.html
http://www.radicaleye.com/lifepage/patterns/contents.html
http://www.bitstorm.org/gameoflife/
```

Applying the Problem Solving Strategies

In class we will brainstorm about how to write a simulation of the Game of Life, focusing on the representation of the grid and on the actual birth and death processes.

Understanding the Requirements

We have already been working toward understanding the requirements. This effort includes playing with small examples by hand to understand the nature of the game, and a preliminary outline of the major issues.

Getting Started

- What are the important operations?
- How do we organize the operations to form the flow of control for the main program?
- What data/information do we need to represent?
- What will be the main challenges for this implementation?

Details

- New Classes? Which STL classes will be useful?

Testing

- Test Cases?