

CSCI-1200 Data Structures — Spring 2016

Lecture 14 — Problem Solving Techniques

Review from Lecture 13

- Rules for writing recursive functions:
 1. Handle the base case(s).
 2. Define the problem solution in terms of smaller instances of the problem. Use *wishful thinking*, i.e., if someone else solves the problem of `fact(4)` I can extend that solution to solve `fact(5)`. This defines the necessary recursive calls. It is also the hardest part!
 3. Figure out what work needs to be done before making the recursive call(s).
 4. Figure out what work needs to be done after the recursive call(s) complete(s) to finish the computation. (What are you going to do with the result of the recursive call?)
 5. Assume the recursive calls work correctly, but make sure they are progressing toward the base case(s)!
- Merge sort
- Non-linear maze search

Today's Class

- Today we will discuss how to design and implement algorithms using three steps or stages:
 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 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?

14.4 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: Merge Sort

- In Lecture 13, we saw 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];
}
```

14.6 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;
}

```

14.7 Exercise: Maximum Subsequence Sum

- Problem: Given is a sequence of n values, a_0, \dots, a_{n-1} , find the maximum value of $\sum_{i=j}^k a_i$ over all possible subsequences $j \dots k$.
- 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);
    }
}

```

14.8 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

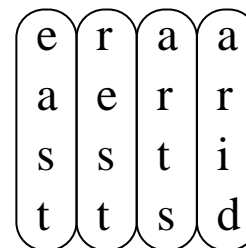
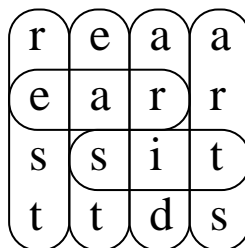
- For larger programs and programs requiring sophisticated classes / functions, these steps may need to be repeated several times over.
- Depending on the problem, some of these steps may be more important than others.
 - For some problems, the data / information representation may be complicated and 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.

14.9 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 that should *not* appear anywhere in the grid. Here’s an example:

Input:

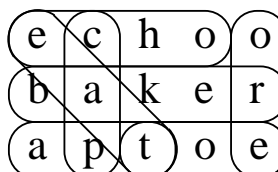
```
4 4
+ arts
+ arid
+ east
+ rest
- ear
- at
- sit
```



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 are available at http://www.cs.rpi.edu/~thompw4/CSCI-1200/Spring2016/inverse_word_search.zip.

14.10 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?

14.11 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?

14.12 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?

Code for the inverse word search can be found at http://www.cs.rpi.edu/~thompw4/CSCI-1200/Spring2016/inverse_word_search.zip.

14.13 Another Example: 8-Queens

- The inverse word search and battleship contest programs are examples of *backtracking*.
- The idea behind *backtracking* is to find a consistent partial solution to a problem and then try to extend the partial solution to a complete solution by executing a recursive step. If the recursive step fails to find a solution, it returns to the previous state and the algorithm tries again from a new consistent partial solution.
- Backtracking takes "1 step forward and n steps backward".
- The **8 queens problem** is a classic example of a problem solved by backtracking.
- The queen is the most powerful piece on the chessboard. It can move horizontally, vertically, and diagonally.
- In the 8 queens problem, we attempt to position 8 queens on a chessboard in such a way that no two can attack each other.
- On an 8 X 8 chess board, there are 4,426,165,368 ways to arrange 8 queens.
- We want to build a solution position by position starting at *row 0* and *col 0*.

- In order to find a square to position a new queen, we check to see if the current spot lies on the same row, column, or diagonal as an already placed queen.
- If the spot cannot be attacked, we add a queen on that spot and recursively try to add more queens.
- Backtracking occurs if the recursive call returns a failure. In that case, we remove the queen and try a new position.
- if the spot is not safe, we try the next row or column until we either find a safe spot or exhaust the board.

```

#ifndef __QUEEN_H__
#define __QUEEN_H__

#include <iostream>

class Queen {
public:
    Queen(int row, int col) : row_(row), col_(col) {}

    int getRow() const {return row_;}
    int getCol() const {return col_;}
    void Print() const {std::cout << "(" << row_ << ", " << col_ << ")" << std::endl;}

    void setPosition(int row, int col) {row_ = row; col_ = col;}

private:
    int row_;
    int col_;
};

#endif

////////////////////////////////////

#include <iostream>
#include <vector>
#include "queen.h"

void PrintBoard(const std::vector<Queen> queens,
int num_rows, int num_cols) {
    std::vector<std::vector<char> > grid(num_rows, std::vector<char>(num_cols, '_'));

    for (std::vector<Queen>::const_iterator it = queens.begin();
        it != queens.end(); ++it) {
        (*it).Print();
        grid[(*it).getRow()][(*it).getCol()] = 'Q';
    }

    for (int r = 0; r < num_rows; ++r) {
        for (int c = 0; c < num_cols; ++c) {
            std::cout << grid[r][c];
        }
        std::cout << std::endl;
    }
}

// Is this position safe? - search for an attack by other queens
bool SafeSquare(const std::vector<Queen> queens,
int row, int col) {
    for (std::vector<Queen>::const_iterator it = queens.begin();
        it != queens.end(); ++it) {
        int qrow = (*it).getRow();
        int qcol = (*it).getCol();

        if (qrow == row)
            return false;
    }
}

```

```

        else if (qcol == col)
            return false;
        else if (qcol - qrow == col - row || qcol + qrow == col + row)
            return false;
    }

    return true;
}

// Place a new queen
bool PlaceQueens(std::vector<Queen>& queens,
    int num_rows, int num_cols) {
    // done if we have a queen on each row and column
    if (int(queens.size()) == num_rows)
        return true;

    // search for a new spot
    for (int r = 0; r < num_rows; ++r) {
        for (int c = 0; c < num_cols; ++c) {
            if (SafeSquare(queens, r, c)) {
                Queen q(r, c); // add a new queen
                queens.push_back(q);
                if (PlaceQueens(queens, num_rows, num_cols))
                    return true;
                queens.pop_back(); // search failed, try the next spot
            }
        }
    }

    return false;
}

int main() {
    std::vector<Queen> queens;
    int num_rows = 8;
    int num_cols = 8;

    // Place first queen
    Queen q(0, 0);
    queens.push_back(q);

    if (PlaceQueens(queens, num_rows, num_cols))
        PrintBoard(queens, num_rows, num_cols);
    else
        std::cout << "No Solution." << std::endl;

    return 0;
}

```

The source is available at http://www.cs.rpi.edu/~thompw4/CSCI-1200/Spring2016/notes_2016_03_22.txt.