CSCI-1200 Data Structures — Fall 2024
Lecture 12 — Problem Solving Techniques

Test 2 Information

- Test 2 will be held **Thursday, October 17th, 2024 from 6:00-7:50pm**.
  - Student's assigned test room, row, and seat assignments will be re-randomized. Test 2 seating assignments will be posted and emailed Tuesday, October 15th.
  - If you did not already indicate your right- or left-handedness for Test 1, you may do that ASAP for Test 2 through the "Left-/Right- Handed Exam Seating" gradeable.
  - No make-ups will be given except for pre-approved absence or emergency or illness, and a written excuse from the Dean of Students or the Student Experience office or the RPI Health Center will be required.
  - If you have a letter from Disability Services for Students and you have not already emailed it to `ds_instructors@cs.rpi.edu`, please do so IMMEDIATELY. Meredith Widman will be in contact to make arrangements for your test accommodations.
- Coverage: Lectures 1-12, Labs 1-8, and Homeworks 1-5.
  - Practice problems from previous tests are available on the course website.
  - Sample solutions to the practice problems will be posted on Wednesday morning.
  - The best way to prepare is to completely work through and write out your solution to each problem, *before* looking at the answers.
  - You should practice timing yourself as well. The test will be 110 minutes and there will be 100 points. If a problem is worth 25 points, budgeting 25 minutes for yourself to solve the problem is a good time management technique.
  - The exam will be handwritten on paper. You're also encouraged to practice *legibly* handwriting your answers to the practice problems on paper.
- OPTIONAL: Prepare a 2 page, black & white, 8.5x11", portrait orientation .pdf of notes you would like to have during the test. This may be digitally prepared or handwritten and scanned or photographed. The file may be no bigger than 2MB. You will upload this file to Submitty gradeable "Test 2 Notes Page (Optional)" before Wednesday October 16th @11:59pm. We will print this and attach it to your test. No other notes may be used during the test.
- Going in to the test, you should know what big topics will be covered on the test. As you skim through the problems, see if you can match up those big topics to each question. Even if you are stumped about how to solve the whole problem, or some of the details of the problem, make sure you demonstrate your understanding of the big topic that is covered in that question.
- Re-read the problem statement carefully. Make sure you didn't miss anything.
- Additional Notes:
  - Please use the restroom before entering the exam room. Except for emergencies, students must remain in their seats until they are ready to turn in their exam. You may leave early if you finish the exam early.
  - Bring your Rensselaer photo ID card. We will be checking IDs when you turn in your exam.
  - Bring your own pencil(s) & eraser (pens are ok, but not recommended). The test *will* involve handwriting code on paper (and other short answer problem solving). Neat legible handwriting is appreciated. We will be somewhat forgiving to minor syntax errors – it will be graded by humans not computers :)
  - Do not bring your own scratch paper. The exam packet will include sufficient scratch paper.
  - Computers, cell-phones, smart watches, calculators, music players, headphones, etc. are not permitted. Please do not bring your laptop, books, backpack, etc. to the test room – leave everything in your dorm room. *Unless you are coming directly from another class or sports/club meeting.*

Review from Lecture 11

- 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)!
- Suggestions for Big "O" Notation
  1. Assign variable(s) to the data size (problem size) that will have an impact on the running time / memory usage of the problem.
  2. Study the code:
     - Identify the explicit `for` or `while` loops.
     - Identify implicit loops via function call recursion.
     - Look for calls to non-constant library/helper functions; for example, STL `sort` or STL `vector::erase`.
  3. Determine the Big "O" Notation of each part, and the number of times each loop will execute.
  4. Combine the parts:
     - Loops in series will *add*.
     - Nested loops will *multiply*.
     - Draw a tree or make a table to understand the pattern of recursion and combine the parts - *this can look very different for different problems!*
  5. Simplify your answer.
- Standard Example Recursion: Binary Search – (Can easily be re-written iteratively)
- Non-trivial Recursion: Merge sort

Today's Class

- Finish Lecture 11: Another example of Recursion & Big O Notation: Non-linear maze search
- 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
- Design Example: Conway's Game of Life
- Problem Solving Example: Quicksort (& compare to Mergesort)

12.1 Generating and Evaluating Ideas

- **Ask questions!** Make notes of your questions as you read the problem.
  Can you answer them? Do a little research.
  Ask your lab study group. Ask your peers and colleagues.
  Ask your TA, instructor, interviewer, project manager, supervisor, etc.
- **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**?
- Does **sorting the data** help?
- What if you **split the problem in half** and solved each half (recursively) separately?
- 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 one or more ideas that you think will work,
  you should **evaluate your ideas**:
  - Will it indeed work?
  - Are there other ways to approach it that might be better / faster?
  - If it doesn't work, why not?

12.2 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) {

  }
```

12.3 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);
  }
```

12.4 Mapping Ideas Into Code

- How are you going to **represent the data** ?
- What structures are **most efficient** and what is **easiest**?
  Note: Might be different answers!
- Can you **use classes (object-oriented programming)** to organize the data?
  - What data should be stored and manipulated as a unit?
  - What information needs to be stored for each object?
  - What public 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 Big O 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.

12.5 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/`
  `http://en.wikipedia.org/wiki/Conway's_Game_of_Life`

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?

12.6 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?
- Did you **combine / format / return / print your final answer**?
  Don't forget to return the correct data from each function.

12.7 Example: Quicksort

- Quicksort (also known as the partition-exchange sort) is another efficient sorting algorithm. Like mergesort, it is a divide and conquer algorithm.
- 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.
- Let's walk through a small example first. Let's choose the middle element as the pivot on each iteration.

```
  2    9    5    1    6    3    8    4    7

[ 2    9    5    1   (6)   3    8    4    7 ]
[ 2    4    5    1   (6)   3    8    9    7 ]
[ 2    4    5    1    3   (6)   8    9    7 ]

[ 2    4   (5)   1    3 ]  6   [ 8   (9)   7 ]
[ 2    4    3    1   (5)]  6   [ 8    7   (9)]

[ 2   (4)   3    1 ]  5    6   [(8)   7 ]  9
[ 2    1    3   (4)]  5    6   [ 7   (8)]  9

[ 2   (1)   3 ]  4    5    6   [(7)]  8    9
[(1)   2    3 ]  4    5    6    7    8    9
  1  [(2)   3 ]  4    5    6    7    8    9
  1    2  [(3)]  4    5    6    7    8    9
  1    2    3    4    5    6    7    8    9
```

```
// Choose a "pivot" and rearrange the vector.  Returns the location of the
// pivot, separating top & bottom (hopefully it's near the halfway point).
int partition(vector<double>& data, int start, int end, int& swaps) {
  int mid = (start + end)/2;
  double pivot = data[mid];















  }
}

void quickSort(vector<double>& data, int start, int end) {
  if(start < end) {
    int pIndex = partition(data, start, end);
    // after calling partition, one element (the "pivot")
    // will be at its final position
    quickSort(data, start, pIndex-1);
    quickSort(data, pIndex+1, end);
  }
}
void quickSort(vector<double>& data) {
  quickSort(data,0,data.size()-1);
}
```

- What value should you choose as the pivot? What are our different options?

- What is the big O notation for the running time of this algorithm?
  What is the big O notation for the additional memory use of this algorithm?

- What is the best case for this algorithm? What is the worst case for this algorithm?

- Compare the design of Quicksort and Mergesort. What is the same? What is different?

12.8 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.