CSCI-1400 Data Structures — Fall 2025
Lecture 14 — Problem Solving Techniques

Test 2 Information

- Test 2 will be held **Thursday, October 23rd, 2025 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 on Tuesday, October 21st.
  - If you did not already indicate your right- or left-handedness for Test 1, please do that ASAP for Test 2 through the "Left-/Right- Handed Exam Seating" gradeable.
  - There will be no make-up exam. If you miss one exam due to illness in the course, your next exam will count double in the final semester average. A formal excused absense from the Dean of Students Office or the RPI Health Center is required if you miss two exams or if you miss the final exam.
  - 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 1-15, 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 22nd @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.*
- SASE https://saserpi.org/ is hosting a Data Structures Exam 2 Review Session on Sunday October 19th from 6-8pm in Low 4050. All are welcome! Bring your questions :)

Review from Lecture 13

- Review Recursion vs. Iteration: Binary Search
- Suggestions for Big "O" Notation
- Suggestions for writing recursive functions:
- Non-trivial Recursion: Merge sort
- Non-trivial Recursion: Non-linear word 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
- Problem Solving Example: Conway's Game of Life
- Problem Solving Example: Quicksort (& compare to Mergesort)
- Problem Solving Example: Inverse Word Search

14.1 Stage 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 to solve a **simpler version of the problem** first and 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?
- 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?

14.2 Practice "Generating and Evaluating Ideas"

- 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) {

  }
```

- 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 start with a straightforward implementation. Then, we can think about how to make it more efficient.

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

14.3 Stage 2: Mapping Ideas Into Code

- How are you going to **represent the data** ?
- What structures are **most efficient** and what is **easiest**? *Note: They might not have the same answer!*
- 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?
- **What is the expected 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.

14.4 Practice "Mapping Ideas into Code": Conway's "Game of Life"

http://en.wikipedia.org/wiki/Conway's_Game_of_Life
http://www.math.com/students/wonders/life/life.html
http://www.radicaleye.com/lifepage/patterns/contents.html
http://www.bitstorm.org/gameoflife/

A brief overview of the Conway's "Game of Life":

- 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.

Let's focus different choices for the data structures necessary to store, update, and display the simulation.

- 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?
- Should we create new classes?
- Which basic datatypes and/or STL containers will be useful?

14.5 Stage 3: Getting the Details Right

- Has everything been **initialized** correctly?
  E.g., boolean flag variables, accumulation variables, max / min variables?
- Is the **logic of your conditionals** correct? Walk through several examples by hand.
- Do you have the **bounds on the loops** correct? Should you end at n, $n-1$ or $n-2$?
- An invariant is a condition or property that remains true at certain points during a program's execution. For example, something that we assume to be true of the arguments passed into a specific function. Or something that is true at the start of each repeat of the body of a for loop.

  Document the assumptions you are making, **formalize the invariants**. When possible **use assertions to explicitly test your invariants**. *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.**

14.6 Practice "Getting the Details Right": 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 on the intervals above & below the pivot.
- 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    (5)    1     3  ]  6     8     9     7
[  2     4     3     1    (5) ]  6     8     9     7
[  2     4     3     1    (5) ]  6     8     9     7

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

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

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

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

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

```
template <class T>
int partition(std::vector<T>& data, int start, int end, int& partitions, int& compares, int& swaps) {
  partitions++;
  // We have a few options for choosing the pivot:
  // int pivot = start;
  // int pivot = end;
  int pivot = (start+end)/2;
  // int pivot = start+std::rand()%(end+1-start);
  print("partition: ",data,start,end,pivot);
  // Simultaneously walk from both ends of the range looking for a pair of values to swap
  int i = start;
  int j = end;
  while (true) {
    // Search the low range for an element greater than the pivot
    while (data[i] <= data[pivot] && i < pivot) { compares++; i++; }
    // Search the high range for an element less than the pivot
    while (data[j] >= data[pivot] && j > pivot) { compares++; j--; }
    // Swap the values
    if (i < j) {
      compares++; swaps++; std::swap(data[i], data[j]);
      // If the pivot value was swapped, update the pivot index
      if (i == pivot) pivot=j;
      else if (j == pivot) pivot=i;
      print("  swap "+std::to_string(i)+"&"+std::to_string(j)+" ",data,start,end,pivot);
    }
    else { compares++; break; }
  }
  print("finished:  ",data,start,end,pivot);
  return j;
}

template <class T>
void quicksort(std::vector<T>& data, int start, int end, int& partitions, int& compares, int& swaps) {
  if (start < end) {
    // partition the data -- after this call the PIVOT will be in its final position
    int pivot = partition(data, start, end, partitions, compares, swaps);

    // DEBUGGING: verify partition is complete
    for (int i = start; i < pivot; i++) { assert (data[i] <= data[pivot]); }
    for (int i = pivot+1; i <= end; i++) { assert (data[i] >= data[pivot]); }

    // recurse on data before and after the pivot
    quicksort(data, start, pivot-1, partitions, compares, swaps);
    quicksort(data, pivot+1, end, partitions, compares, swaps);

    // DEBUGGING: verify range is sorted
    for (int i = start; i < end; i++) { assert (data[i] <= data[i+1]); }
  }
}

// DRIVER FUNCTION
template <class T>
void quicksort(std::vector<T>& data) {
  int num_partitions = 0;
  int num_compares = 0;
  int num_swaps = 0;
  print("before:    ",data);
  quicksort(data, 0, data.size()-1, num_partitions, num_compares, num_swaps);
  print("after:     ",data);

  // ANALYSIS:  measure the cost of the algorithm
  std::cout << "num partitions = " << num_partitions << std::endl;
  std::cout << "num compares = " << num_compares << std::endl;
  std::cout << "num swaps = " << num_swaps << std::endl;
}
```

- How do we test & debug the correctness of the implementation?
  What are the invariants? What should be true before and after each call to `quicksort` and `partition`?
  We can add *temporary* debugging statements to verify these properties.
  Comment out or remove these statements before large-scale, performance-critical usage!

- What are the "corner cases" for sorting? For this algorithm specifically?

- 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?
  We implemented this algorithm for an STL `vector`. Could we also implement it for a linked list?

Quicksort Performance Data:

input	pivot choice	partition calls	comparisons	swaps
1000 random	middle	672	13,765	2,441
1000 random	first	689	15,041	3,694
1000 random	random	687	15,633	2,806
1000 sorted	middle	511	8,498	0
1000 sorted	first	999	500,499	0
1000 sorted	random	670	11,607	0
1000 reverse sorted	middle	511	8,998	500
1000 reverse sorted	first	999	500,999	500
1000 reverse sorted	random	668	11,816	500

14.7 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.

14.8 A "Homework 6" from the Past: 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:

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

r	e	a	a
e	a	r	r
s	s	i	t
t	t	d	s

e	r	a	a
a	e	r	r
s	s	t	i
t	t	s	d

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.

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

e	c	h	o	o
b	a	k	e	r
a	p	t	o	e

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

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

14.9 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 big O 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 big O notation be improved?

14.10 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?

14.11 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?