Announcements: Test 2 Information

- **Thursday, Oct 22nd, 2020 from 6:55-8:45pm America / New York.**

- Coverage: Lectures 1-14, Labs 1-8, and Homeworks 1-5.

- The procedures will be the same as for Test 1. See: [https://www.cs.rpi.edu/academics/courses/fall20/csci1200/lectures/05_memory.pdf](https://www.cs.rpi.edu/academics/courses/fall20/csci1200/lectures/05_memory.pdf)

  - You may start up to 15 minutes early (to spread the load on Submitty). But you must make your final submission no more than 1 hour 50 minutes after you first loaded the test page.
  - You will lose 1 point per minute over your time limit (unless verified excuse).
  - If you have a personal emergency, complete and submit the test as fairly and promptly as possible, then email ds_instructors@cs.rpi.edu.

- No internet references, no communication.

- Closed book, closed notes, except for 2 pages (equivalent) of prepared notes.

- **Post Test Interview**
  - Early next week ALL STUDENTS will fill out a Submitty form to indicate availability for the interview on Thursday evening or Friday morning or Friday afternoon.
  - Approximately 30% of all students will be selected for post-exam interview. 
    *Even if you were interviewed for Test 1, you may be selected again for Test 2, or a future test.*

- 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.

**Test Taking Skills**

- Look at the point values for each problem, allocate time proportional to the problem points – Roughly 1 minute per point. (Don’t spend all of your time on one problem and neglect other big point problems). Practice these time management skills when doing the practice problems this weekend.

- Look at the size of the answer box & the sample solution code line estimate for each problem. If your solution is going to take a lot more space than the box allows, we are probably looking for the solution to a simpler problem or a simpler solution to the problem.

- 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.
Review of Lecture 12

- General Problem Solving Techniques:
  1. Generating and Evaluating Ideas
  2. Mapping Ideas into Code
  3. Getting the Details Right

- Small exercises to practice these techniques

- Problem Solving Strategies / Checksheet

Today!

- One more exercise from Lecture 12: Maximum Subsequence Sum
- Problem Solving Example: Quicksort (& compare to Mergesort)
- Design Example: Conway’s Game of Life
- Another Design Example: Inverse Word Search

13.1 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.

```c++
// 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];
```

```c++
```
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?

13.2 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.radicaley.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?

13.3 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

13.4 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?

13.5 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?

13.6 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?