CSci 160 Algorithms

Welcome!
Study of Algorithms

● Classical/important Data Structures and Algorithms
  - What are they?
  - Why do we care about them? What are some interesting/important applications of these DS/Algs?
  - How do they work? Main ideas, subtle details … in their design and implementations?
  - Why are the approaches correct and efficient?
    - What are alternative ways to implement them? What are the pros and cons...?
  - What to do when there are (minor) changes to the problem specifications?
    - Minor revision to existing approach, or a completely new approach?
  - ...

...
Study of Algorithms

- Algorithmic problem solving
  - Develop a systematic, effective approach for dealing with new problems and generating correct and efficient solutions
  - Learn from the classical data structures and algorithms
  - Break a big problem into more manageable parts (e.g. HW APIs)
  - Use different levels of abstractions: rough ideas → fine details
  - Use smaller cases, simpler variations to gain insights
  - Make connections to old problems
  - Adapt from existing solutions
  - ...
Study of Algorithms

• Important, Broadly Applicable Skills
  – Read/understand technical materials
  – Learn new materials mostly on your own
  – Think critically
  – Design, analyze and test your solutions thoroughly
  – Conduct your work in a disciplined and systematic manner

• Good Work Ethics
  – Work hard and effectively
  – Produce high-quality results within given time
  – Hold yourself to high academic and integrity standards
  – ...
Key Factors to Success

- Backgrounds, preparations
  - Programming Fundamentals
  - Computational Thinking and Problem Solving

- Academic Interests and Strengths
  - Interests in studying and developing correct and efficient algorithms, professional code, systematic process
  - Good analytical skills, critical thinking …

- Work Ethics → can be improved with your efforts
  - Work hard and effectively
  - Be disciplined, hold yourself to a high standard
  - Be determined, persistent, and optimistic
Hypothetical Situation

- Cs160 assignments: challenging/interesting problems, firm deadlines, rigorous testing

- Imagine that during an interview process, an applicant told a hiring manager that, for a challenging assignment, the applicant
  - rushed to do his assignment before the deadline,
  - submitted his code and report right before the submission site was closed,
    - Without testing his code, thinking through his solution, or checking his results/report
  - because he was so busy with many obligations for other courses and extra curriculum activities

- How likely do you think that the hiring manager will make an offer to the applicant, especially when there are other competitive applicants?

- If this applicant gets to develop a robot surgery software, will the applicant himself willing to be operated by the robot?
Key Factors to Success

• Take the ownership of your products and study
  – You are the only one that can determine your learning outcomes
  – Set a high standard to yourself, work hard and effectively

• Improve your understanding and problem solving skills by doing your learning the hard way, the only way
  – Study course materials, think through the topics, have discussions with students and staff ...
  – Solve problems yourself
    • watching other people solve problems and understanding other people's solution is very different from solving problems yourself

• We are here to help you
  – We help you by helping you learn how to fish, not by giving you a fish
  – Assignments: discussions, hints
  – Other problems: discussions, hints, solutions

• Use the available resources (incl. office hours, pretests, Piazza Q&A ...) to help yourself and work with us to make this course a success for your learning
## Course Organization

<table>
<thead>
<tr>
<th></th>
<th>Traditional Approach</th>
<th>Flipped Classroom</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Before class</strong></td>
<td>Student preparation: read slides/textbook (??)</td>
<td>Student preparation</td>
</tr>
<tr>
<td></td>
<td>(how well do students do this??)</td>
<td>* watch video, read slides/textbook</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* do summary and exercise problems</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* develop good understanding of overall/easy topics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(use office hours for help if needed)</td>
</tr>
<tr>
<td><strong>During Class</strong></td>
<td>Instructor gives a lecture:</td>
<td>* review major topics</td>
</tr>
<tr>
<td></td>
<td>* impossible to cover all topics or slides</td>
<td>* discuss hard/confusing issues</td>
</tr>
<tr>
<td></td>
<td>* need to spend time on easy materials</td>
<td>* solve problems</td>
</tr>
<tr>
<td></td>
<td>* have to skip some materials, incl. important/hard materials sometimes</td>
<td>→ in theory, helpful for students</td>
</tr>
<tr>
<td></td>
<td>* be able to respond to student questions</td>
<td>* better understanding</td>
</tr>
<tr>
<td><strong>After class</strong></td>
<td>Students study: reflect on course materials, figure out hard topics, solve problems</td>
<td>* less confusion</td>
</tr>
<tr>
<td></td>
<td>(how well do students do this??)</td>
<td>* more systematic and effective problem solving</td>
</tr>
</tbody>
</table>
Assignment Strategies

- Start early, finish early, be systematic and disciplined
- Understand the problem
  - Study and understand the info. in web pages and video
  - Be able to articulate the problem, explain examples and suggested approach (e.g. the virtual top and bottom)
  - Understand the given resources: sample test data, java code --> useful for testing (you need to figure out how)

- Good understanding
  - clear picture of overall process and individual classes: what they are supposed to do
  - rigorous and comprehensive testing plan
Assignment Strategies

- Devise a solution → the goal is to reach a correct and efficient solution
  - Be prepared to go through multiple iterations
  - Use and develop algorithmic problem solving strategies

- Pay close attention to functionality and performance requirements
- Figure out and describe your big ideas (English) --> fine details (pseudo-code)
  - Be very careful about your big ideas → largely determine the quality of your solution
- Figure out, choose and design proper data structures
  - What information to keep? How is the information used/changed?
  - Data structures → significant impact on solution efficiency
- General development sequence: big ideas → proper D.S. → ... → fine details,
  - revision and iteration along the way

- Think through, describe, analyze and test your algorithmic solution on paper
Assignment Strategies

- Implement the solution: be systematic and disciplined
  - Incremental implementation
  - Thorough testing along the way
  - Good coding styles
  - More testing, revisions, iterations
  - ...

- Look back, reflect on your work
  - What were the hard issues? How did I resolve them?
  - What other problems can I use my new approaches to solve?
  - What ineffective/incorrect solutions did I use?
  - What have I learned from this problem and solving process?
  - ...

Example: Percolation Assignment

- Understand the problem?
  - Underlying physics problem?
  - Class specifications? Percolation and PercolationStats
- Key pieces of information → data structures?
  - PercolationStats ← Percolation ← UF, 2D grid array
    (overall process) (individual functionality) (basic data structures)
    (utility class, high-level D.S.)
  
- GuitarHero ← GuitarString ← RingBuffer
  - RingBuffer: fixed size array based queue implementation

- Levels of abstraction: a client of UF uses UF public methods and does not worry about or have access to the underlying UF implementation
PreConditions and PostConditions

• Method/algorithm (functional) interface
  – **preconditions**: promised/expected conditions (properties) at the start of a method/algorithm
  – **postconditions**: promised/expected conditions (properties) at the end of a method/algorithm
  – Together, they specify WHAT is computed, not how
  – e.g. UF: union(int p, int q)
    • preconditions: id[ ] represents the current connected components
    • postconditions: id[ ] is the same as before if p and q are connected, or represents updated connected components, with p and q components merged into one
Data Structure Invariants

• For a data structure, its invariants (properties)
  – true at the beginning and end of every execution of a public method
  – Properties/constraints on internal data fields and method implementations
  – Related to method preconditions and postconditions

• Example: QuickFind
  – int[ ] id: id[i] is the component id of element i
Data Structure Invariants

• Example: QuickUnion
  – int[ ] id
    • id[i] is the parent id of element i
    • Elements in one connected component form a (directed, rooted-in) tree → the overall system is a forest in general
    • A node j is a root if and only if id[j] = j

• Example: Weighted QuickUnion
  – int[ ] id:
    • same invariants as in QuickUnion
    • Any (subtle) property from weighted union? (hint: yes. But what?)
  – Int[ ] size
    • size[i] is the size of a subtree (component subset) rooted at i
Data Structure Invariants

- Stack
  - (conceptual) top: operation point for push and pop

- Singly Linked List Based Implementation
  - Use a linked list to store all elements on a stack
  - the first node is the top
    - Question: assume that we keep track of both the first and last nodes of a linked list, why not use the last as top?

- Resizing Array Based Implementation
  - Item[ ]: Use a large enough array and the beginning part (0, N-1) to store stack elements
  - Int N: the size of the stack, i.e. the number of items in a stack
  - top: the last element in the stack part of the subarray
    - Question: why don't we use the first array element, a[0], as the top?

- What about fixed size array based implementation?