Sorting

• **Insertion Sort:**
  – Consider each element in the array, starting at the beginning.
  – Shift the preceding, already sorted, elements one place to the right, until finding the proper place for the current element.
  – Insert the current element into its place.
  – Worst-case time is $O(n^2)$.

• **Treesort:**
  – Insert the $n$ items one by one into a binary search tree.
  – Then do an inorder traversal of the tree.
  – For a basic BST, worst-case time is $O(n^2)$, but average time is $O(n \log n)$.
  – For a balanced BST, worst-case time is $O(n \log n)$, although code is more complicated.
Sorting (cont’d)

• **Heapsort:**
  – Insert the \( n \) items one by one into a heap.
  – Then remove the minimum element one by one. Elements will come out in sorted order.
  – Worst-case time is \( O(n \log n) \).

• **Mergesort:** Apply the idea of divide and conquer:
  – Split the input array into half.
  – Recursively sort the first half.
  – Recursively sort the second half.
  – Then merge the two sorted halves together.
  – Worst-case time is \( O(n \log n) \) like heapsort; however, it requires more space.
Object-Oriented Software Engineering

References:

- Standish textbook, Appendix C

Outline of material:

- Introduction
- Requirements
- Object-oriented analysis and design
- Verification and correctness proofs
- Implementation
- Testing and debugging
- Maintenance and documentation
- Measurement and tuning
- Software reuse
Small Scale vs. Large Scale Programming

**Programming in the small:** programs done by one person in a few hours or days, whose length is just a few pages (typically under 1000 lines of code).

**Programming in the large:** projects consisting of many people, taking many months, and producing thousands of lines of code. Obviously the complications are much greater here.

The field of software engineering is mostly oriented toward how to do programming in the large well. However, the principles still hold (although simplified) for programming in the small. It’s worth understanding these principles so that

- you can write better small programs and
- you will have a base of understanding for when you go on to large programs.
Object-Oriented Software Engineering

Software engineering studies how to define, design, implement and maintain software systems.

Object-oriented software engineering uses notions of classes, objects and inheritance to achieve these goals.

Why object-oriented?

• use of abstractions to control complexity, focus on one subproblem at a time

• benefits of encapsulation to prevent unwanted side effects

• power of inheritance to reuse software

Experience has shown that object-oriented software engineering

• helps create robust reliable programs with clean designs and

• promotes the development of programs by combining existing components.
Object-Oriented Software Engineering (cont’d)

Solutions to specific problems tend to be fragile and short-lived: any change to the requirements can result in massive revisions.

To minimize effects of requirement changes capture general aspects of the problem domain (e.g., student record keeping at a university) instead of just focusing on how to solve a specific problem (e.g., printing out all students in alphabetical order.)

*Usually the problem domain is fairly stable, whereas a specific problem can change rapidly.*

*If you capture the problem domain as the core of your design, then the code is likely to be more stable, reusable and adaptable.*

More traditional structured programming tends to lead to a strictly top-down way of creating programs, which then have rigid structure and centralized control, and thus are difficult to modify.
Object-Oriented Software Engineering (cont’d)

In OO analysis and design, identify the abstractions needed by the program and model them as classes. Leads to middle-out design:

- go downwards to flesh out the details of how to implement the classes
- go upwards to relate the classes to each other, including inheritance relationships, and use the classes to solve the problem

This approach tends to lead to decentralized control and programs that are easier to change. For instance, when the requirements change, you may have all the basic abstractions right but you just need to rearrange how they interact.

Aim for a core framework of abstract classes and interfaces representing the core abstractions, which are specialized by inheritance to provide concrete classes for specific problems.
Software Life Cycle

- inception: obtain initial ideas
  - requirements: gather information from the user about the intended and desired use

- elaboration: turn requirements into a specification that can be implemented as a program
  - analysis: identify key abstractions, classes and relationships
  - design: refine your class model to the point where it can be implemented
  - identify reuse: locate code that can be reused

- implementation
  - program and test each class
  - integrate all the classes together
  - make classes and components reusable for the future

- testing

- delivery and maintenance
Software Life Cycle (cont’d)

Lifecycle is not followed linearly; there is a great deal of iteration.

An ideal way to proceed is by iterative prototyping:

- implement a very simple, minimal version of your program
- review what has been achieved
- decide what to do next
- proceed to next iteration of implementation
- continue iterating until reaching final goal

This supports exploring the design and implementation incrementally, letting you try alternatives and correct mistakes before they balloon.
Requirements

Decide what the program is supposed to do before starting to write it. Harder than it sounds.

Ask the user

- what input is needed
- what output should be generated

Involve the user in reviewing the requirements when they are produced and the prototypes developed.

Typically, requirements are organized as a bulleted list.

Helpful to construct scenarios, which describe a sequence of steps the program will go through, from the point of view of the user, to accomplish some task.
Requirements (cont’d)

An example scenario to look up a phone number:

1. select the find-phone-number option
2. enter name of company whose phone number is desired
3. click search
4. program computes, to find desired number (do NOT specify data structure to be used at this level)
5. display results

Construct as many scenarios as needed until you feel comfortable, and have gotten feedback from the user, that you’ve covered all the situations.

This part of the software life cycle is no different for object-oriented software engineering than for non-object-oriented.
Object-Oriented Analysis and Design

Main objective: identify the classes that will be used and their relationships to each other.

Analysis and design are two ends of a spectrum: Analysis focuses more on the real-world modeling, while design focuses more on the programming aspects.

For large scale projects, there might be a real distinction: for example, several programming-level classes might be required to implement a real-world level “class”.

For small scale projects, there is typically no distinction between analysis and design: we can directly figure out what classes to have in the program to solve the real-world problem.
Object-Oriented Analysis and Design (cont’d)

To decide on the classes:

- Study the requirements, brainstorming and using common sense.

Look for *nouns* in the requirements: names, entities, things (e.g., courses, GPAs, names), roles (e.g., student), even strategies and abstractions.

These will probably turn into classes (e.g., student, course) and/or instance variables of classes (e.g., GPA, name).

See how the requirements specify interactions between things (e.g., each student has a GPA, each course has a set of enrolled students).

- Use an *analysis method*: a collection of notations and conventions that supposedly help people with the analysis, design, and implementation process. (Particularly aimed at large scale projects.)