## Today's topics

Designing and Implementing Algorithms
- Problem solving
- Pseudocode
- Java
- Syntax and Grammars

Upcoming
- More Java

Acknowledgements
- Marti Hearst, UC Berkeley
- David Smith, Georgia tech

Reading
- *Computer Science*, Chapter 5
- *Great Ideas*, Chapter 2
- Back of the Envelope Calculations, excerpt from Jon Bentley's *Programming Pearls*

## Problem Solving

*Programming is a strenuous exercise in problem solving*

G. Polya. *How to Solve It,* 2nd ed., Princeton University Press, 1957.

- Understand the problem
  - ➢ What are its parts? unknown, data, condition
  - ➢ Does the problem make sense? Is it feasible?
  - ➢ Think about the problem, get a sense of what it needs
- Make a plan
  - ➢ Find the connection between givens and result
  - ➢ What kind of problem is it? Is it familiar?
  - ➢ Think about generalizations, specializations, variants
- Carry out the plan
  - ➢ Check each step
- Examine the result
  - ➢ Does it make sense?

## Back of the envelope calculations

Jon Bentley. *Programming Pearls.* 2nd edition. A-W, 2000.

http://www.vendian.org/envelope/

- Engineering technique to approximate and check answers
  - ➢ Two answers are better than one
  - ➢ Quick checks
  - ➢ Rules of thumb
  - ➢ Practice
- Ad claims that salesperson drove 100,000 miles in a year. True?
- Newspaper article states that a United States quarter dollar coin has "an average life of 30 years." How can you check that claim?

## Why "back of the envelope" estimates?

- Often need to make rapid estimates
  - ➢ to eliminate candidate solutions
  - ➢ establish feasibility
  - ➢ sketch out potential trade-offs
- Most remember key numbers related to their field, not every detail
- Hence we need to estimate
  - ➢ *which* numbers are important
  - ➢ *values* of numbers needed
  - ➢ *how* to perform the calculation
- Emphasis is on "order of magnitude" estimates
  - ➢ to nearest factor of 10 (or 2)

## Orders of Magnitude

- How far away is home? Is it more like 1, or 10, or 100 miles?
  - Probably do not know exactly
  - Is it approximately "a couple", or "a few", or "a lot"
  - Estimate based on powers rather than multiples of 10
- How tall is your dorm? More like 1, 10, 100, 1000 feet?
  - 1 foot tall is like a doll house, so that's out
  - What do we know that is about 10 feet big? Hmm... People
  - If building is a couple of people high, 10 sounds good.
  - But that means 1000, would be 100 people high, so that's out
  - So 10 or 100 depending on how many people tall the building is
- Use orders of magnitude as brackets to find reasonable range

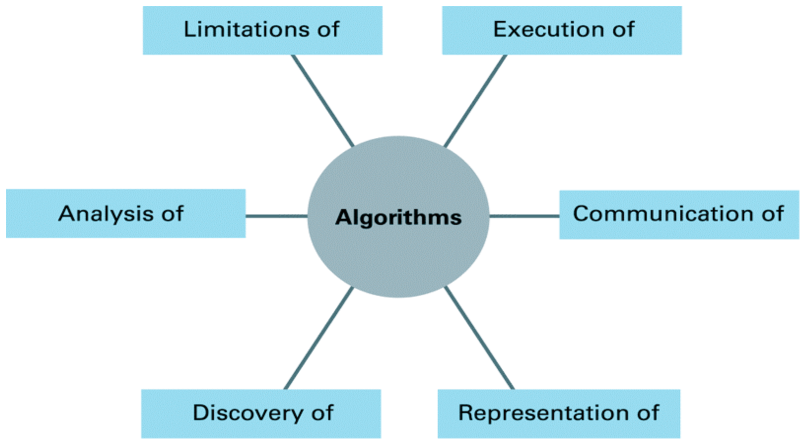
## Example: How many piano tuners in NYC

- Approximately how many people are in New York City?
  - 10,000,000
- Does every individual own a piano?
  - No
- Reasonable to assert "individuals do not own pianos; families do"?
  - Yes
- About how many families are there in a city of 10 million people?
  - Perhaps there are 2,000,000 families
- Does every family own a piano?
  - No
- Perhaps one out of every five does
  - That would mean there are about 400,000 pianos in NYC

## Estimation General Principles

- Recall Einstein's famous advice
  - Everything should be made as simple as possible, but no simpler
- Do not worry about constant factors of 2, π, etc.
  - Round to "easy" number or nearest order of magnitude
- Guess numbers you do not know
  - Within bounds of common sense (accuracy increases with experience)
- Adjust geometry, etc., to suit you
  - Assume a cow is spherical if it helps
- Extrapolate from what you do know
  - Use ratios to assume unknown value is similar to known quantity
- Apply a 'plausibility' filter
  - If answer seems unbelievable, it probably is
  - Can usually set range of reasonable values that indicates major mistake (e.g., speed cannot be faster than light!)

## Central role of algorithms in CS



## What's wrong with this algorithm?

**(From back of shampoo bottle)**

**<u>Directions:</u>**

**Wet Hair**
**Apply a small amount of shampoo**
**Lather**
**Rinse**
**Repeat**

## Properties of good algorithms

- **Good algorithms must be**
  - **Correct**
  - **Complete**
  - **Precise**
  - **Unambiguous**
- **And should be**
  - **Efficient**
  - **Simple**
  - **Contain levels of abstraction**

An algorithm is an ordered set
of unambiguous, executable steps,
defining a terminating process.

## Algorithms

- **Hand-waving not allowed!**
- **Specifying algorithms requires you to say what is really involved in making it work.**
- **Example:**
  - **How does a computer work?**
  - **Hand-wave: zeros & ones**
  - **Real answer: see later part of class.**
- **You learn to know when you don't know**
  - *"I know nothing except the fact of my ignorance."*
  - **Socrates, from Diogenes Laertius, Lives of Eminent Philosophers**

## Describing Algorithms

Pictures

More easily expressed ↑

Natural language (English)

Pseudo-code

↓ More precise

Specific high-level programming language

## Pseudocode

- A shorthand for specifying algorithms
- Leaves out the implementation details
- Leaves in the essence of the algorithm

```
procedure Greetings
Count ← 3;
while (Count < 0) do
    (print the message "Hello" and
     Count ← Count +1)
```

- What does this algorithm do?
- How many times does it print Hello?

## Sequential search

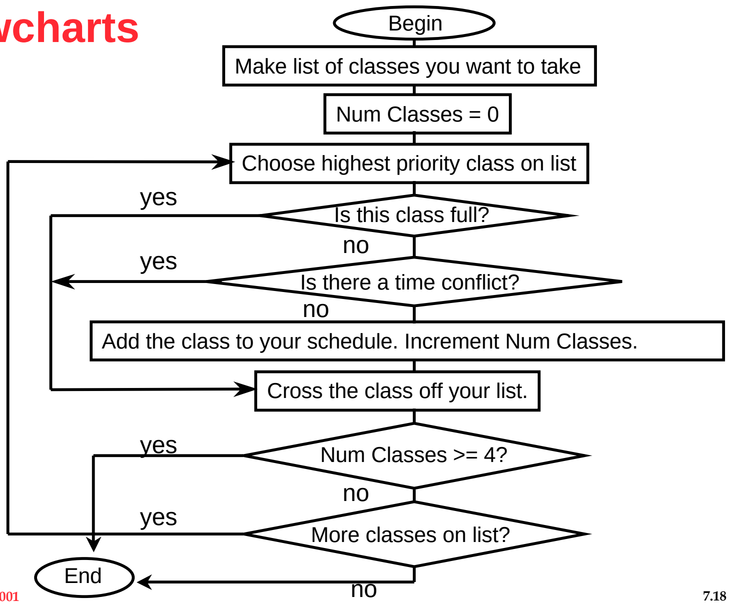
```
procedure Search (List, TargetValue)
if (List empty)
    then
      (Declare search a failure)
    else
      (Select the first entry in List to be TestEntry;
       while (TargetValue > TestEntry and
                   there remain entries to be considered)
            do (Select the next entry in List as TestEntry.);
       if (TargetValue = TestEntry)
            then (Declare search a success.)
            else (Declare search a failure.)
      ) end if
```

## Picking courses

1. Make a list of courses you want to register for, in order of priority
2. Start with empty schedule. Number of courses = 0.
3. Choose highest priority class on list.
4. If the chosen class is not full and its class time does not conflict with classes already scheduled, then register for the class (2 steps):
   1. Add the class to the schedule
   2. Increment the number of classes scheduled

5. Cross that class off of your list.
6. Repeat steps 3 through 5 until the number of classes scheduled is >= 4, or until all classes have been crossed out.
7. Stop.

## Flowcharts


Begin
Make list of classes you want to take
Num Classes = 0
Choose highest priority class on list
yes
Is this class full?
no
yes
Is there a time conflict?
no
Add the class to your schedule. Increment Num Classes.
Cross the class off your list.
yes
Num Classes >= 4?
no
yes
More classes on list?
End
no

## Programming Primitive Operations

- Assign a value to a variable
- Call a method
- Arithmetic operation
- Comparing two numbers
- Indexing into an array
- Following an object reference
- Returning from a method

## Components of Computing Algorithms

Any computing algorithm will have AT MOST five kinds of components:

- Data structures to hold data
- Instructions change data values
- Conditional expressions to make decisions
- Control structures to act on decisions
- Modules to make the algorithm manageable by abstraction, i.e., grouping related components

## Java!

- Java is a buzzword-enabled language
- From Sun (the developers of Java),

  *"Java is a simple, object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, high performance, multi-threaded, and dynamic language."*

- What do all of those terms mean?

## A Java Program

```
import java.awt.*;

public class HelloWorld extends
  java.applet.Applet
{
  TextField m1;
  public void init()
  {
    m1 = new TextField(60);
    m1.SetText(“Hello World”);
    add(m1);
  }
}
```

## Definitions

- *Algorithm*: ordered set of unambiguous executable steps, defining a terminating process
- *Program*: instructions executed by a computer
- *Applet*: Java program that is executed in a program such as `appletviewer` or a Java-enabled web browser
- *Class*: family of components sharing common characteristics consisting of:
  - *Data*: information
  - *Method*: functionality
- *Object*: instance of a class
- *Variable*: represent value stored in computer memory. A variable must be defined or declared before being used
  - Sometimes synonymous with object

## Grammar

- English and other natural languages have structure

```
<S> => <NOUN-PHRASE> <VERB-PHRASE>
=> <NOUN> | <ARTICLE> <NOUN> | <PP>
<VERB-PHRASE> => <VERB> | <VERB> <NOUN-PHRASE>
<NOUN> => DOG | FLEAS | PERSON | ...
<VERB> => RAN | BIT | ...
```

- Process of taking sentence and fitting it to grammar is called *parsing*

```
DOG BIT PERSON
<NOUN> <VERB> <NOUN>
<NOUN-PHRASE> <VERB-PHRASE>
<S>
```

- Parsing English is complex because of *context dependence*

## Formal specifications

- Need a precise notation of syntax of a language
- Grammars can be used for generation and also can be used
- Context-free grammars

<name> => *sequence of letters and/or digits that begins with a letter*

<name> => guessB

<name> => msg42

- Substitute as many times as necessary. All legal statements can be generated this way
  - Want `person = firstn + " " + lastn;`
  - `How do we get this from our grammar?`

## A Grammar for Java

- Need a set of rules
- Our first one was a good start:
  - <name> => *any string of alphanumeric symbols that begins with a letter*
- Let's add something to define a simple statement:
  - <statement> => <name> = <expression> ;
- And then work on the details:
  - <expression> => <string-expression> | <int-expression> | <oth-expression>
  - <string-expression> => <string>
  - <string> => <name>
  - <string> => *"any sequence of charcters"*

## A Simple Statement

- Now have enough to generate a statement like: `msg = "hello";`
  - Start with:

  <statement> => <name> = <expression> ;
  - Then using: <name> => *any string of alphanumeric symbols that begins with a letter*

  `msg` = <expression> ;
  - Then, using: <expression> => <string-expression> | <int-expression> | <oth-expression>

  `msg` = <string-expression> ;
  - Using: <string-expression> => <string>

  `msg` = <string> ;
  - Using: <string> => *"any sequence of charcters"*

  `msg = "hello" ;`

## A Grammar for Java

- Including more rules to describe programs we have:
  1. <name> => *any string of alphanumeric symbols that begins with a letter*
  2. <statement> => <name> = <expression> ;
  3. <statement> => <name> = `new` <class>(<arguments>);
  4. <statement> => <name>.<method>(<arguments>); | <method>(<arguments>);
  5. <arguments> => *possibly empty list of* <expression>*s separated by commas*
  6. <expression> => <string-expresseion> | <int-expression> | <oth-expression>
  7. <string-expression> => <string-expression> + <string-expression>
  8. <string-expression> => <string>
  9. <string> = "any sequence of characters"
  10. <string> = <name>

## Using our Grammar

- Use this to generate: `person = firstn + " " + lastn;`

| Rule # | Statement being Generated |
|---|---|
| 2: | <statement> => <name> = <expression>; |
| 1: | <statement> => `person` = <expression>; |
| 6: | <statement> => `person` = <str-expression>; |
| 7: | <statement> => `person` = <str-expression> + <str-expression>; |
| 8: | <statement> => `person` = <string> + <str-expression>; |
| 10: | <statement> => `person` = <name> + <str-expression>; |
| 1: | <statement> => `person = firstn +` <str-expression>; |
| 7: | <statement> => person = firstn + <str-expression> + <str-expression>; |
| 8: | <statement> => `person = firstn +` <string> + <str-expression>; |
| 9: | <statement> => `person = firstn + " "` + <str expression>; |
| 8: | <statement> => `person = firstn + " "` + <string>; |
| 10: | <statement> => `person = firstn + " "` + <name>; |
| 1: | <statement> => <statement> => `person = firstn + " " + lastn;` |

## Proving Grammatical Correctness

- Why go through the process we went through?
  - Shows that desired statement can be generated from this grammar
- Actually *proves* that the statement is grammatically correct!
  - Same rigor as a mathematical proof
- (Doesn't prove that logic is correct, though)

- Actually need more rules to handle the level of Java we've covered so far
  - Summary of rules shown on pages 78-79 of *Great Ideas*
  - Also give an example for a complete applet
  - Too long to go through in class – Please Read!