CS 51 Computer Science Principles

APCSP Module 3: Data, Internet, Computer and Programming

Unit 3: Programming and Algorithms

LECTURE 10 APP LAB OVERVIEW

DR. ERIC CHOU

IEEE SENIOR MEMBER

Overview

LECTURE 1

Topics

1. Variables and Assignments
2. Data Abstraction
3. Mathematical Expressions
4. Strings
5. Boolean Expressions
6. Conditionals
7. Nested Conditionals
8. Iteration
9. Developing Algorithms
10. Lists
11. Binary Search
12. Calling Procedures
13. Developing Procedures
14. Parts of a Procedure
15. **Libraries**
16. **Random Values**
17. **Simulation**
18. **Algorithmic Efficiency**
19. **Undecidable Problems**

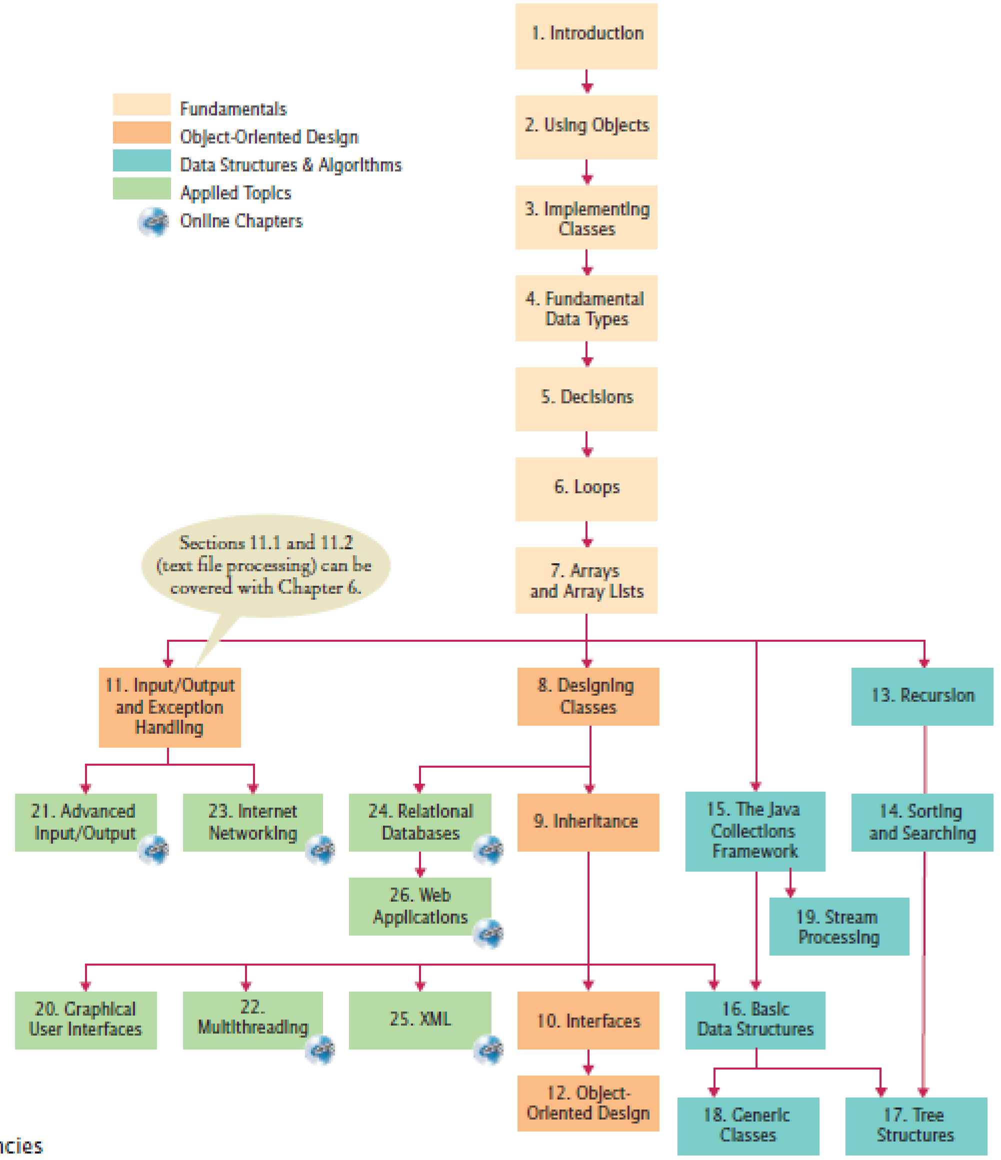
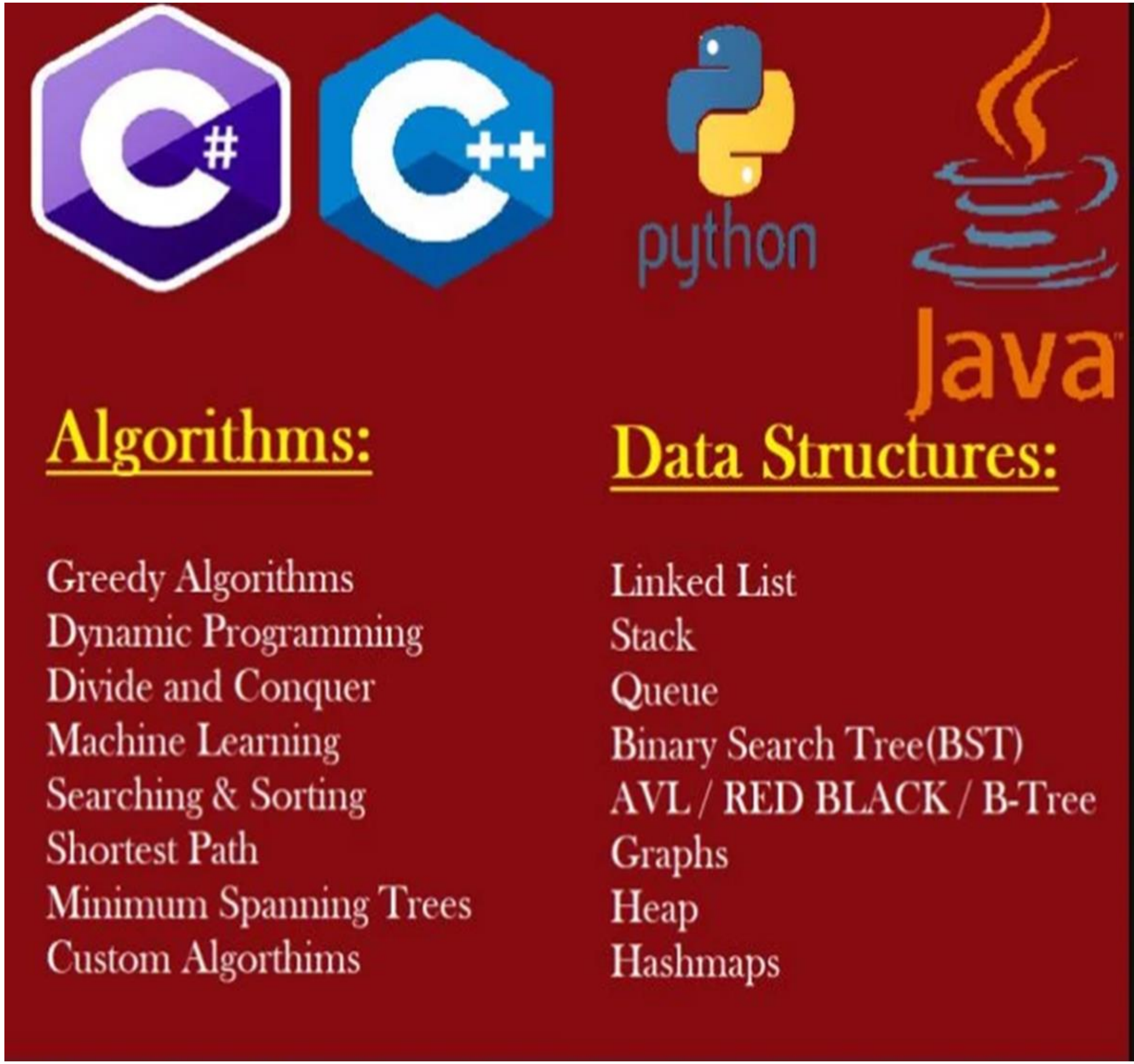


Figure 1
Chapter
Dependencies



Return Value and Parameters

LECTURE 2

Generating Random Numbers

- Many coding languages provide a way to generate random numbers, and College Board's Pseudocode is no exception. It's a good tool for many programs.

Text: `RANDOM(a, b)` Block: `RANDOM` $\boxed{\texttt{a, b}}$	Generates and returns a random integer from `a` to `b`, including `a` and `b`. Each result is equally likely to occur. For example, `RANDOM(1, 3)` could return `1`, `2`, or `3`.

Python

```
import random
c = random.randint(a,b)
```

Notice how you have to import the random module into your program in order to gain access to the random generator.

Library

LECTURE 3

Libraries

- You don't have to define every procedure you use in a program. If you're using already-written code, you can import it into your program. In Python, you do this by importing the module it's contained in. This importing usually takes the form of a line of text at the top of your program that looks like this:

```
from location import module
```

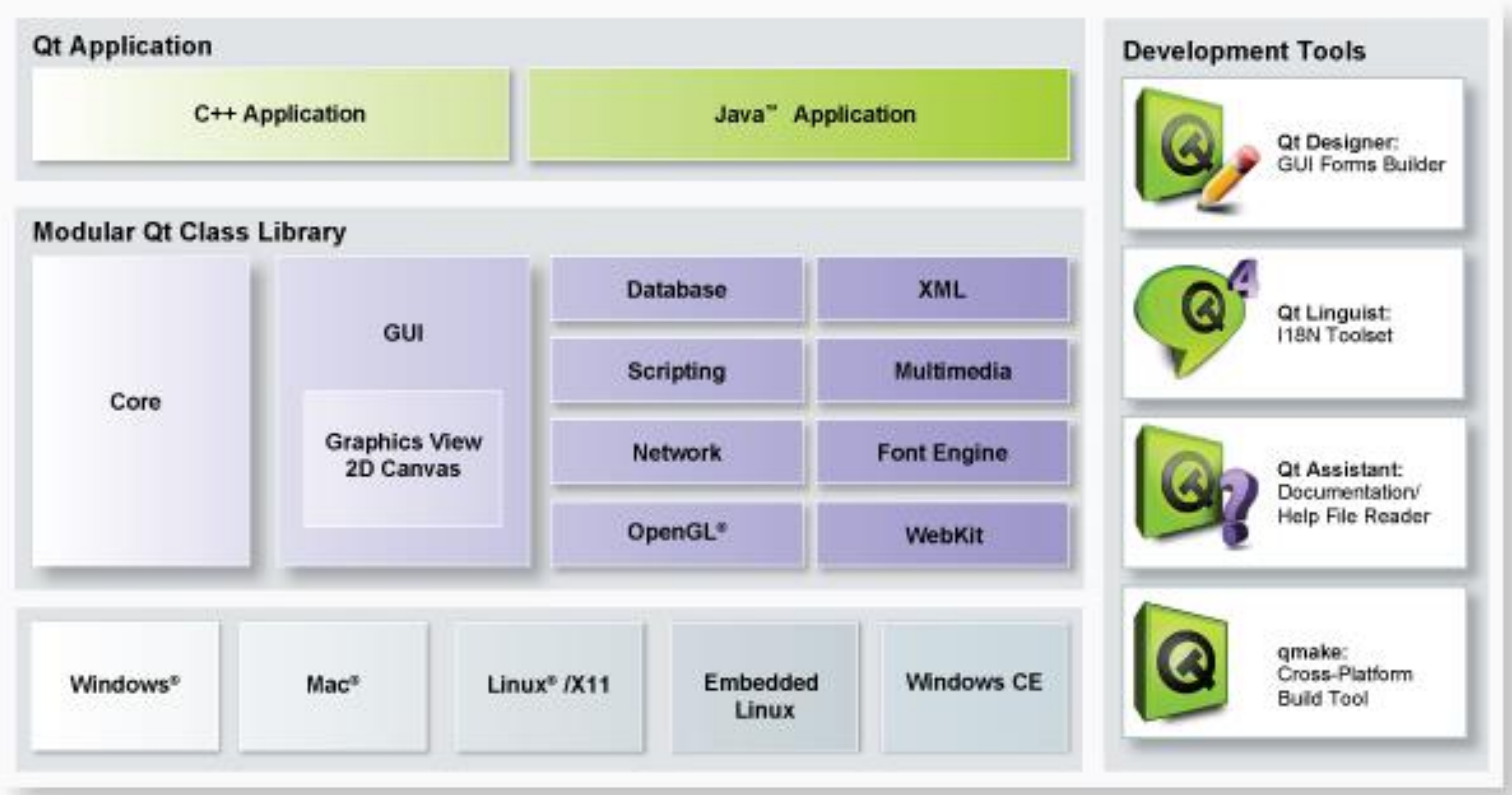
- There are several places where already-written modules come from.

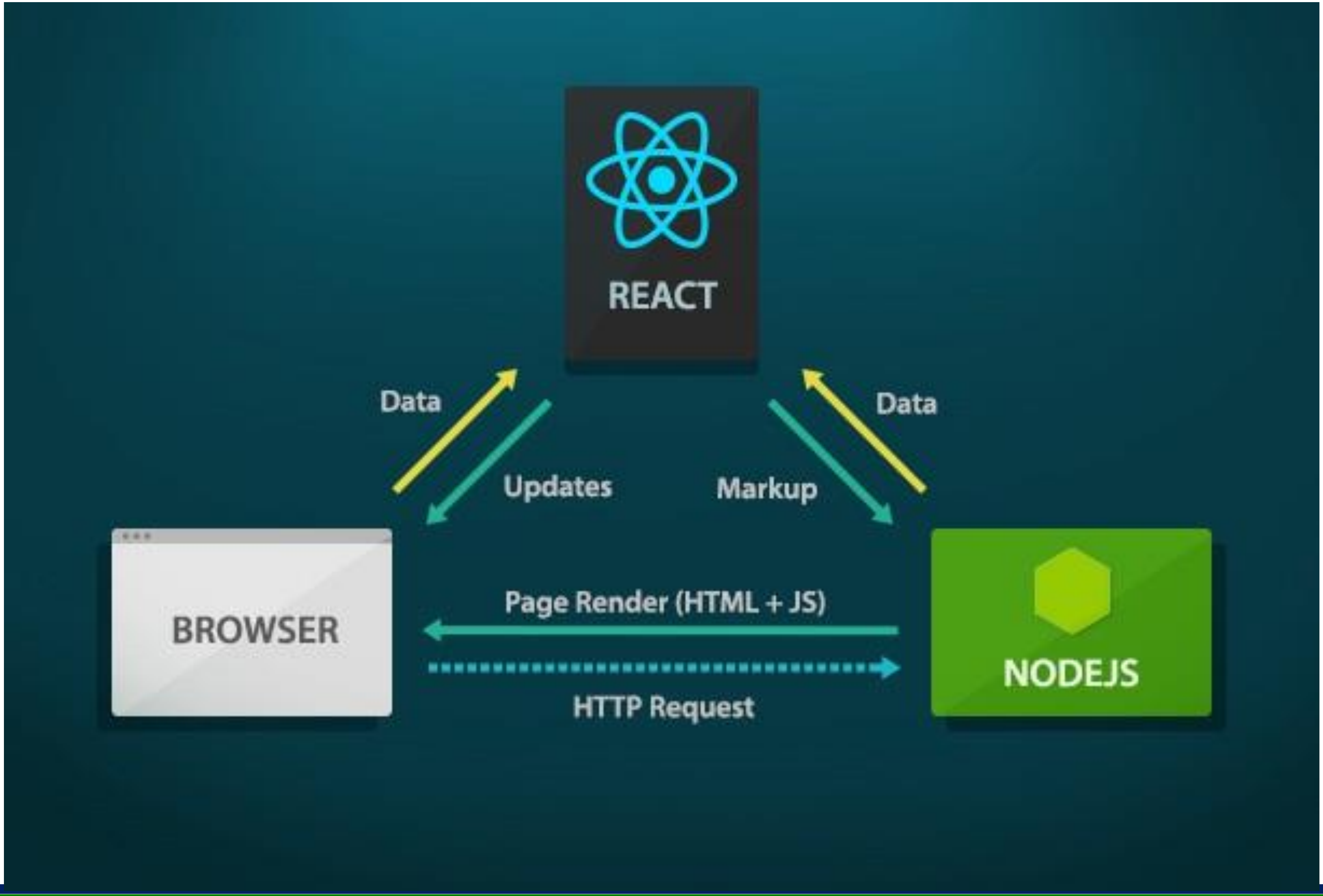
API Software Libraries

- A **software library** contains already-developed procedures that you can use in creating your own programs. You don't have to write new procedures to, for example, display images or find derivatives in your program; someone's already written those procedures and put them in a library for the public to use.
- There are libraries for everything under the sun. Here are some Python examples:
  - **Pillow** allows you to work with images.
  - **Matplotlib** allows you to make 2D graphs and plots.
- You can also **import** some of your own previously written code into a new program.

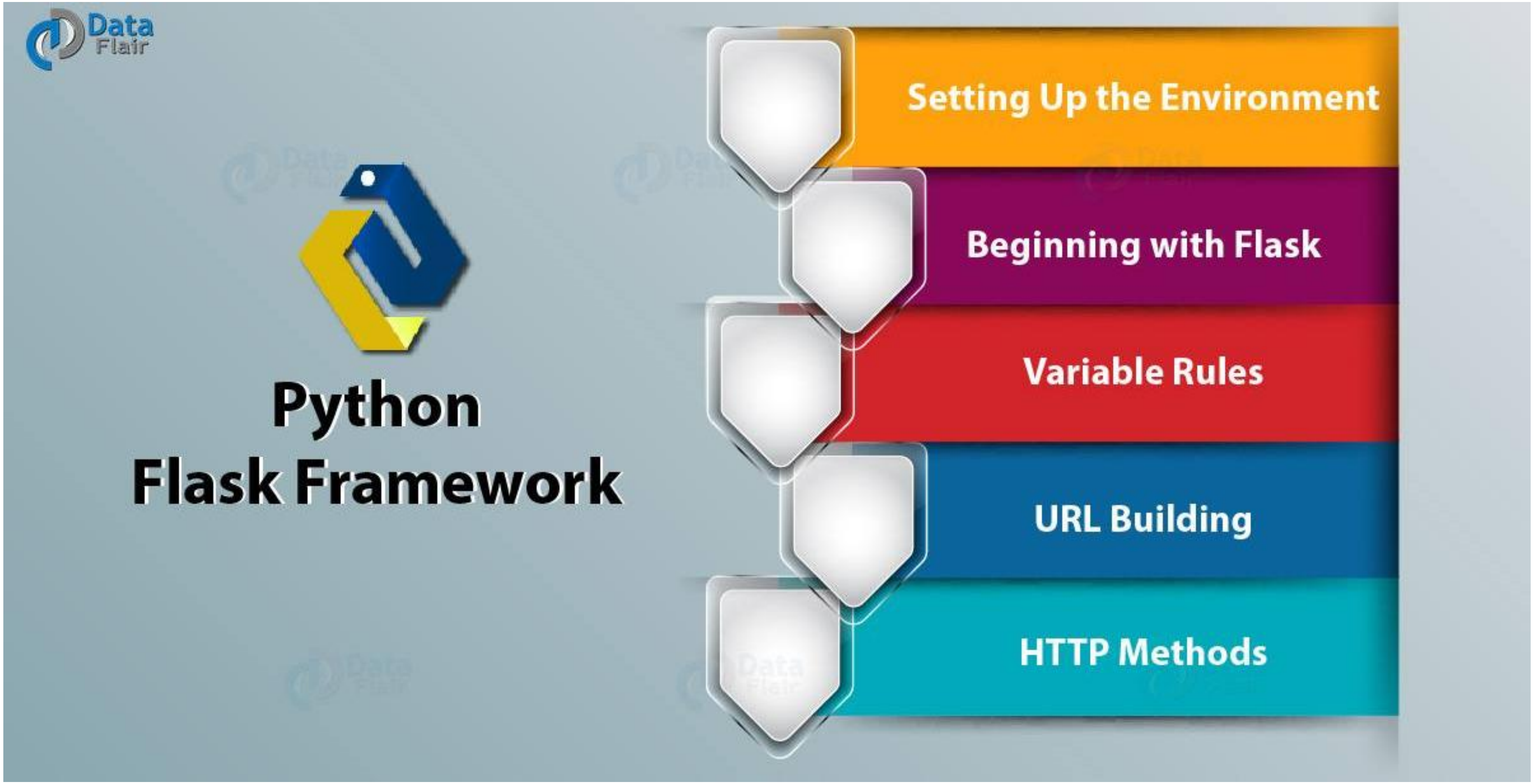
API Software Libraries

- An Application Program Interface, or **API**, contains specifications for how the procedures in a library behave and can be used. It allows the imported procedures from the library to interact with the rest of your code.
- In order to make the most of both APIs and Libraries, both need to be well documented. Libraries are, at their heart, a collection of other people's code, and it would be difficult to understand how the procedures should be used without documentation. APIs explain how two separate pieces of software interact with each other, and they also need documentation to keep this communication going.


Qt Application
C++ Application
Java™ Application
Modular Qt Class Library
Core
GUI
Graphics View 2D Canvas
Database
XML
Scripting
Multimedia
Network
Font Engine
OpenGL®
WebKit
Windows®
Mac®
Linux® /X11
Embedded Linux
Windows CE
Development Tools
Qt Designer: GUI Forms Builder
Qt Linguist: I18N Toolset
Qt Assistant: Documentation/ Help File Reader
qmake: Cross-Platform Build Tool



REACT
Data
Updates
Markup
Data
BROWSER
Page Render (HTML + JS)
HTTP Request
NODEJS


Data Flair
Python
Flask Framework
Setting Up the Environment
Beginning with Flask
Variable Rules
URL Building
HTTP Methods

Simulation

LECTURE 4

Simulations

- **Simulations** are simplifications of complex objects (like the planets) or phenomena (like tornadoes) for a stated goal. They often use varying sets of values to reflect how a phenomenon changes. Using a computer, we can simulate everything from a **science lab** to a nuclear explosion to a zombie apocalypse, and computer simulations are used in industries like weather forecasting and financial planning.
- In order to develop a simulation, you have to remove certain real world details (like language barriers in a historical simulation event) or simplify how something functions.

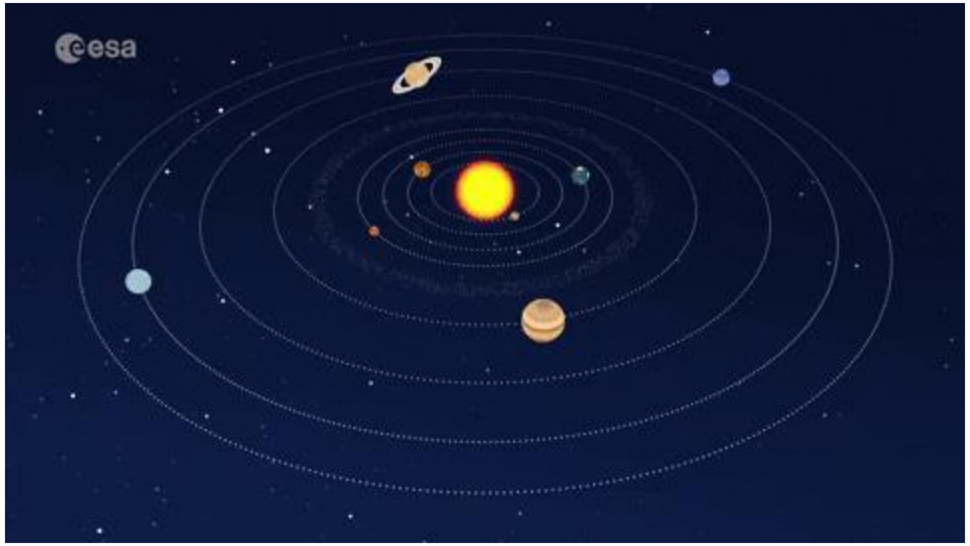
AP CSP

Abstraction in Simulations

- Simplifying details to highlight a main point, where have we heard this before? 🤔
- That's right, simulations are an example of abstraction.
- There are many benefits to creating a simulation. They can be used to represent real-world events and conditions, like the force of gravity or the atmospheric conditions of a battle, so you can investigate and draw conclusions about them without dealing with some of the complications of the real world. Simulations are the most useful when observing the phenomenon in real life would be impractical, like if what you wanted to study was too big (Big Bang, continental drift) or too small (atoms, elements).

Abstraction in Simulations

- However, simulations also have some disadvantages. They run the risk of being too simple or conveying the wrong message about what you're trying to study (simulating the planets with tennis balls, for example, may lead people to think they're closer to each other and more similarly sized than they actually are.)



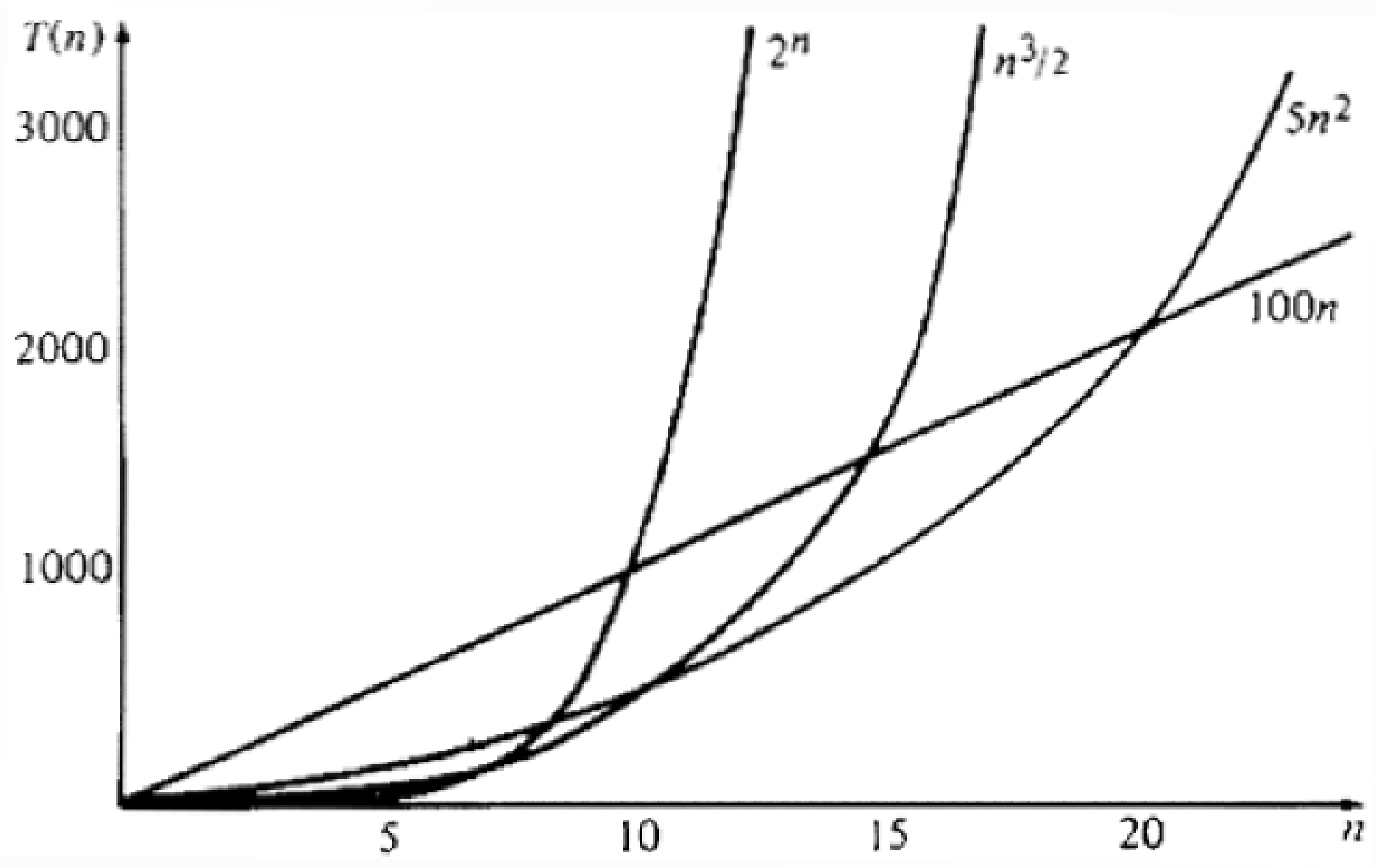
Abstraction in Simulations

- Simulations may also contain bias based on what the simulation creator chose to include or exclude.
- Random number generators can help simulate real-world variability in these simulations: it's a little like rolling a pair of dice.

Algorithm Efficiency

LECTURE 5

Approximate Growth Rate *T(n)*



Comparing Common Growth Functions

$$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3) < O(2^n)$$

$O(1)$	Constant time
$O(\log n)$	Logarithmic time
$O(n)$	Linear time
$O(n\log n)$	Log-linear time
$O(n^2)$	Quadratic time
$O(n^3)$	Cubic time
$O(2^n)$	Exponential time

Best, Worst, and Average Cases

- For the same input size, an algorithm's execution time may vary, depending on the input. An input that results in the shortest execution time is called the best-case input and an input that results in the longest execution time is called the worst-case input. Best-case and worst-case are not representative, but worst-case analysis is very useful. You can show that the algorithm will never be slower than the worst-case.
- **An average-case analysis attempts to determine the average amount of time among all possible input of the same size.** Average-case analysis is ideal, but difficult to perform, because it is hard to determine the relative probabilities and distributions of various input instances for many problems. Worst-case analysis is easier to obtain and is thus common. So, the analysis is generally conducted for the worst-case.

Ignoring Multiplicative Constants

- The linear search algorithm requires n comparisons in the worst-case and **n/2** comparisons in the average-case. Using the Big O notation, both cases require **O(n)** time. The multiplicative constant **(1/2)** can be omitted.
- Algorithm analysis is focused on growth rate. The multiplicative constants have no impact on growth rates.
- The growth rate for n/2 or 100n is the same as n, i.e., **O(n) = O(n/2) = O(100n).**

Ignoring Non-Dominating Terms

- Consider the algorithm for finding the maximum number in an array of n elements. If n is 2, it takes one comparison to find the maximum number. If n is 3, it takes two comparisons to find the maximum number. In general, it takes n-1 times of comparisons to find maximum number in a list of n elements.
- Algorithm analysis is for large input size. If the input size is small, there is no significance to estimate an algorithm's efficiency. As n grows larger, the n part in the expression n-1 dominates the complexity.
- The Big O notation allows you to ignore the non-dominating part (e.g., -1 in the expression n-1) and highlight the important part (e.g., n in the expression n-1). So, the complexity of this algorithm is O(n).

Constant Time

- The Big **O(n)** notation estimates the execution time of an algorithm in relation to the input size. If the time is not related to the input size, the algorithm is said to take constant time with the notation **O(1)**.
- For example, a method that retrieves an element at a given index in an array takes constant time, because it does not grow as the size of the array increases.

Repetition: Simple Loops

$$\text{executed } n \text{ times} \left\{ \begin{array}{l} \texttt{for (i = 1; i <= n; i++) \{} \\ \quad \texttt{k = k + 5;} \\ \texttt{\}} \end{array} \right.$$

```
for (i = 1; i <= n; i++) {
   k = k + 5;
}
```

constant time

Time Complexity

$T(n) = (\text{a constant } c) * n = cn = \mathbf{O(n)}$

Ignore multiplicative constants (e.g., "c").

Repetition: Nested Loops

executed
n times

```
for (i = 1; i <= n; i++) {
   for (j = 1; j <= n; j++) {
      k = k + i + j;
   }
}
```

inner loop
executed
n times

constant time

Time Complexity

$T(n) = (\text{a constant } c) * n * n = cn^2 = O(n^2)$

Ignore multiplicative constants (e.g., "c").

Repetition: Nested Loops

executed
n times

```
for (i = 1; i <= n; i++) {
   for (j = 1; j <= i; j++) {
      k = k + i + j;
   }
}
```

inner loop
executed
i times

constant time

Time Complexity

$T(n) = c + 2c + 3c + 4c + \ldots + nc = cn(n+1)/2 =$
$(c/2)n^2 + (c/2)n = O(n^2)$

Ignore non-dominating terms

Ignore multiplicative constants

Repetition: Nested Loops

executed
n times

```
for (i = 1; i <= n; i++) {
   for (j = 1; j <= 20; j++) {
      k = k + i + j;
   }
}
```

inner loop
executed
20 times

constant time

Time Complexity

$T(n) = 20 * c * n = O(n)$

*Ignore multiplicative constants (e.g., 20*c)*

Sequence

executed *10* times

```
for (j = 1; j <= 10; j++) {
   k = k + 4;
}
```

executed *n* times

```
for (i = 1; i <= n; i++) {
   for (j = 1; j <= 20; j++) {
      k = k + i + j;
   }
}
```

inner loop executed *20* times

Time Complexity

$T(n) = c * 10 + 20 * c * n = O(n)$

Selection

O(n)

```
if (list.contains(e)) {
  System.out.println(e);
}
else
  for (Object t: list) {
    System.out.println(t);
  }
```

Let n be list.size(). Executed n times.

Time Complexity

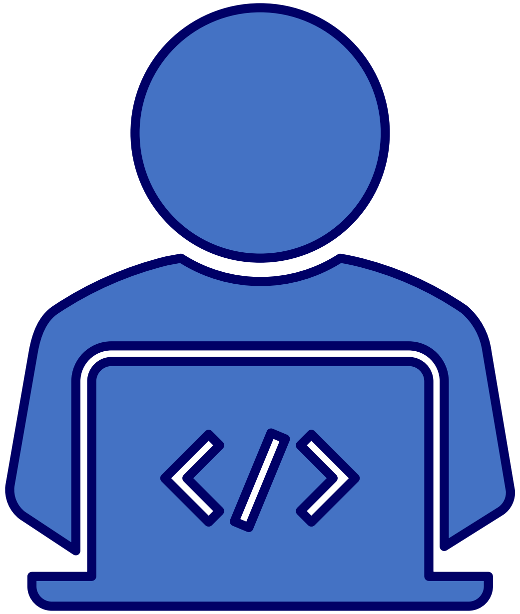
T(n) = test time + worst-case (if, else)
= O(n) + O(n)
= O(n)

Undecidable Problems

LECTURE 6

Undecidable Problems

- A **decidable problem** is a decision problem (one that has a yes/no answer) where an algorithm can be written to produce a correct output for all inputs.
- If an algorithm can't be written that's always capable of providing a correct yes or no answer, it's an **undecidable problem**. An undecidable problem might be able to be solved in some cases, but not in all of them.
- The classic example of an undecidable problem is the halting problem, created by Alan Turing in 1936. The halting problem asks that if a computer is given a random program, can an algorithm ever be written that will answer the question, will this program ever stop running?, for all programs? By proving that there wasn't, Turing demonstrated that some problems can't be completely solved with an algorithm.



Algorithm

SECTION 1

The Need for Algorithms

LECTURE 7

Objectives

- The main purpose of the lesson is to connect the acts of writing "code" and designing algorithms, and to take some steps towards programming with code.
- **Algorithm** - A precise sequence of instructions for processes that can be executed by a computer

Problem Solving

Step 1

- Understand the problem
- Identify program input and output

Step 2

- Design the solution (algorithm)

Step 3

- Writing a program in a programming language to match the algorithm steps

Human-Machine FindMin

ACTIVITY

Activity – Human/Machine

- In this activity you're going to pretend that you are a "**Human Machine**" that operates on playing cards on the table.
- We often get started thinking about algorithms by trying to rigorously act them out ourselves as a sort of "Human Machine". When acting as a machine, we can keep the limitations of a computer in mind.

Activity – Human/Machine

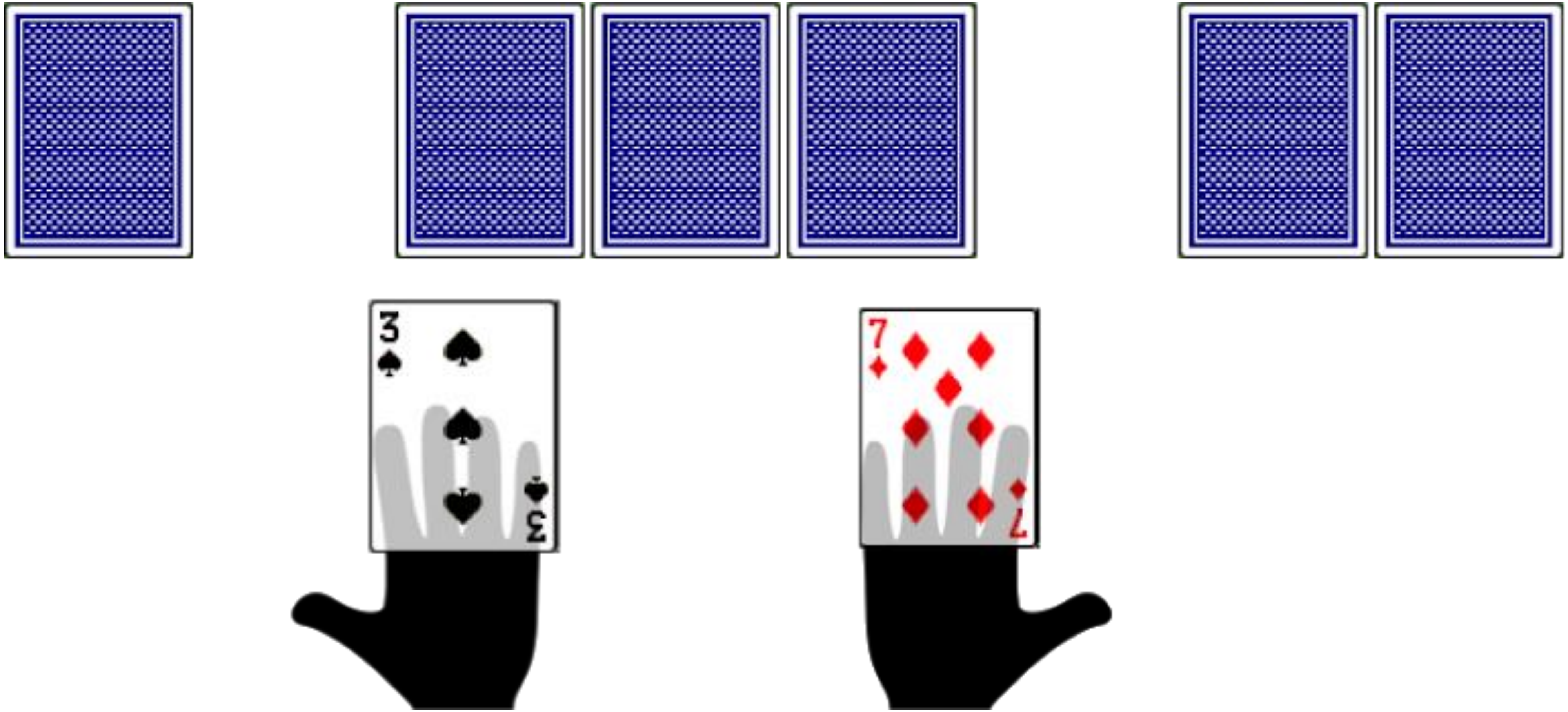
- In this activity, you'll design an algorithm to find the smallest item in a list. Obviously, if we were really writing instructions for a person, we could simply tell them: "find the smallest item in a list." But that won't work for a computer.
- We need to describe the process that a person must go through when they are finding the smallest item. What are they really doing?

Activity

- Everyone gets Minimum Card Algorithm - Activity Guide
- Each pair gets a deck of cards.
- Or you can use Online-Deck-of-Cards

Setup and Rules:

- We'll use playing cards face down on the table to represent a list of items. Start with 8 random cards face down in a row.
- Any card on the table must be face down.
- When acting as the machine, you can pick up a card with either hand, but each hand can only hold one card at a time.
- You can look at and compare the values of any cards you are holding to determine which one is greater than the other.
- You can put a card back down on the table (face down), but once a card is face down on the table, you cannot remember (or memorize) its value or position in the list.



Task:

- **Goal:** The algorithm must have a clear end to it. The last instruction should be to say: "I found it!" and hold up the card with the lowest value.
- The algorithm should be written so that it would theoretically work for any number of cards (1 or 1 million).
- Write your algorithm out on paper as a clear list of instructions in "pseudocode." Your instructions can refer to the values on cards, and a person's hands, etc., but you must invent a systematic way for finding the smallest card.

Activity

- Get clear on the task, rules, instructions
- With a partner act out an algorithm
- Write down the steps

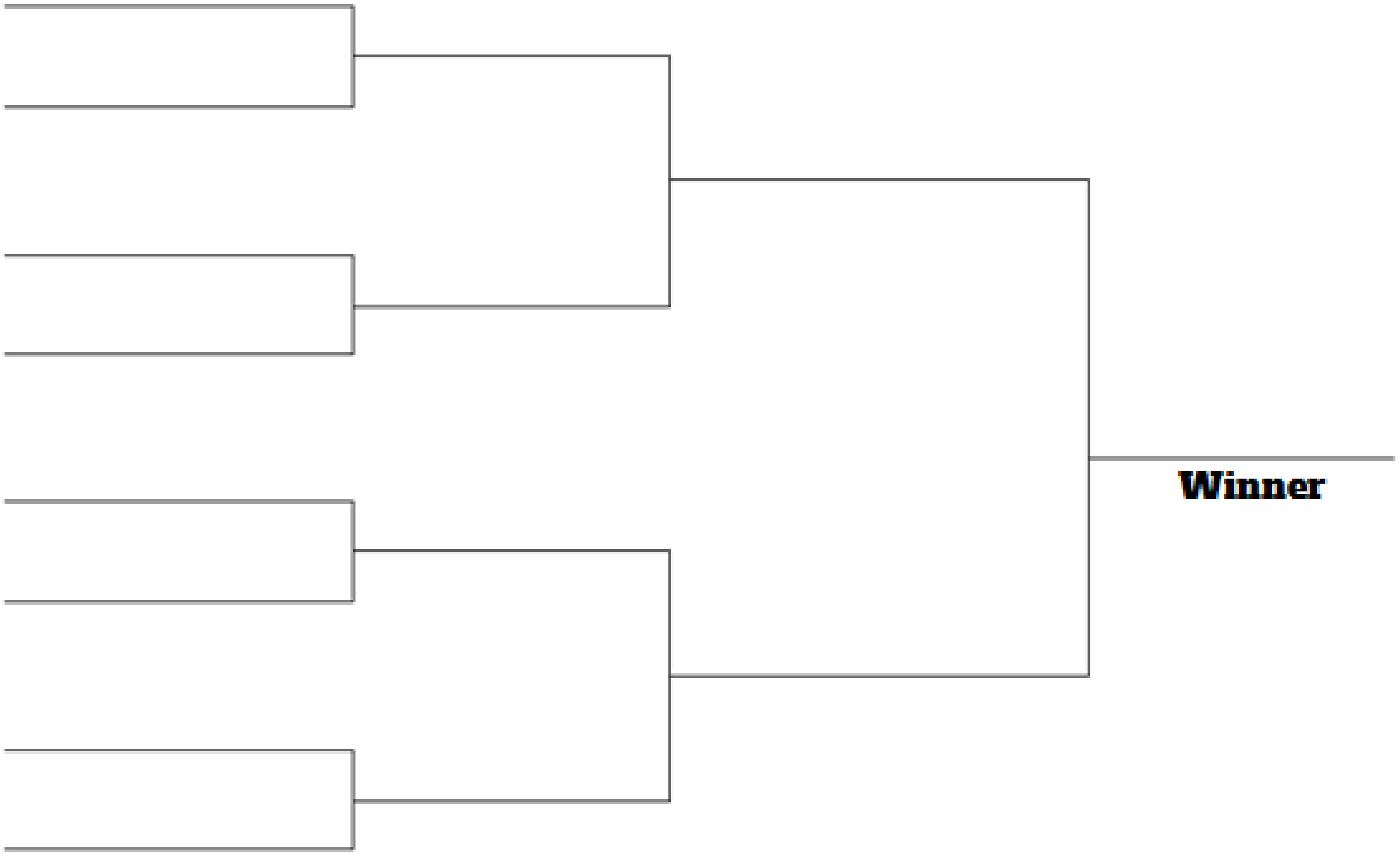
Discussion

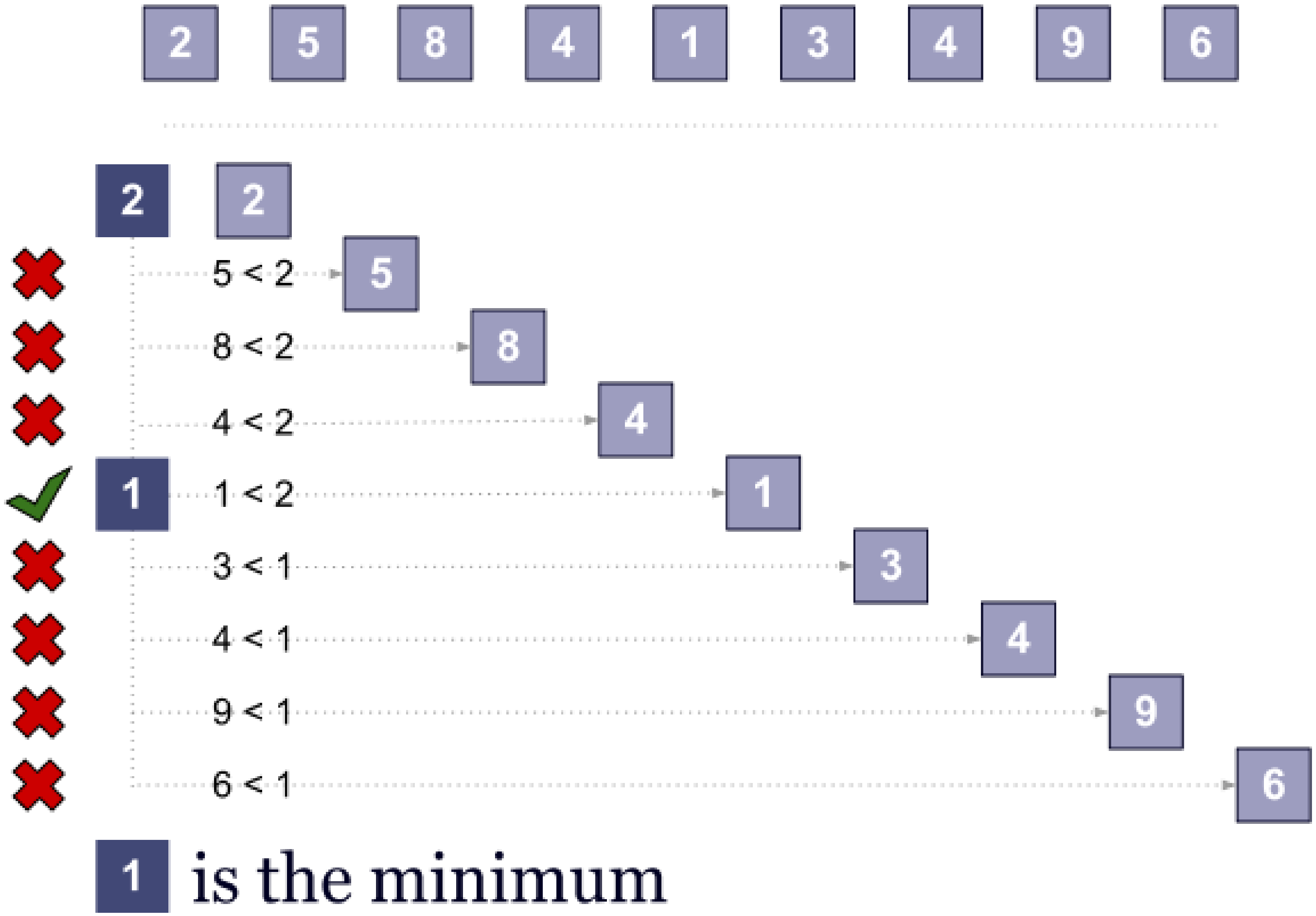
- How do you know when to stop?
- Do your instructions state where and how to start?
- Is it clear where to put cards back down after you've picked them up?

Discussion

- As we look at these algorithms you came up with, we can see they are not all the same.
- However, there are common things that you are all making the human machine do and commonalities in some of your instructions.
- Can we define a language of common Human Machine commands for moving cards around? What are the commands or actions most of these instructions have in common?"

8 Team Single Elimination




2 5 8 4 1 3 4 9 6
2
2
5 < 2
5
8 < 2
8
4 < 2
4
1
1 < 2
1
3 < 1
3
4 < 1
4
9 < 1
9
6 < 1
6
1 is the minimum

Creativity in Algorithms

LECTURE 8

Objectives

- The purpose of this lesson is to see what "creativity in algorithms" means.

Vocabulary

- **Algorithm** - A precise sequence of instructions for processes that can be executed by a computer
- **Iterate** - To repeat in order to achieve, or get closer to, a desired goal.
- **Selection** - A generic term for a type of programming statement (usually an if-statement) that uses a Boolean condition to determine, or select, whether or not to run a certain block of statements.
- **Sequencing** - Putting commands in correct order so computers can read the commands.

Vocabulary

- **Boolean** – when you have two choices, on or off, yes or no.
- **Creativity** - has to do with both the process you invent (an algorithm) to solve a new problem in clever ways that can be executed by a machine.

Creativity

- **Creativity** often means combining or using algorithms you know as part of a solution to a new problem.
- Different algorithms can be developed to solve the same problem

Different programs (or code) can be written to implement the same algorithm.

Program Sequence

- Let's learn a little program sequence:

What is the answer to this program?

```
X = 2
X = 5
X = X + 1
```

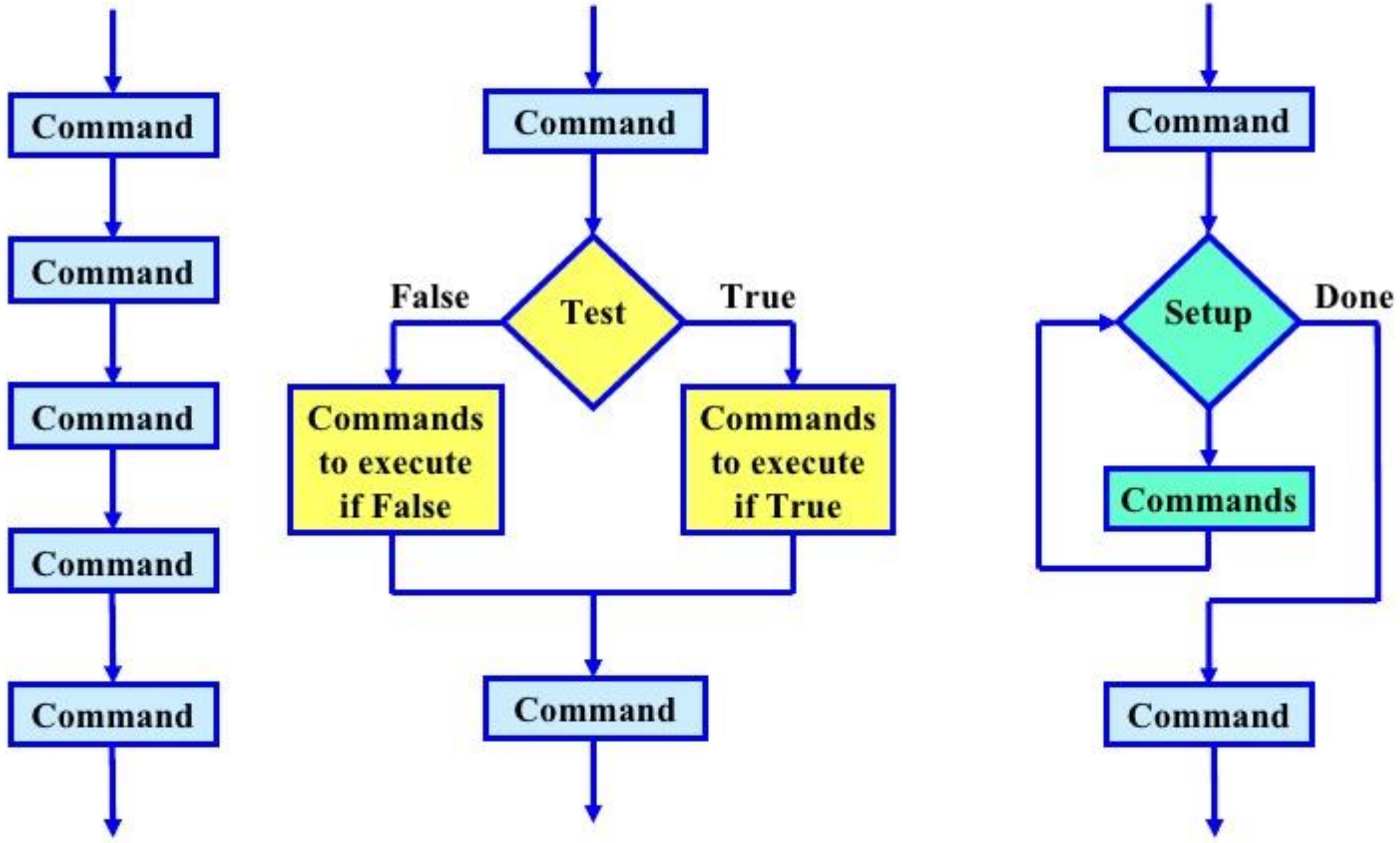
Structured Programming

LECTURE

Structured Programming

- If these statements are true then we should be able to identify these elements of **sequencing, selection and iteration** in our **Find-Min** and **Min-to-Front** algorithms.
- I'll give you a quick definition of each and you tell me if or where we saw it in our Human Machine Language programs.

Flowcharts for sequential, selection, and iterative control structures



Sequential Structure
(straight-line structure)

Example Selection Structure
(decision or branching structure)

Iterative Structure
(looping structure)

Sequencing

- **“4.1.1B Sequencing** is the application of each step of an algorithm in the order in which the statements are given." -- Does our human machine language have sequencing?
- **Sequencing** is so fundamental to programming it sometimes goes without saying. In our lesson, the sequencing is simply implied by the fact that we number the instructions with the intent to execute them in order.

Selection

- "4.1.1C **Selection** uses a [true-false] condition to determine which of two parts of an algorithm is used." -- Where did we see "**selection**" in our human machine language programs?
- The JUMP…IF command in the Human Machine Language is a form of selection. It gives us a way to compare two things (numbers) and take action if the comparison is true, or simply proceed with the sequence if false.
- NOTE: **Selection** is also known as "branching" most commonly seen in if-statements in programs.

Iteration

- "4.1.1D **Iteration** is the repetition of part of an algorithm until a condition is met or for a specified number of times." -- Where did we see iteration in our human machine language programs?
- The JUMP command (as well as JUMP...IF) in the Human Machine Language allows us to move to a different point in the program and start executing from there. This allows us to re-use lines of code, and this is a form of **iteration** or **looping**.
- NOTE: **Iteration** is also known as “looping” in most programming languages.

Development of Algorithm

- Important points:

```
Algorithms can be combined to make new
algorithms
```

- Low-Level languages exist - most basic, primitive, set of commands to control a computer. The Human Machine Language is similar to something called **Assembly Language**

Assembly Language

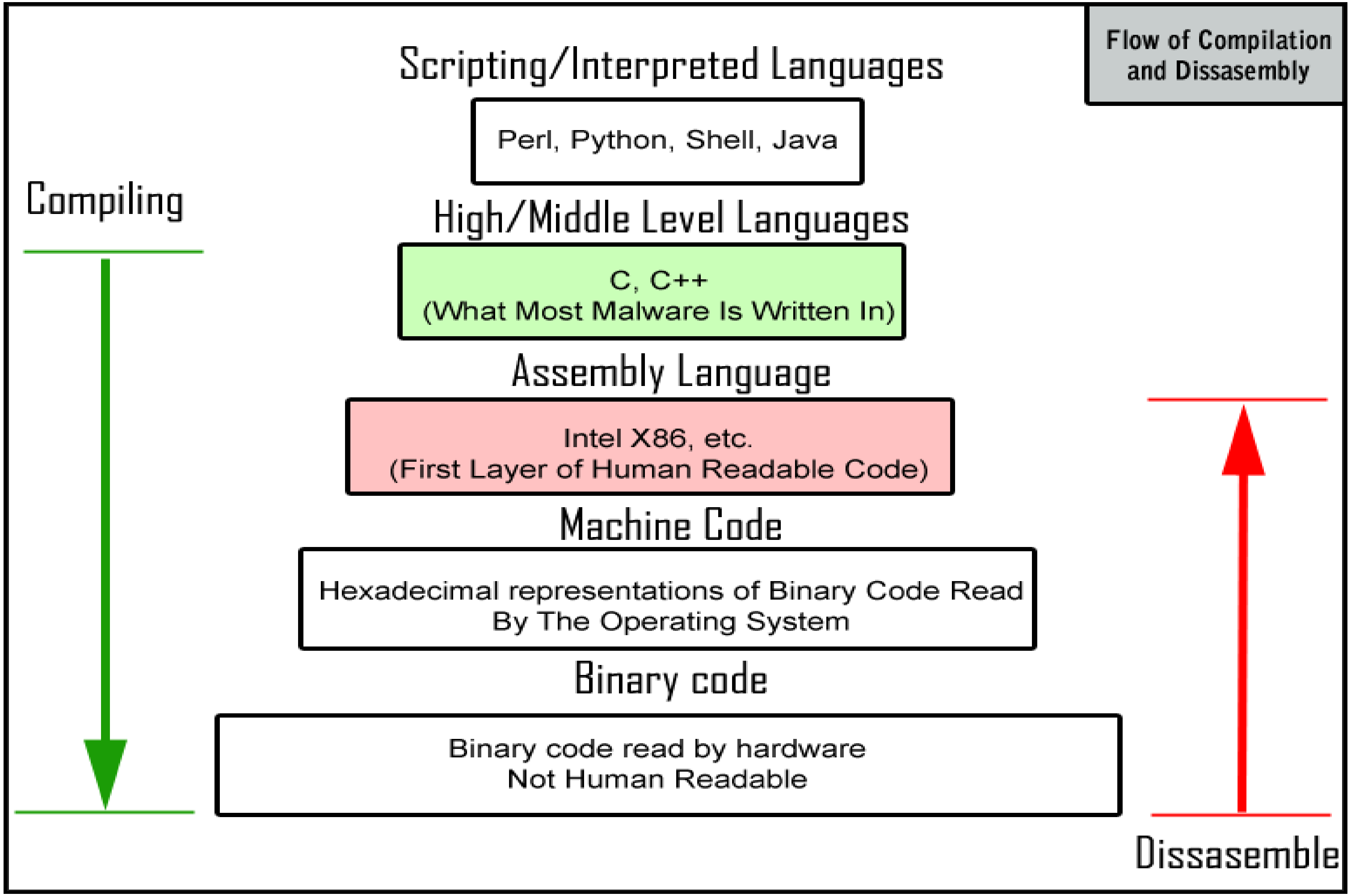
Here is an example of a simple program written in IBM Assembly Language.

```
; Example of IBM PC assembly language
; Accepts a number in register AX;
; subtracts 32 if it is in the range 97-122;
; otherwise leaves it unchanged.

SUB32  PROC           ; procedure begins here
       CMP   AX,97    ; compare AX to 97
       JL    DONE     ; if less, jump to DONE
       CMP   AX,122   ; compare AX to 122
       JG    DONE     ; if greater, jump to DONE
       SUB   AX,32    ; subtract 32 from AX
DONE:  RET            ; return to main program
SUB32  ENDP           ; procedure ends here
```

Programming Language

- From the CSP Framework: 2.2.3C Code in a programming language is often translated into code in another (lower level) language to be executed on a computer.
- The Human Machine Language is a "low level" language because the commands are very primitive and tie directly specific functions of the "human machine".


Flow of Compilation and Dissasembly
Scripting/Interpreted Languages
Perl, Python, Shell, Java
Compiling
High/Middle Level Languages
C, C++
(What Most Malware Is Written In)
Assembly Language
Intel X86, etc.
(First Layer of Human Readable Code)
Machine Code
Hexadecimal representations of Binary Code Read By The Operating System
Binary code
Binary code read by hardware
Not Human Readable
Dissasemble

Magical Powers

- Learning to program is really learning how to think in terms of algorithms and processes. And it can be really fun and addicting. It also can make you feel like you have **magical powers**.

Parameters, Return, and Libraries

LECTURE 9

Unit 7 - Parameters, Return, and Libraries ('22-'23)

This unit introduces parameters, return, and libraries. Learn how to use these concepts to build new kinds of apps as well as libraries of code that you can share with your classmates. End the unit by designing a library of functions around any topic of your choosing.

Teacher resources | Printing Options | View calendar

For your owned section:

APCSP 2023 Banana | ✔ Assigned

Parameters and Return Values

Lesson 1: Parameters and Return Explore

Students work with envelopes and paper to model functions with parameters and return values. Students create their own physical function envelope for drawing a house that takes in different parameters, and then build another function to calculate and return the cost of building that house.

1 Check For Understanding

View Lesson Plan

Student Resources

Send to students

Rate this Lesson

Visible | Hidden