

# CSC 121 Computers and Scientific Thinking

David Reed  
Creighton University

## Algorithms and Programming Languages

1

## Algorithms



the central concept underlying all computation is that of the *algorithm*

- an algorithm is a step-by-step sequence of instructions for carrying out some task

programming can be viewed as the process of designing and implementing algorithms that a computer can carry out

- a programmer's job is to:
  - create an algorithm for accomplishing a given objective, then
  - translate the individual steps of the algorithm into a programming language that the computer can understand

example: programming in JavaScript

- we have written programs that instruct the browser to carry out a particular task
- given the proper instructions, the browser is able to understand and produce the desired results

2

# Algorithms in the Real World



the use of algorithms is not limited to the domain of computing

- e.g., recipes for baking cookies
- e.g., directions to your house

there are many unfamiliar tasks in life that we could not complete without the aid of instructions

- in order for an algorithm to be effective, it must be stated in a manner that its intended executor can understand
  - a recipe written for a master chef will look different than a recipe written for a college student
- as you have already experienced, computers are more demanding with regard to algorithm specifics than any human could be

**EASY COOK DIRECTIONS:**

**TOP OF STOVE**

- **BOIL 6 cups of water.** Stir in Macaroni. Boil 11 to 14 minutes, stirring occasionally.
- **DRAIN. DO NOT RINSE.** Return to pan.
- **ADD 3 Tbsp. margarine, 3 Tbsp. milk and Cheese Sauce Mix;** mix well. Makes about 2 cups.

**NO DRAIN MICROWAVE**

- **POUR** Macaroni into 1 or 2 quart microwavable bowl. Add 1 cups hot water.
- **MICROWAVE** uncovered, on HIGH 12 to 14 minutes or until water is absorbed, stirring every 5 minutes. Continue as directed above.

3

# Designing & Analyzing Algorithms



4 steps to solving problems (George Polya)

1. understand the problem
2. devise a plan
3. carry out your plan
4. examine the solution

EXAMPLE: finding the oldest person in a room full of people

1. understanding the problem
  - initial condition – room full of people
  - goal – identify the oldest person
  - assumptions
    - ✓ a person will give their real birthday
    - ✓ if two people are born on the same day, they are the same age
    - ✓ if there is more than one oldest person, finding any one of them is okay
2. we will consider 2 different designs for solving this problem

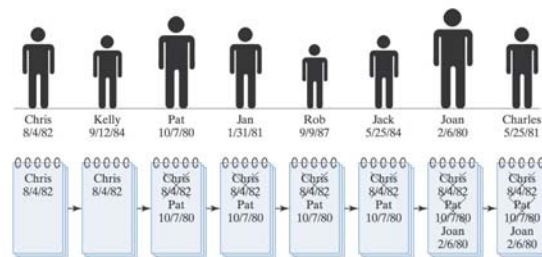
4

## Algorithm 1

### Finding the oldest person (algorithm 1)

1. line up all the people along one wall
2. ask the first person to state their name and birthday, then write this information down on a piece of paper
3. for each successive person in line:
  - i. ask the person for their name and birthday
  - ii. if the stated birthday is earlier than the birthday on the paper, cross out old information and write down the name and birthday of this person

when you reach the end of the line, the name and birthday of the oldest person will be written on the paper



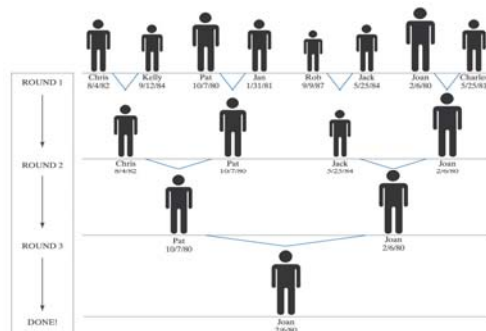
5

## Algorithm 2

### Finding the oldest person (algorithm 2)

1. line up all the people along one wall
2. as long as there is more than one person in the line, repeatedly
  - i. have the people pair up (1<sup>st</sup> with 2<sup>nd</sup>, 3<sup>rd</sup> with 4<sup>th</sup>, etc) – if there are an odd number of people, the last person will be without a partner
  - ii. ask each pair of people to compare their birthdays
  - iii. request that the younger of the two leave the line

when there is only one person left in line, that person is the oldest



6

# Algorithm Analysis

determining which algorithm is "better" is not always clear cut

- it depends upon what features are most important to you
  - if you want to be sure it works, choose the /clearer algorithm
  - if you care about the time or effort required, need to analyze performance

algorithm 1 involves asking each person's birthday and then comparing it to the birthday written on the page

- the amount of time to find the oldest person is *proportional to the number of people*
- if you double the amount of people, the time needed to find the oldest person will also double

algorithm 2 allows you to perform multiple comparisons simultaneously

- the time needed to find the oldest person is *proportional to the number of rounds it takes to shrink the line down to one person*
  - which turns out to be the logarithm (base 2) of the number of people
- if you double the amount of people, the time needed to find the oldest person increases by a factor of one more comparison

# Algorithm Analysis (cont.)

when the problem size is large, performance differences can be dramatic

for example, assume it takes 5 seconds to compare birthdays

- for algorithm 1:
  - 100 people →  $5 * 100 = 500$  seconds
  - 200 people →  $5 * 200 = 1000$  seconds
  - 400 people →  $5 * 400 = 2000$  seconds
  - ...
  - 1,000,000 people →  $5 * 1,000,000 = 5,000,000$  seconds
- for algorithm 2:
  - 100 people →  $5 * \lceil \log 100 \rceil = 35$  seconds
  - 200 people →  $5 * \lceil \log 200 \rceil = 40$  seconds
  - 400 people →  $5 * \lceil \log 400 \rceil = 45$  seconds
  - ...
  - 1,000,000 people →  $5 * \lceil \log 1,000,000 \rceil = 100$  seconds

N	$\lceil \log_2(N) \rceil$
100	7
200	8
400	9
800	10
1,600	11
...	...
10,000	14
20,000	15
40,000	16
...	...
1,000,000	20

## Big-Oh Notation



to represent an algorithm's performance in relation to the size of the problem, computer scientists use what is known as *Big-Oh* notation

- executing an  $O(N)$  algorithm requires time proportional to the size of problem
  - given an  $O(N)$  algorithm, doubling the problem size doubles the work
- executing an  $O(\log N)$  algorithm requires time proportional to the logarithm of the problem size
  - given an  $O(\log N)$  algorithm, doubling the problem size adds a constant amount of work

based on our previous analysis:

- algorithm 1 is classified as  $O(N)$
- algorithm 2 is  $O(\log N)$

9

## Another Algorithm Example



SEARCHING: a common problem in computer science involves storing and maintaining large amounts of data, and then searching the data for particular values

- data storage and retrieval are key to many industry applications
- search algorithms are necessary to storing and retrieving data efficiently
- e.g., consider searching a large payroll database for a particular record
  - if the computer selected entries at random, there is no assurance that the particular record will be found
  - even if the record is found, it is likely to take a large amount of time
  - a systematic approach assures that a given record will be found, and that it will be found more efficiently

there are two commonly used algorithms for searching a list of items

- sequential search – general purpose, but relatively slow
- binary search – restricted use, but fast

10

## Sequential Search

*sequential search* is an algorithm that involves examining each list item in sequential order until the desired item is found

### sequential search for finding an item in a list

1. start at the beginning of the list
2. for each item in the list
  - i. examine the item - if that item is the one you are seeking, then you are done
  - ii. if it is not the item you are seeking, then go on to the next item in the list

if you reach the end of the list and have not found the item, then it was not in the list

sequential search guarantees that you will find the item if it is in the list

- but it is not very practical for very large databases
- *worst case*: you may have to look at every entry in the list

11

## Binary Search

*binary search* involves continually cutting the desired search list in half until the item is found

- the algorithm is only applicable if the list is ordered
  - e.g., a list of numbers in increasing order
  - e.g., a list of words in alphabetical order

### binary search for finding an item in an ordered list

1. initially, the potential range in which the item could occur is the entire list
2. as long as items remain in the potential range and the desired item has not been found, repeatedly
  - i. examine at the middle entry in the potential range
  - ii. if the middle entry is the item you are looking for, then you are done
  - iii. if the middle entry is greater than the desired item, the reduce the potential range to those entries left of the middle
  - iv. if the middle entry is less than the desired item, the reduce the potential range to those entries right of the middle

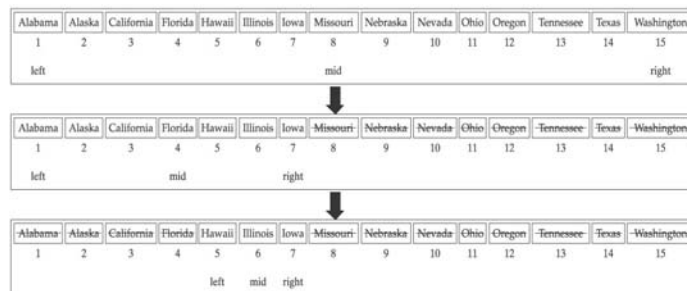
by repeatedly cutting the potential range in half, binary search can hone in on the value very quickly

12

## Binary Search Example

suppose you have a sorted list of state names, and want to find *Illinois*

1. start by examining the middle entry (*Missouri*)  
since *Missouri* comes after *Illinois* alphabetically, can eliminate it and all entries that appear to the right
2. next, examine the middle of the remaining entries (*Florida*)  
since *Florida* comes before *Illinois* alphabetically, can eliminate it and all entries that appear to the left
3. next, examine the middle of the remaining entries (*Illinois*)  
the desired entry is found



13

## Search Analysis

### sequential search

- in the worst case, the item you are looking for is in the last spot in the list (or not in the list at all)
  - as a result, you will have to inspect and compare every entry in the list
- the amount of work required is proportional to the list size  
→ sequential search is an  $O(N)$  algorithm

### binary search

- in the worst case, you will have to keep halving the list until it gets down to a single entry
  - each time you inspect/compare an entry, you rule out roughly half the remaining entries
- the amount of work required is proportional to the logarithm of the list size  
→ binary search is an  $O(\log N)$  algorithm

imagine searching a phone book of the United States (280 million people)

- sequential search requires at most 280 million inspections/comparisons
- binary search requires at most  $\lceil \log(280,000,000) \rceil = 29$  inspections/comparisons

14

## Another Algorithm Example



### Newton's Algorithm for finding the square root of N

1. start with an initial approximation of 1
2. as long as the approximation isn't close enough, repeatedly
  1. refine the approximation using the formula:  
$$\text{newApproximation} = (\text{oldApproximation} + N/\text{oldApproximation})/2$$

example: finding the square root of 1024

```
Initial approximation = 1
Next approximation = 512.5
Next approximation = 257.2490243902439
Next approximation = 130.16480157022683
Next approximation = 69.22732405448894
Next approximation = 42.00958563100827
Next approximation = 33.19248741685438
Next approximation = 32.02142090500024
Next approximation = 32.0000071648159
Next approximation = 32.00000000000008
Next approximation = 32
```

### algorithm analysis:

- Newton's Algorithm does converge on the square root because each successive approximation is closer than the previous one
  - however, since the square root might be a nonterminating fraction it becomes difficult to define the exact number of steps for convergence
- in general, the difference between the given approximation and the actual square root is roughly cut in half by each successive refinement
  - demonstrates  $O(\log N)$  behavior

15

## Algorithms and Programming



programming is all about designing and coding algorithms for solving problems

- the intended executor is the computer or a program executing on that computer
- instructions are written in programming languages which are more constrained and exact than human languages

the level of precision necessary to write programs can be frustrating to beginners

- but it is much easier than it was 50 years ago
- early computers (ENIAC) needed to be wired to perform computations
- with the advent of the von Neumann architecture, computers could be programmed instead of rewired
  - an algorithm could be coded as instructions, loaded into the memory of the computer, and executed

16

# Machine Languages

the first programming languages were known as *machine languages*

- a *machine language* consists of instructions that correspond directly to the hardware operations of a particular machine
  - i.e., instructions deal directly with the computer's physical components including main memory, registers, memory cells in CPU
  - very low level of abstraction
- machine language instructions are written in binary
  - programming in machine language is tedious and error prone
  - code is nearly impossible to understand and debug

excerpt from a machine language program:

```
0000000000000110100001100101011011000110110001101110010111001100011011100000
1110000000000001100111011000110110001100110010010111110110001101101111011010
10111000001101001011011000110010101100100001011100000000001011110101000101011
11011100010110100011011101100100000000001011101011110110110011000110011010
1111101011110011011101101110110011011010001100100110010101100001011011010
101000010001100101001000110110110111011001101101000110010011001010110000
10110110101011110101001000110110110111011001101110100011100100110010101100
001010110100000000101111010111101101100011100110101111010111100110111011
011110111001101110100011100100110010101100001011010101000001000011011000110
00000001100101011011001100100011011000101111010111101000110010100100011011
101101110110011011101000111001001100101011000010110110100000000110110101100
00101101001011011100000000011000110110111011101101011101000000000000000000
```

# High-Level Languages

in the early 1950's, *assembly languages* evolved from machine languages

- an assembly language substitutes words for binary codes
- much easier to remember and use words, but still a low level of abstraction (instructions correspond to hardware operations)

in the late 1950's, *high-level languages* were introduced

- high-level languages allow the programmer to write code closer to the way humans think (as opposed to mimicking hardware operations)
- a much more natural way to solve problems
- plus, programs are machine independent

two high level languages that perform the same task (in JavaScript and C++)

<pre>&lt;html&gt; &lt;!-- hello.html          Dave Reed --&gt; &lt;!-- This page displays a greeting. --&gt; &lt;!-- -----&gt; &lt;head&gt;   &lt;title&gt;Greetings&lt;/title&gt; &lt;/head&gt; &lt;body&gt;   &lt;script type="text/javascript"&gt;     username = prompt("Enter your name", "");     document.write("Hello " + username + "!");   &lt;/script&gt; &lt;/body&gt; &lt;/html&gt;</pre>	<pre>// hello.cpp          Dave Reed // This program displays a greeting. // ----- #include &lt;iostream&gt; #include &lt;string&gt; using namespace std;  int main() {   string userName;   cout &lt;&lt; "Enter your name" &lt;&lt; endl;   cin &gt;&gt; userName;   cout &lt;&lt; "Hello " &lt;&lt; userName &lt;&lt; "!";   return 0; }</pre>
--	---

## Program Translation

using a high-level language, the programmer is able to reason at a high-level of abstraction

- but programs must still be translated into machine language that the computer hardware can understand/execute

there are two standard approaches to program translation

- interpretation
- compilation

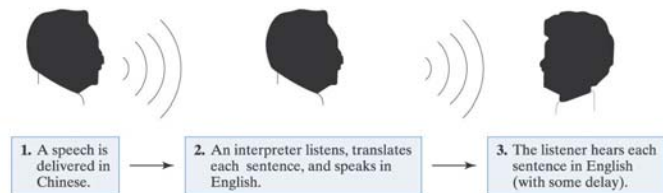
real-world analogy: translating a speech from one language to another

- an *interpreter* can be used provide a real-time translation
  - the interpreter hears a phrase, translates, and immediately speaks the translation
  - ADVANTAGE: the translation is immediate
  - DISADVANTAGE: if you want to hear the speech again, must interpret all over again
- a *translator (or compiler)* translates the entire speech offline
  - the translator takes a copy of the speech, returns when the entire speech is translated
  - ADVANTAGE: once translated, it can be read over and over very quickly
  - DISADVANTAGE: must wait for the entire speech to be translated

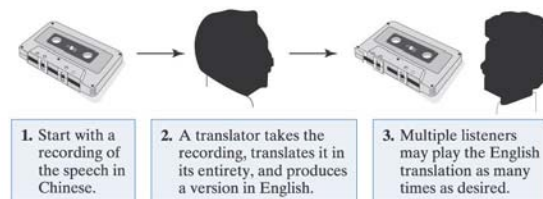
19

## Speech Translation

Interpreter:



Translator (compiler):

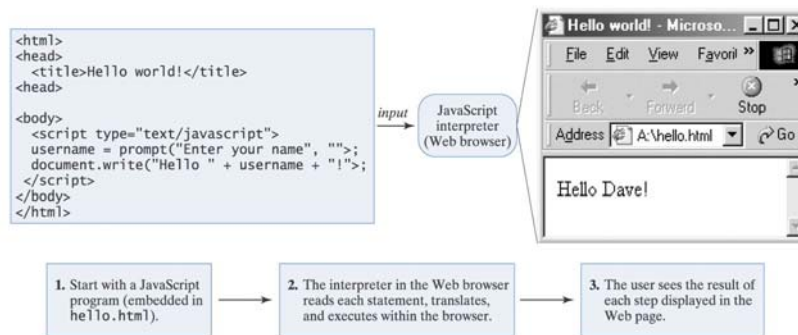


20

# Interpreters

for program translation, the interpretation approach relies on a program known as an *interpreter* to translate and execute high-level statements

- the interpreter reads one high-level statement at a time, immediately translating and executing the statement before processing the next one
- JavaScript is an interpreted language

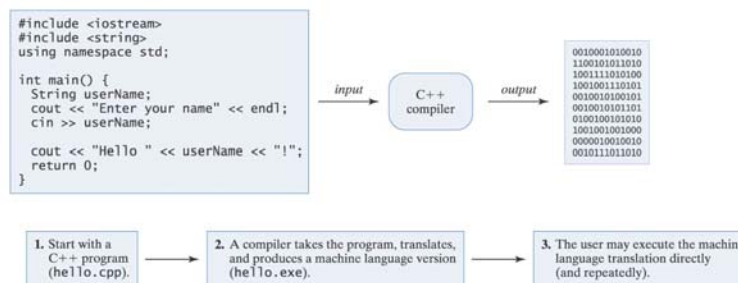


21

# Compilers

the compilation approach relies on a program known as a *compiler* to translate the entire high-level language program into its equivalent machine-language instructions

- the resulting machine-language program can be executed directly on the computer
- most languages used for the development of commercial software employ the compilation technique (C, C++)



22

# Interpreters and Compilers



tradeoffs between interpretation and compilation

interpreter

- produces results almost immediately
- particularly useful for dynamic, interactive features of web pages
- program executes more slowly (slight delay between the execution of statements)

compiler

- produces machine-language program that can run directly on the underlying hardware
- program runs very fast after compilation
- must compile the entire program before execution
- used in large software applications when speed is of the utmost importance