Internet Programming

Programming and Algorithms

Dan Brandon, Ph.D., PMP

Copyright Dan Brandon, PhD, PMP

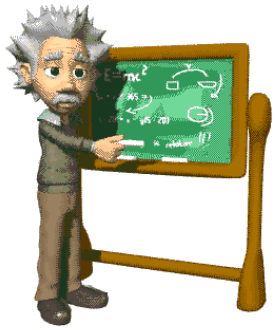
Half of the high-paying jobs in America now require this skill

By Catey Hill in MarketWatch [www.marketwatch.com]

Published: Aug 29, 2016 9:30 a.m. ET

Roughly half of the jobs in the top income quartile — defined as those paying $57,000 or more per year — are in occupations that commonly require applicants to have at least some computer coding knowledge or skill, according to an analysis of 26 million U.S. online job postings released this month by job market analytics firm Burning Glass and Oracle Academy, the philanthropic arm of Oracle focused on computer science education, in Redwood City, Calif. In simple terms, coders write the instructions that tell computers what to do; in-demand programming languages include SQL, Java, JavaScript, C# and Python.

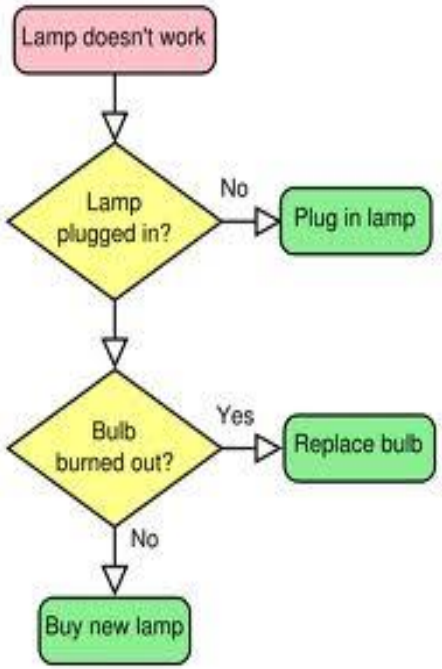
This high number is thanks, in part, to the fact that it's not just technology jobs that now require at least some coding knowledge, says Alison Derbenwick Miller, the vice president of Oracle Academy. "Computing has become a tool in every industry," which means that coding knowledge is now needed for workers across fields, she says. Indeed, everyone from business people who work with data to designers and marketers who create websites to scientists who conduct research now need at least some coding knowledge.



Employers and employees — even those who aren't in the technology field — say the same. Jake Lane, a growth analyst at lawn care company LawnStarter, says that "having some knowledge of coding is essential for job seekers these days," as it can help them understand the tasks of — and work more effectively with — other departments, including their tech and engineering

Algorithm

- Wikipedia:
  - In mathematics, computer science, and other areas, an **algorithm** is an effective method expressed as a finite list of well-defined instructions for calculating a result

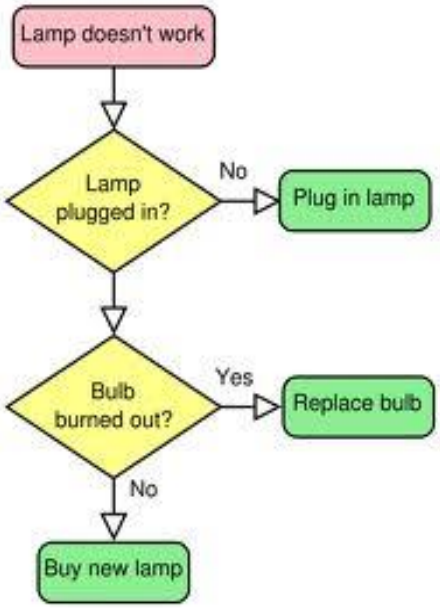


The Google Way

[Infoworld, 2/23/04]

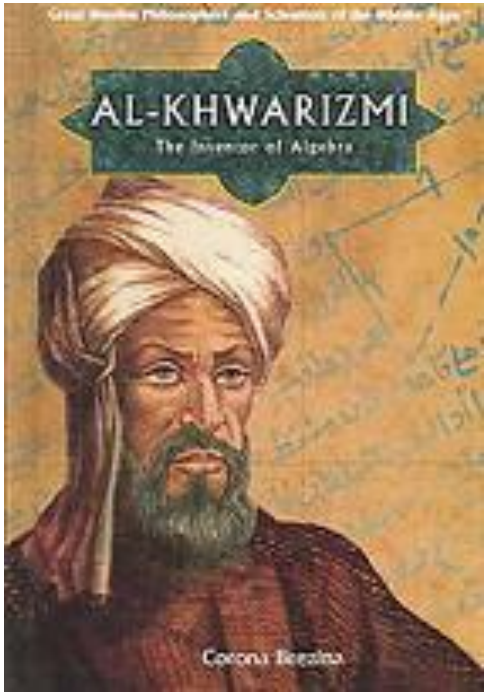
- The Google corporate philosophy is expressed in five principles:
  - "Work on things that matter
  - Affect everyone in the world
  - Solve problems with algorithms if possible
  - Hire bright people and give them lots of freedom
  - Don't be afraid to try new things“

Where did the word “algorithm come from ?”



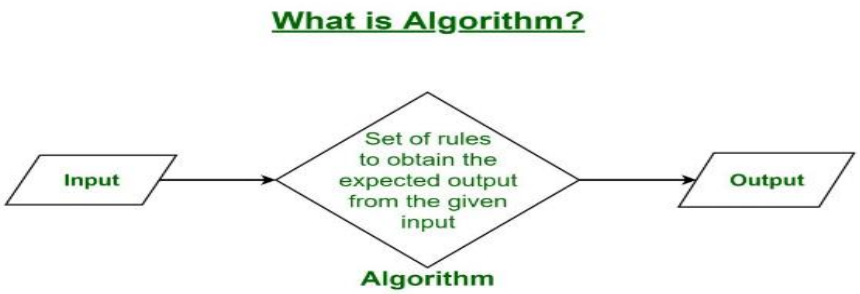
Don’t look ahead !

- Last name of Persian mathematician Mohammad ibn-Musa Al-Khowarizmi, which sounds more like algorithm when written in Latin



Algorithms

- Briefly: The systematic solution to a specific problem, represented in a series of well defined steps
- Computer application - the steps are performed by a computing agent, or the interaction between a computing agent and some other form of intelligence (ie human)



Is this an algorithm?

- Chocolate “mousse” :
  - 1. Melt chocolate and 2 tablespoons water in double boiler. Set aside.
  - 2. Beat egg yolks until thick and lemon-colored, about 5 minutes.
  - 3. Gently fold in 8 ounces of semi-sweet chocolate.
  - 4. Reheat slightly to melt chocolate, if necessary.
  - 5. Stir in 1 tablespoon of rum and vanilla. Beat egg whites until frothy.
  - 6. Beat in 2 tablespoons sugar until stiff peaks form.
  - 7. Gently fold whites into chocolate-yolk mixture.
  - 8. Pour into individual serving dishes. Chill at least 4 hours.
  - 9. Serve with whipped cream, if desired.
    - Makes 6 to 8 servings

Don’t look ahead !

What's wrong with the algorithm to make chocolate mousse?

- It uses words and phrases whose meanings are not clear to some people (it’s ambiguous)
  - double boiler, fold in, frothy, stiff peaks form

To correct the mousse algorithm:

- We'd have to use more specific and well understood terms to express such things as "frothy" and "fold" to make it not so ambiguous

Recipe with "Flow Chart"

Recipe
CHOCOLATE CAKE

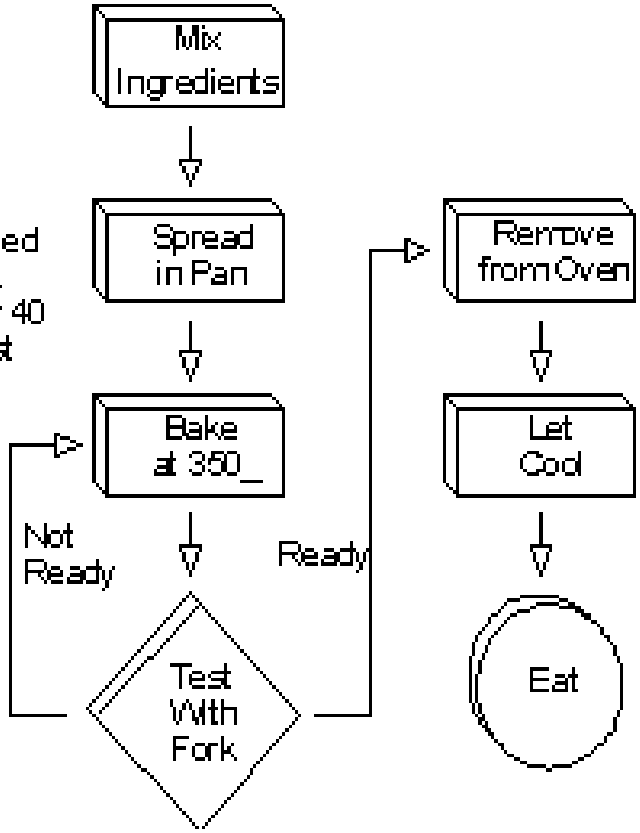
4 oz. chocolate	3 eggs
1 cup butter	1 tsp. vanilla
2 cups sugar	1 cup flour

Melt chocolate and butter. Stir sugar into melted chocolate. Stir in eggs and vanilla. Mix in flour. Spread mix in greased pan. Bake at 350_ for 40 minutes or until inserted fork comes out almost clean. Cool in pan before eating.

Program Code

```
Declare variables:
    chocolate   eggs        mix
    butter      vanilla
    sugar       flour

mix = melted ((4*chocolate) + butter)
mix = stir (mix + (2*sugar))
mix = stir (mix + (3*eggs) + vanilla)
mix = mix + flour
spread (mix)
While not clean (fork)
bake (mix, 350)
```



Is this an algorithm?

1. Wet hair
2. Lather
3. Rinse
4. Repeat

Don’t look ahead !

What's wrong with the algorithm to wash your hair?

- It has no end !

To correct the hair-washing algorithm:

- We'd have to say something like:
  - "Repeat steps 2 and 3 twice and then stop to make it a finite process."

Table of Contents

IS THIS AN ALGORITHM?

- 1. Make a list of all the prime numbers
- 2. Put the list in ascending order
- 3. Print each 10th element
- 4. STOP

Prime Number – a number that has only **two** factors, itself and 1.

7 is prime because the only numbers that will divide into it evenly are 1 and 7

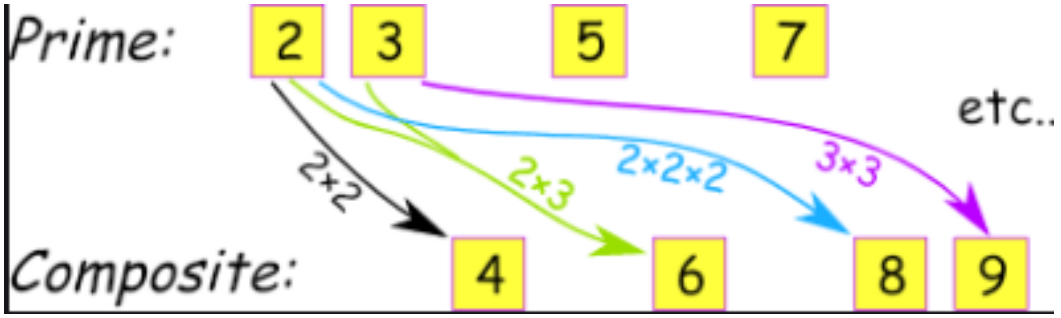
$1 \div 7 = 7$ (two factors)

$1 \times 7 = 7$ (two factors)

Don’t look ahead !

What's wrong with this algorithm?

- One of the steps is impossible to do !

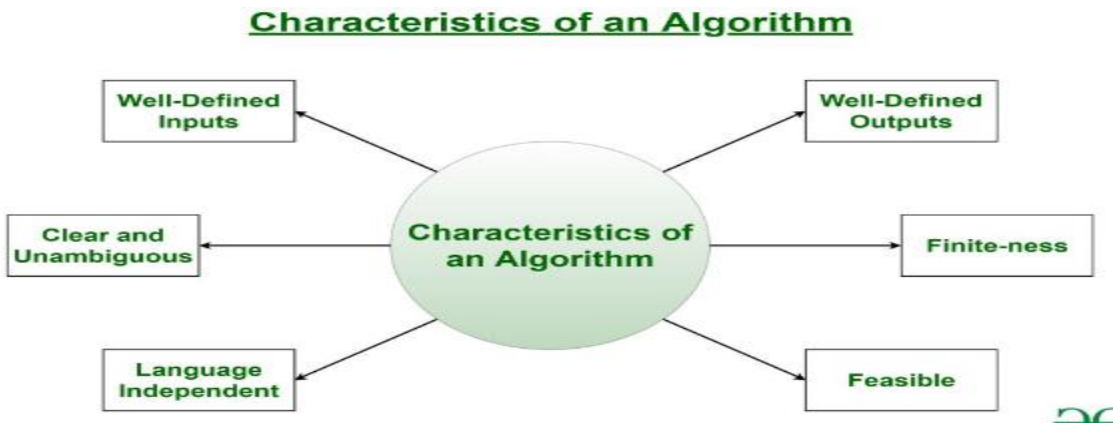


To correct this algorithm:

- We'd have to say something like:
  - "Make a list of the prime numbers less than X"
- So that it is computable

A Good Algorithm:

- Provides a step-by-step process in the correct order
- Is unambiguous
- Is “do-able”
- Has an end
- Provides some useful output or result



CS Algorithm Definition

- An algorithm is a well-ordered collection of unambiguous and effectively computable operations that when executed produces a result and halts in a finite amount of time

Writing An Algorithm

- In order to write an algorithm, following things are needed:
  - The **problem** that is to be solved by this algorithm
  - The **constraints** of the problem that must be considered
  - The **input** to be taken to solve the problem
  - The **output** to be expected when the problem the is solved
  - The **solution** to this problem, in the given constraints

It follows that in order to write a good algorithm, you must know..

- The set of words that your computing agent can understand
- The set of operations that your computing agent is capable of performing

The “Find The Maximum” Algorithm

- Let’s say I handed you 16 pieces of paper, each of which had a number on it
- Now write down the process you would go through to find the largest of the numbers

Did your algorithm have all of these features ?

- Provides a step-by-step process in the correct order
- Is unambiguous
- Is “do-able”
- Has an end
- Provides some useful output or result

How Do You Tell A Computer How To Perform A Process?

- 1. You give it step by step instructions using a computer language
- 2. The computer will perform those steps in the order in which they are presented to it

Tell A Computer How To Perform A Process (con't)

- 3.You can, however, tell the computer to:
  - Start at a certain step, and proceed sequentially step by step
  - Repeat certain steps
  - Skip certain steps under certain circumstances

In Other Words, The Computer Can Perform Operations Three Ways:

- Sequentially
- Conditionally
- Iteratively

The Branching/Conditional Constructs

- IF
- IF/ELSE

- Format of: IF
  - IF (THIS IS TRUE)
    - DO SOMETHING;

Conditional (con't)

- Format of: IF/ELSE
  - IF (SOME CONDITION IS TRUE)
    - DO SOMETHING;
  - ELSE
    - DO SOMETHINGELSE;

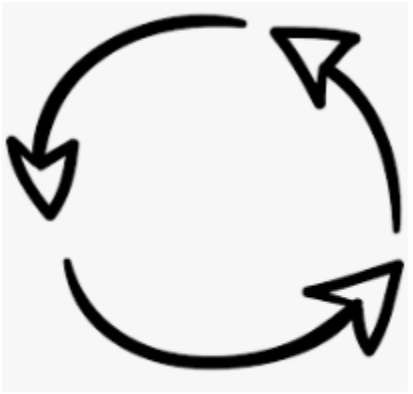
Here is an example of when you would use if:

- IF (USER PRESSES THE ESCAPE KEY)
  - EXIT THE PROGRAM

Here is an another example of when you would use if/else:

- PROMPT USER FOR NUMBER OF HOURS WORKED THIS WEEK
  - IF (NUMBER OF HOURS <=40)
    - GROSS = RATE * HOURS
  - ELSE
    - GROSS=(RATE*40)+(HOURS-40)*(RATE*1.5)

The Looping/Iterative Construct:



- WHILE “loop”

- The format for: WHILE
- WHILE (“THIS STATEMENT” IS TRUE)
  - D0 SOMETHING;
- (“SOMETHING” IS STATEMENT (s) THAT WILL EVENTUALLY CAUSE “ THIS STATEMENT” TO BE FALSE)

Loops (con't)

- Other looping structures
  - Do ... While
  - Repeat...Until
  - For ...

“Find The Maximum Value" Algorithm

- 1. Say you have a stack of index cards with a number on each one. You want to find the largest number.
- 2. You look at the first number. It is the largest so far so you set it aside as the "max".
- 3. You look at the next number. You compare it to the "max" value. If its larger, it becomes the max; otherwise discard it.
- 4. You repeat step 3 until you reach the end of the "pile". The number set aside as the max is the largest.

Next, tell a computer how to do it.

- Computers don't deal with "index cards", "setting things aside" and "piles"
- They deal with values stored in memory cells
- In addition, data must be stored in the memory cell(s) using a specific format or structure

I/O

- In order to have the computer find the maximum value, we have to give it some input and it will have to provide us with some output
- In other words, we have to give the computer several data items that it must store in the appropriately defined memory cells

We refer to these memory cells as variables:

- In high school algebra, a variable was a number with an unknown value that was represented by a letter (Ex: x=2y + 3)

- To a computer, a variable is a storage cell in memory that can store a single data item

Basic (Intrinsic) Items

- There are some basic or “intrinsic”data items (variable types) that an algorithm might need to manipulate (each must be stored in an appropriately-defined memory cell):
  - Integers
  - Real numbers (floating point or real)
  - Characters (text)

Complex Items

- There are also some not-so-basic data items that an algorithm might need to manipulate (and each of these must be stored in an appropriately defined memory cell or cells):
  - Character strings (names, etc.)
  - Arrays (a series of numbers)
  - Objects (user defined data items)
  - Data Structures (Groups of Items)
    - Stacks (LIFO arrays)
    - Queues (FIFO arrays)
    - Trees

It is necessary to use the correct type of memory cell when storing data because:

- Putting a value in a particular type of memory cell (the memory cell has been defined to hold a certain type) gives that value meaning or context
- Without structure, the data has no defined meaning Its just 010110110111000
- The computer has to know how to interpret those 1's and 0's (bits)
  - Perhaps as the ASCII or UNICODE code for a character
  - Perhaps as the binary representation for a number

Variable types (con't)

- Also, a value must be in the appropriate type (format) so that the program will only perform appropriate processes on that data (i.e. it won't try to multiply characters)

Now that we know something about writing algorithms, and data types, we can make a second attempted to re-write the “find max” algorithm assuming that our computing agent is a basic comp

- MAX = FIRST NUMBER
- WHILE (MORE NUMBERS TO LOOK AT)
  - {
    - GET NEXT NUMBER
    - IF THIS NUMBER IS BIGGER THAN MAX THEN MAX = THIS NUMBER
  - }
  - OUTPUT MAX

Implementing the Algorithm -Programming

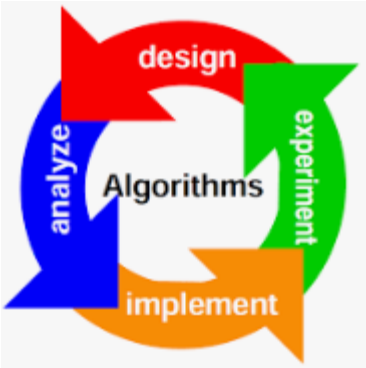
- Writing the program
  - Specific computer language
  - Editor
- Checking the program (not for interpreted languages)
  - Compiling
  - Removing syntax errors
  - Linking
- Running Program
  - Debugging (removing runtime errors)

Languages

- Machine Language
  - Ones & Zeros
- Assembly Language
- High Level Languages
  - Third Generation Languages
    - C and C++
    - COBOL
    - Fortran
    - Java
    - JavaScript (an interpreted language – no compiler)
    - PHP
    - Python
    - R

Representing Algorithms

- Pseudocode
- Flowcharts
- Dataflow Diagrams
- Event Diagrams & Use Cases



Pseudocode

- Informal language to express algorithm
- Not executed on computers
- Consists only of action statements, leaving out declarations
- Avoids syntax details (ie semicolons, etc)

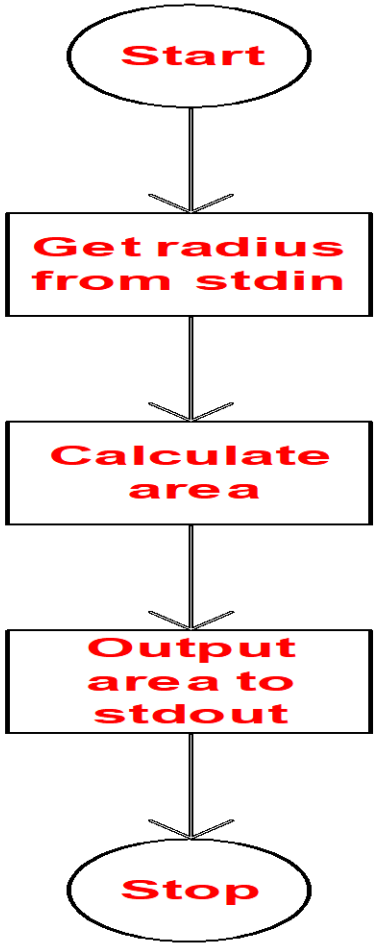
- Pseudocode:
  - if a is greater than b
    - print “a is bigger”
- JavaScript code:
  - if (a > b)
    - alert (“a is bigger”);

```
start
   input testScore, classRank
   if testScore >= 90 then
      if classRank >= 25 then
        output "Accept"
      else
        output "Reject"
      endif
   else
      if testScore >= 80 then
        if classRank >= 50 then
          output "Accept"
        else
          output "Reject"
        endif
      else
        if testScore >= 70 then
          if classRank >= 75 then
            output "Accept"
          else
            output "Reject"
          endif
        else
          output "Reject"
        endif
      endif
   endif
stop
```

Flowcharts

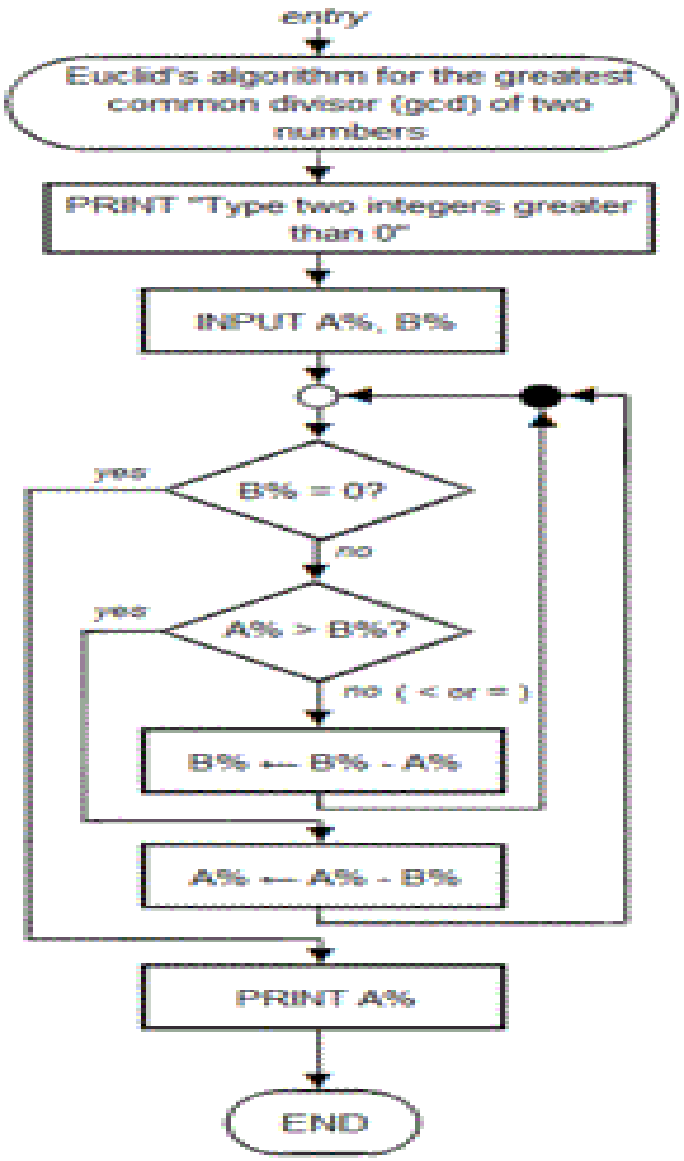
- A graphical representation of an algorithm or portion thereof
- Arrows - flowlines
- Symbols
  - rectangle (action) - some processing
  - diamond (decision) - a decision is made
  - trapezoid (i/0) - input is obtained, or data output
  - oval - a starting or stopping point

Sample Flowchart



When writing algorithms, we can use the following flow control structures :

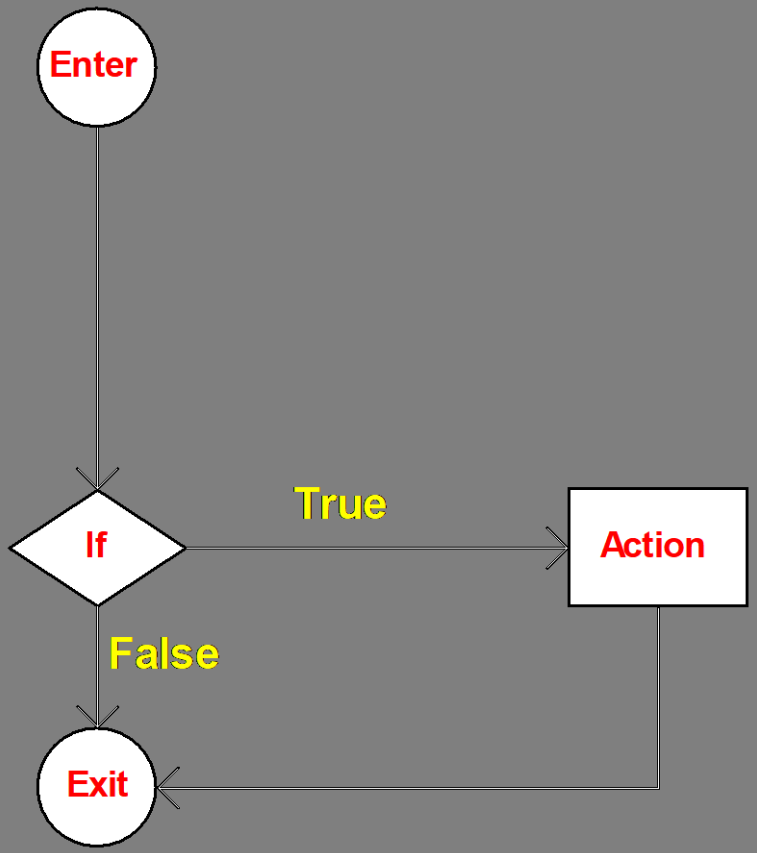
- SEQUENTIAL
  - Do this and then this
- CONDITIONAL
  - Do this or don't do this [if]
  - Do this or do that [if/else]
- REPETITION (LOOP)
  - Do something until this happens (repeat/until)
  - While this is true do something


entry
Euclid's algorithm for the greatest common divisor (gcd) of two numbers
PRINT "Type two integers greater than 0"
INPUT A%, B%
yes
B% = 0?
no
yes
A% > B%?
no (< or =)
B% ← B% - A%
A% ← A% - B%
PRINT A%
END

if

- A single selection structure
- Selects or ignores an action
- Pseudocode:
  - if a is greater than b, then print “a more than b”
- JavaScript code
  - if (a > b)
    - alert (“a more than b”);

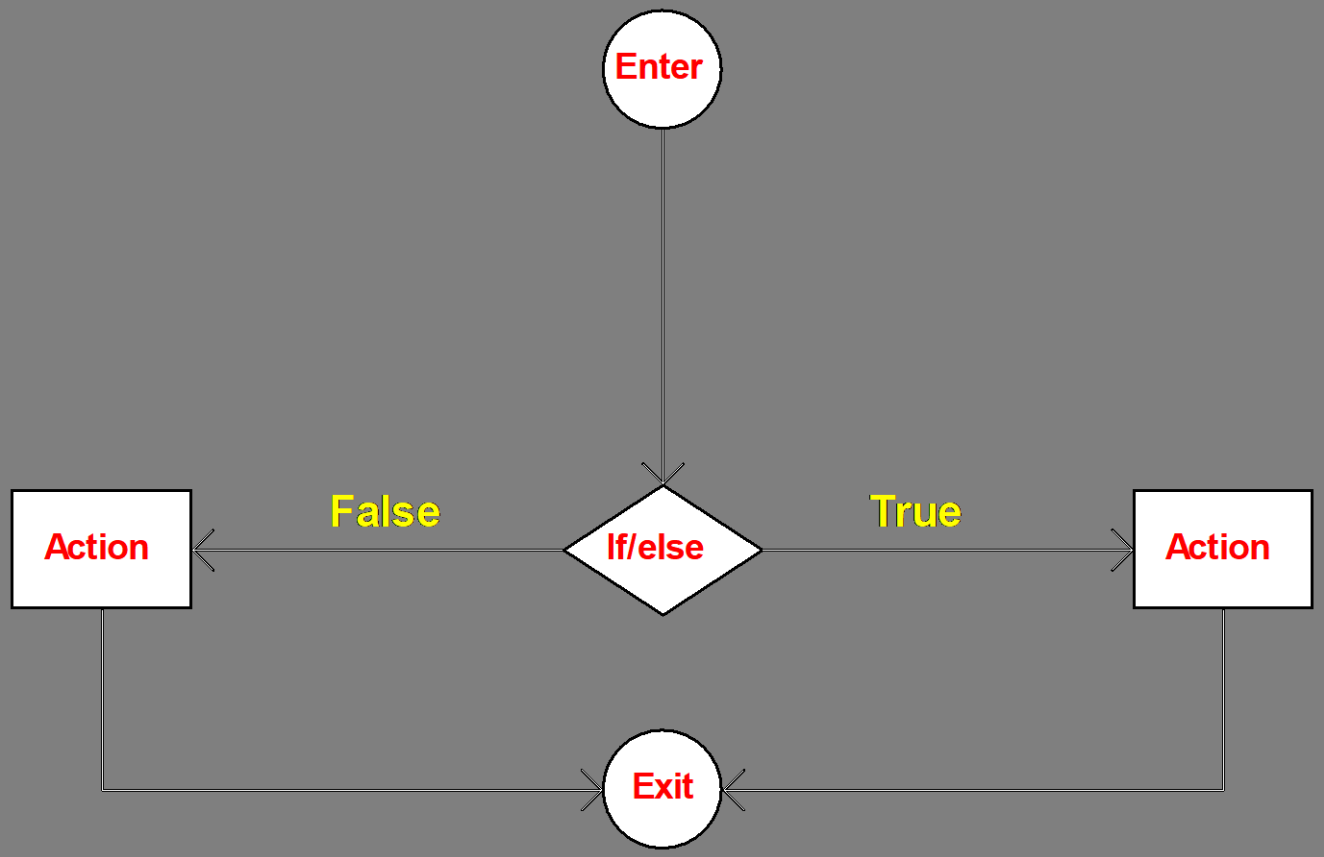
if Flowchart



if/else

- Double Selection
- Pseudocode:
  - If a greater than b then
    - print “yes”
  - else
    - print “no”
- JavaScript code:
  - if (a > b)
    - alert (“yes”);
  - else
    - alert(“no”);

if/else Flowchart



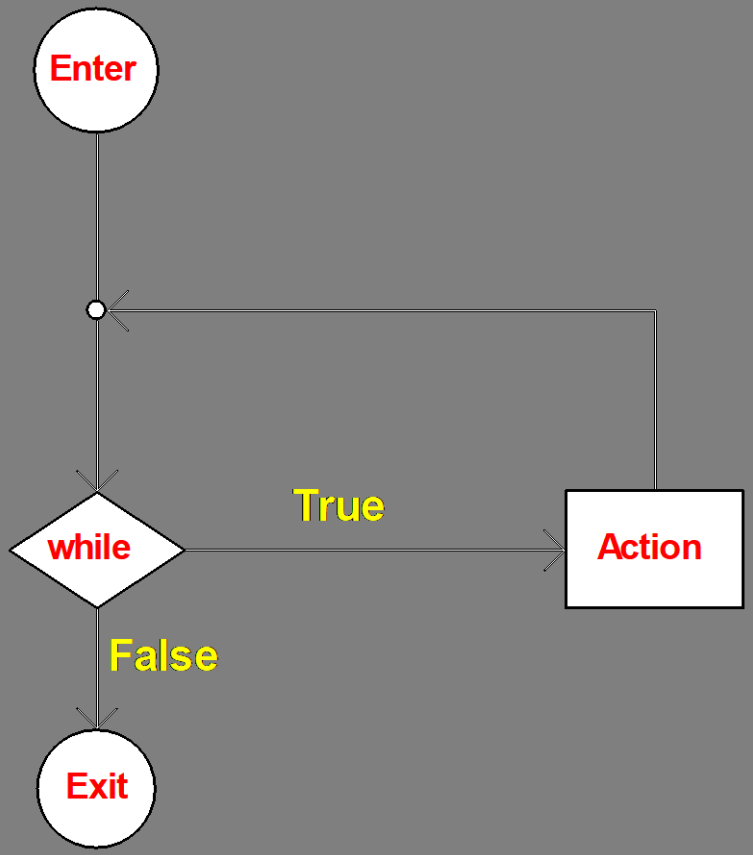
JavaScript while loop

- while (some condition is true)
  - statement or block of statements

- count = 1;
- while (count <= 20)
- {
  - alert (count);
  - count = count + 1; or count++;
- }

What does this code do ?

while Flowchart



While Cautions

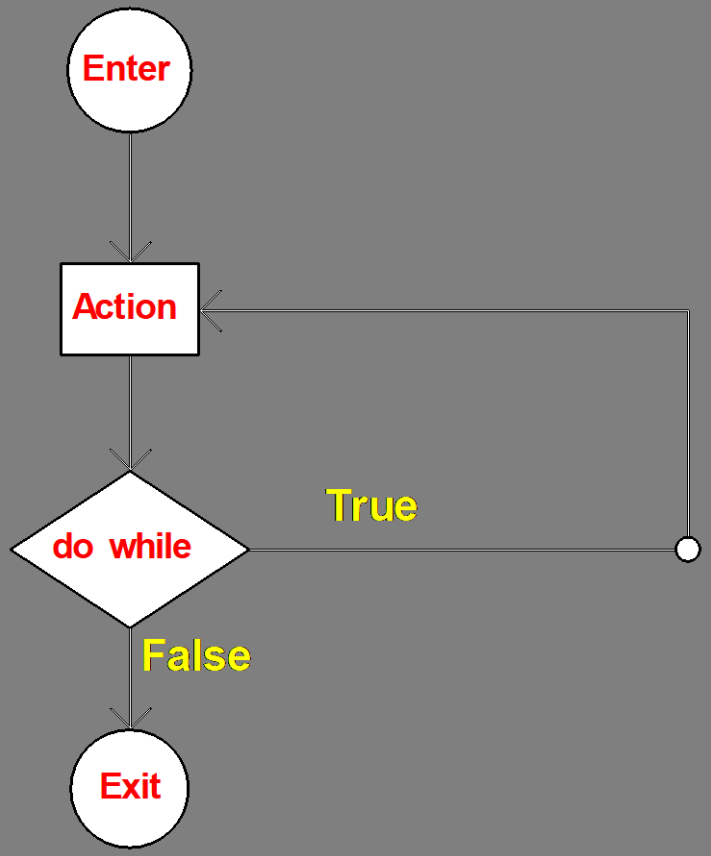
- Remember while tests condition *before* repetitions!
- Remember to initialize counter
- Remember to include a termination condition, so you do not get an *infinite loop*

do/while loop

- do
  - statement(s);
- while (condition);

- do/while does statement(s) first time before testing condition

Do/While Flowchart

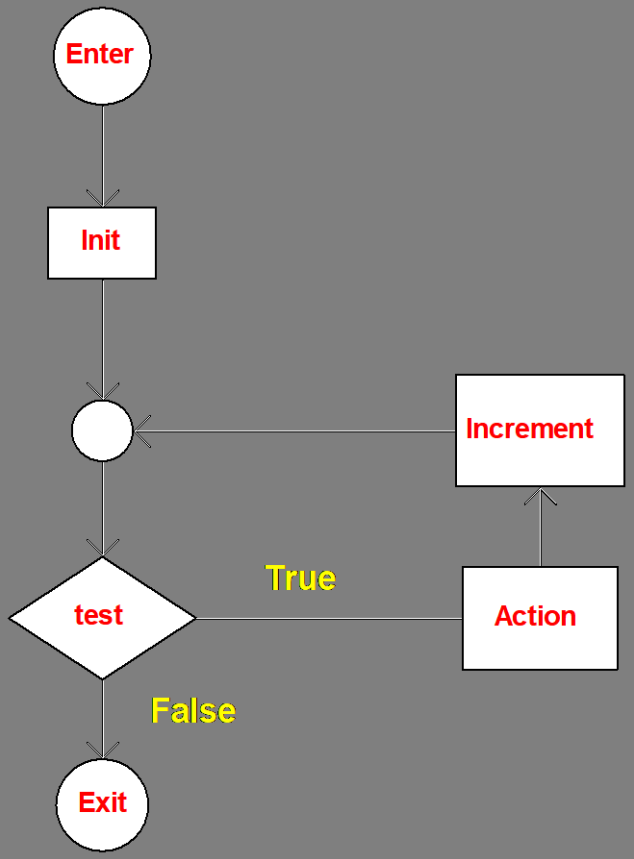


for

- for (initial; test condition; increment)
  - statement or block of statements

- for (count =1; count <=20; count = count + 1)
  - alert (count);
  - can use: count++ instead of count = count+1

For Flowchart

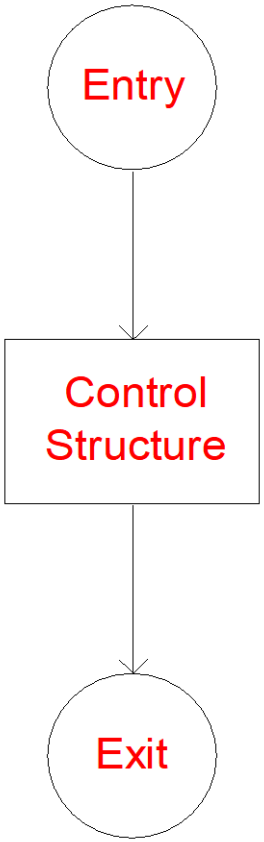


Structured Programming
[goto less programming]

- Single entry for each control structure
- Single exit for each control structure
- Combine control structures by
  - Stacking
    - Exit of one structure is entry for another
  - Nesting

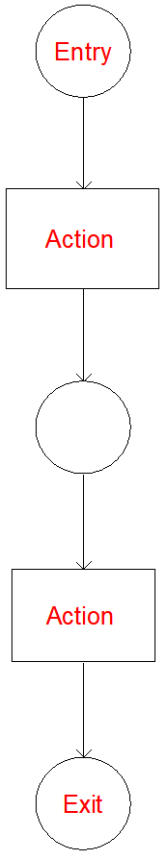
Copyright: Dan Brandon, PhD

Single Entry - Single Exit

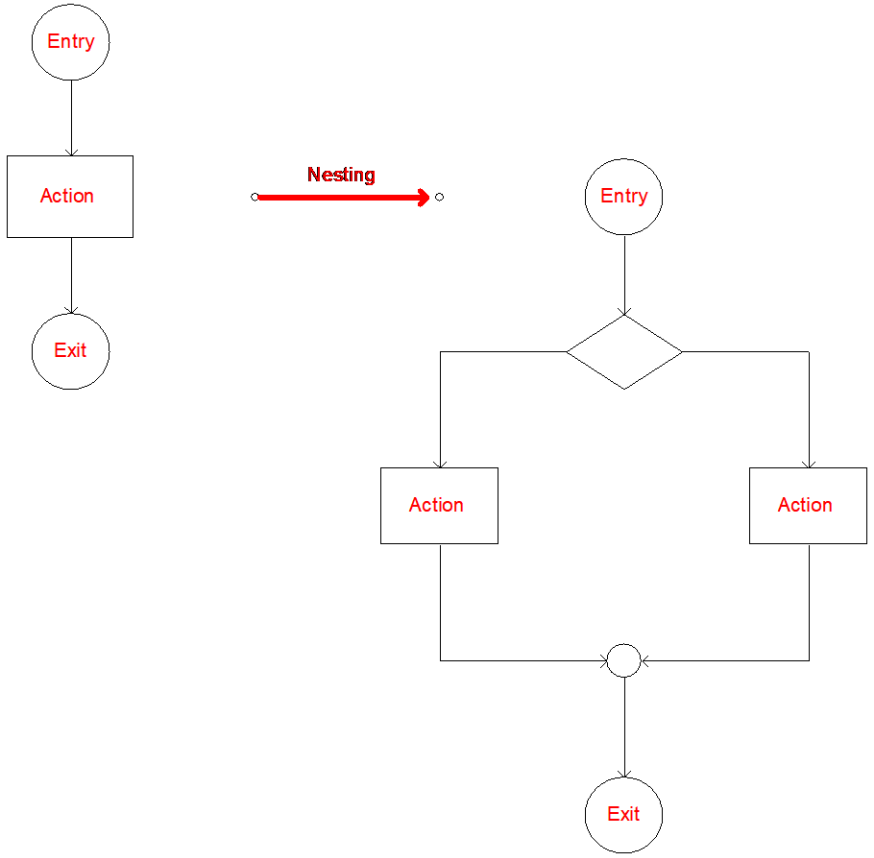


Copyright: Dan Brandon, PhD

Stacking

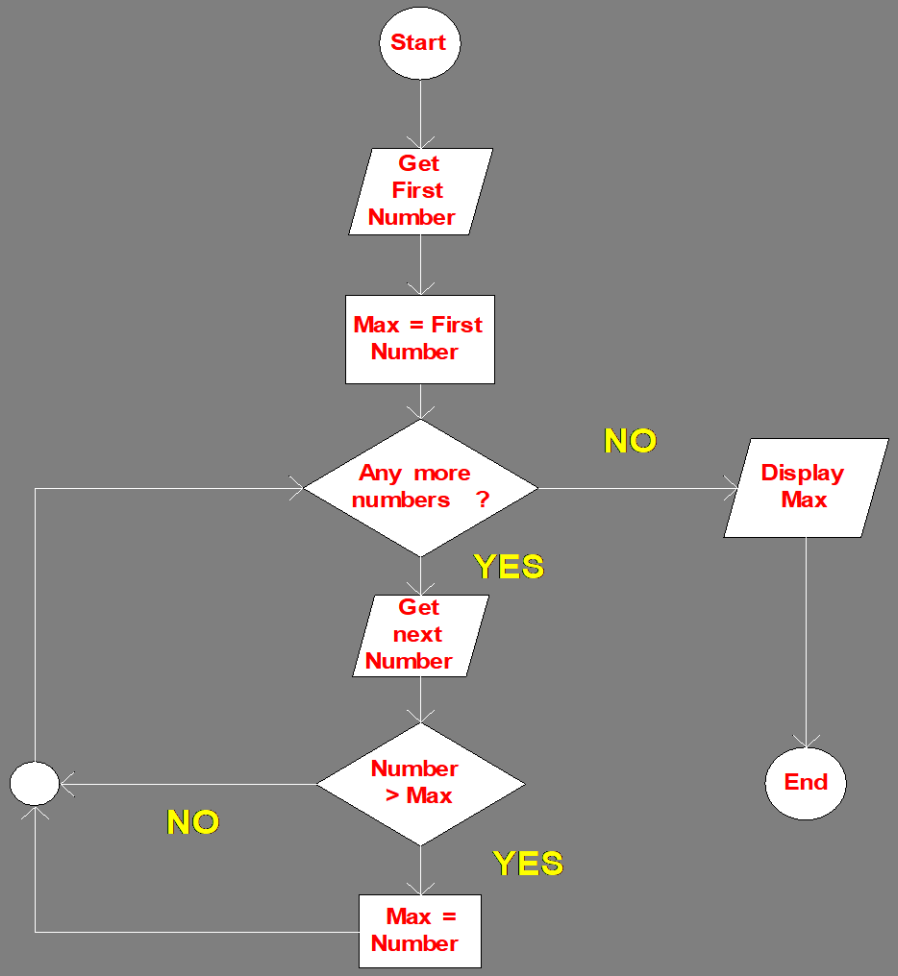


Nesting



Copyright: Dan Brandon, PhD

Find Maximum Flowchart



Find Maximum Algorithm [“sentenel” = 0]

- Get first number from user (enter zero to end)
- MAX = first number
- While (number not zero)
  - {
    - if number greater than MAX then MAX = number
    - get next number from user
  - }
- Display MAX

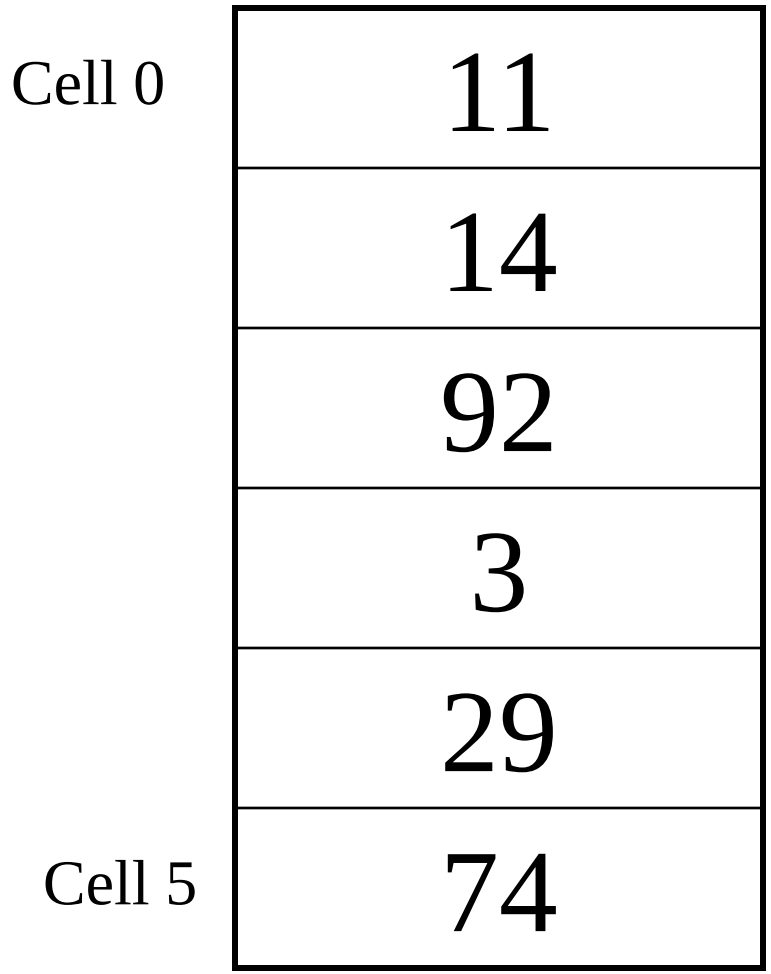
Arrays

- *Consecutive* cells in memory, of a particular *type*
- The array is given a name - myArray
- Cells are numbered consecutively (starting with zero in JavaScript)

Arrays (con't)

- Reference to individual cells is by giving array name and relative position (index or subscript) in array:
  - MyArray [12]
  - JavaScript starts numbering from zero, so this is the thirteenth item in the array

MyArray



MyArray[2] is 92

"for" loops – normally used with Arrays

[when you know how many times you want to do something]

```
for (i = 0; i < n; i=i+1)
{
    //do something n times
}
```

Alternatively: `for (i = 0; i < n; i++)`

Functions

- Programming languages typically have functions some of which may be predefined, and others can be created by the programmer
- Functions can have input arguments and return values
- For example, one might create a function to calculate pay:
  - Pay(hourly-rate, hours-worked)
  - Which would presumably multiply the two arguments together and return the result

Object Orientation

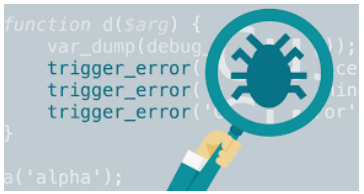
- In OO languages, objects are built and/or created (either directly or thru classes)
- These objects have associated:
  - Properties
  - Functions
- For example if we have a “rectangle object”:
  - Properties: length and width
  - Functions: area and perimeter
- Access to properties and functions is via an object (encapsulation):
  - Object.property or Object.function
- A type of object(class) can de derived from another class (inheritance) and the derived class (sub class) can modify the functions (polymorphism)

Algorithm Design vs Programming

- Some students at this point may feel programming is difficult, and perhaps more difficult that algorithm design
- However as you become more familiar with the programming, you will see that algorithm design is typically more demanding and difficult
- Analogy:
  - When you were a baby, learning English was difficult (you had lots to say, if only you knew how to say it)
  - When you are older and know the language well, the hard part is finding the right thing to say !

Debugging

- Learning how to “debug” an algorithm and the associated program is a vital skill:
  - Syntax errors
  - Logic (run time) errors



- Carefully walk thur algorithm and code
- Develop and check code in small “chunks”
- Display intermediate results and check for validity
- Students must debug their own code as a key part of learning !

Debugging (con't)

Debugging Tips

- #1. Print things a lot
- On every single line of code, you should have a sense of what all of the variables values' are. If you're not sure, print them out!
- Then when you run your program, you can look at the console and see how the values might be changing or getting set to null values in ways you're not expecting.
- Sometimes it's helpful to print a fixed string right before you print a variable, so that your print statements don't all run together and you can tell what is being printed from where
- print "about to check some_var"
- print some_var
- Sometimes you may not be sure if a block of code is being run at all. A simple print "got here" is usually enough to see whether you have a mistake in your control flow like if-statements or for-loops.

- #2. Start with code that already works
- When in doubt, start with someone else's existing code that already works. If you're a beginner, you're still more of a Hacker than an Engineer, and so it's better to start with an existing structure and tweak it to meet your needs.
- If you're working on your own project, try googling around for a script that does what you're trying to do. In my web scraping class, I provide working python code that completes the tasks in each lesson.
- Make sure you run the code you find before you make any changes to verify that it works properly and does what it claims to do. Then make small changes to the existing code and test it often to see if your changes have introduced bugs.

Debugging Tips (con't)

- #3. Run your code every time you make a small change
- Do not start with a blank file, sit down and code for an hour and then run your code for the first time. You'll be endlessly confused with all of the little errors you may have created that are now stacked on top of each other. It'll take you forever to peel back all the layers and figure out what is going on.
- Instead, you should be running any script changes or web page updates every few minutes – it's really not possible to test and run your code too often.
- The more code that you change or write between times that you run your code, the more places you have to go back and search if you hit an error.
- Plus, every time you run your code, you're getting feedback on your work. Is it getting closer to what you want, or is it suddenly failing?

- #4. Read the error message
- It's really easy to throw your hands up and say "my code has an error" and feel lost when you see a stacktrace. But in my experience, about 2/3rds of error messages you'll see are fairly accurate and descriptive.
- The language runtime tried to execute your program, but ran into a problem. Maybe something was missing, or there was a typo, or perhaps you skipped a step and now it's not sure what you want it to do.
- The error message does its best to tell you what went wrong. At the very least, it will tell you what line number it got to in your program before crashing, which gives you a great clue for places to start hunting for bugs.

- #5. Google the error message

- If you can't seem to figure out what your error message is trying to tell you, your best bet is to copy and paste the last line of the stacktrace into Google. Chances are, you'll get a few stackoverflow.com results, where people have asked similar questions and gotten explanations and answers. This can sometimes be hit-or-miss, depending on how specific or generic your error is. Make sure you try to read the code in the question and see if it's similar to yours – make sure it's the same language! Then read through the comments and the first 2-3 answers to see if any of them work for you.

Copyright Dan Brandon, PhD, PMP

Debugging Tips (con't)

- #6. Guess and Check
- If you're not 100% sure how to fix something, be open to trying 2 or 3 things to see what happens. You should be running your code often (see #3), so you'll get feedback quickly. Does this fix my error? No? Okay, let's go back and try something else.
- There's a solid possibility that the fix you try may introduce some new error, and it can be hard to tell if you're getting closer or farther away. Try not to go so far down a rabbit hole that you don't know how to get back to where you started.
- Trying a few things on your own is important because if you go to ask someone else for help (see #10), one of the first things they'll ask you for is to list 2-3 things you've tried already. This helps save everyone time by avoiding suggestions you've already tried, and shows that you're committed to solving your problem and not just looking for someone else to give you free, working code.

Debugging Tips (con't)

- #7. Comment-out code
- Every programming language has a concept of a comment, which is a way for developers to leave notes in the code, without the language runtime trying to execute the notes as programming instructions.
- You can take advantage of this language feature by temporarily "commenting out" code that you don't want to lose track of, but that you just don't want running right now. This works by basically just putting the "comment character" for your language at the start (and sometimes at the end) of the lines that you're commenting out.
- If your script is long, you can comment out parts of the code that are unrelated to the specific changes you're working on. This might make it run faster and make it easier to search the remainder of the code for the mistake.
- Just make sure you don't comment out some code that sets variables that your program is using later on – the commented-out code is skipped entirely, so statements that run later on won't have access to any variables that are set or updated in the commented out sections.
- And of course, make sure you remove the comment characters so that it turns back into instructions when you're done testing the other sections.

Copyright Dan Brandon, PhD, PMP

Debugging Tips (con't)

- #8. If you're not sure where the problem is, do a binary search
- The more code you have that's running, the more places you have to check for an error. Especially as your project grows past a few dozen lines, it can get more and more difficult to find out where errors are happening. Your stack trace and error message should give you a clue as to where things are going wrong, but sometimes they're not too helpful.
- In that case, it's helpful to do a binary search to hone in on the section of code that's misbehaving.
- At a high level, a binary search involves splitting something in half and searching each of the halves for what you're looking for. Once you decide which half it's in, you repeat the process again on that half. This is one of the quickest ways to hone in on where something is, in an otherwise large list of instructions.
- For finding bugs in your script or web app, just run the first half of your code, comment out the second half, and then print the half-way done results. If those look right, then the first half of your code is running fine and the problem you're encountering must be in the second half. If there's a problem with the half-way done results, then the error is occurring somewhere in the first half.
- Repeat this process over and over, and you'll be able to quickly hone in on the 2 or 3 lines that seem to be leading your program astray.
- You may have noticed that this method combines a lot of the earlier steps of printing variables out, commenting code out and reading the error messages looking for line numbers.

Debugging Tips (con't)

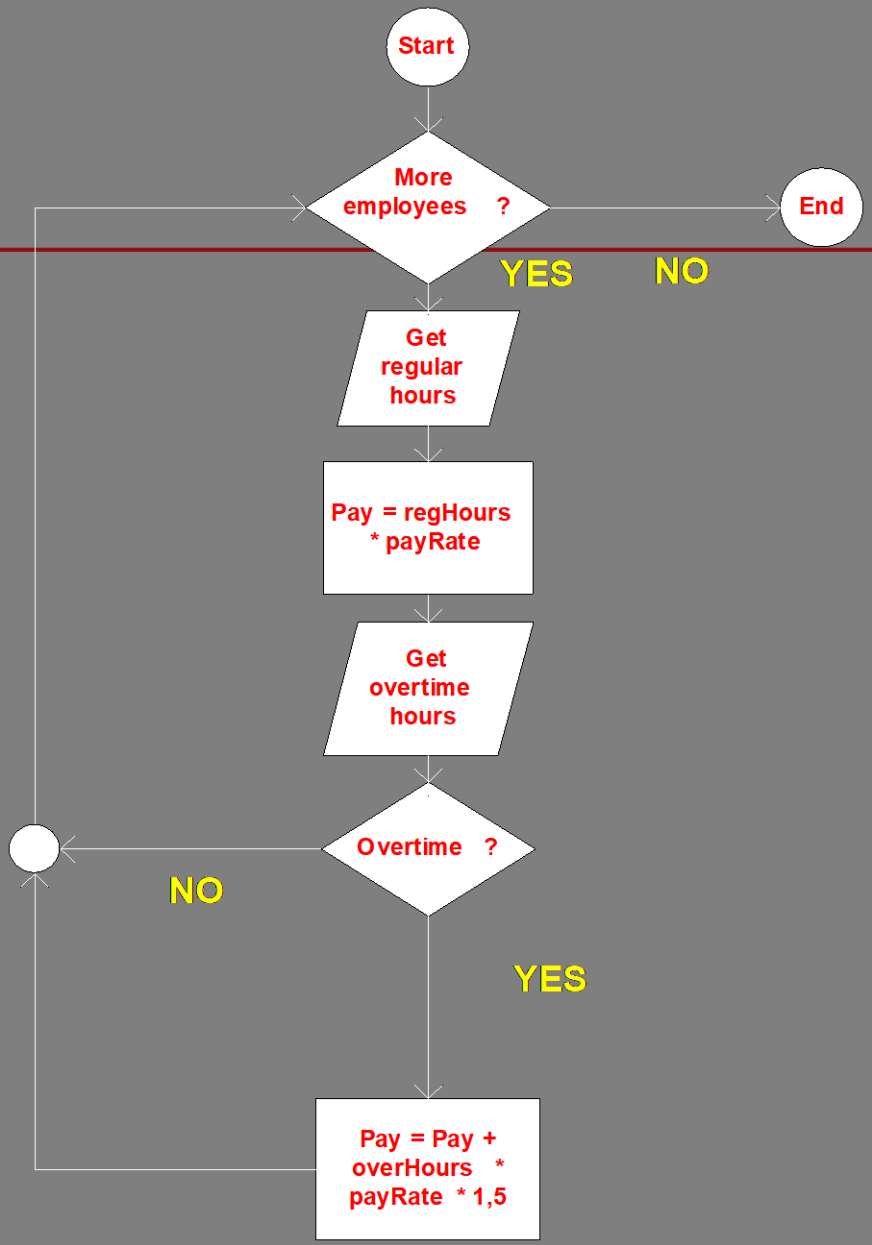
- #9. Take a break and walk away from the keyboard
- It's really easy to get caught up in the details of your current implementation. You get bogged down in little details and start to lose sight of the forest through the trees.
- In cases where I feel like I've been spinning my wheels for the last 20 or 30 minutes, I try to step away from the code and do some other activity for a little while before coming back. I'll go get a drink of water, meander around a bit or have a snack. It's a great way to reset your mind, pull back and start to think of another approach.
- I realize that it's also seemingly unsatisfying if you haven't tried it. "If I walk away now, I'll forget all of these details! I'll have to start all over again. Plus I don't feel satisfied leaving code in a broken state. What if I never fix it and I'm a failure. I can't leave until it's working again." I used to think all of those things as well.
- But it has become one of my favorites tips and has helped me past dozens of bugs over the years. If you try it you might be surprised just how helpful that can be.

Copyright Dan Brandon, PhD, PMP

Class Exercise

- Design a flowchart for the following process:
  - Calculate an employee's gross pay
    - They get paid by the hour
    - They get time-and-a-half for over 40 hours
  - There is more than one employee

Don’t look ahead !



Copyright Dan Brandon, PhD, PMP

Pseudocode

- Input first employee
- While (employee not null)
- {
  - Get employee data (pay-rate, reg-hrs, ovt-hrs)
  - Pay = reg-hours * pay-rate
  - If (ovt-hrs > 0) {
    - Pay = pay + ovt-hrs * pay-rate * 1.5
  - }
  - Output pay
  - Input next employee
- }

Design an Algorithm to Add n Numbers (in an array) - Express in pseudocode

- $Z = A_1 + A_2 + \ldots A_N$
- Numbers are in an array
- Write pseudocode

Don’t look ahead !

- Input array []
- sum = 0 {some languages initialize variables to zero, some don't}
- i = 1 {arrays start at zero in JavaScript}
- while i <= n
  - {
  - sum = sum + array [i]
  - i = i + 1
  - }
- Output sum

Design an Algorithm to Multiply n Numbers

- $Z = A_1 * A_2 * \ldots A_N$
- Numbers are in an array

Don’t look ahead !

- Input array []
- prod = 1
- i = 1
- while i <= n
  - {
  - prod = prod * array [i]
  - i = i + 1
  - }
- Output prod

Design An Algorithm That Can Find N! (N factorial)

- 1! = 1
- 2! = 2 * 1
- 3! = 3 * 2 * 1
- ...

Don’t look ahead !

Forward & Backward Loops

- Input n
- prod = 1
- i = 2
- while i <= n
  - {
  - prod = prod * i
  - i = i + 1
  - }
- Output prod

- Input n
- prod = n
- i = n - 1
- while i > 0
  - {
  - prod = prod * i
  - i = i – 1
  - }
- Output prod

Design An Algorithm That Sums The Numbers From 1 To N

- Do not use any kind of loop
- Use formulas

Don’t look ahead !

Example:

- Add numbers from 1 to 100
  - 1 + 100 = 101
  - 2 + 99 = 101
  - 3 + 98 = 101
  - And so on fifty times

- SUM = (N + 1) * (N/2)
- But this only works for even numbers
- So an algorithm would be:
  - if N is even then SUM = (N + 1) * (N/2)
  - if N is odd then SUM = (N * (N-1)/2) + N
- How do you check for an even number in code?

- if (int(N/2)*2 == N) {or “floor” function}
- Or use mod function if available which calculates the remainder when one divides by X
  - if (mod(N,2) == 0)
- There may be a modulus operator:
  - if((N mod 2) ==0)
- % in JavaScript:
  - if((N % 2) ==0)