CSc 28
Discrete Structures

Chapter 4
Data Structures & Algorithms

Herbert G. Mayer, CSU CSc

Status 1/1/2021

Syllabus

- Algorithm
- Searching
- Interesting Experience
- Number Theory
- Prime Numbers
- Representation of Integers
- Congruence
- Graphs

Algorithm

Definition: Algorithm

- **Informally: Algorithm is a discrete sequence of precise instructions to compute a solution to a problem in finite time**
- **Alternatively from Wikipedia [1]:**
  - **"The word *algorithm* itself is derived from the 9th century mathematician *Muḥammad ibn Mūsā al-Khwārizmī,* Latinized *Algoritmi.***
  - **A partial formalization of what would become the modern concept of algorithm began with attempts to solve the Entscheidungsproblem, posed by Hilbert in 1928.**
  - **Later formalizations were framed as attempts to define Effective Calculability or Effective Method."**

Algorithm

Purpose & Properties of algorithm:

- Algorithm performs operations on data structures
- Input from a specified data set, e.g. file (data source)
- Output onto specified data set or file (solution)
- Unambiguous definition of every compute step
- Correct mapping defined for every possible input, even erroneous input data
- Finiteness of number of calculation steps
- Effectiveness of each calculation step

Algorithm

Named after Persian Mathematician al-Khwarizmi 780-850

Algorithm

Dijkstra's general programmer attributes:

- **Critical thinker, uses good judgment**
- **Creative ideas, about algorithms, computing**
- **Open to others' ideas**
- **Humble ☺, see E. Dijkstra [1]**

Practicing and mastering a craft:

- **Programmer analyzes, improves programs**
- **Designs programs, i.e. SW solutions of computable problems**
- **Practices, reviews, corrects, refines programming process**

Experience and practice:

- **Knows how to design, implement, document, test, validate, improve, maintain, judge complex software**

Mastery and skills:

- **Advances the *state of the art* of SW development to a higher level**
- **Designs, implements cooperatively; true SW jobs generally are not solvable by 1 engineer; except Linus Torvalds implementing Linux ☺**

System Programmer

Necessary professional (Pro) requirements for a successful system programmer:

- The Pro masters and uses some stable, well-defined, high-level programming language L
- Language L allows visibility of, and access to, low-level target machine, or to some OS resources
- Possible languages: Java, C, C++, assembler, . . .
- Pro knows target machine thoroughly
- Understands which operating system services can be expressed in L to manage system resources
- Pro can combine system calls to low-level machine from L, with overall goal of effective machine use

Good Judgment comes from Experience,
and
Experience comes from Bad Judgment

provided that one is smart!

Only a CS genius learns exclusively from the mistakes of others

The smart Sac State CS student learns from her own mistakes

A dumb person –not studying CS ☺ of course– repeats errors

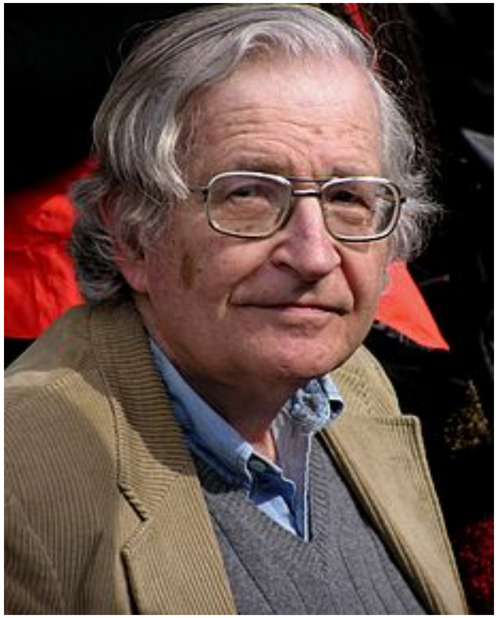
Software Development Practices

SW with true value should be:

- Functional
- Correct
- Reliable
- Efficient
- Easily readable: C. A. R. Hoare [4]
- Re-Usable
- Extendible, AKA open-ended
- Maintainable
- Complete; exceptions are documented clearly!
- Does such SW exist? Then why don't we see & touch it all the time? Ever touch an OS that locks up, freezes, slows down?

Language Complexity

Chomsky complexity measure for grammars, or for languages defined by such grammars, classified according to Noam Chomsky hierarchy:



- Type 3 Regular Expressions
- Type 2 Context Free Languages
- Type 1 Context Sensitive Languages
- Type 0 Recursively Enumerable (AKA free)

Listed in *simple (3)* to *complex (0)* order

See later detailed lecture on language complexity, and program complexity

Chomsky Hierarchy

Ref: http://www.cs.nuim.ie/~jpower/Courses/Previous/parsing/node21.html

Language Class	Grammar	Automaton
3	Regular	NFA or DFA
2	Context-Free	Push-Down Automaton
1	Context-Sensitive	Linear-Bounded Automaton
0	Unrestricted (or *Free*)	Turing Machine

Interesting Experiences

1. I re-typed ~50 lines of Pascal-like system code to render buggy SW fully functional; mystery never solved why the re-typed, "identical" program worked!
2. Designed and coded some days and nights w. almost no sleep and food, and crafted a few 1,000 lines of highly functional C++ code for Ada Case Statement in Ada compiler; worked! Until years later I observed an error in that Case Statement implementation!
3. Colleague left 1 page of pseudo-code for *Symbolic Differentiation* on copy machine; was intuitive, comprehensible, beautiful code; I stole with pride
4. Positive feedback about an almost *religious credo*: do not trust your own SW; but insert checks instead! Catch your own assumptions! Better: make few assumptions

What Can Be Programmed?

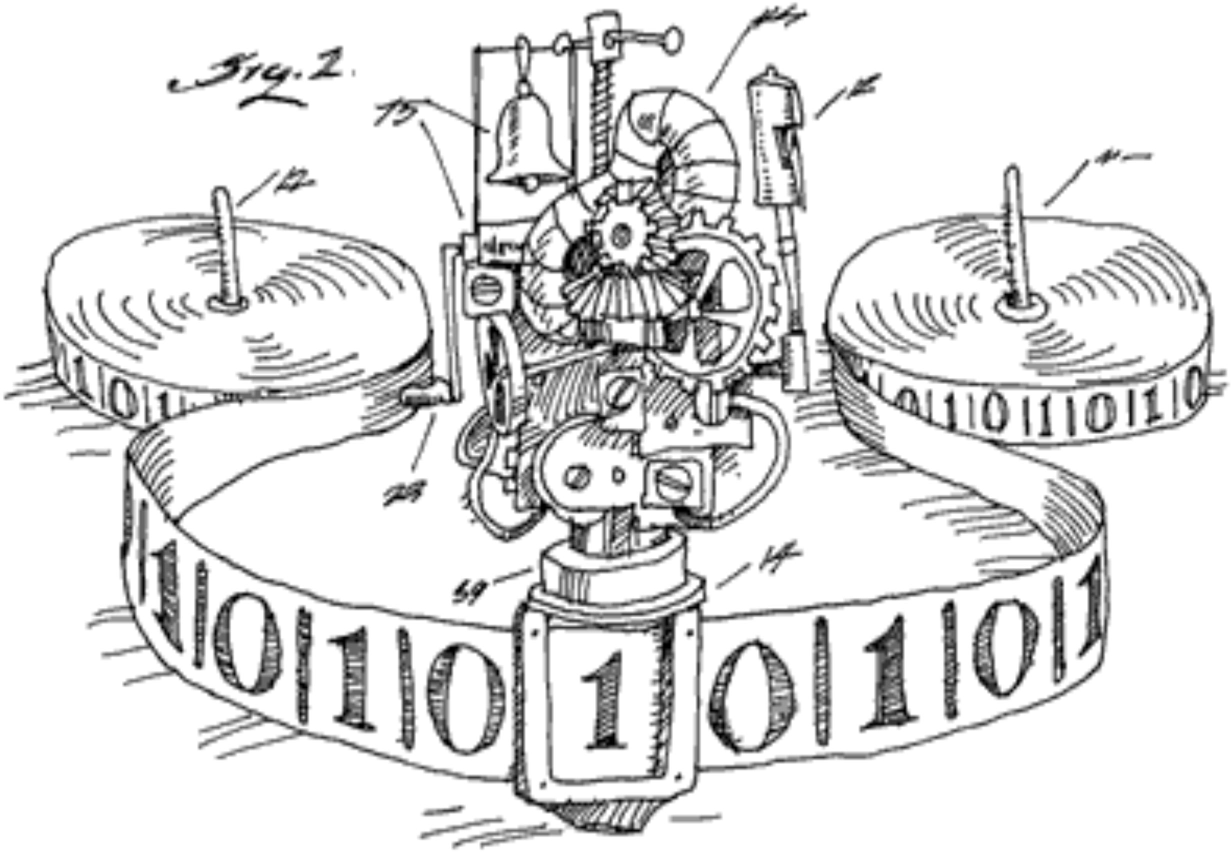
- Definition of algorithm; see [2]
- Guideline for what can be programmed: Church Thesis [6]: “any algorithmically computable function can be programmed and executed on a regular computer.” I.e. can be expressed as a C++ or Java program
- What is “Computable?
  - see Alonzo Church’s Lambda Calculus
  - or Alan Turing’s “Turing Machine”
- Yet some problems remain very hard to solve programmatically, though being computable

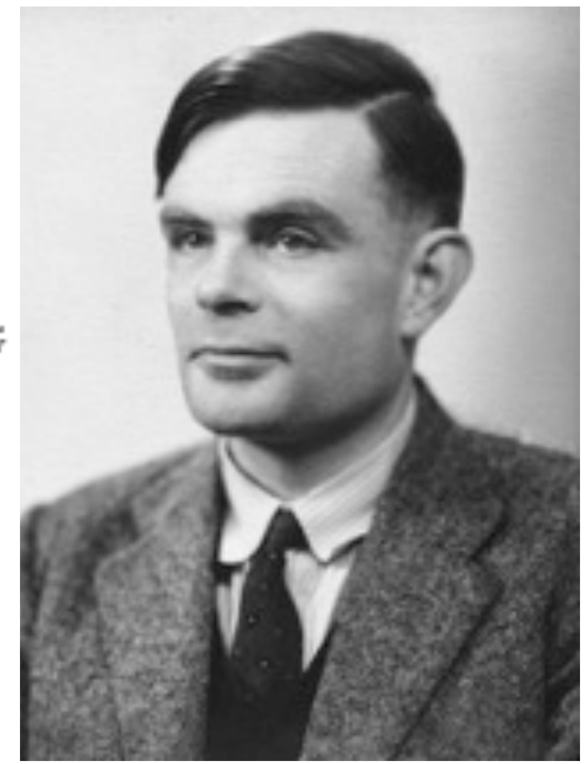
What Can Be Programmed?

American Mathematician Alonzo Church 1903 - 1995

What Can Be Programmed?

Maybe this playful Turing Machine, with bells & whistles, zeros & ones, can be programmed?





What Cannot be Programmed?

- How to become a Perfect System Programmer?
  - There is no methodical, algorithmic, guaranteed way ☹
  - But there *are* ways ☺ Don't give up! Includes learning, practicing, thinking
- Design SW to win lottery?
  - Is easy, provided unbounded time and all permutations allowed; you'd win, but then you'd *lose* money due to large number of ticket bought ☺ for all combinations
  - So you can't win!
- Natural language translator?
  - Yes, there exist automated translation tools! And your teacher uses them! Like Babel Fish, Google translate, etc.
  - If it were possible, how then would common ambiguities in human interaction be avoided? Forever?

What Cannot be Programmed?

- **Decryption of any encrypted code?**
  - May require too much time to render result interesting
  - Hence some nations impose limitations on encryption complexity (e.g. 128 bits) in the US for outside-national communication
- **Medical diagnosis?**
  - Yet great steps achieved to automate some diagnostic steps
  - Mainly to save time; final diagnosis generally left to MD
- **Judicial judgment?**

Algorithm

Markov (1954)

- Algorithm is … an exact prescription, defining a computational process, leading from various initial data to the desired result

Donald Knuth (1968)

- A precisely defined sequence of finite steps to compute a result from given inputs and initial values –paraphrased by HGM ☺

Stone (1972)

- "An algorithm is a set of rules that precisely defines a sequence of operations such that each rule is effective and definite and that the sequence terminates in (very) finite time."

Algorithm

Minsky (1967)

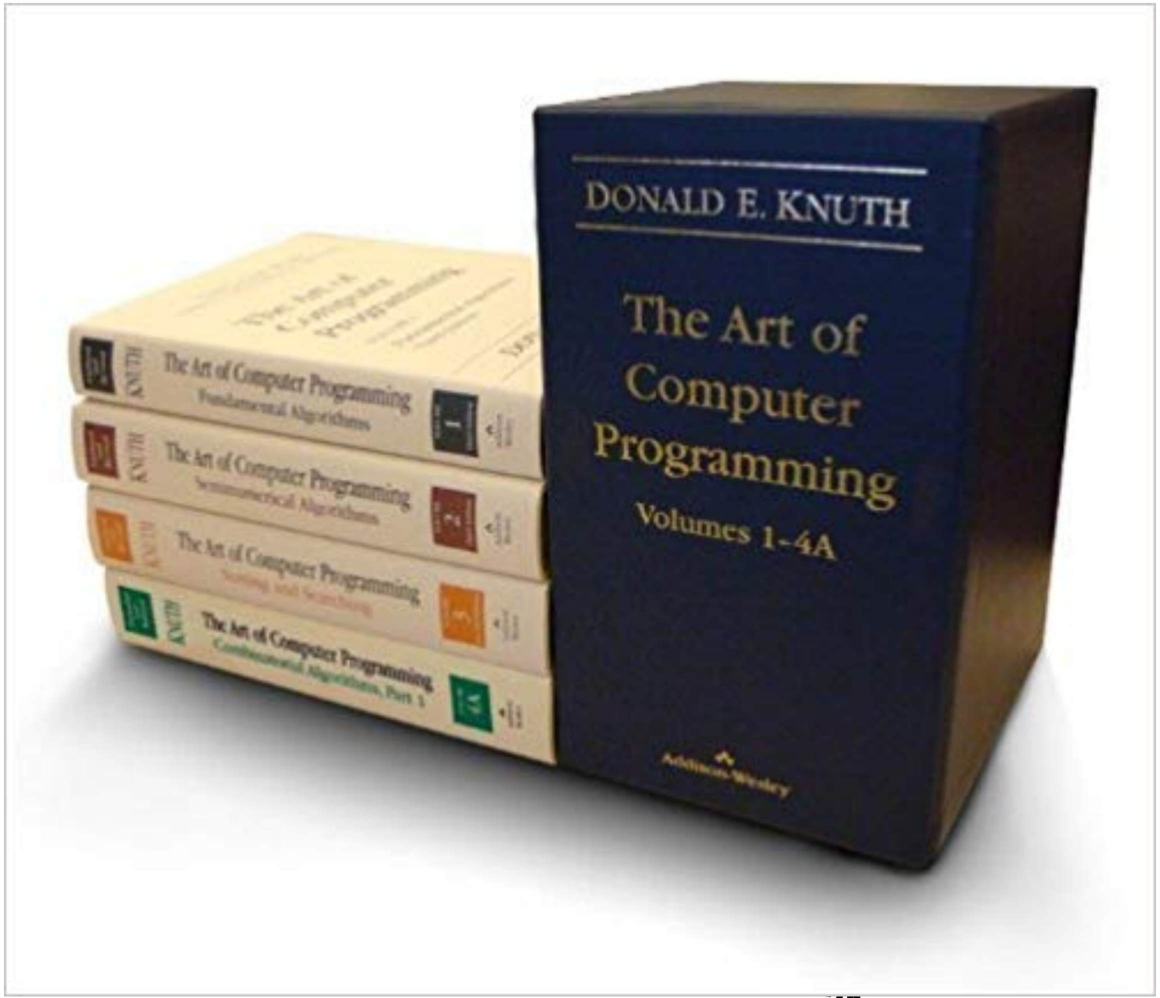
- Algorithm is a synonym for "effective procedure"

Berlinsky (2000)

- Algorithm is a finite procedure, written in a fixed symbolic vocabulary, governed by precise instructions, moving in discrete steps, 1, 2, 3, . . ., whose execution requires no insight, cleverness, intuition, intelligence, or perspicuity, and that sooner or later comes to an end

The Art of Computer Programming

Donald Knuth's Opus Magnum today 2021



Defensive Programming

Automation

- Some aspects of programming can be automated
- Many web interfaces to users/customers are automated
- How many times did you have to retype correct web-page information because 1 item further down was misspelled?
- Learning: simple things can be automated, but even for those, best to use good programming principles, consistency, clear common sense
- See Richard Sites, main designer of Alpha processor [5]: "I'd rather write programs that write programs than write programs!"
- Automate what can be; and "manually" program the rest
- Programming the remaining portion, the hard problems, is a challenge

Programming

Defensive Programming

Future for Sac State CS students:

- Programmers will highly likely have very good work opportunities for a long time to come, in an exciting, fun profession!
- Like in all professions: The good ones will be in high demand, and craft new technologies through their work
- Others write web interfaces that clear out ☹ after a user makes 1 typo!

Don't be complacent; consider future enhancements!

- Today your SW works perfectly fine for some given spec.
- Tomorrow's spec. will change, and you need to adapt your SW to the modified requirements: Craft maintainable SW
- Tomorrow new errors pop up ☺

Defensive Programming

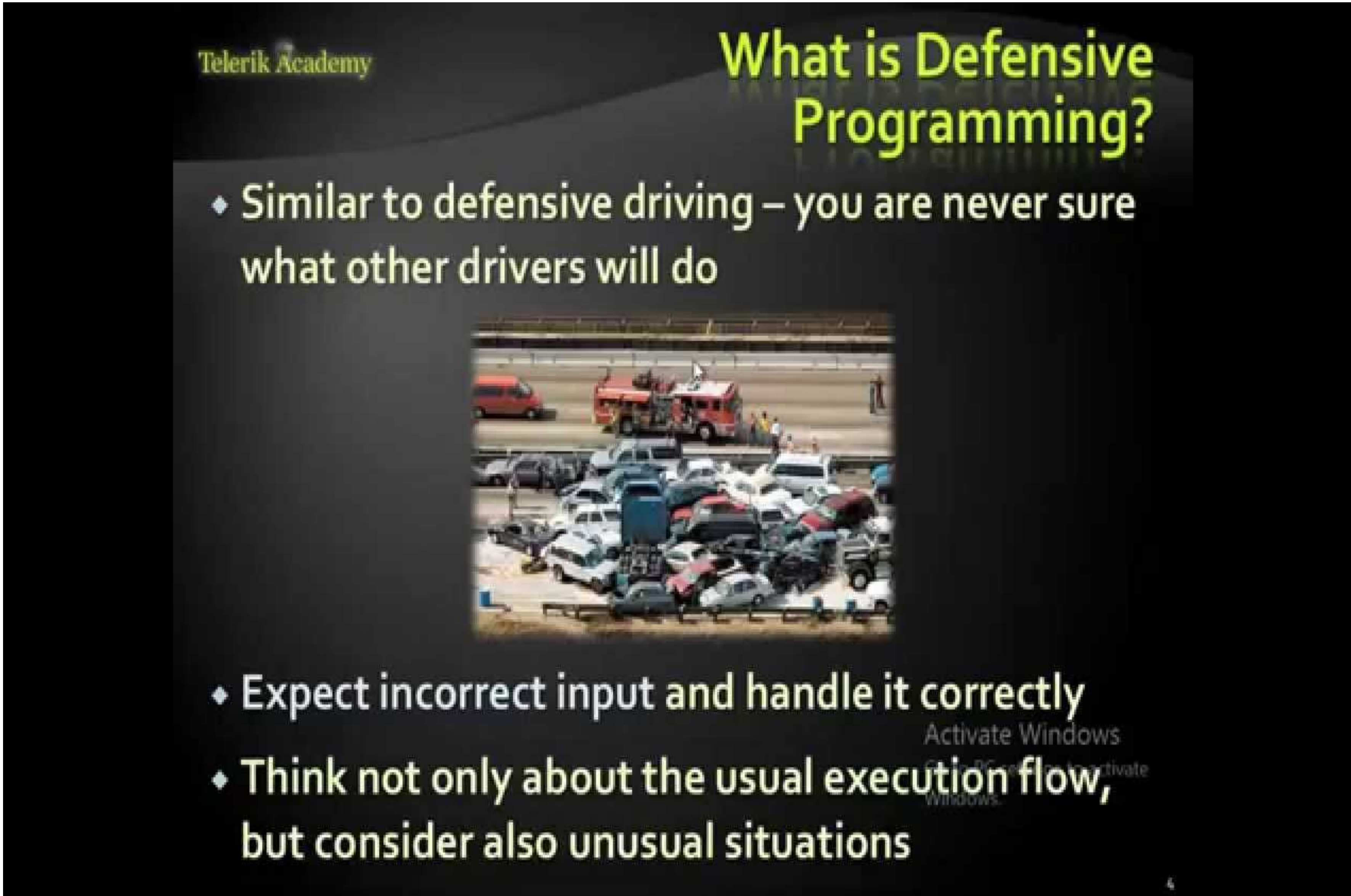
Don't trust your interface; *verify* even if checks seem unnecessary!

- Defined interface dictates your SW inputs, and specifies the output your SW is to generate
- Verify the accuracy of input, even your own; maybe especially!
- Generate messages where applicable and beneficial
- May not work for embedded SW, space mission 200 Mio miles from home planet; but appropriate default action should be meaningful

Don't trust your SW; check and verify all input instead!

- Other SW may automatically generate your program's input
- And even you!! make errors: so check, verify, report, mistrust
- Make on-line, live "reports" of suspicious logic, use #ifdef while developing SW product

Defensive Programming



Algorithm Sample: Search

Goal of Search

- Have a data structure and an interesting datum:
- Searching: The process of inspecting a data structure for a specified datum
- Goal of search may be to determine, whether the datum is included in the searched structure
  - Upon finding the datum, the search may end successfully
  - Frequently the goal of a search is to know the specific location or index, where the datum was found
  - If not found, the complete data structure had to be searched
  - Yet in case of sorted data structures this may not be as costly as it sounds! I.e. the cost may be way << O(n)
- Another goal of a search may be to determine the number of occurrences of a datum in the data structure being inspected

Algorithm Sample:

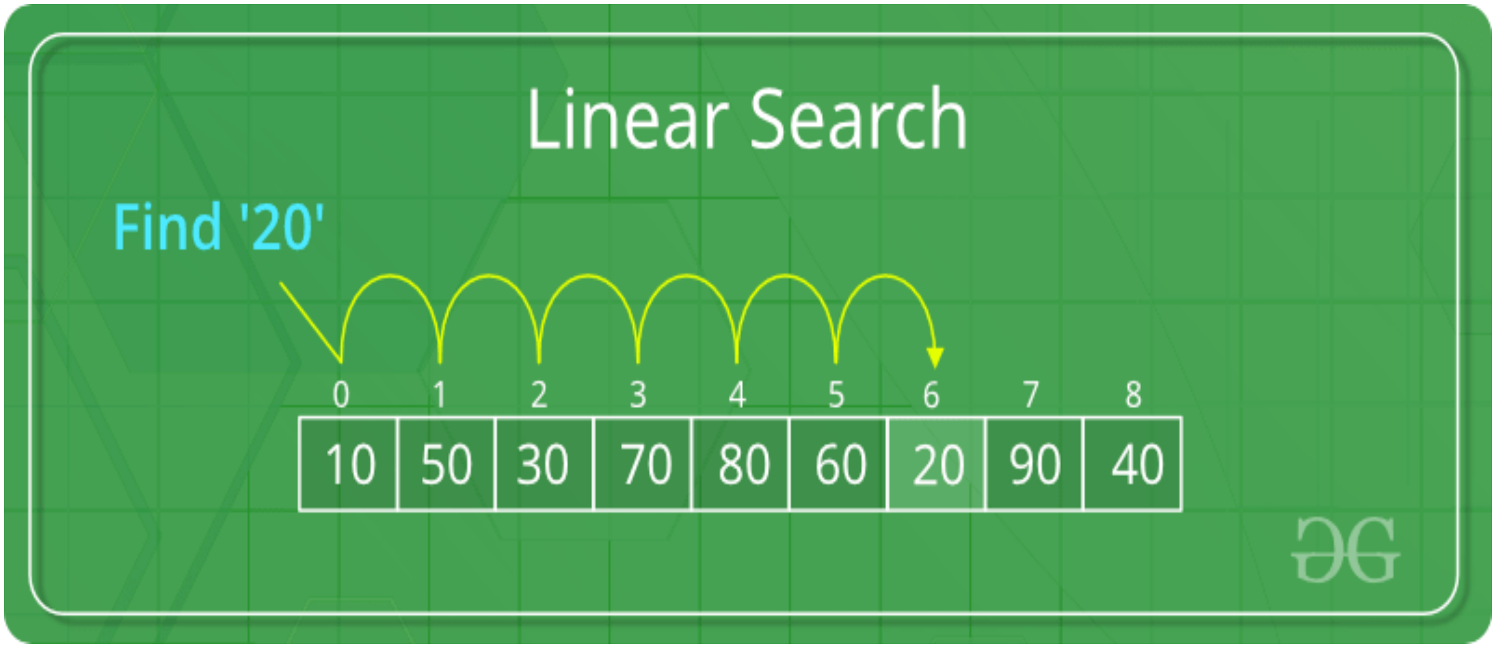
Cost of Search

- In linear search, the cost to find a datum is n, with n being the size of the data structure
- Or more concisely: Must look at up to all n elements in linear search to locate an item
- In Big O notation we say, the cost function for 1 lookup in a linear search is $O(n)$ (Big O, see later!)
- It doesn't matter that sometimes the first try may be successful, the overall cost function is still $O(n)$
- For the binary search in a sorted list, the lookup cost is $O(\log_2(n))$, since the worst case, to find an item, is to inspect $\log_2(n)$ elements
- For a balanced binary tree, cost is really $O(\log_2(n))$, but for an unbalanced tree it can be as bad as $O(n)$
- Redundant to say $\log_2$ as in Big O notation, the ratio of $\log_2$ by $\log_{10}$ is constant, i.e. "noise" in Big O

Linear Search

- Linear search makes no assumption on the order, in which data are stored in a data structure, the data generally are unsorted!
- Data structure is often an array; but could be a linked list or any equivalent data structure
- If it is known that datum x is stored uniquely (occurs at most once) then search may terminate at first successful find
- To prove that a datum is not included, the complete data structure must be inspected in linear search
- Hence the cost function for a linear search is O(n), with n being size of data structure
- If datum may be included more than once, then even after successful find the complete data structure must be inspected; result is count of occurrences

Linear Search



Linear Search

```
#include <stdio.h>
#include <iostream.h>
#define MAX 20
// slot at index 0 is NOT used to store data
// instead, index 0 used to indicate: "not found"
int unsorted[ MAX ] =
{
        0, 9, -99, 99, 999, -999, 7, -7, 77, 88,
        1, 2,   9,  8,   3,    4, 7, 16, 77, 22
};
```

Linear Search

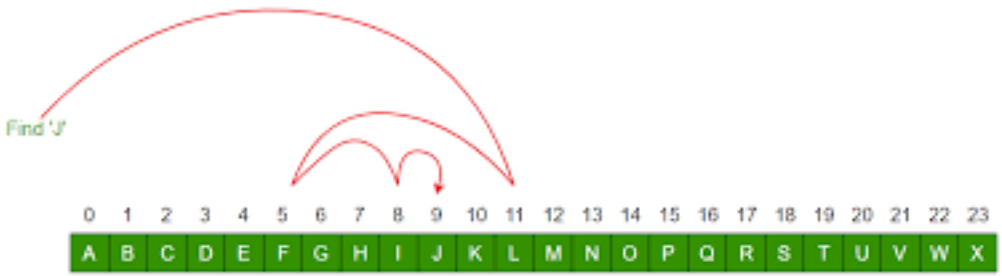
```
// assumes global array unsorted[] with MAX elements
// Note element unsorted[ 0 ] is NOT used:
// as index 0 does signify: Not Found!
// return 0 if element n is NOT found in unsorted[]
// searching n, return index of first occurrence
int linear( int n )
{ // linear
      // start search at index 1; index 0 not used
      for( int i = 1; i < MAX; i++ ) {
            if( unsorted[ i ] == n ) {
                  // found it
                  return i;
            } //end if
      } //end for
      return 0;   // 0 used for "not found!"
} //end linear
```

Binary Search, Play Game

- Practice the Guessing Game in class:
- Students think of an integer number between 1 and 1,000; keep it secret
- I can guess your secret number in at most 10 tries; exactly that secret number!
- No cheating ☺ and no changing of mind!
- Each time not yet found, all you need to say is: “no” and whether the actual number if > or <
- Why does that work?
- Binary search cost is $O(\log_2(n))$

Binary Search

- Binary search assumes that data structure be sorted, generally in some array[] or equivalent data structure
- If data are unique –occurring at most once– then we say the sorted array holds data in ascending or descending order
- Alternatively we say: *non-ascending* and *non-descending*; sounds a bit awkward
- The method is:



Binary Search: Method

- Use: left and right indices; array[] sorted in ascending order, wanted element is: n
- Initially left = 0 in C++, right is index of last array element: right = size - 1
- Loop: Repeatedly do the following “guess” and verify:
- Make a wild guess, that n be in the middle, named mid, between left and right, i.e. mid = (left + right) / 2
- If array[mid] == n the algorithm terminates: Found!

Binary Search: Method

- Else adjust indices: left: upward, or right: downward
- If wanted element is greater than the found one, search continues, with left = mid + 1; i.e. search in upper portion
- Else continue with right = mid – 1; i.e. search in lower portion
- When left is beyond right, in that case n is not found and the algorithm stops
- Else continue at loop

Binary Search 1

```
#include <stdio.h>
#include <iostream.h>
#define MAX 20
// also slot at index 0 not used in this binary search!
// simply a convention used here!
int sorted[ MAX ] =
{
    0, -88, -77, -12, -4, -2,  -1,    0,    1,    4,
    5,  14,  17,  66, 77, 99, 100, 1001, 2015, 9999
};
```

Binary Search 1

```
// binary search for element n in sorted[] array
// sorted[] size MAX is known globally ☹
// element sorted[ 0 ] is NOT used
// stored in ascending order! (non-descending!)
// if n not found, return 0; else return index of n
int binary( int n ) // search for n
{ // binary
      int left  = 1;      // don't use index 0
      int right = MAX-1;  // left, right approach
      int mid   = ( left + right ) / 2;
      . . .
```

Binary Search 1

```
int binary( int n )     // search for n, return 0 if not found
{ // binary             // use of globals not always best
   int left  = 1;       // don't use index 0
   int right = MAX-1;   // left, right, middle approach
   int mid   = ( left + right ) / 2;
   while( ( sorted[ mid ] != n ) && ( left < right ) ) {
       // n not yet found, and not looked at all elements
       if( sorted[ mid ] > n ) {
          right = mid - 1;    // search in "lower" half
       }else if( sorted[ mid ] < n ) {
          left = mid + 1;     // search in "upper" half
       }else{
          break;              // found n
       } //end if
       mid = ( left + right ) / 2;
   } //end while
   return sorted[ mid ] == n ? mid : 0; // conditional expr.
} //end binary
```

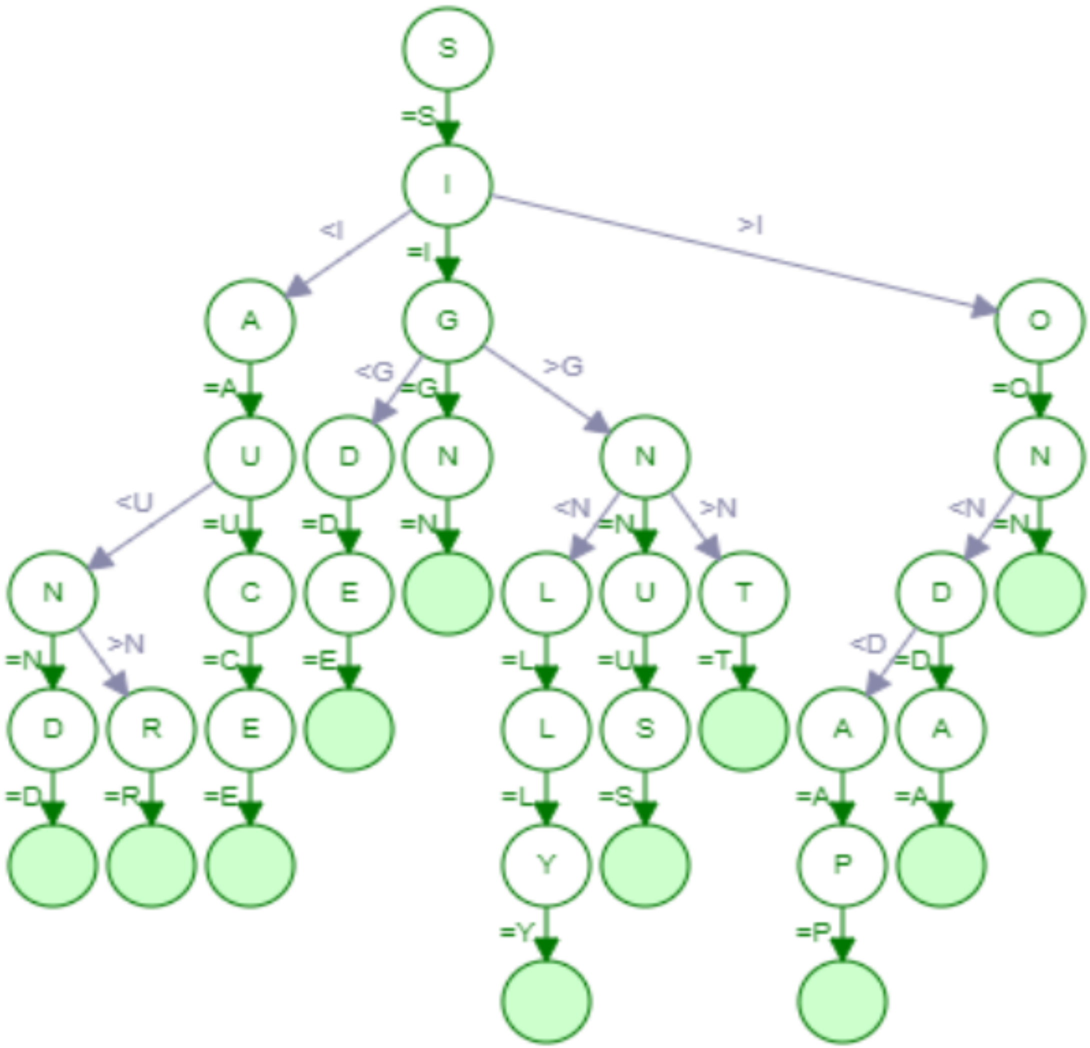
Better Binary Search 2

```
int sorted[ MAX ] = { -12, -6, -2, -1, 0, 1, 2, 16, 44, 120 };
// here element at index 0 is used
int binary_search( int first, int last, int key )
{ // binary_search
        int mid;
        if ( first > last ) {
             return NIL;      // not in range 0..MAX-1
        } //end if
        mid = ( first + last ) / 2;
        if ( key == sorted[ mid ] ) {
             return mid;
        }else if( key < sorted[ mid ] ) {
             return binary_search( first, mid-1, key );
        }else{
             // key < sorted[ mid ], so: first = mid+1
             return binary_search( mid+1, last, key );
        } // end if
} //end binary_search
```

Binary Search in Trees

- Searching for a datum n in a sorted binary tree is similar:
  - Initially node pointer p is set to the root
- If wanted element n is at p->data, the algorithm ends successfully, returning p
- Else, if n > p->data, search the right subtree: rooted at slot: p = p->right
- Else search the left subtree: p = p->left
- If p is ever null, n is not found; n is not in the tree!
- Careful, if tree is completely unbalanced, tree traversal can be as slow as linear search: O(n)
- Ternary Trees, next page, used in auto-completion techniques

Ternary Trees

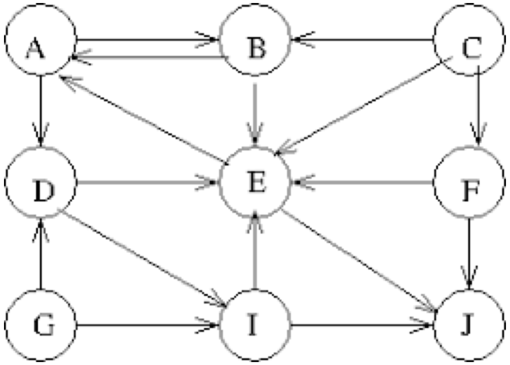


Search in Graph G

- A graph G is a data structure of nodes and connecting edges
- Edges may be undirected or directed
- A simplification is: to view undirected edges a bi-directional
- Any node in G may have any number incident edges
- Unlike tree: In a tree, nodes have exactly 1 incident edge; always!
- In contrast, graph G may have any # of incident edges
- A graph G may even be unconnected! In which case there are no incident edges to some nodes
- Use “trick” to traverse! See work-around later!

Search in Graph G

- From “G has any number incident edges” follows:
- . . . that G may be unconnected, when there are no incident edges to some nodes
- Hence a binary search for a general G is not feasible
- Unless an ancillary data structure is provided, even the linear search would be infeasible



Search in Graph G

- Work-around: add a field to each node of G; but this field is NOT logically part of G proper!
- Instead, this field named finger, allows an algorithm to weave through all nodes of G, in linear (or other) fashion, even if unconnected
- Don't forget to add a Boolean visited field to avoid repeat visits!

Bubble Sort

Sorting

- Sorting to be handled in detail in later section
- Here outline of so called: “Bubble Sort”
- Repeatedly search through –the remainder of– an array, to identify the largest (or smallest) elements, and exchange with the current element
- Have 2 nested loops
- Repeated extract the one (smallest, or largest) and place into right array position
- Cost if high: doubly nested loop

Bubble Sort Array

```
// bubble sort array a[] of ints; global an exception ☺
// outer loop start index 0; ends 1 before list-end
// inner loop starts at index+1; ends at last index

void sort()
{ // sort
   for( int left = 0; left < MAX-1; left++ ) {
     for( int right = left+1; right < MAX; right++ ) {
       // is right element greater? if so: manual swap
       if( a[ right ] > a[ left ] ) { // descending
          int temp    = a[ right ];
          a[ right ]  = a[ left ];
          a[ left ]   = temp;
       } //end if
     } //end for
   } //end for
} //end sort
```

Print Sorted Array

```
int main( void )
{ // main
  // print global array a[] unsorted
  print( "before sorting:" );

  sort();

  // print global array a[] sorted
  print( "after sorting:" );
} //end main
```

Growth of Functions: Big O Notation

Big O Notation

- Big O notation: A mathematical notation expressing the limiting behavior of a function when its argument grows towards a particular boundary value, usually toward infinity
- Is a member of a family of notations invented by Paul Bachmann, Edmund Landau, et al.
- Practiced widely in Math and Computer Science
- Collectively called Bachmann–Landau notation, AKA asymptotic notation
- Used in CS to characterize algorithms according their runtime or space requirements, as a function of the input size or time for lookup

Big O Notation

- Use of letter capital O, as growth rate of a function is referred to as the: Order of the function
- Characterizing a function via big O notation provides Upper Bounds on the growth of such a function
- In typical use Big O Notation is asymptotical: It refers to very large *x* of O(x)
- Hence the contribution of terms that grow most quickly eventually renders other terms irrelevant
- Following simplification rules can be applied:
  1. If $f(x)$ is a sum of several terms, and one of these has a largest growth rate, it can be kept; all others can be omitted in Big O
  2. If $f(x)$ is a product of several factors, any other constant factors (that are not f(x)) can also be omitted

Growth of Functions

- **Computational growth of functions is usually described using that Big O notation**
- **Definition: Let f and g be functions f(x) and g(x) from integer numbers x to the real numbers**
- **We say that f(x) is O(g(x)) if there are constants C and k such that:** (Operator symbols | | used for ***absolute value***)

$$|f(x)| \leq C \cdot |g(x)|$$

whenever $x > k$

Growth of Functions

- **When we analyze the growth of complexity functions, typically named f(x) and g(x), we view f(x) and g(x) always as positive**

- **Therefore, we can simplify the Big O requirement to**

$$\mathbf{f(x) \leq C \cdot g(x)} \quad \text{whenever } x > k$$

- **If we want to show that f(x) is O(g(x)), we only need to find some pair (C, k)**

Growth of Functions

- **Idea behind Big O notation: Establish upper bound for the growth of a function f(x) for large x**

- **This boundary is specified by a function g(x) that is usually much simpler than f(x)**

- **We accept (i.e. ignore) constant C in the requirement**

$$f(x) \leq C \cdot g(x) \quad \text{whenever } x > k$$

- **because C does not grow, or even change with x**

- **We are interested in large x, so it is OK if**

$$f(x) > C \cdot g(x) \text{ for } x \leq k$$

Growth of Functions

Example:

- Show that $f(x) = x^2 + 2x + 1$ is really just: $O(x^2)$

- For $x > 1$ we have:

$$x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2$$
$$\Rightarrow x^2 + 2x + 1 \leq 4x^2$$

- Therefore, for $C = 4$ and $k = 1$:

- $f(x) \leq C \cdot x^2$ whenever $x > k$

- $\Rightarrow f(x)$ is $O(x^2)$

Growth of Functions

- Question: If f(x) is $O(x^2)$, would its cost function also be bounded by (i.e. no greater than) $O(x^3)$?
- Or even higher
- Yes, f(x) of order $O(x^2)$ is bounded by $O(x^3)$
- After all, x^3 grows faster than x^2, so x^3 grows also faster than f(x) which is limited by x^2
- We lose precision computing a cost function
- Should always aim to identify the simplest (cheapest) function g(x) for which f(x) is O(g(x))

Growth of Functions

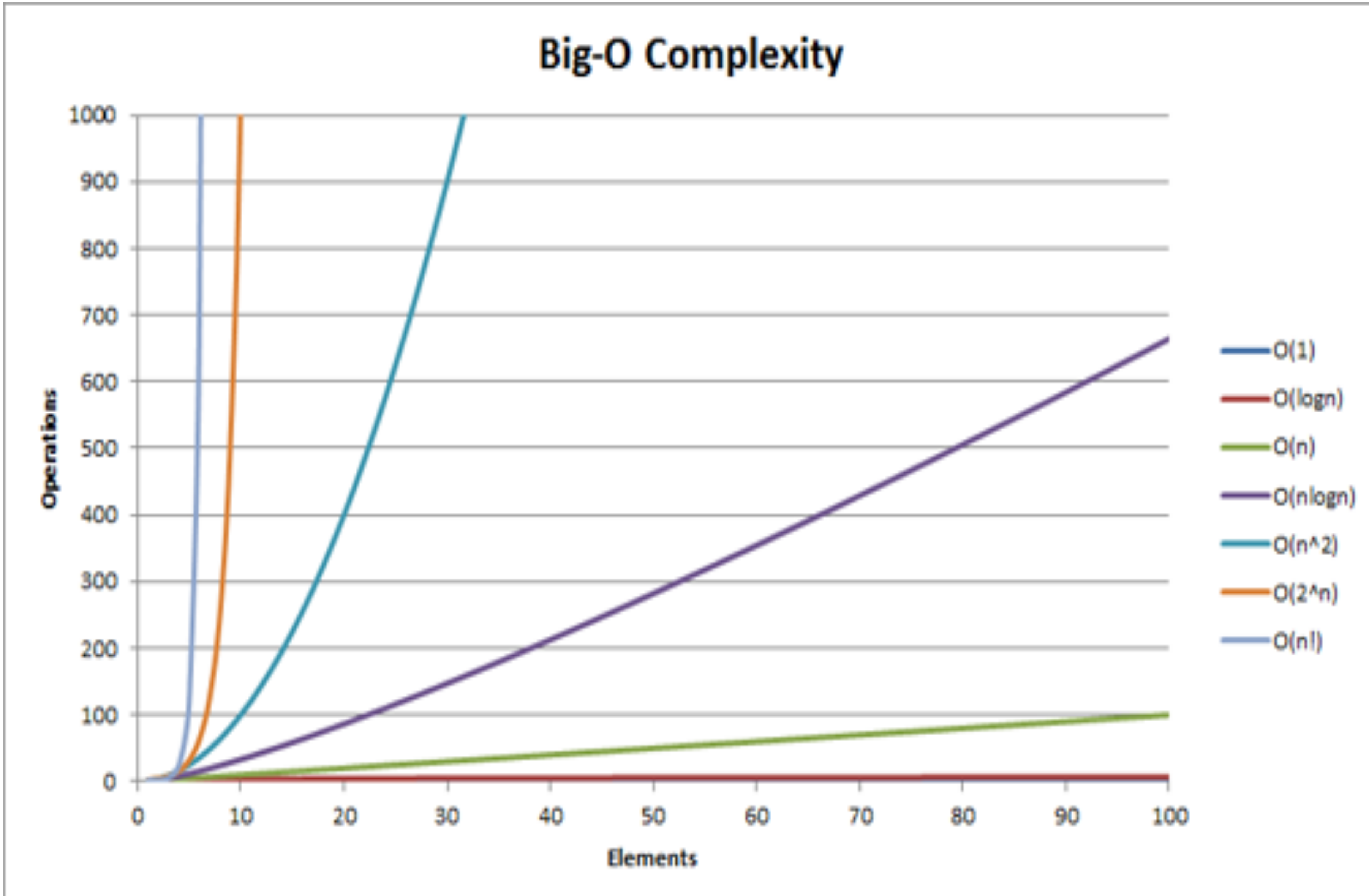
- Sample unordered generic functions g(n) in Big O notation:

  n, $\log n$, 1, 2^n, n^2, $n!$, n, n^3, $n \log n$

- Listed top-down from smallest to largest in terms of cost:

  1
  $\log n$
  n
  $n \log n$
  n^2
  n^3
  2^n
  $n!$
  n^n

Growth of Functions



Growth of Functions

- A problem that can be solved with polynomial worst-case complexity is called: tractable

- Problems of higher complexity are called: intractable

- Problems that no algorithm can solve are called: unsolvable

Useful Rules for Big O

- For any polynomial $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0$, where $a_0, a_1, \ldots, a_n$ are real numbers:
- Such an $f(x)$ is $O(x^n)$

If cost function for $f_1(x)$ is $O(g_1(x))$ and for $f_2(x)$ is $O(g_2(x))$:
$(f_1 + f_2)(x)$ is $O(\max(g_1(x), g_2(x)))$

If $f_1(x)$ is $O(g(x))$ and $f_2(x)$ is $O(g(x))$, then
$(f_1 + f_2)(x)$ is $O(g(x))$

Complexity Examples

What does the following algorithm compute?

```
procedure who_knows( a[n] : int )
m := 0
for i := 1 to n-1
   for j := i + 1 to n
      if |a[i]-a[j]| > m then m := |a[i]-a[j]|
   end for
end for
. . .

?
```

Complexity Examples

What does the following algorithm compute?

```
procedure who_knows( a[n] : int )
m := 0
for i := 1 to n-1
   for j := i + 1 to n
      if |a[i]-a[j]| > m then m := |a[i]-a[j]|
   end for
end for
--m is max abs(diff) between any pair
```

Number of comparisons is ~ (n-1) n / 2 = $0.5n^2 - 0.5n$

Time complexity is $O(n^2)$

Factor 0.5 of n^2 falls by wayside; Summand - 0.5n as well!

Complexity Examples

Another algorithm solving a different search problem:

```
procedure max_diff( a[n] : int)
min := a1
max := a1
for i := 2 to n
   if a[i] < min then
      min := a[i]
   elsif a[i] > max then  -- elsif a la Ada
      max := a[i]
   end if
m := max – min
. . .
```

Comparisons: 2(n – 2); so time complexity is O(n)

Number Theory

Introduction to Number Theory

- **Number theory treats integers & their properties**
- **We will start with the basic principles of**
  - **divisibility**
  - **greatest common divisors gcd**
  - **least common multiples, and**
  - **modular arithmetic**

Pierre de Fermat

And analyze some relevant algorithms
Aside from Fermat's Conjecture

(No longer ***Fermat's Conjecture***!
Proven in 2016:
No three positive integers a, b, and c satisfy
equation $a^n + b^n = c^n$ for any integer $n > 2$)

Division

- If a and b are integers with a $\neq$ 0, we say that a divides b if there is an integer c so that b = ac
- When a divides b we say that a is a factor of b, or equivalently that b is a multiple of a
- Notation a | b means: “integer a divides integer b” evenly
- We write a χ b when integer a does not divide b

Divisibility Theorems

For integers a, b, and c it is true that:

- if a | b and a | c, then a | (b + c)
- Example:
  - 3 | 6 and 3 | 9, so 3 | 15

- if a | b, then a | b * c for all integers c
- Example:
  - 5 | 10, so 5 | 20, 5 | 30, 5 | 40, …

- if a | b and b | c, then a | c
- Example:
  - 4 | 8 and 8 | 24, so 4 | 24

Primes

- **A positive integer p greater than 1 is called prime if the only positive factors of p are 1 and p**

- **Note: Here 1 itself is not a considered a prime**

- **A positive integer that is greater than 1 and not prime is called a composite**

- **A fundamental theorem of arithmetic:**
  - **Every positive integer can be written uniquely as the product of primes; AKA prime factorization**
  - **Whose prime factors are conventionally written in increasing order**

Primes

Examples of Prime Factorization:

15	=	3 * 5		
48	=	2 * 2 * 2 * 2 * 3		
17	=	17 AKA 1 * 17		
100	=	2 * 2 * 5 * 5	=	$2^2 * 5^2$
512	=	2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2	=	2^9
515	=	3 * 103		
28	=	2 2 * 7		

Division Algorithm

- Let a be an integer and d be another positive integer
- Then there are unique integers q and r, with $0 \le r < d$, that:

$$a = d * q + r$$

- In the above equation:
  - a is called the dividend
  - d is called the divisor
  - q is called the quotient, and
  - r is called the remainder

Division Algorithm

Example: When we divide 17 by 5, we have

$$17 = 5 * 3 + 2$$

- 17 is the dividend
- 5 is the divisor
- 3 is called the quotient, and
- 2 is called the remainder

Division Algorithm

Another example:

What happens when we divide -11 by 3 ?

Note that a *remainder can never be negative*

$$-11 = 3 * (-4) + 1$$

- **-11 is the dividend**
- **3 is the divisor**
- **-4 is called the quotient, and**
- **1 is called the remainder**

Greatest Common Divisor

- Let a and b be integers, not both zero: The largest integer d such that d l a and d l b is called the greatest common divisor of a and b

- Greatest common divisor of a and b: is denoted by gcd(a, b)

- Example 1: What is the gcd(48, 72) ?

- The positive common divisors of 48 and 72 are 1, 2, 3, 4, 6, 8, 12, 16, and 24, so gcd(48, 72) = 24

- Example 2: What is gcd(19, 72) ?

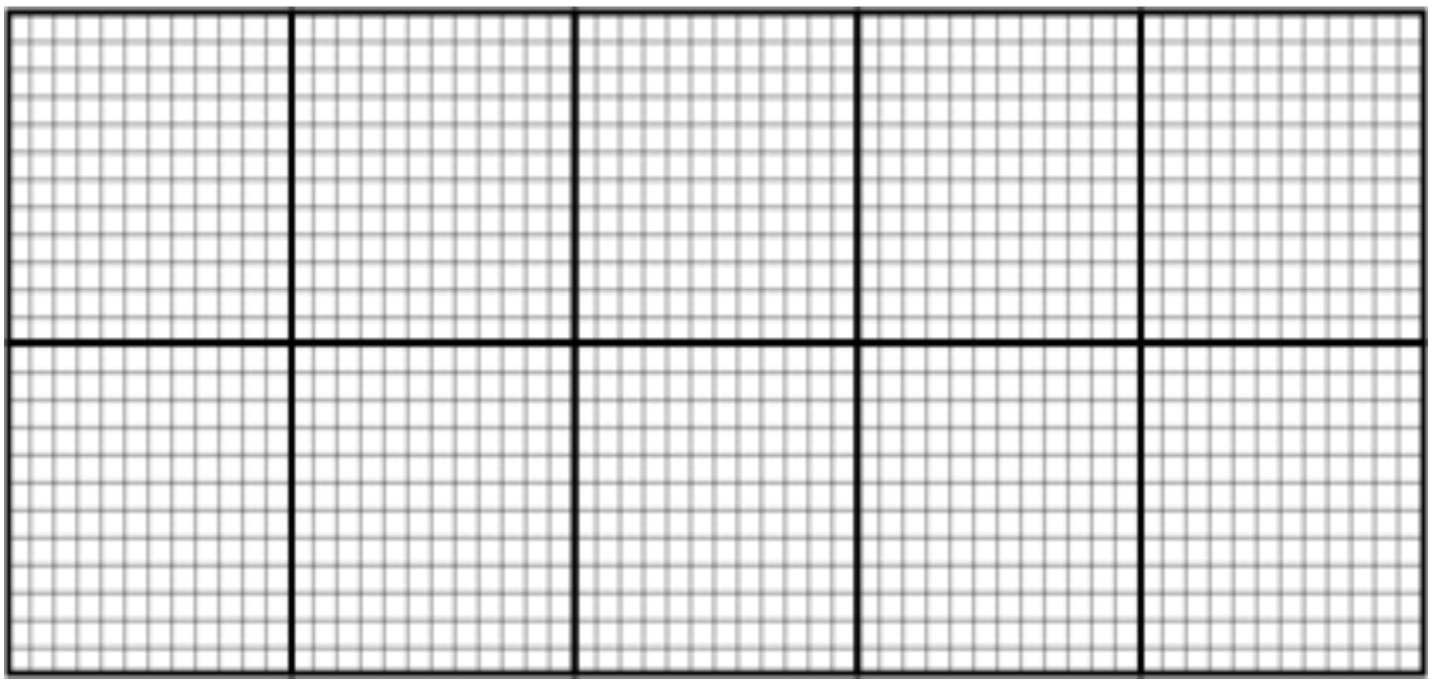
- The only positive common divisor of 19 and 72 is 1, so gcd(19, 72) = 1

Greatest Common Divisor

- There exists a nice geometric view of the GCD
- For example, a 24-by-60 rectangular area can be divided into a grid of 1-by-1 squares, 2-by-2 squares, 3-by-3 squares, 4-by-4 squares, 6-by-6 squares or 12-by-12 squares
- Thus we see that 12 is the greatest common divisor of 24 and 60!
- A 24-by-60 rectangular area can be divided into a grid of 12-by-12 squares, with two squares along one edge (24/12 = 2) and five squares along the other (60/12 = 5)
- See below:

Greatest Common Divisor

- **Below you see the geometric partition of a 24 * 60 rectangle partitioned into 60/12 = 5; 12 is GCD**
- **1,440 mini-squares; found at Wiki: [9]**



Relatively Prime Integers

Definition: Two integers a and b are relatively prime if gcd(a, b) = 1. Needed for example to define suitable size of hash table in fast compilers

Examples:

- **Are 15 and 28 relatively prime?**
- **Yes, gcd(15, 28) = 1**

- **Are 55 and 28 relatively prime?**
- **Yes, gcd(55, 28) = 1**

- **Are 35 and 28 relatively prime?**
- **No, gcd(35, 28) = 7**

Relatively Prime Integers

- Definition: The integers $a_1, a_2, \ldots, a_n$ are pairwise relatively prime if $\gcd(a_i, a_j) = 1$ whenever $1 \leq i < j \leq n$

- Example 1: Are 15, 17, and 27 pairwise relatively prime?
- No, because
  $\gcd(15, 27) = 3$

- Example 2: Are 15, 17, and 28 pairwise relatively prime?
- Yes, because
  $\gcd(15, 17) = 1$
  $\gcd(15, 28) = 1$
  $\gcd(17, 28) = 1$

Challenge

- **Name the largest prime number!**
- **Or else prove, there exists no largest prime!**

Modular Arithmetic

Let a be an integer and m be a positive integer. Then we denote by: a mod m the integer remainder, when a is divided by m

Examples:

9 mod	4	=	1	
9 mod	3	=	0	
9 mod	10	=	9	-- quotient being 0
-13 mod	4	=	3	-- Note quotient is -4

Congruence

- Let a and b be integers and m be a positive integer. We say that a is congruent to b modulo m if

  m divides (a - b)

- We use the notation $a \equiv b \pmod{m}$ to indicate that a is congruent to b modulo m

- In other words:
  $a \equiv b \pmod{m}$ if and only if (a mod m) = (b mod m)

Congruence


rapport
congruence
balance
compatibility
synchronous
match
parallelism
consonance
congruity
agreement
harmony

Congruence

Examples:

- **Is 46 $\equiv$ 68 (mod 11) ?**
- **Yes, because 11 | (46 – 68)**

- **Is 46 $\equiv$ 68 (mod 22)?**
- **Yes, because 22 | (46 – 68)**

- **For which integers z is it true that z $\equiv$ 12 mod 10?**
- **It is true for any z $\in$ {…, -28, -18, -8, 2, 12, 22, 32, … }**

Theorem: Let m be a positive integer. Then integers a and b are congruent modulo m if and only if there is an integer k such that a = b + k * m

Congruence

Theorem: Let m be a positive integer
If $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$, then:
$a + c \equiv b + d \pmod{m}$ and $a * c \equiv b * d \pmod{m}$

- **Proof: We know that $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$ implies that there are integers s and t with:**
  $b = a + s\, m$ and $d = c + t * m$

- **Therefore,**
- **$b + d = (a + s * m) + (c + t * m) = (a + c) + m*(s + t)$**
  and
- **$b * d = (a + s * m) * (c + t * m) = a * c + m * (a * t + c * s + s * t * m)$**

- **Hence, $a + c \equiv b + d \pmod{m}$ and $a * c \equiv b * d \pmod{m}$**

Congruence

Theorem: Let m be a positive integer. $a \equiv b \pmod{m}$ iff $a \bmod m = b \bmod m$

Proof:

Let $a = mq1 + r1$, and $b = mq2 + r2$.

Only if part: $a \bmod m = b \bmod m \rightarrow r1 = r2$, therefore

$a - b = m(q1 - q2)$, and $a \equiv b \pmod{m}$.

If part: $a \equiv b \pmod{m}$ implies

$$a - b = mq$$

$$mq1 + r1 - (mq2 + r2) = mq$$

$$r1 - r2 = m(q - q1 + q2).$$

Since $0 \le r1, r2 < m$, $0 \le |r1 - r2| < m$. The only multiple in that range is 0

Therefore $r1 = r2$, and $(a \bmod m) = (b \bmod m)$

Euclid of Alexandria

323 BC – 283 BC

Euclidean Algorithm

The Euclidean Algorithm finds the greatest common divisor of two integers a and b

- **For example, if we want to find gcd(287, 91), we divide 287 by 91, and find:**

$$287 = 91 * 3 + 14$$

- **We know that for integers a, b and c:**
  if a l b and a l c, then a l (b + c)

- **Therefore, any divisor (including their gcd) of 287 and 91 must also be a divisor of 287 - 91·3 = 14**

- **Consequently, gcd(287, 91) = gcd(14, 91)**

Euclidean Algorithm

In the next step, we divide 91 by 14:

- **91 = 14 * 6 + 7**

- **This means that gcd(14, 91) = gcd(14, 7)**
- **So we divide 14 by 7:**

  14 = 7 * 2 + 0

- **We find that 7 | 14, and thus gcd(14, 7) = 7**

- **Therefore, gcd(287, 91) = 7**

Euclidean Algorithm

In Algol-like pseudo code, the algorithm can be implemented as follows, comments in { }:

```
procedure gcd( a, b: positive integers )
    x := a
    y := b
    while y ≠ 0 do
        r := x mod y
        x := y
        y := r
    end
    {x is gcd(a, b ) }
end
```

Integer Representation Review

Representation of Integers

- Let b be a positive integer greater than 1: base = b
- Then if n is a positive integer, it can be expressed uniquely in the form:

$$n = a_k b^k + a_{k-1} b^{k-1} + \ldots + a_1 b^1 + a_0 b^0$$

- where k is a nonnegative integer,
- $a_0, a_1, \ldots, a_k$ are nonnegative integers less than b

Example 859_{10} for base $b = 10$:

$$859 = 8 * 10^2 + 5 * 10^1 + 9 * 10^0$$

Representation of Integers

Example for b = 2 (binary expansion):

$$10110_2 = 1*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 0*2^0 = 22_{10}$$

Example for base b=16 (hexadecimal expansion), using letters A to F to indicate hex digit values 10 to 15:

$$3A0F_{16} = 3*16^3 + \underbrace{10}_{A}*16^2 + 0*16^1 + \underbrace{15}_{F}*16^0 = 14863_{10}$$

Representation of Integers

- How to construct base b expansion of an integer n?
- First, divide n by b to obtain a quotient q_0 and remainder a_0, that is:

$$n = bq_0 + a_0, \text{ where } 0 \le a_0 < b$$

- The remainder a_0 is the rightmost digit in the base b expansion of n
- Next, divide q_0 by b to obtain:

$$q_0 = bq_1 + a_1, \text{ where } 0 \le a_1 < b$$

- a_1 is the second digit from right in base b expansion of n. Continue these steps til you obtain a 0 quotient

Representation of Integers

Example: Base 8 expansion of 12345_{10}

First, divide 12345 by 8:

12345	= 8*1543	+ 1
1543	= 8*192	+ 7
192	= 8*24	+ 0
24	= 8*3	+ 0
3	= 8*0	+ 3

The result is: $12345_{10} = 30071_8$

Adding Integers

How do we (humans) add two integers?

```
Example:        7583
            +   4932
                -------
      carry:   11100
                ====
               12515
```

Binary Numbers

Binary Numbers

- Binary Digit –AKA Bit-- is the smallest unit of computation on digital computers
- Bit has two states:
  - 0 represents 0 Volt [V], or ground; used for logical False, or for numeric 0
  - 1 represents positive Voltage [+V]; used for logical True, or for numeric 1
- Computer word consists of multiple bits, typically 32, 60, today mostly 64 bits
- Often words are composed of bytes, units of 8 bits that are addressable as one unit: byte-addressable
- Binary number representation works like decimal system

$$\underset{\uparrow \atop \text{MSB}}{b_{n-1}} b_{n-2} \ldots b_1 \underset{\uparrow \atop \text{LSB}}{b_0} = b_{n-1}\times 2^{n-1} + b_{n-2}\times 2^{n-2} + \ldots + b_1\times 2^1 + b_0\times 2^0$$

Binary Numbers Using TCR

- Possible representations of binary numbers: *sign-magnitude (sm), one's complement (ocr)*, and *two's complement representation (tcr)*
- Advantage of *tcr:* machine needs no subtract unit, the sole adder is sufficient
- When subtraction is needed, just add a negative number
- To create a negative number: invert the positive one; see full method below!
- In TCR, there is no need for signed- *and* unsigned arithmetic; unsigned is sufficient
- C++ allows signed & unsigned integers. In fact, arithmetic units with *tcr* effectively ignore the sign bit
- *Tcr* just needs an adder, inverter, and check for carry

Binary Numbers Using TCR

- Binary numbers in *tcr* use a sign bit and a fixed number of bits for the magnitude
- For example, old PCs have 32-bit integers, 1 bit for the sign, 31 bits for the magnitude
- PCs today use 64-bit integers, also use 1 for the sign
- When processed on *tcr* architecture, the most significant bit (MSB) is the *sign bit*, the other 63 bits hold the actual signed value of interest (AKA magnitude); usually in two's complement
- By convention, the sign bit 0 stands for positive and 1 for negative numbers

Binary Numbers Using TCR

- **Inverting positive numbers:** To create a negative number from a positive in *tcr*, start with the binary representation for the positive one, invert all bits, ***and add 1***
- Note that overflow cannot happen by inversion alone: the negative value of positive numbers can always be created
- But **there is one more negative number** in *tcr* than positive numbers!
- There exists only **one single 0**, i.e. **no negative 0 in *tcr***, as we find it in *one's complement* and *sign-magnitude* representations

Binary Numbers Using TCR

- Inverting negative numbers: To invert a negative number, complement all bits of the original negative number and add a 1in *tcr*, do not subtract it!
- However, there will be one negative value, whose positive inverse cannot be represented; it will cause overflow instead!
- That value is the smallest, negative number. For example, an 8-bit signed *tcr* integer can hold integers in the range from -128 .. 127. See the asymmetry? See the one negative value that cannot be inverted?
- On a 32-bit architecture, the range is from -2,147,483,648 to +2,147,483,647; similar for 64-bits
- Note the asymmetry of the numeric range!

Hexadecimal Numbers

- Hexadecimal (hex) numbers are simply numbers with a base 16; not 10, not 2, just 16, no magic ☺
- They have 16 different digits, 0..9 and, purely by convention, the digits a . . f
- Symbol ‘a’ or ‘A’ stands for some hex digit with value 10_{10}, while the symbol ‘f’ stands for the value 15_{10}
- Programming tools are not picky; they permit 6 extra digits to be lower- as well as uppercase letters ☺
- Here are a few hex numbers and their equivalent decimal values:

Hexadecimal Numbers

Decimal	Hexadecimal
0	0
1	1
9	9
10	a
11	b
15	f
16	10
33	21
127	7f
128	80
129	81
255	ff
256	100

Decimal	Hexadecimal
257	101
258	102
300	12c
16,383	3fff
16,384	4000
16,385	4001
32,767	7fff
32,768	8000
32,769	8001
65,535	ffff
65,536	1,0000
65,637	1,0001
4,294,967,295	ffff,ffff

Adding Hexadecimal Numbers

Adding Hexadecimal Numbers, $af_{16} + 65_{16}$, $10a42_{16} + 5be_{16}$

			a	f
			6	5
		1	1	4

1	0	a	4	2
		5	b	e
1	1	0	0	0

Building Graphs

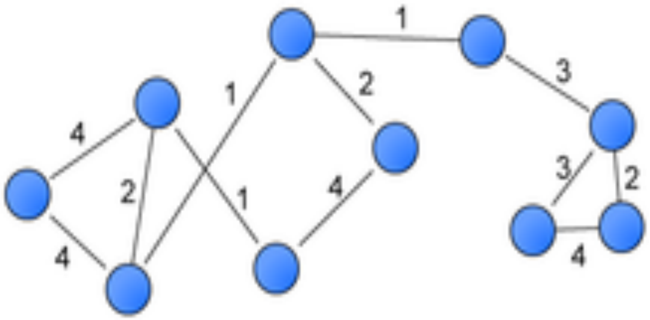
Formal Definition of Graph

- ***Graph***: A graph G is a data structure $G = \{ V, E \}$ consisting of a set E of edges and a set of V vertices, AKA **nodes**. Any node $v_i \in V$ may be connected to any other node v_j. Such a connection is called an **edge**. Edges may be **directed**, even **bi-directed**
- Different from a tree, a node in G may have any number of predecessors –or incident edges: **THE main difference** between graph and tree!
- ***Empty Graph***: For expediency we ignore the possibility of a graph G being empty; in an empty graph the data structure that points the graph is simply NIL
- ***Connected Graph***: If all $n > 0$ nodes v_n in G are connected somehow, the graph G is called connected, regardless of edge directions

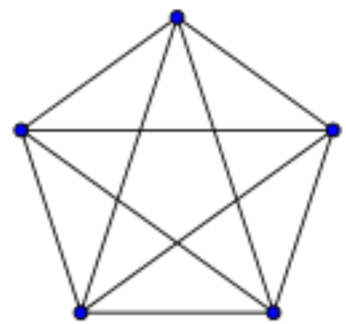
Formal Definition of Graph

- ***Strongly Connected Component*** **(SCC): A subset SG $\subseteq$ G is strongly connected, if for every node v_i in SG, i > 0, such a v_i can reach all v_j nodes in SG**
- ***Directed Acyclic Graph*** **(DAG): A DAG is a graph with directed edges but no cycles. A node may still have multiple predecessors and/or successors**
- **When programming graphs, it is convenient to add fields to the node type for auxiliary functions; e.g. it is possible to process all nodes in a linear fashion by adding a link field, often called a "finger" or "link"**
- **Possible use: traversing all nodes in G, though G may be unconnected!**

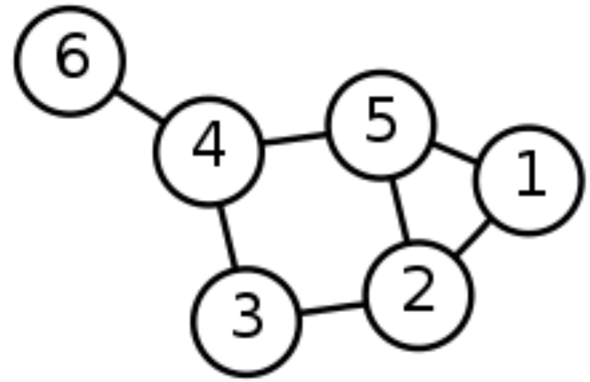
Sample Graphs



Weighted graph with 10 vertices and 12 weighted edges



Complete graph with 5 vertices and all possible 10 edges

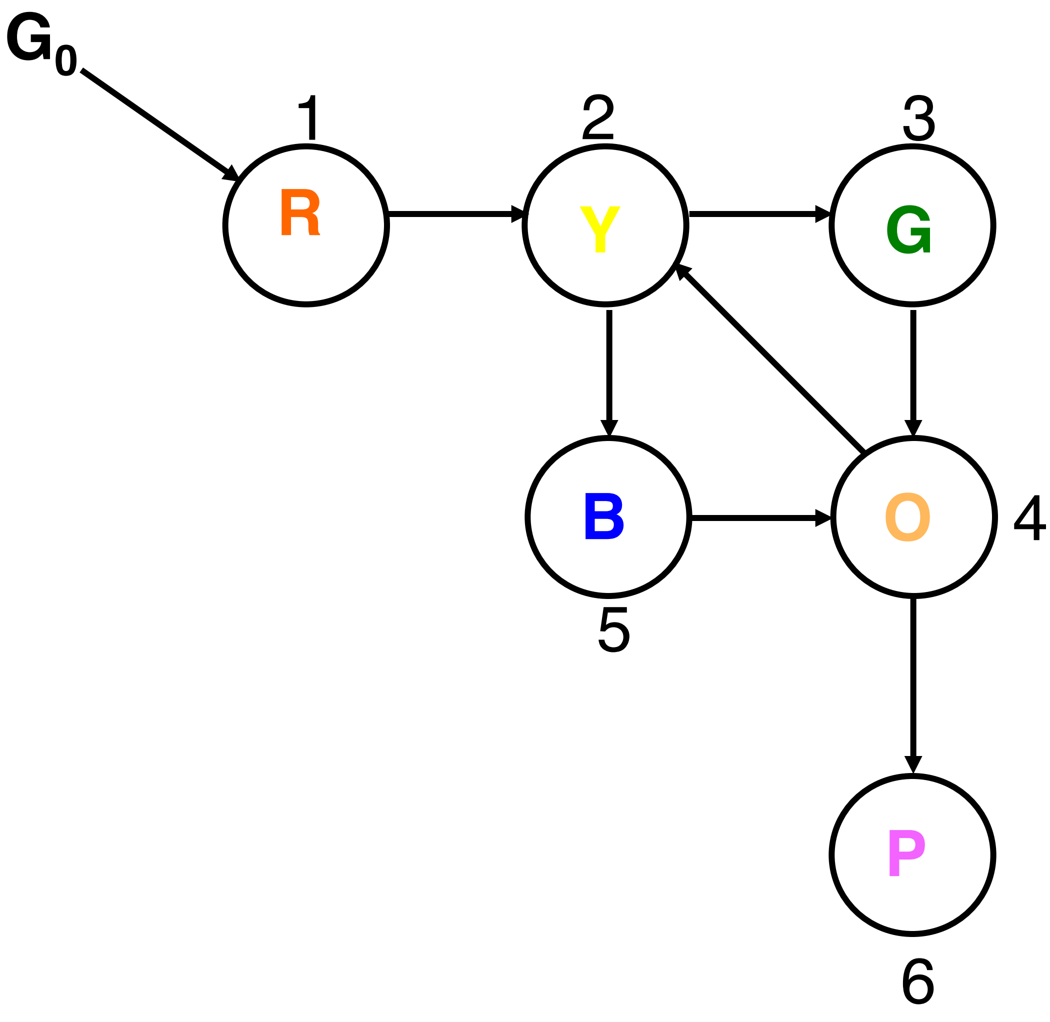


Sample graph with 6 named vertices and 7 edges

Graph Data Structure

- A graph G(v, e) consists of nodes v and edges e
  - Implemented via some node_type data structure
- G often identified and accessible via one select node, called *entry node*, or simply entry, AKA *head*
- Or identified by a pointer to node_type; if NIL, the graph is empty
- Caveat: G is not necessarily connected
  - If parts of G are unconnected, how can they be retrieved in case of a necessary, complete graph traversal?
- Several methods of forcing complete traversal:
  - Either create a super-node, not part of G proper, in a way that each unconnected region is pointed at, thus reachable
  - Or have a linked-list (LL) meandering through each node of G, without this link field being part of G proper; e.g. "finger"

Sample Graph G_0



Graph Data Structure, Cont'd

- Sample Graph G_0 above has 6 nodes
- The ID of a node, AKA *name* of each node is shown next to the nodes, e.g. 1, 2, 3, 4, 5, 6
- The graph's node type data structure includes such name information as part of struct node_type
- In addition, each node in G_0 has *attributes*, such as R, G, Y etc. in the sample above; such attributes can be arbitrarily complex consisting of many fields
- There may be more attributes belonging to each node, depending on what the graph is used for
- Any of these attributes must also be declared in the node_type data structure
- Successors, if any, of each node must be encoded in the node; there is no upper limit on the number!
- G_0 has 3 SCCs; 2 of those trivial, i.e. not interesting!

Graph Data Structure

- There is no upper bound on number of successor nodes; suitable way to define successors: via linked list of tuples, AKA link nodes
- Thus a possible data type for successor nodes is: pointer to a link node
- Link nodes can be allocated off the heap, as needed; they are not of type node_type, but of type link_type
- And each link is a tuple, i.e. consists of just 2 fields:
  - One field pointing to the next link; the type is pointer to link_type, in some languaged expresses as *link_type
  - The other field pointing to the successor node; the type is pointer to node_type
- For convenience, the last link inserted is added at the head of the list, simplifying searches for the list end

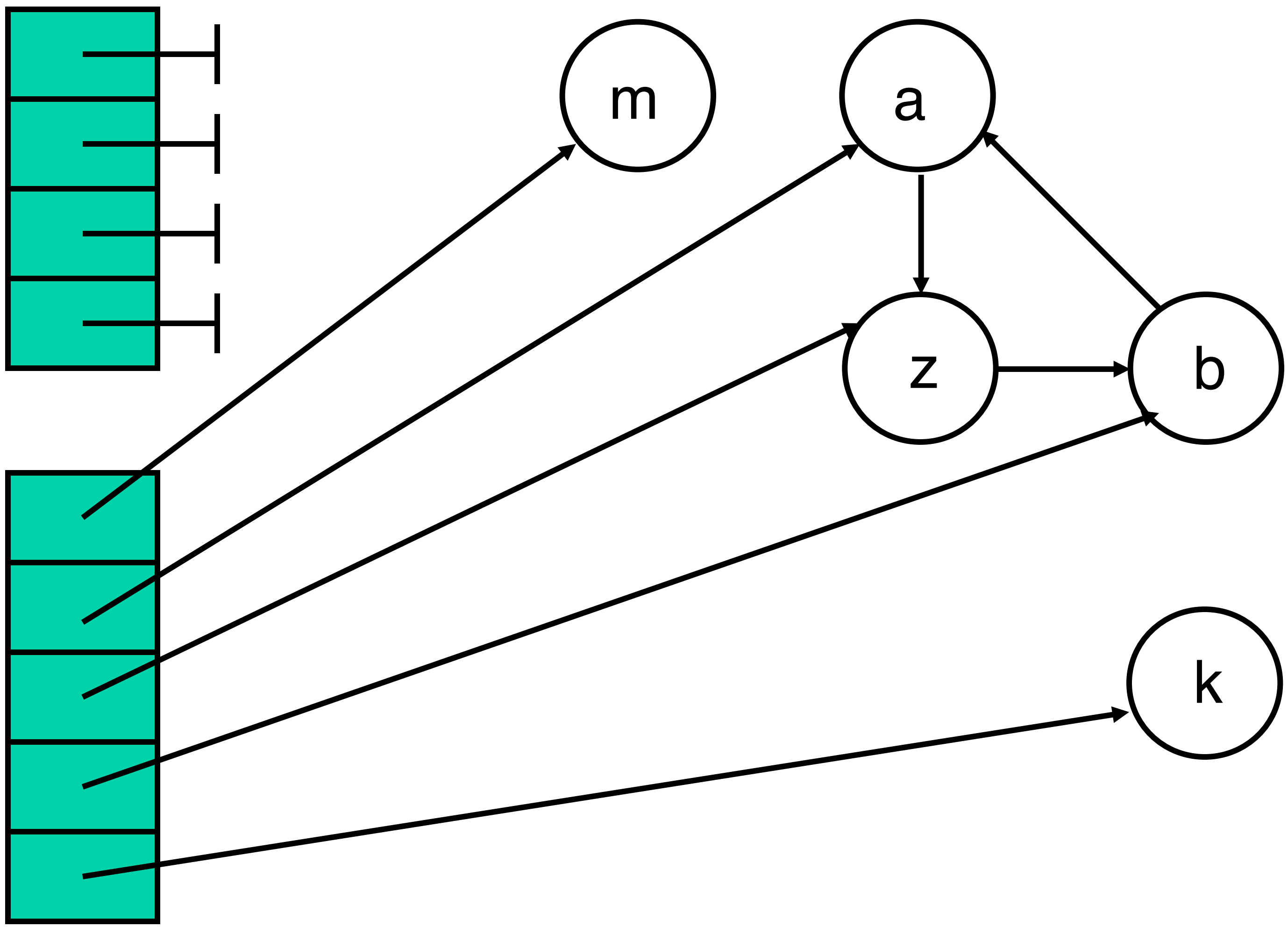
Graph Traversal

- Graph G with i nodes may be unconnected, and yet each unconnected region is part of G
- An algorithm may require visiting each node $n_i \in G$
- Requires additional data structure to guarantee a possible visit to node n_i
- Moreover, any full traversal of G must start with some node of G; which one?

1. Graph Traversal: Fixed Array

- Specify some static array a[k] of k node pointers, all initially null, not necessarily all used, in which each used element a[i], $i < k$ points to node n_i
- Say, in increasing order of indices, starting at index 0
- As soon as a[i] yields a null pointer, no more nodes are identified in a[]
- Thus G is known to have i nodes; or $k = i$
- Advantage:
  - Simple to implement
- Disadvantage:
  - Not recommended!
  - Wasted space almost always, since `a[]` needs to be large
  - Too limited, the method fails if `i >= k`

1. Graph Traversal Fixed Array



2. Graph Traversal Linked List

- Or use separate linked list of links w. pointer to nodes
- Each link element has 2 fields
  - Field next points to next link element, if any; ends with NIL
  - Field finger points to corresponding node $n_i \in G$
- Link element is allocated dynamically off the heap only when another node n_i in G is added
- Advantage: simple to implement
  - Space consumption directly proportional to the number of nodes in G; thus no wasted static space
  - Runs out of space only when all memory space is exhausted
- Disadvantage:
  - Is data structure separate from actual graph G, ends up with 2 data structures to be synchronized
  - Also not recommended

3. Graph Traversal Added Field

- Extend the data structure of the graph G proper!
- Keep all fields needed for the graph, but add a finger field to each node, i.e. field of type pointer to node
- Traversing G requires an outside data structure, of type pointer to node, initially nil, to point to one select start node of the nodes $n_i \in G$
- Thus finger fields form a linked list of graph nodes
- Advantage:
  - Simple to implement
  - Space consumption proportional to number of nodes in G
  - Runs out of space only when all memory space is exhausted
- Disadvantage:
  - One more field, but the cost is contained!
  - You never get something for nothing, unless previously … ☺

Graph Data Structure, Link

```
// node may have any number of successors, incl. 0
// all must be retrievable
// Hence each node in G has a link pointer,
// pointing to LL of all successor nodes
// Last node connected will be inserted at head

typedef struct link_tp * link_ptr_tp;  // forward ref!
typedef struct node_tp * node_ptr_tp;  // forward ref!

typedef struct link_tp
{
  link_ptr_tp next_link;  // point to next link
  node_ptr_tp next_node;  // point to successor node
} str_link_tp;

#define LINK_SIZE sizeof( str_link_tp )
```

Graph Data Structure, Node

```
// "name" is some unique ID given during creation
// Could use another way of "naming" nodes
// "link" is head of LList of successor nodes, while
// "finger" is direct linear link through all nodes
// "visited" only true after visit; initially FALSE

typedef struct node_tp
{
   link_ptr_tp  link;     // points to LL of successors
   node_ptr_tp  finger;   // finger through all nodes
   int          name;     // name=ID given at creation
   bool         visited;  // to check traversal
   others ...             // other fields: attributes
} str_node_tp;

#define NODE_SIZE sizeof( str_node_tp )
```

Building a Graph, one Node

```
// create a node in graph G, identified by "name"
// connect to the global "finger" at head of LList
// assumption: no such node "name" exists in graph
// assume: global "finger" pointer, initially NULL

node_ptr_tp make_node( int name )
{ // make_node
   node_ptr_tp node = (node_ptr_tp) malloc( NODE_SIZE );

   // check once non-Null, not repeatedly on user side!
   ASSERT( node, "space for node missing" );
   node->finger    = finger;   // set: global finger!!
   node->link      = NULL;     // no successors yet
   node->name      = name;     // IDs this node
   node->visited   = FALSE;    // initially
   finger          = node;     // now link to "this"
   return node;
} //end make_node
```

Building a Graph from Linked Pairs

```
// input is list of pairs, each element being a node name
// craft edge from first node name=a to second node name=b;
// If a node is new: create it; else use ptr = exists()
while( scanf( "%d%d", &a, &b ) ) {        // a, b are ints
    if( ! ( first = exists( a ) ) ) {     // 'a' new node?
        first = make_node( a );           // allocate 'a'
    } //end if
    if( ! ( second = exists( b ) ) ) {    // 'b' new node?
        second = make_node( b );          // allocate 'b'
    } //end if
   // both exist. Either created, or pre-existed: Connect!
    if( new_link( first, second ) ) {
        link = make_link( first->link, second );
        ASSERT( link, "no space for link node" );
        first->link = link;
    }else{
        // link was there already, no need to add again!
        printf( "<><> skip duplicate link %d->%d\n", a, b );
    } //end if
} //end while
```

Building a Graph from Linked Pairs

```
// check, whether link between 2 nodes already exists
// if not, return true: New! Else return false, NOT new!
bool new_link( node_ptr_tp first, node_ptr_tp second )
{ // new_link
   int target        = second->name;
   link_ptr_tp link = first->link;

   while( link ) {
       if( target == link->next_node->name ) {
           return FALSE; // it is an existing link, NOT new
       } //end if
       // check next node; if any
       link = link->next_link;
   } //end while
   // none of successors equal the second node's name
   return TRUE;          // is a new link
} //end new_link
```

Building a Graph

- Any graph node may have 0 or more successors
- Even if some nodes have 0 successor nodes, graph must be traversable
- Needs extra data structure to link all nodes together
- If a link exists already, no need to duplicate
- A separate presentation later discusses detail of graph analysis and graph construction

Summary

- Defined algorithm as a finite sequence of discrete steps to perform computations
- Discussed algorithm, computational complexity, prime numbers and Euclid's method and other to test for primality (sic!)
- Introduced data structure graph, and representation in programming languages

References

1. The Humble Programmer: http://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html
2. Algorithm Definitions: http://en.wikipedia.org/wiki/Algorithm_characterizations
3. Solvability: http://www.cs.nott.ac.uk/~nxk/TEACHING/G53COM/G53COMLecture8.pdf
4. C. A. R. Hoare's comment on readability: http://www.eecs.berkeley.edu/~necula/cs263/handouts/hoarehints.pdf
5. Dr. Richard Sites' phrase even on sweatshirt: http://www.cafepress.com/+id_rather_write_programs_hooded_sweatshirt,63975143

References

6. Church-Turing Thesis: http://plato.stanford.edu/entries/church-turing/
7. Linux design: http://www.livinginternet.com/i/iw_unix_gnulinux.htm
8. Words of wisdom: http://www.cs.yale.edu/quotes.html
9. GCD: https://en.wikipedia.org/wiki/Greatest_common_divisor