Algorithms

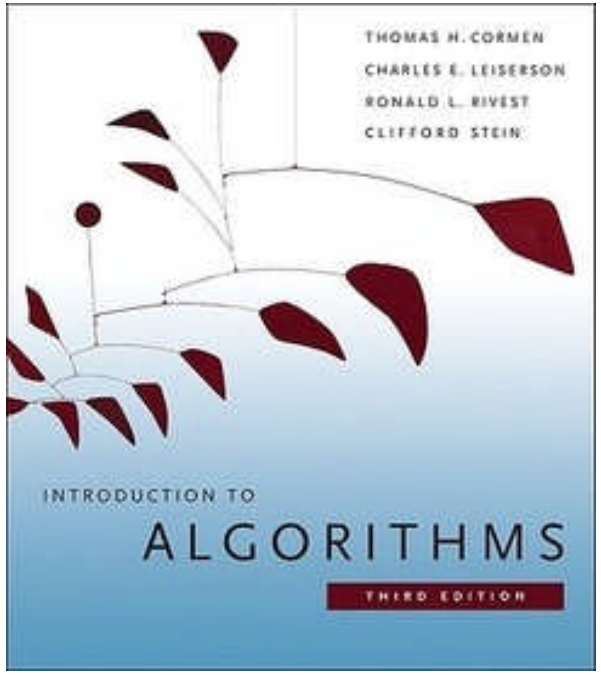
Slides compiled by moshahmed@gmail for NITK class

sources: internet (google wikipedia, books).

Textbook and references

- Cormen, 3rd edition.

  http://www.flipkart.com/introduction-algorithms-8120340078

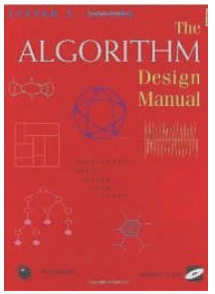


References

- Design and Analysis of Computer Algorithms, by Ullman and Hopcroft
- Algorithm Design Manual, 2nd ed, by Skiena.

Google, Wikipedia

- Math for CS, by Meyer, Lehman, MIT 2010.
- MIT courseware video lectures

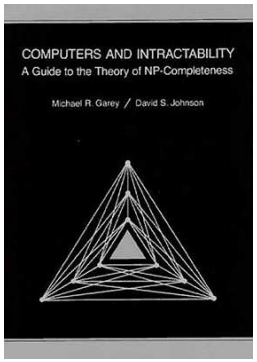


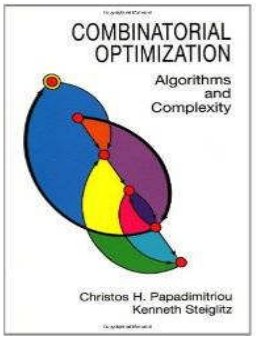
WIKIPEDIA

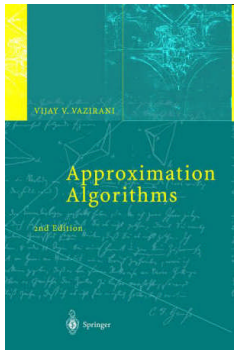
Google

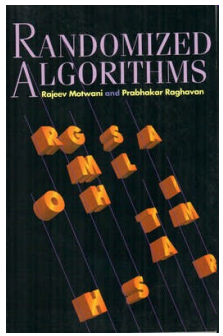
References (advanced)

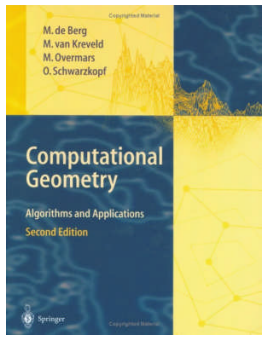
1. Combinatorial Optimization: Algorithms and Complexity, by Papadimitriou and Steiglitz.
2. Computer and Intractability, by Garey and Johnson.
3. Approximation Algorithms, by Vazirani.
4. Randomized Algorithms, by Motwani and Raghavan.





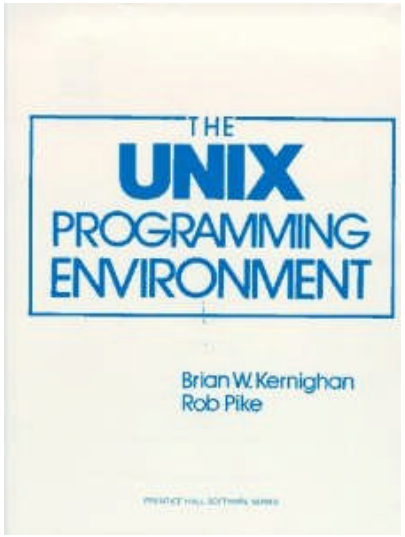


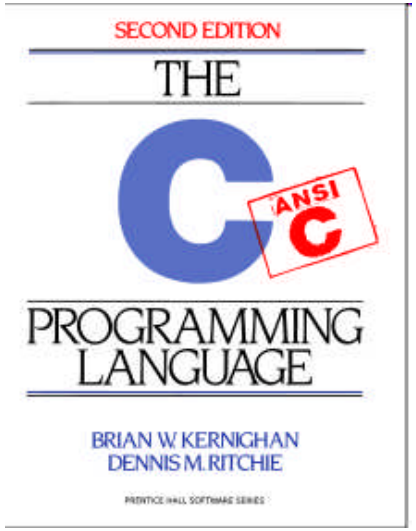


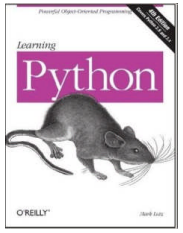


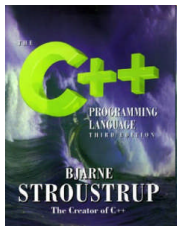
Programming textbooks

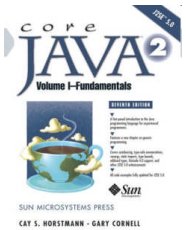
- K&R: C Programming Language, by Kernighan and Ritchie, 2nd ed (not 1st ed).
- Pike: Unix Programing Environment, by Kernighan and Pike.
- C++ Language, by Stroustrup, 3rd ed (not 1st or 2nd ed).
- Learning Python (and docs.python.org)
- Core Java, Vol 1 and 2, by Horstmann.





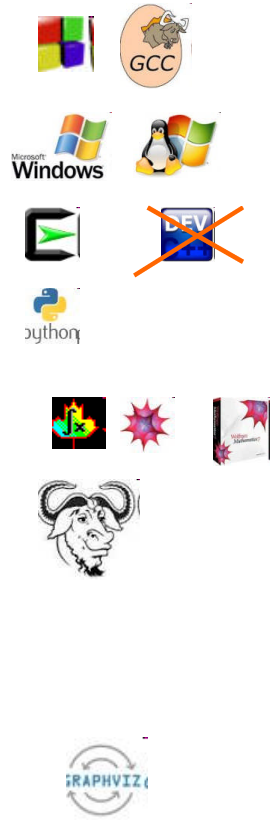






Lab

- C programs using gcc/g++ IDE from http://www.codeblocks.org or MS-VC, not dev-cpp. recent gcc 4.
- On Linux or Windows (install cygwin for bash, make, configure, vim, ctags).
- we will also use Python, Java, C++, js
- Demonstration packages:
  - Maple, Mathematica
  - LP: glpk (gnu linear programming), Excel solver.
  - Optimization: Concorde TSP Solvers, Coin-OR
  - Number Theory: gnupg, gmp, pari, js
  - graphviz for drawing graphs



Topics

1. Algorithm running times
2. Searching & Sorting: insertion, bubble, merge, quick, heap, radix.
3. Data structures: stacks, queues, lists, hashing, heaps, trees, graphs.
4. Searching: lists, binary, dfs, bfs.
5. Greedy algorithms, minimum spanning tree.
6. Dynamic programming: matrix chain multiplication, longest common subsequence,
7. Graph algorithms: shortest path, matchings, max-flow.
8. Matrix operations, LUP decomposition, Strassen matrix mult.
9. Polynomial multiplication and fft.
10. String matching: kmp, rabin karp, boyer moore, suffix trees.
11. Linear programming, simplex.
12. Number theory: gcd, ext-gcd, power-mod, crt, rsa,gpg.
13. NP completeness, SAT, vertex cover, tsp, hamiltonian, partition…
14. Approximation algorithms

Website and discussions

- Code https://sites.google.com/site/algorithms2013/
- IT-235 https://www.facebook.com/groups/algorithms.nitk
- CS-830
  https://www.facebook.com/groups/applied.algorithms.nitk
- To ask questions, reply to questions, discussions, schedules.

Grading scheme

- 50% endsem
- 25% midsem exam
- 10+10% quizzes/handwritten assignments
- 5% attendance

0 marks to all copied assignments/exams.

What is an algorithm?

Algorithm is a step-by-step procedure for calculations. More precisely, an algorithm is an effective method expressed as a finite list of well-defined instructions for calculating a function

A program is an implementation of a algorithm.

Examples

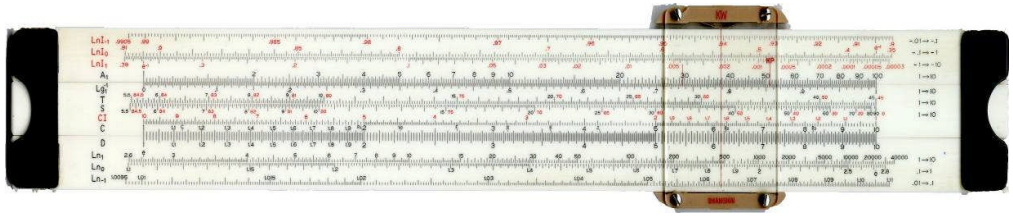
- Compute area of circle, given the radius?
- Driving Directions from NITK to Manipal?
- Recipe to bake a cake?
- Predict tomorrow's weather?
- Autofocus a camera?
- Recognize face? Voice? Songs?
- Oldest algorithm? Euclid's GCD.

Reactive systems

- Examples: Traffic lights, lift, factory alarms
- Properties:
  - Always running, realtime input/output
  - No beginning, no end.
  - Correctness
  - Liveliness
  - Deadlock free
  - Fairness

What is computable?

- Alonzo Church created a method for defining functions called the λ-calculus
- Alan Turing created a theoretical model for a machine, now called a universal Turing machine
- Church–Turing **thesis** states that a function is algorithmically computable if and only if it is computable by a Turing machine.



Turing machine (TM)

- **Turing machine** is a hypothetical device that manipulates symbols on a strip of tape according to a table of rules.
- Despite its simplicity, a TM can be adapted to simulate the logic of any computer algorithm
- Used for explaining real computations.
- A universal Turing machine (UTM) is able to simulate any other Turing machine.
- Church–Turing thesis states that Turing machines capture the informal notion of effective method in logic and mathematics, and provide a precise definition of an algorithm or a 'mechanical procedure'.

Lambda Calculus (λ)

- Formulated by Alonzo Church as a way to formalize mathematics through the notion of functions
- (λ var body) ..Lambda expression/function
- Application:

(λ x y (* (+ 3 x) (- y 4))) 10 20

→ (* (+ 3 10) (- 20 4)) → (* 13 16) → 208

Church–Turing Thesis, the untyped lambda **calculus** is claimed to be capable of computing all effectively calculable functions.

The typed lambda **calculus** is a variety that restricts function application, so that functions can only be applied if they are capable of accepting the given input's "type" of data.

Combinatory Logic

- **SKI combinator calculus** is a computational system that may be perceived as a reduced version of untyped lambda calculus.
- Used in mathematical theory of algorithms because it is an extremely simple Turing complete language.
- Only two operators or combinators: **K***xy = x and* **S***xyz = xz*(*yz*).

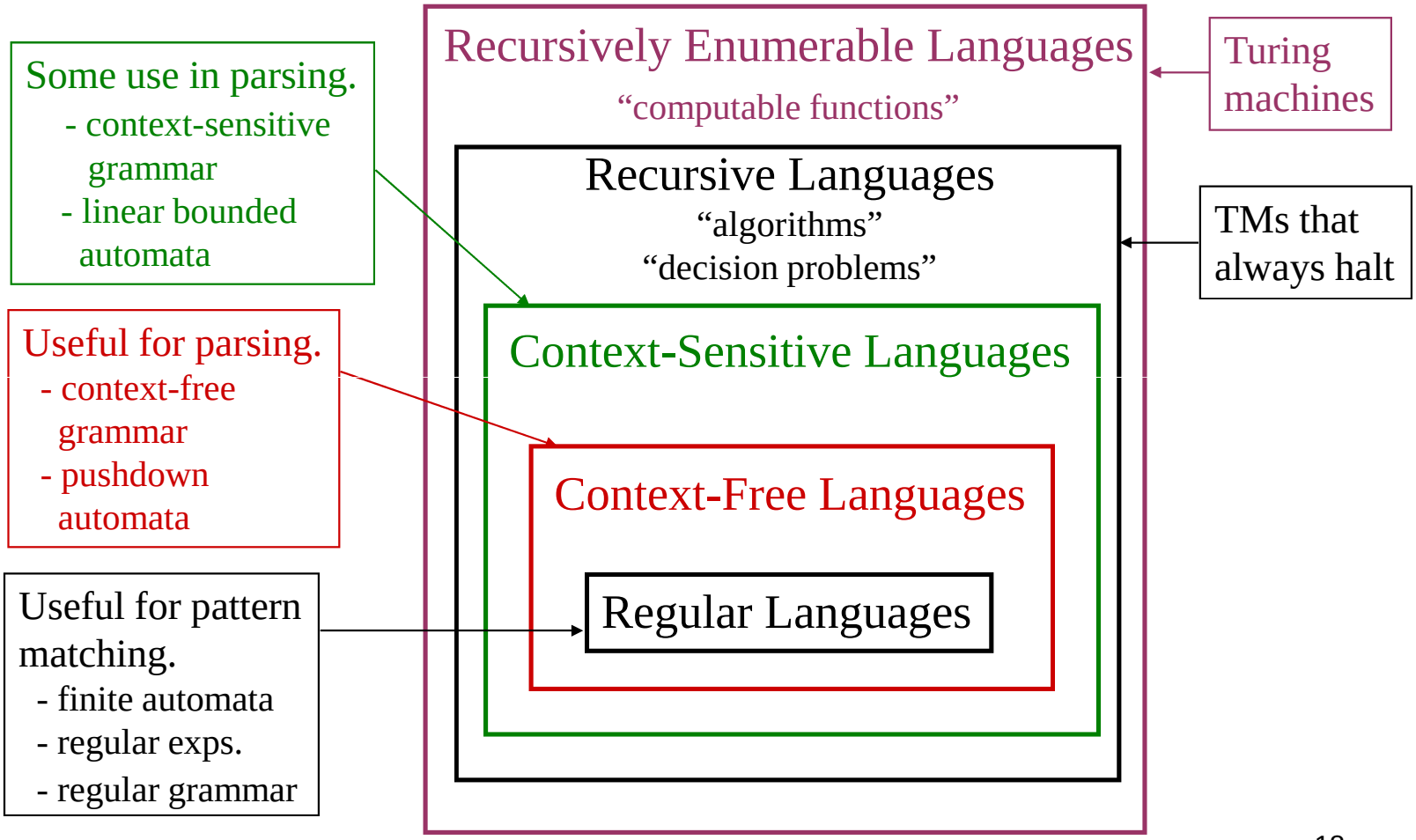
- **K** := λ*x*.λ*y*.*x*
- **S** := λ*x*.λ*y*.λ*z*.*x z* (*y z*)
- **I** := λ*x*.*x*
- **B** := λ*x*.λ*y*.λ*z*.*x* (*y z*)
- **C** := λ*x*.λ*y*.λ*z*.*x z y*
- **W** := λ*x*.λ*y*.*x y y*
- **ω** := λ*x*.*x x*
- **Ω** := **ω ω**
- **Y** := λ*g*.(λ*x*.*g* (*x x*)) (λ*x*.*g* (*x x*))

Example computation or reduction: SKKX ==> KX(KX) ==> X

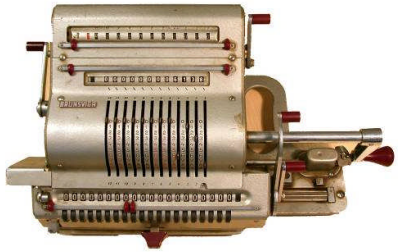
Functional programming

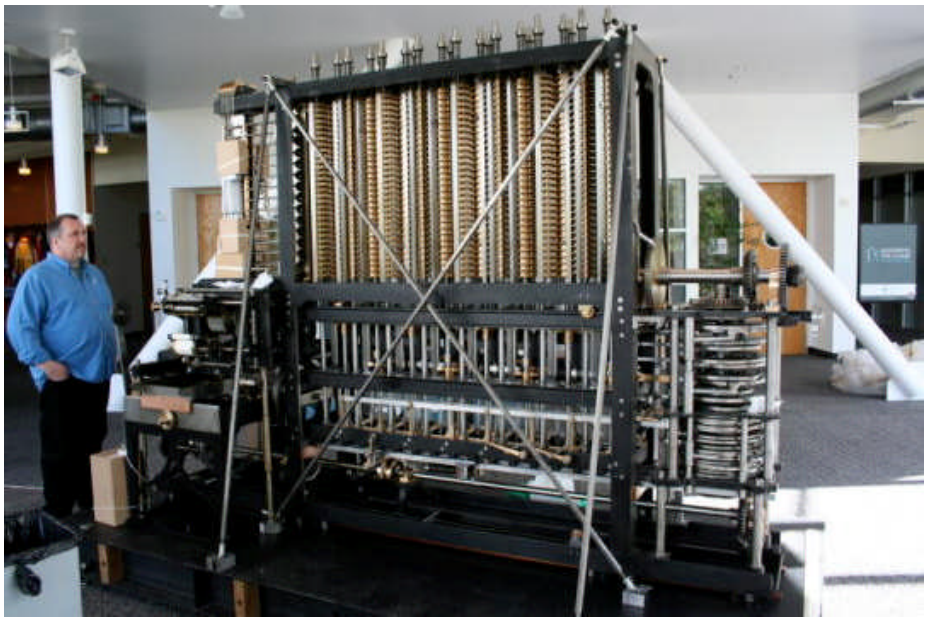
- FP is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data.
- Based on just application of functions without side effects.
- Compare to the imperative programming style, which emphasizes changes in state.
- Functional programming has its roots in lambda calculus
- Languages: Haskell, Ocaml, ML, Lisp, Scala, Clojure,..
- Lisp example: `(defun fibonacci (n &optional (a 0) (b 1)) (if (= n 0) a (fib (- n 1) b (+ a b))))`
- Python Example: >>> (lambda x: x*2)(3)
  6

Languages


Recursively Enumerable Languages
"computable functions"
Turing machines
Some use in parsing.
- context-sensitive grammar
- linear bounded automata
Recursive Languages
"algorithms"
"decision problems"
TMs that always halt
Context-Sensitive Languages
Useful for parsing.
- context-free grammar
- pushdown automata
Context-Free Languages
Regular Languages
Useful for pattern matching.
- finite automata
- regular exps.
- regular grammar

What are these?

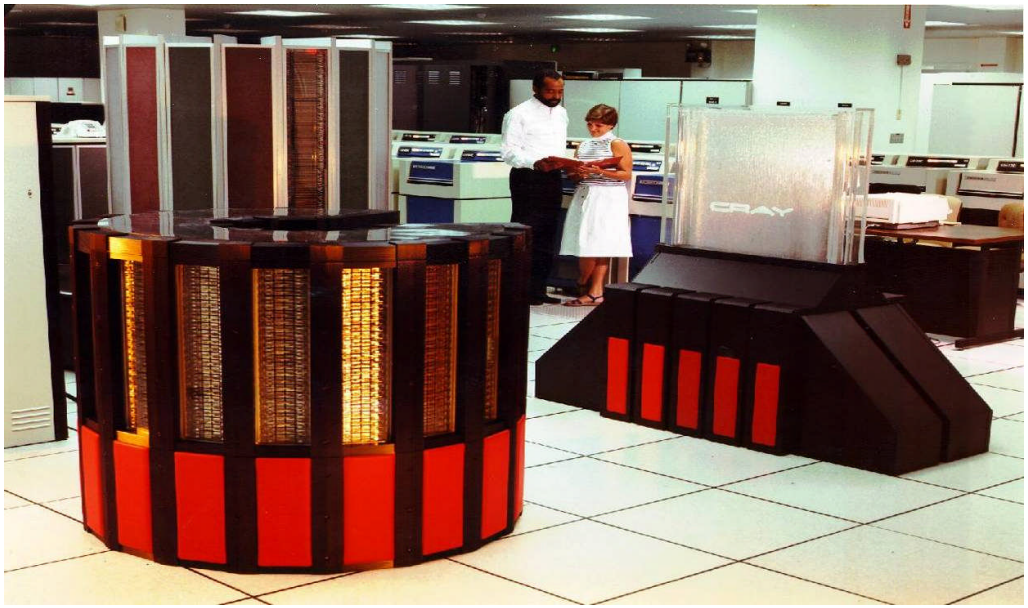




Oracle of Delphi

- A **Turing** machine with a black box, called an oracle, which is able to decide certain decision problems in a single operation.
- The problem can be of any complexity class.
- Even undecidable problems, like the halting problem can be answered by an oracle.

Cray super computer



Data center = Millions of PCs

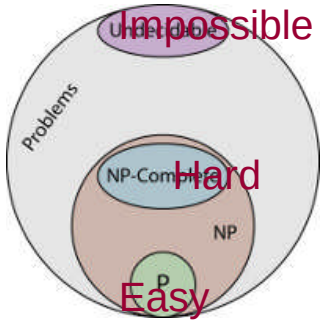
FFT algorithm is used in audio/video/signal processing

Algorithms Part 2

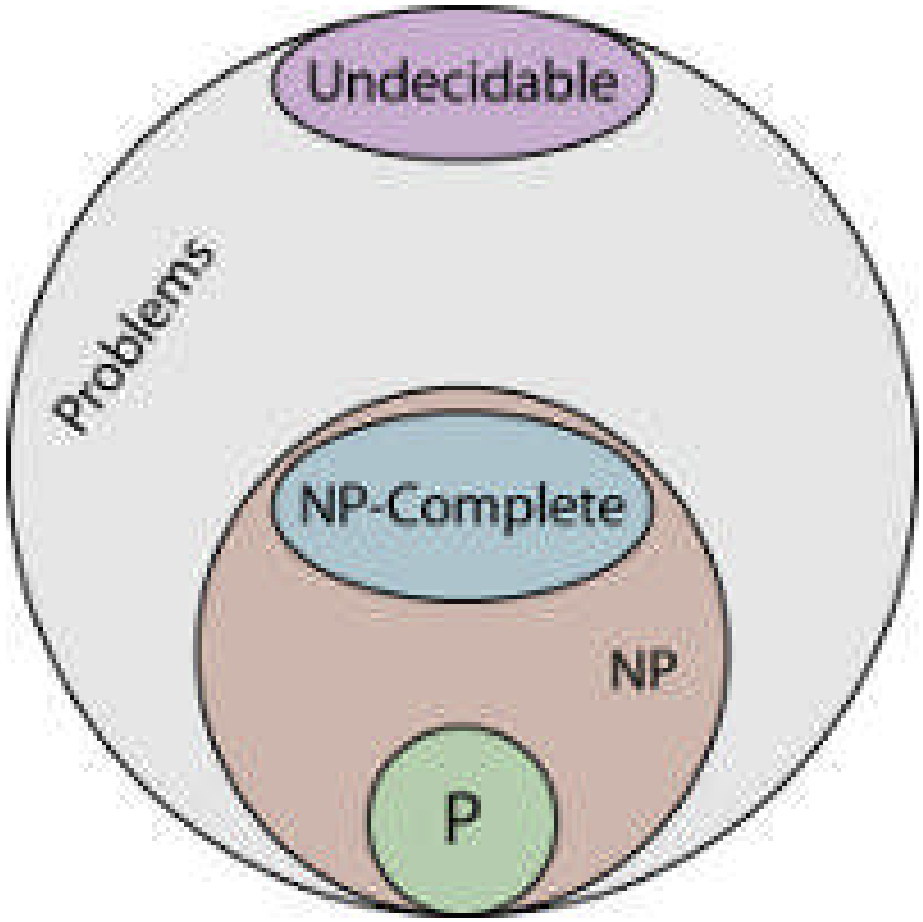
1. Starting from an initial state and initial input (perhaps empty), the instructions describe a computation that, when executed, will proceed through a finite number of well-defined successive states, eventually producing "output" and terminating at a final ending state.
2. The transition from one state to the next is not necessarily deterministic
3. Some algorithms, known as randomized algorithms, incorporate random input.

Some algorithms are harder than others

- Some algorithms are easy
  - Finding the max
  - Searching in a list
- Some algorithms are a bit harder
  - Sorting
- Some algorithms are very hard
  - TSP - Finding the cheapest road tour of n cities
- Some algorithms are practically impossible
  - Factoring large numbers
- Some problems are undecidable
  - (halting problem).



Problem Hierarchy



Algorithm: Maximum element

1. Given a list, how do we find the maximum element in the list?
2. To express the algorithm, we'll use pseudocode. Pseudocode is like a programming language, without worrying about syntax.

Findi the Maximum (psuedo code)

```
procedure max (a[1..n] : array of numbers)
    max := a_1
    for i := 2 to n
      if max < a_i then
        max := a_i
       // {max is largest of a[1..i} // invariant
    endfor
    // {max is the largest of a[1..n]} // invariant
    return max
end procedure
```

Maximum element running time

1. How long does this take?
2. If the list has *n* elements
3. Worst case?
4. Best case?
5. Average case?

Properties of algorithms

1. Input
2. Output
3. Definiteness: the steps are defined precisely
4. Correctness
5. Finiteness: finish in finite steps.
6. Effectiveness: each step must be performed in a finite amount of time.
7. Generality: the algorithm *should* be applicable to all problems of a similar form.

Types of algorithms

- Deterministic.
- Randomized, answer maybe different each time, but correct.
- Approximation, for hard problems answer is close to the best (optimal)
- Online, solve items as they come, without seeing all the items (choices). E.g. Hiring.

Algorithm Design Techniques

1. Incremental: *Insertion sort, Selection sort*.
2. Divide and Conquer: *Merge sort, Quick sort, Binary Search.*
3. Dynamic: *Fibonacci numbers, Matrix chain multiplication, Knapsack problem.*
4. Greedy: *Shortest path problem, knapsack problem, spanning tree.*
5. Graph: *Depth-first search, Breadth-first search, Traveling salesman.*
6. *Reduction: NP completeness*
7. *Brute force: Combinatorial puzzles*

Algorithm Complexity

Time complexity : Specifies how the running time depends on the size of the input. A function mapping "size" n of input to "time" T(n) executed. (Independent of the type of computer used).

Space complexity : Function specifying mapping "size" n of input to space used.

Complexity

- Asymptotic Performance: *How does the algorithm behave as the problem size gets very large?*
  1. Running time
  2. Memory/storage requirements
- asymptotic (big-O) notation : *What does $O(n)$ running time mean? $O(n^2)$? $(n \lg n)$?*

Design and Analysis of Algorithms IT-235

Timings and location

- Tue 10-12pm
- Thu 11-12 and 1-2pm
- Classroom L601
- Office hours: After class
- Lab: Thu 2-5pm

Applied Algorithms
CS-830

- Timings and location

Tue 2-4pm

Thu 2-3pm

Classroom: Seminar Room

Office hours: After class

Marking scheme

- Final 50 %
- Midterm 25 %
- In semester 25%
  - Quizzes 10 %
  - Homework 10%
  - Attendance 5%

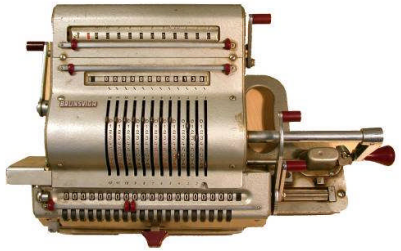
Website and discussions

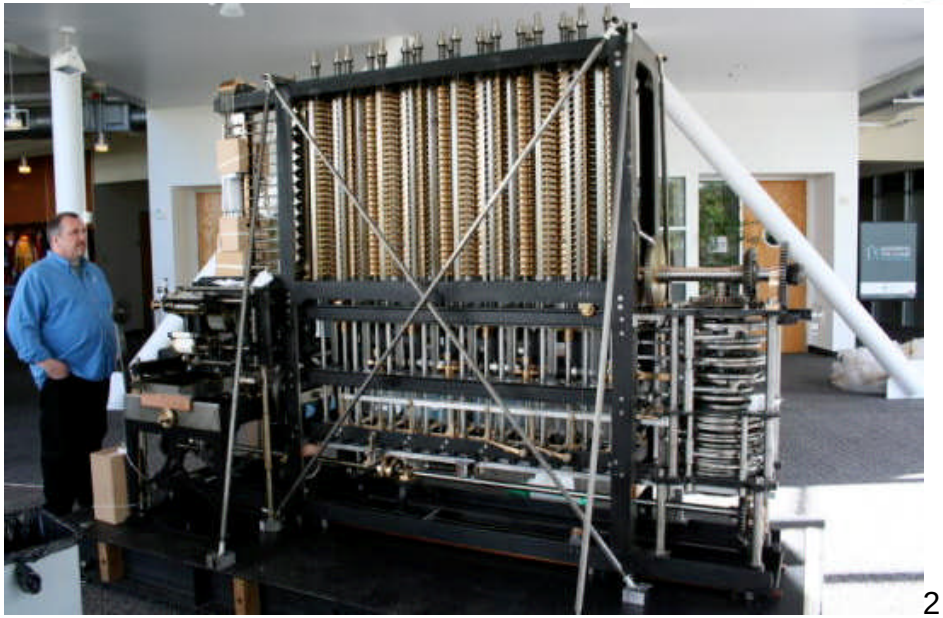
- Code https://sites.google.com/site/algorithms2013/
- IT-235 https://www.facebook.com/groups/algorithms.nitk
- CS-830 https://www.facebook.com/groups/applied.algorithms.nitk
- Ask Class Representative to add you to the group
- To ask questions, reply to questions, discussions, schedules.
- Email: moshahmed/at/gmail

Growth of Functions

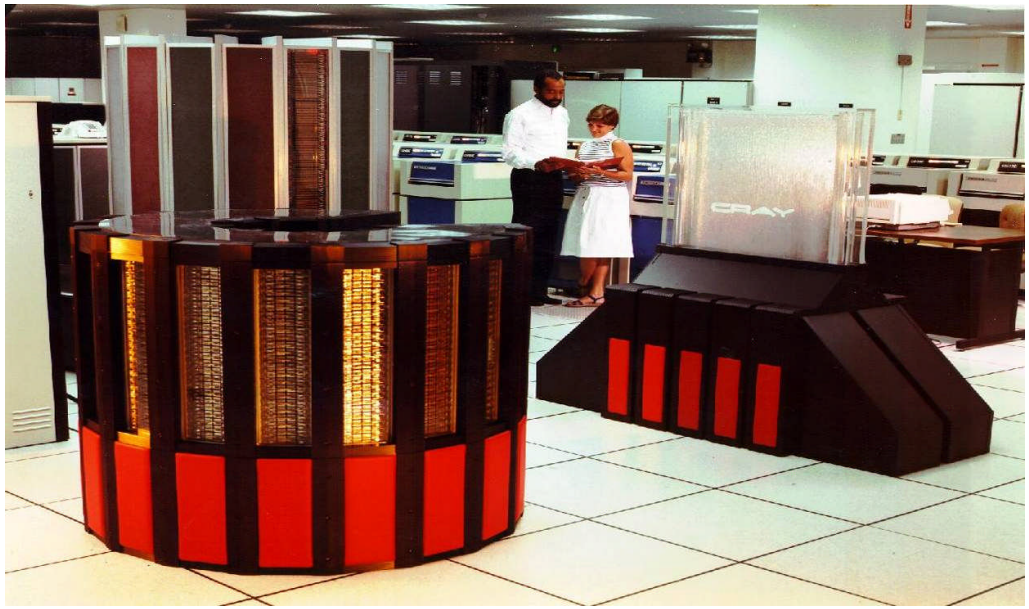
(and computers algorithms)

Pocket calculators

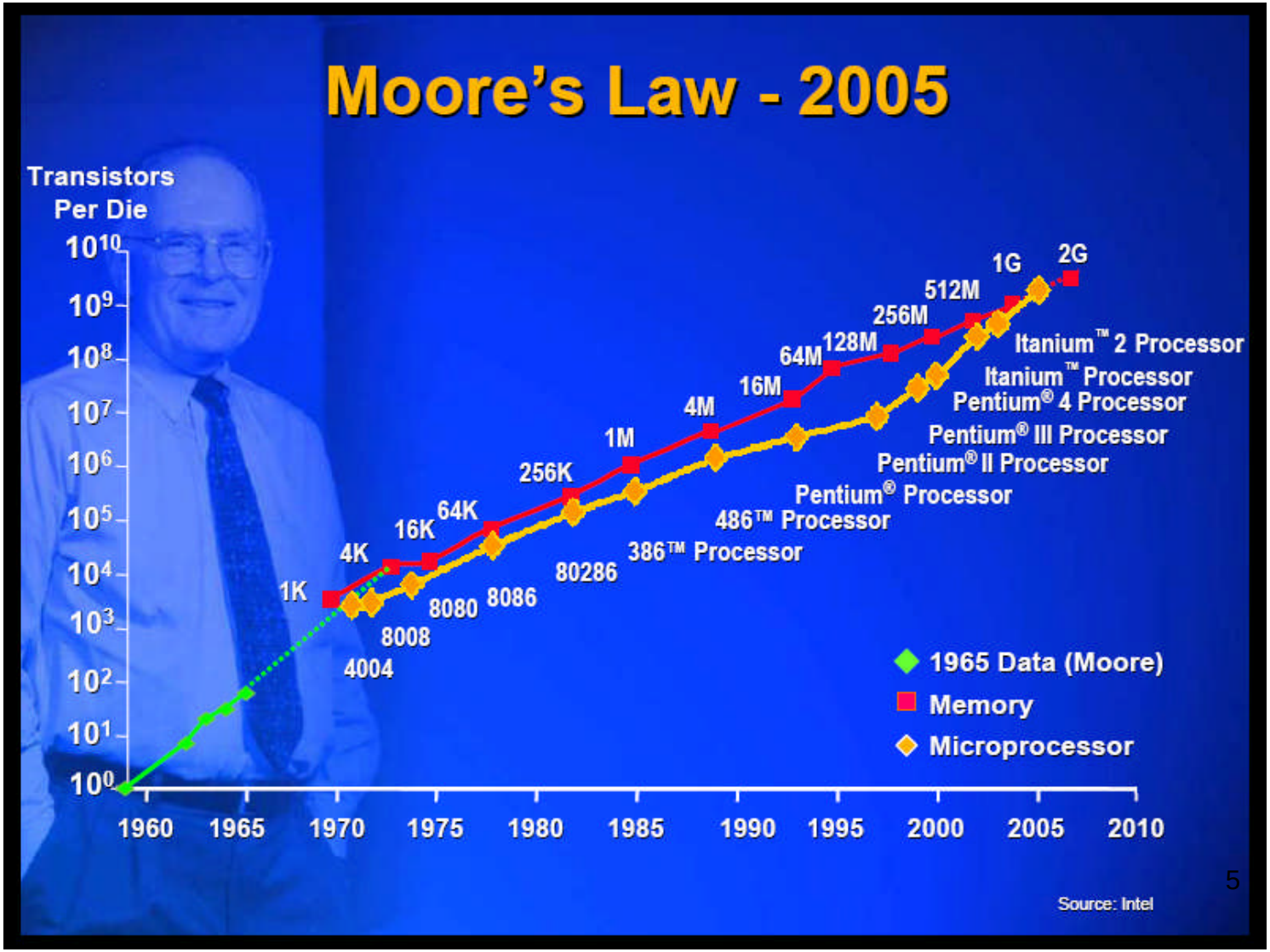


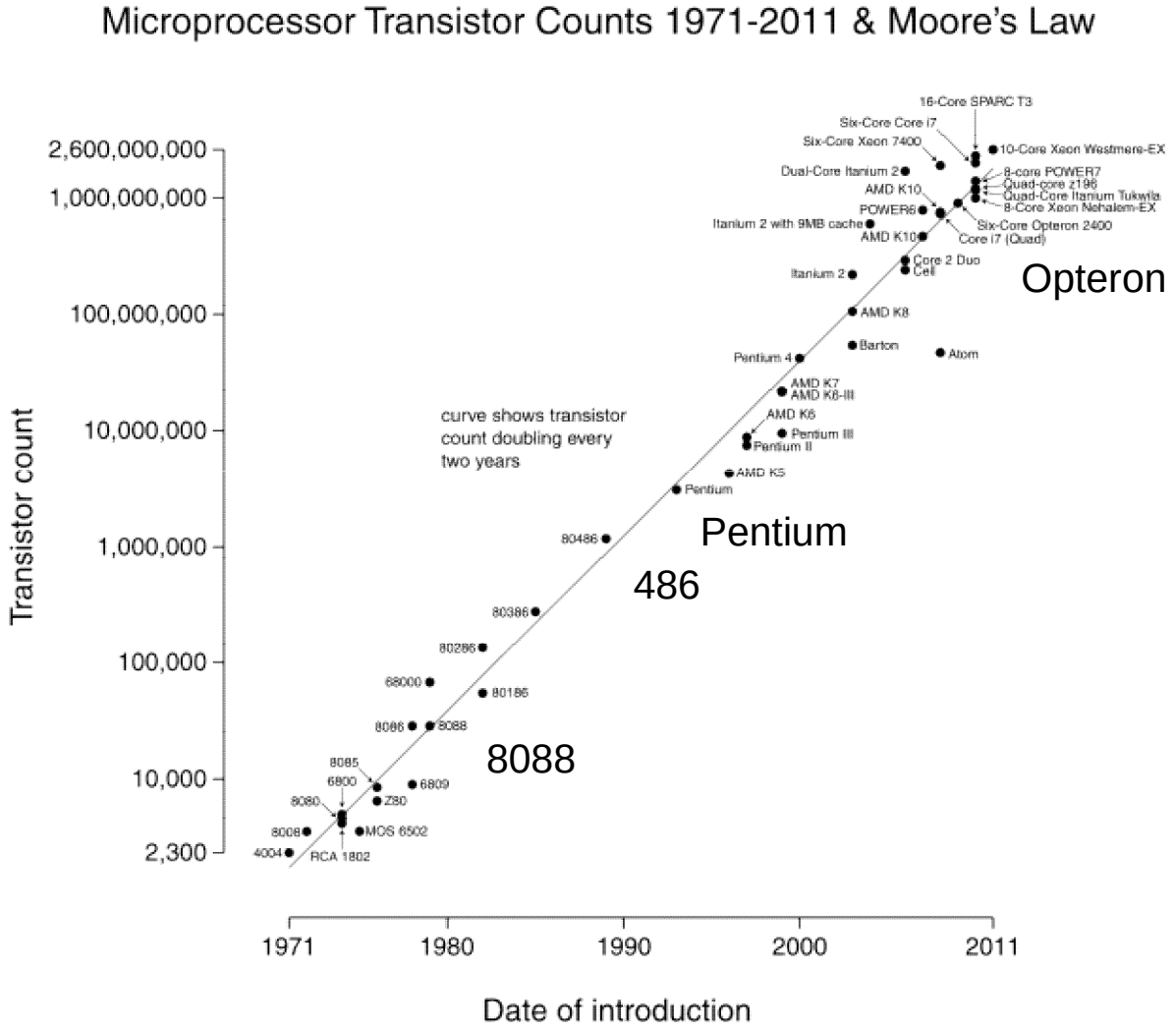


Cray super computer



Data center = Million PCs


Moore's Law - 2005
Transistors
Per Die
10^{10}
10^{9}
10^{8}
10^{7}
10^{6}
10^{5}
10^{4}
10^{3}
10^{2}
10^{1}
10^{0}
1960
1965
1970
1975
1980
1985
1990
1995
2000
2005
2010
1K
4K
16K
64K
256K
1M
4M
16M
64M
128M
256M
512M
1G
2G
4004
8008
8080
8086
80286
386™ Processor
486™ Processor
Pentium® Processor
Pentium® II Processor
Pentium® III Processor
Pentium® 4 Processor
Itanium™ Processor
Itanium™ 2 Processor
1965 Data (Moore)
Memory
Microprocessor
Source: Intel


Microprocessor Transistor Counts 1971-2011 & Moore's Law
Transistor count
2,600,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
2,300
1971
1980
1990
2000
2011
Date of introduction
curve shows transistor count doubling every two years
16-Core SPARC T3
Six-Core Core i7
Six-Core Xeon 7400
10-Core Xeon Westmere-EX
Dual-Core Itanium 2
8-core POWER7
Quad-core z196
Quad-Core Itanium Tukwila
AMD K10
8-Core Xeon Nehalem-EX
POWER6
Six-Core Opteron 2400
Itanium 2 with 9MB cache
Core i7 (Quad)
AMD K10
Core 2 Duo
Cell
Itanium 2
Opteron
AMD K8
Barton
Atom
Pentium 4
AMD K7
AMD K8-III
AMD K6
Pentium III
Pentium II
AMD K5
Pentium
Pentium
80486
486
80386
80286
68000
80186
8086
8088
8088
8085
6800
6809
8080
Z80
8008
MOS 6502
4004
RCA 1802

How does one measure algorithms?

- We can time how long it takes a computer
  - What if the computer is doing other things?
  - And what happens if you get a faster computer?
    - A 3 Ghz Windows machine will run an algorithm at a different speed than a 3 Ghz Linux

Single Step Computer

1. We can loosely define a “step” as a single computer operation
    1. A comparison, an assignment, etc.
    2. Regardless of how many machine instructions it translates into.
2. This allows us to put algorithms into broad categories of efficient-ness
    1. An efficient algorithm on a slow computer will *always* beat an inefficient algorithm on a fast computer on a large problem.

Asymptotic

- Asymptotic efficiency of algorithms
  - How does the running time of an algorithm increase as the input size (n) increases infinitely $n \rightarrow \infty$
- Asymptotic notation (“the order of”)
  - Define sets of functions that satisfy certain criteria and use these to characterize time and space complexity of algorithms

Function growth rates

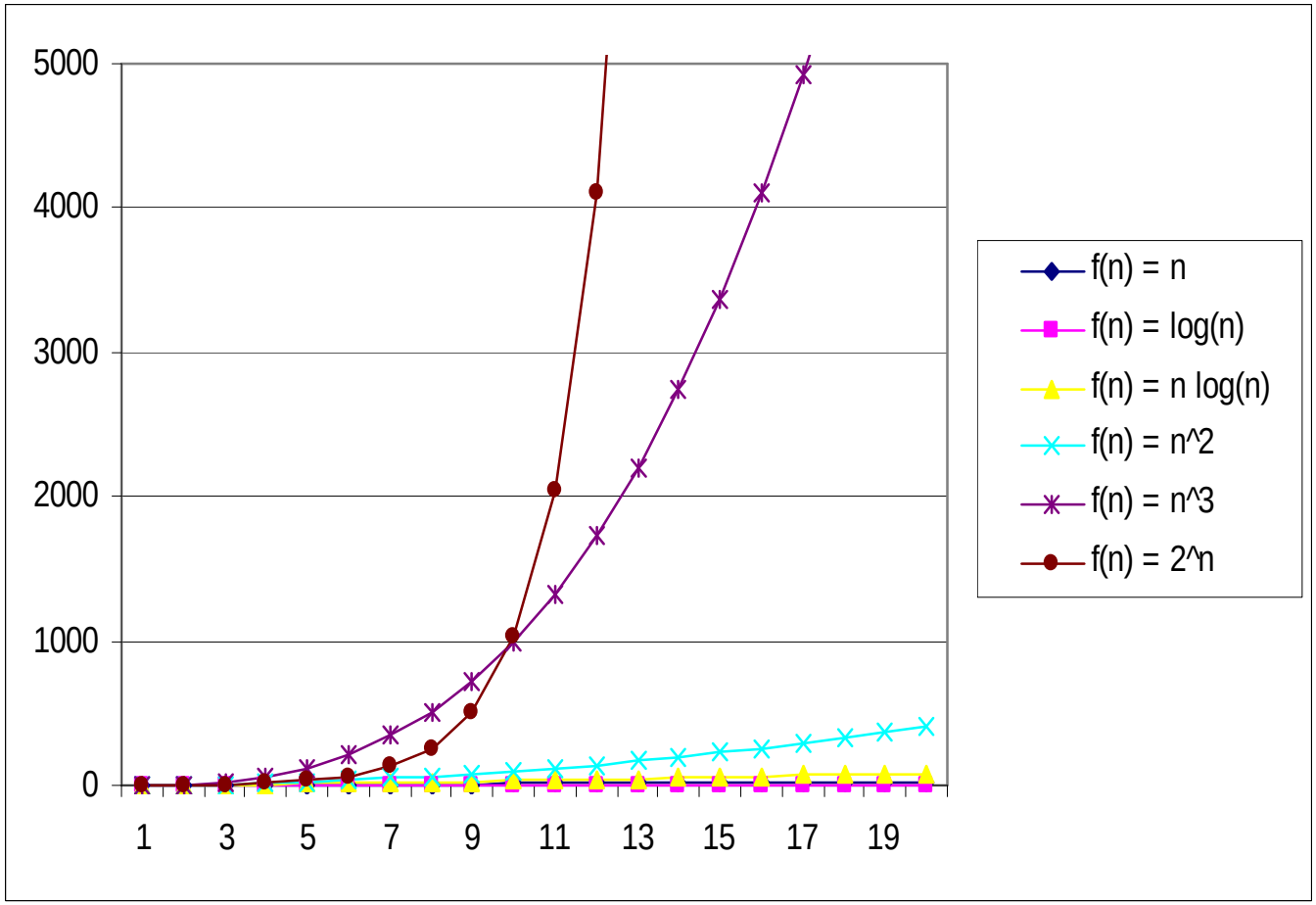
- For input size n = 1000

• $O(1)$	1	
• $O(\log n)$	≈ 10	
• $O(n)$	10^3	
• $O(n \log n)$	$\approx 10^4$	
• $O(n^2)$	10^6	Million
• $O(n^3)$	10^9	Billion
• $O(n^4)$	10^{12}	Trillion
• $O(n^c)$	10^{3*c}	*c* is a consant
• 2^n	$2^{1000} \approx 10^{301}$	
• n!	$\approx 10^{2568}$	
• n^n	10^{3000}	

Complexity nomenclature

Complexity	Term
O(1)	constant
O(log n)	logarithmic
O(n)	linear
O(n lg n)	n log n
$O(n^b)$	polynomial
$O(b^n)$ b > 1	exponential
O(n!)	factorial

Growth of functions



Exponential versus Polynomial

Logarithmic scale!

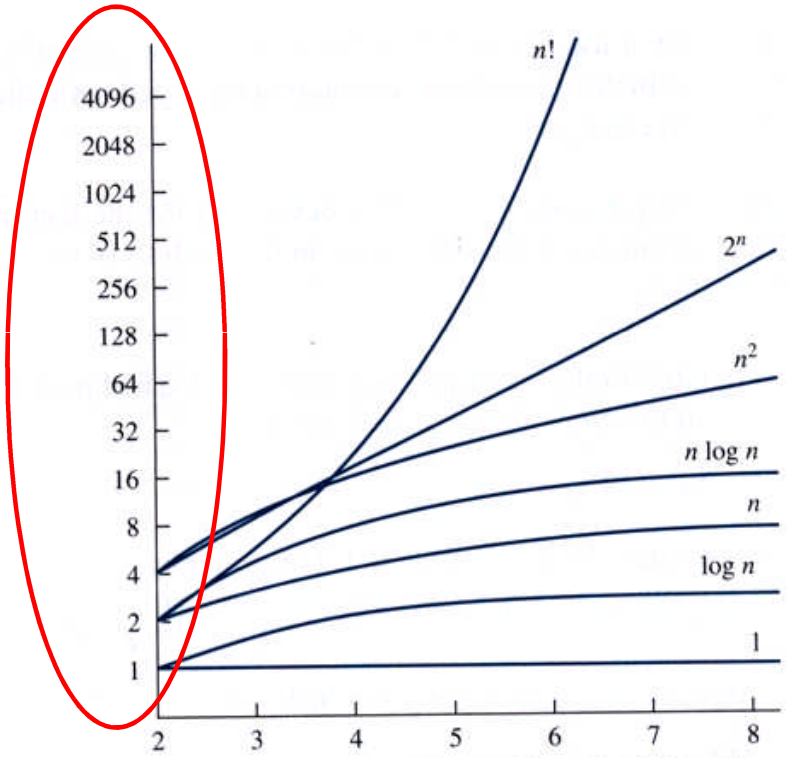


FIGURE 3 A Display of the Growth of Functions Commonly Used in Big-*O* Estimates.

Algorithm Running Time

Given a size n problem, an algorithm runs $O(f(n))$ time:

1. $O(f(n))$: upper bound. (Ω :lower θ: equal)
2. Polynomial: $f(n)=1$(constant), n(linear), n^2(quadratic), n^k. (poly..)
3. Exponential: $f(n)=2^n$, $n!$, n^n.

Time	$n=1$	$n=10$	$n=100$	$n=1000$
n	1	10	10^2	10^3
n^2	1	10^2	10^4	10^6
n^{10}	1	10^{10}	10^{20}	10^{30}
2^n	2	$>10^3$	$>10^{30}$	$>10^{300}$
$n!$	1	$>10^6$	$>10^{150}$	$>10^{2500}$

Too big to solve

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3 nsec	0.01 μ	0.02 μ	0.06 μ	0.51 μ	0.26 μ
16	4 nsec	0.02 μ	0.06 μ	0.26 μ	4.10 μ	65.5 μ
32	5 nsec	0.03 μ	0.16 μ	1.02 μ	32.7 μ	4.29 sec
64	6 nsec	0.06 μ	0.38 μ	4.10 μ	262 μ	5.85 cent
128	0.01 μ	0.13 μ	0.90 μ	16.38 μ	0.01 sec	10^{20} cent
256	0.01 μ	0.26 μ	2.05 μ	65.54 μ	0.02 sec	10^{58} cent
512	0.01 μ	0.51 μ	4.61 μ	262.14 μ	0.13 sec	10^{135} cent
2048	0.01 μ	2.05 μ	22.53 μ	0.01 sec	1.07 sec	10^{598} cent
4096	0.01 μ	4.10 μ	49.15 μ	0.02 sec	8.40 sec	10^{1214} cent
8192	0.01 μ	8.19 μ	106.50 μ	0.07 sec	1.15 min	10^{2447} cent
16384	0.01 μ	16.38 μ	229.38 μ	0.27 sec	1.22 hrs	10^{4913} cent
32768	0.02 μ	32.77 μ	491.52 μ	1.07 sec	9.77 hrs	10^{9845} cent
65536	0.02 μ	65.54 μ	1048.6 μ	0.07 min	3.3 days	10^{19709} cent
131072	0.02 μ	131.07 μ	2228.2 μ	0.29 min	26 days	10^{39438} cent
262144	0.02 μ	262.14 μ	4718.6 μ	1.15 min	7 mnths	10^{78894} cent
524288	0.02 μ	524.29 μ	9961.5 μ	4.58 min	4.6 years	10^{157808} cent
1048576	0.02 μ	1048.60 μ	20972 μ	18.3 min	37 years	10^{315634} cent

Table of common time complexities

Further information: Computational complexity of mathematical operations

The following **table** summarises some classes of commonly encountered time complexities. In the **table**, poly(x) = $x^{O(1)}$, i.e., polynomial in x.

Name	Complexity class	Running time ($T(n)$)	Examples of running times	Example algorithms
constant time		$O(1)$	10	Determining if an integer (represented in binary) is even or odd
inverse Ackermann time		$O(\alpha(n))$		Amortized time per operation using a disjoint set
iterated logarithmic time		$O(\log^* n)$		Distributed coloring of cycles
log-logarithmic		$O(\log \log n)$		Amortized time per operation using a bounded

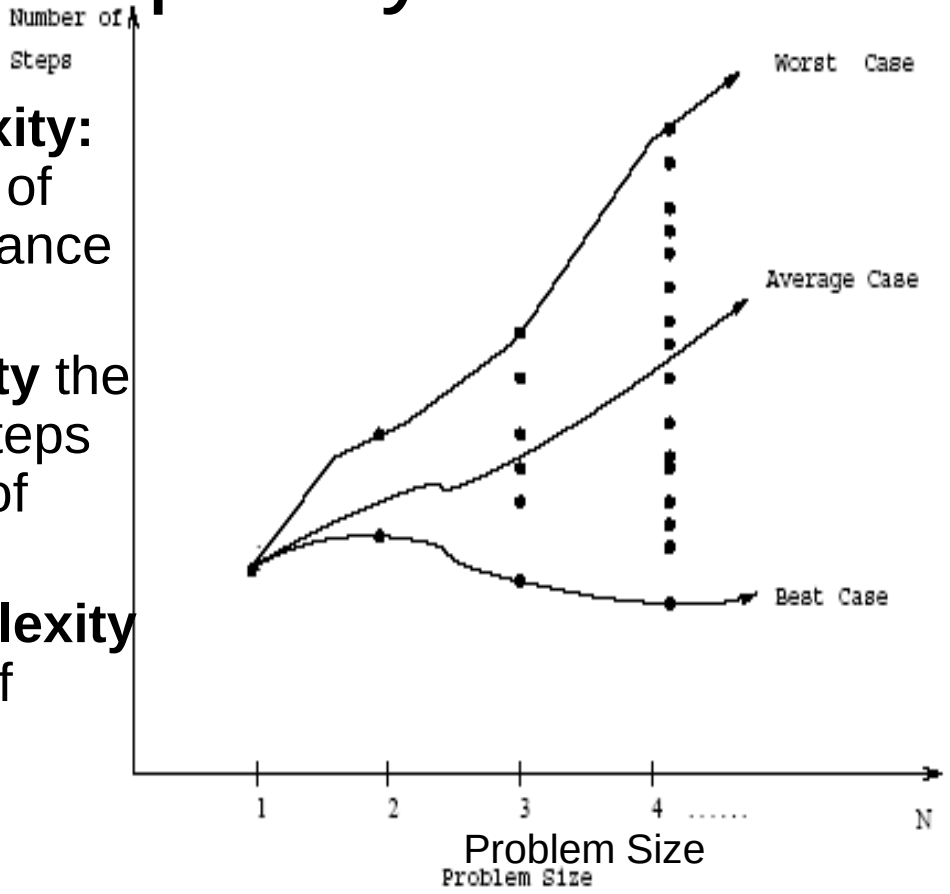
				priority queue[2]
logarithmic time	DLOGTIME	$O(\log n)$	$\log n$, $\log(n^2)$	Binary search
polylogarithmic time		$\text{poly}(\log n)$	$(\log n)^2$	
fractional power		$O(n^c)$ where $0 < c < 1$	$n^{1/2}$, $n^{2/3}$	Searching in a kd-tree
linear time		$O(n)$	n	Finding the smallest item in an unsorted array
"n log star n" time		$O(n \log^* n)$		Seidel's polygon triangulation algorithm.
linearithmic time		$O(n \log n)$	$n \log n$, $\log n!$	Fastest possible comparison sort
quadratic time		$O(n^2)$	n^2	Bubble sort; Insertion sort
cubic time		$O(n^3)$	n^3	Naive multiplication of two $n \times n$ matrices. Calculating partial correlation.
polynomial time	P	$2^{O(\log n)} = \text{poly}(n)$	n, $n \log n$, n^{10}	Karmarkar's algorithm for linear programming; AKS primality test
quasi-polynomial time	QP	$2^{\text{poly}(\log n)}$	$n^{\log \log n}$, $n^{\log n}$	Best-known $O(\log^2 n)$-approximation algorithm for the directed Steiner tree problem.

sub-exponential time (first definition)	SUBEXP	$O(2^{n^{\varepsilon}})$ for all $\varepsilon > 0$	$O(2^{\log n^{\log n}})$	Assuming complexity theoretic conjectures, BPP is contained in SUBEXP.[3]
sub-exponential time (second definition)		$2^{o(n)}$	$2^{n^{1/3}}$	Best-known algorithm for integer factorization and graph isomorphism
exponential time	E	$2^{O(n)}$	1.1^n, 10^n	Solving the traveling salesman problem using dynamic programming
factorial time		$O(n!)$	$n!$	Solving the traveling salesman problem via brute-force search
exponential time	EXPTIME	$2^{\text{poly}(n)}$	2^n, 2^{n^2}	
double exponential time	2-EXPTIME	$2^{2^{\text{poly}(n)}}$	2^{2^n}	Deciding the truth of a given statement in Presburger arithmetic

From http://en.wikipedia.org/wiki/Time_complexity

Complexity

1. **Worst-Case Complexity:** the maximum number of steps taken on an instance of size n.
2. **Best-Case Complexity** the minimum number of steps taken on an instance of size n.
3. **Average-Case Complexity** the average number of steps taken on any instance of size n.



Example of difficult problem

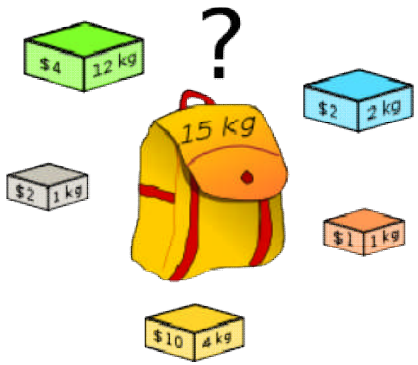
- Factoring a composite number into it's component primes is $O(2^n)$
  - Where n is the number of bits in the number, i.e. The length of the number in (binary) bits.
- This, if we choose 2048 bit number for an RSA keys, it will take 2^{2048} (10^{617}) steps to factor it.

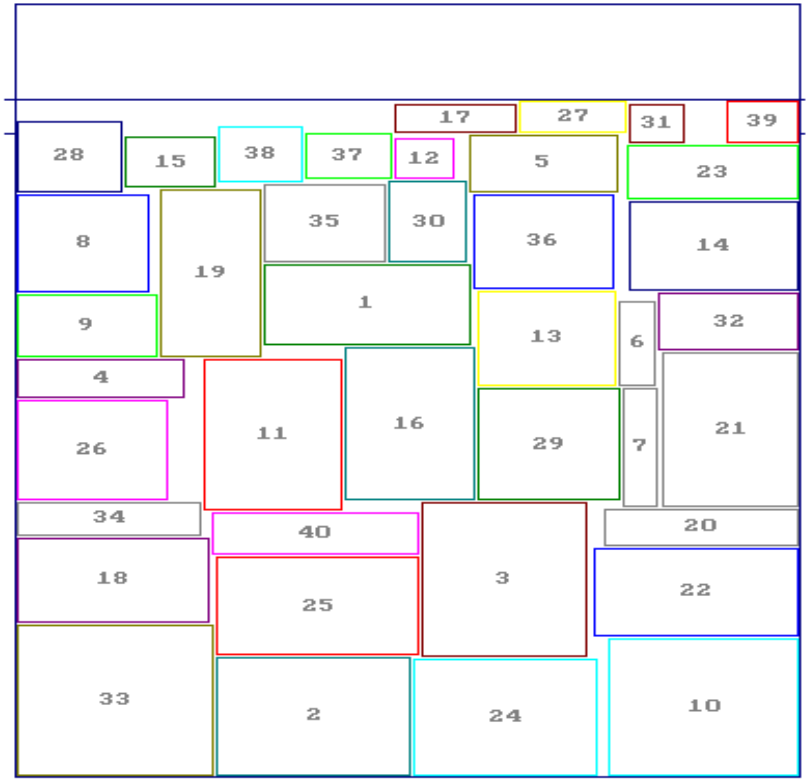
NP Hard problems

Solving hard problems in practice

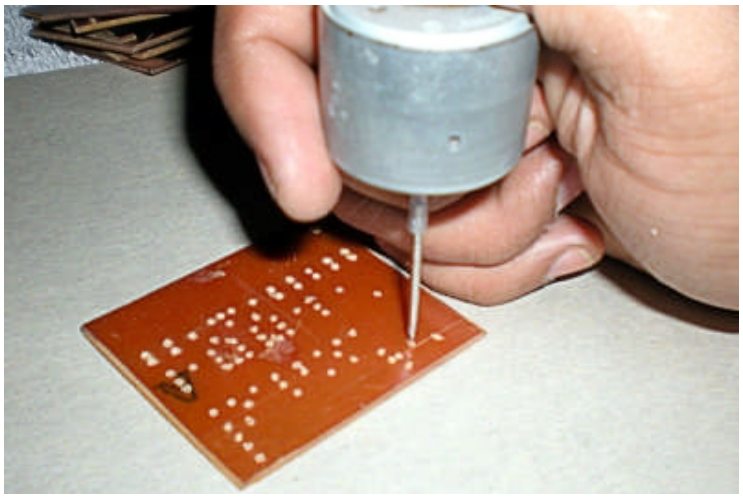
- SIMPLEX - Optimization using simplex method can take exponential steps in worst case, but in practice it finishes quickly.
- Hard problems can be solved quickly using fast approximation algorithms, if you don’t want the very best answer.

Knapsack and bin packing are NP complete problems

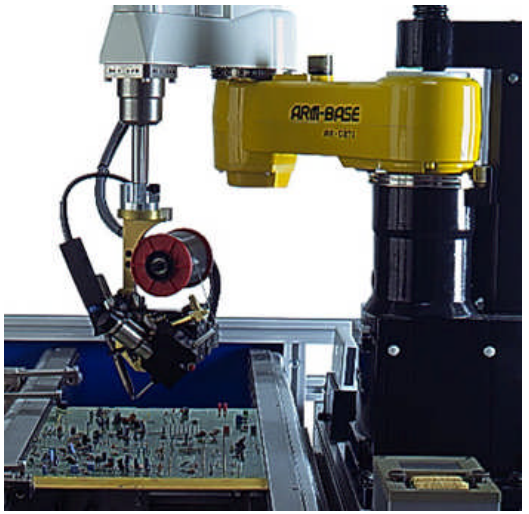


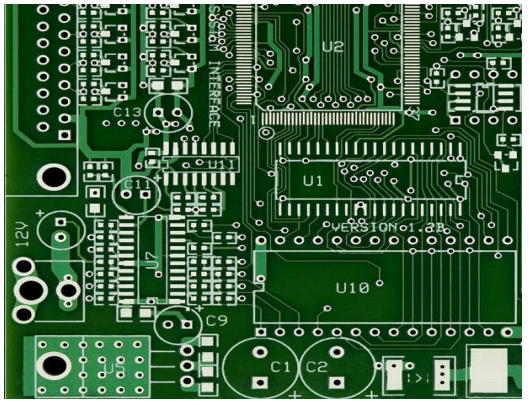


Moving a drill minimal amount to make a circuit board



Soldering on a circuit board





Beyond Polynomial

1. Polynomial: Any expression of the form n^c, where c is a constant Thus, any function that is $O(n^k)$ is a polynomial-time function, for a fixed k.
2. Exponential 2^n, $n!$, n^n (but not polynomial)
3. Elementary: +,-,*,/,log,exp,n!
4. Non-elementary: Ackerman, BusyBeaver

Ackerman function

(among the fastest growing functions)

```
ack 0  n = n + 1                          - - haskell
ack m 0 = ack (m-1) 1
ack m n = ack (m-1) (ack m (n-1))
```

Values of $A(m, n)$

m\n	0	1	2	3	4	n
0	1	2	3	4	5	$n+1$
1	2	3	4	5	6	$n+2 = 2+(n+3)-3$
2	3	5	7	9	11	$2n+3 = 2\cdot(n+3)-3$
3	5	13	29	61	125	$2^{(n+3)}-3$
4	13 $=2^{2^2}-3$	65533 $=2^{2^{2^2}}-3$	$2^{65536}-3$ $=2^{2^{2^{2^2}}}-3$	$2^{2^{65536}}-3$ $=2^{2^{2^{2^{2^2}}}}-3$	$2^{2^{2^{65536}}}-3$ $=2^{2^{2^{2^{2^{2^2}}}}}-3$	$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}_{n+3}-3$

Slowest growing function α

- α(*n*), slowest growing function.
- As slow as ackermann is fast, not inverse
- Algorithm 'Union-Find' runtime is O(*m* α(*n*) + *n*), where *n* is the number of elements and *m* is the total number of Union and Find operations performed.

The following table shows the first values of $\alpha_k(n)$:

n =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	...	n
$\alpha_1(n)$ =	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8	9	9	10	10	11	11	12	12	13	...	$[n / 2]$
$\alpha_2(n)$ =	0	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	5	...	$[\log_2 n]$
$\alpha_3(n)$ =	0	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	4	4	4	4	4	4	4	4	4	...	$\log^* n$
$\alpha_4(n)$ =	0	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	...	
...																											

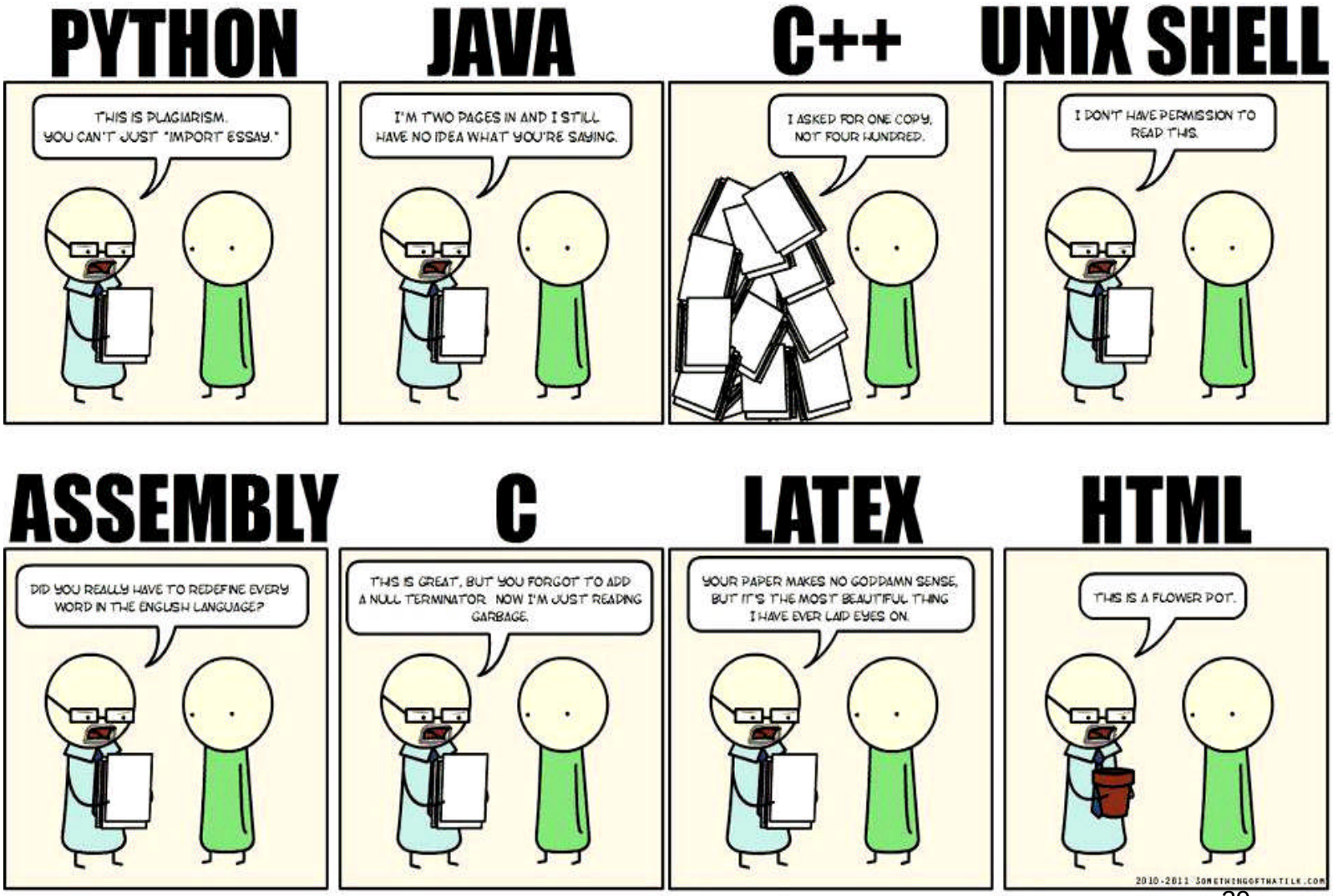
Install the following software

Windows:

- Cygwin with gcc, g++, xwindows
- Codeblocks (gcc and g++ with ide)
- python 2.7 in c:\python27
- Java openjdk17 in c:\java\openjdk17

Linux

- Codeblocks

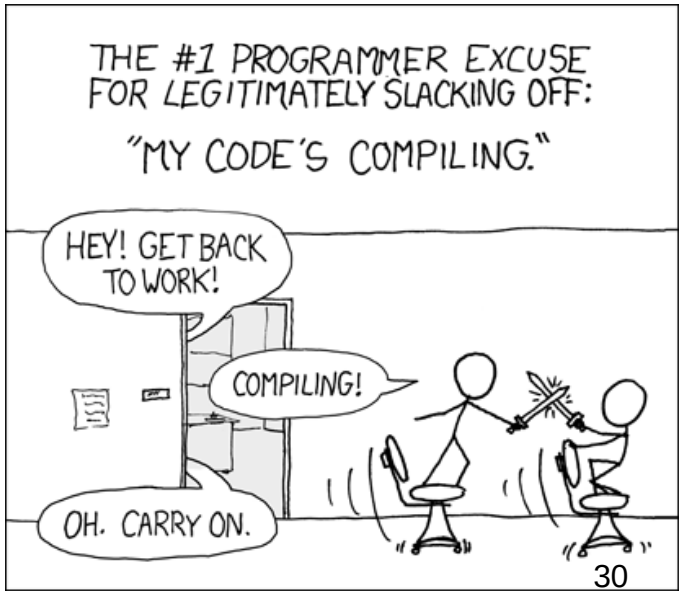

PYTHON
THIS IS PLAGIARISM. YOU CAN'T JUST "IMPORT ESSAY."
JAVA
I'M TWO PAGES IN AND I STILL HAVE NO IDEA WHAT YOU'RE SAYING.
C++
I ASKED FOR ONE COPY, NOT FOUR HUNDRED.
UNIX SHELL
I DON'T HAVE PERMISSION TO READ THIS.
ASSEMBLY
DID YOU REALLY HAVE TO REDEFINE EVERY WORD IN THE ENGLISH LANGUAGE?
C
THIS IS GREAT, BUT YOU FORGOT TO ADD A NULL TERMINATOR. NOW I'M JUST READING GARBAGE.
LATEX
YOUR PAPER MAKES NO GODDAMN SENSE, BUT IT'S THE MOST BEAUTIFUL THING I HAVE EVER LAID EYES ON.
HTML
THIS IS A FLOWER POT.
2010-2011 SOMETHINGOFTHATILK.COM

Home work

Write a program to print the table of

Ackermann [0..3, 0..4] in

C, Python and Java.



Solutions: Ackerman in python, C, Java

```
1.    # ackerman.py
2.    # naïve ackermann function
3.    def ackermann(m, n):
4.        global calls
5.        calls += 1
6.        if m == 0:
7.            return n + 1
8.        elif n == 0:
9.            return ackermann(m - 1, 1)
10.       else:
11.           return ackermann(m - 1, ackermann(m, n - 1))
```

```
1. /* ackerman.c */
2.  int A (int m, int n)  {
3.      if (m == 0) return n + 1;
4.      if (n == 0) return A(m - 1, 1);
5.      return A(m - 1, A(m, n - 1));
6.  }
```

```
1. //  Ackerman.java
2.  public class Ackermann {
3.    public static long ackermann(long m, long n) {
4.      if (m == 0) return n + 1;
5.      if (n == 0) return ackermann(m - 1, 1);
6.      return ackermann(m - 1, ackermann(m, n - 1));
7.    }
8.    public static void main(String[] args) {
9.      long M = Long.parseLong(args[0]);
10.     long N = Long.parseLong(args[1]);
11.     System.out.println(ackermann(M, N));
12.   }
13. }
```

Questions

Q. Which has higher complexity:

1. O(n!) or O(exp(n))?
2. O(n log n) or O(n^1.5)?
3. O(log n) or O(sqrt(n))?
4. O(n log n) or O(n)?
5. O(n log n) or O((log n)^2) ?

Answers

Q. Which has higher complexity:

1. O(n!) or O(exp(n))?
2. O(n log n) or O(n^1.5)?
3. O(log n) or O(sqrt(n))?
4. O(n log n) or O(n)?
5. O(n log n) or O((log n)^2) ?

See http://math.stackexchange.com/questions/145739/prove-that-logn-o-sqrtn
O(log n) = o(sqrt(n)), and log(x) < sqrt(x) for large x.

Growth of functions

Notation: O o Ω ω Θ

Meaning of **O o Ω ω Θ**

- $f(n) = O(g(n)) \approx a \leq b$ — f no faster than **g**
- $f(n) = o(g(n)) \approx a < b$ — f slower than **g**

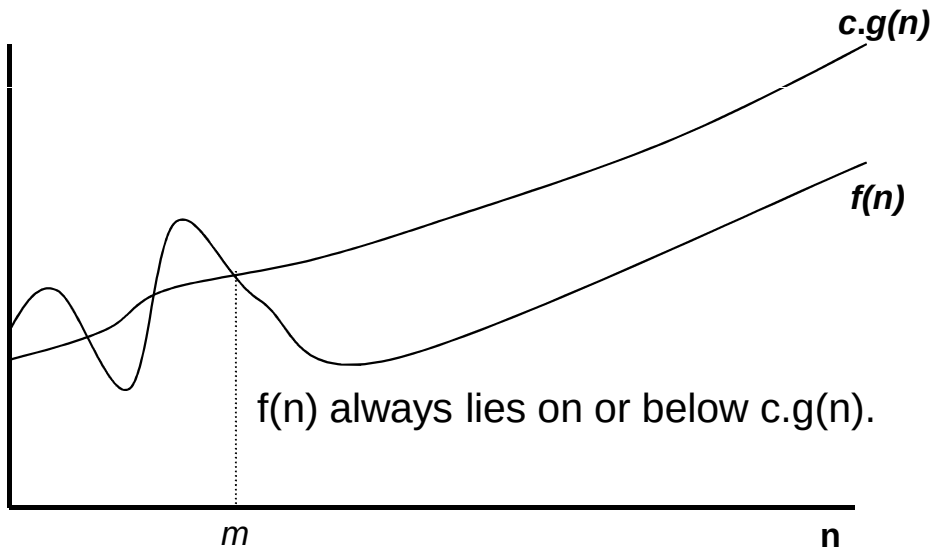
- $f(n) = \Theta(g(n)) \approx a = b$ — f about as fast as **g**

- $f(n) = \Omega(g(n)) \approx a \geq b$ — f no slower than **g**
- $f(n) = \omega(g(n)) \approx a > b$ — f faster than **g**

Big O (Upper bound, Worst case)

f(n) in O(g(n)) = Commonly used notation f(n) = O(g(n))

$$\exists\, c, m > 0 : 0 \leq f(n) \leq c\, g(n), \forall\, n \geq m$$

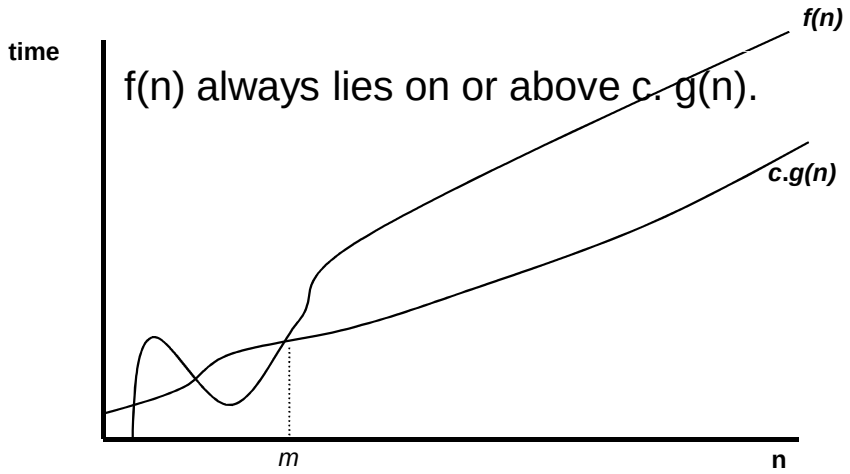


f(n) = O(g(n)

Big Omega Ω-notation (Lower bound, Best case)

f(n) is Ω(g(n)) =

$\exists\ c, m > 0 : 0 \leq c, g(n) \leq f(n), \ \forall\ n \geq m$



f(n) = Ω(g(n)

Think of the equality as meaning *in the set of functions*.

Big Theta Θ-notation (Tight bound)

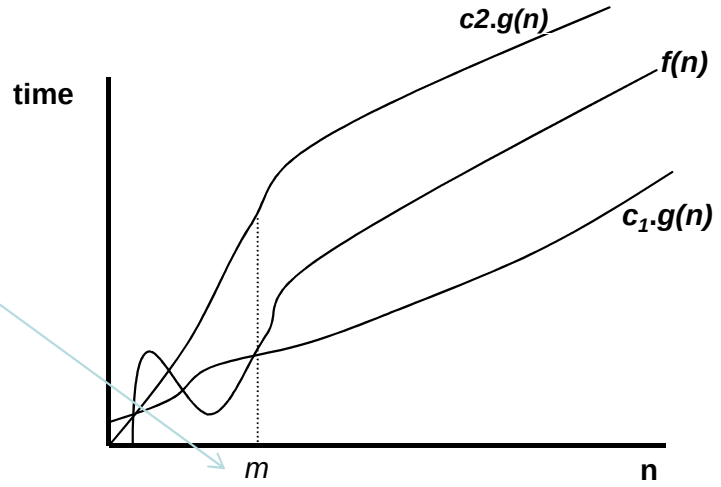
f(n) always lies between c1.g(n) and c2.g(n).

f(n) in Θ(g(n)) =

$\exists\ c_1, c_2 > 0, m$: such that

$$0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)$$

$$\forall\ n \geq m\ \}$$



f(n) = Θ(g(n)

Other Asymptotic Notations

- o **Little o** **-** A function f(n) is o(g(n))

  if $\exists$ c, $m > 0$:

  $f(n) < c\ g(n) \quad \forall\ n \geq m$

- ω **Little-omega -** A function f(n) is ω(g(n))

  if $\exists\ c, m > 0$:

  $c\ g(n) < f(n) \quad \forall\ n \geq m$

Space Complexity

1. Temp memory (work space) needed to solve an instance of the problem (excluding input size).
2. In many problems there is a time-space tradeoff.

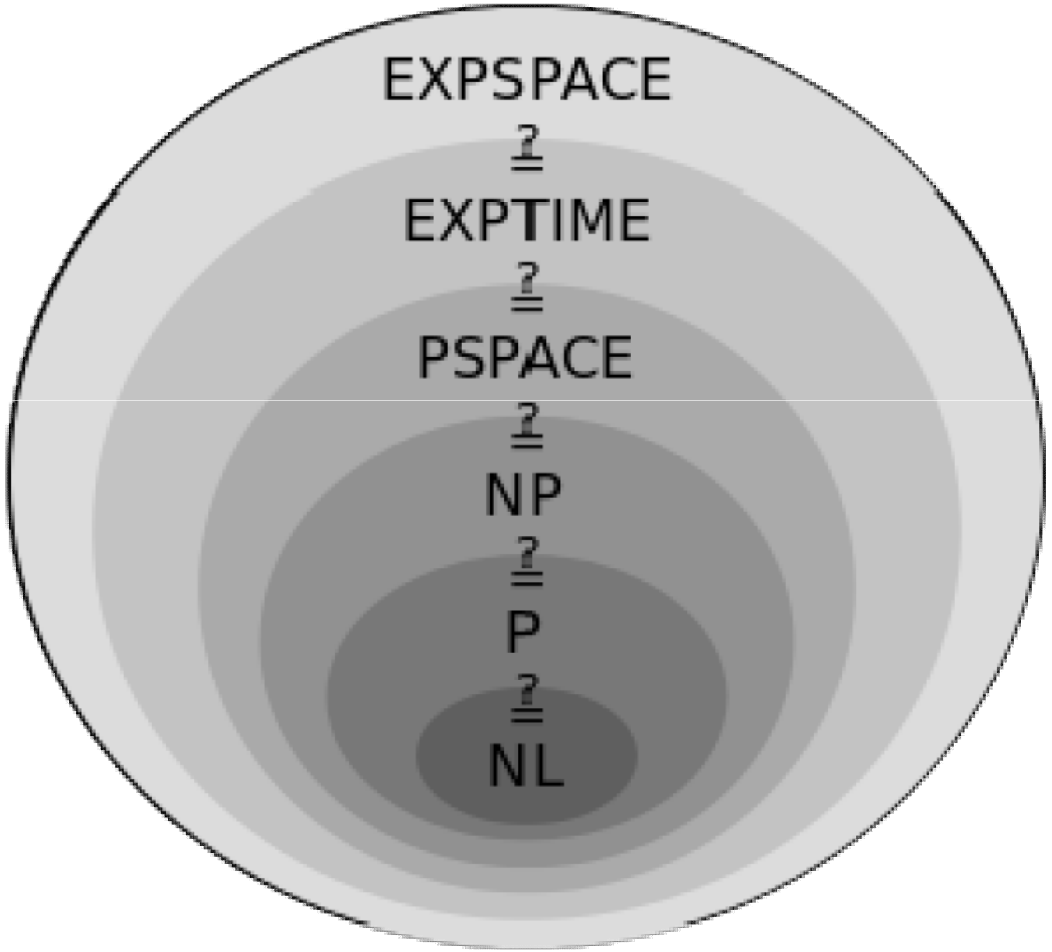
   The more space we allocate for the algorithm the faster it runs, and vice versa.
3. All previous asymptotic notation definitions also apply to space complexity.

Space time hierarchy

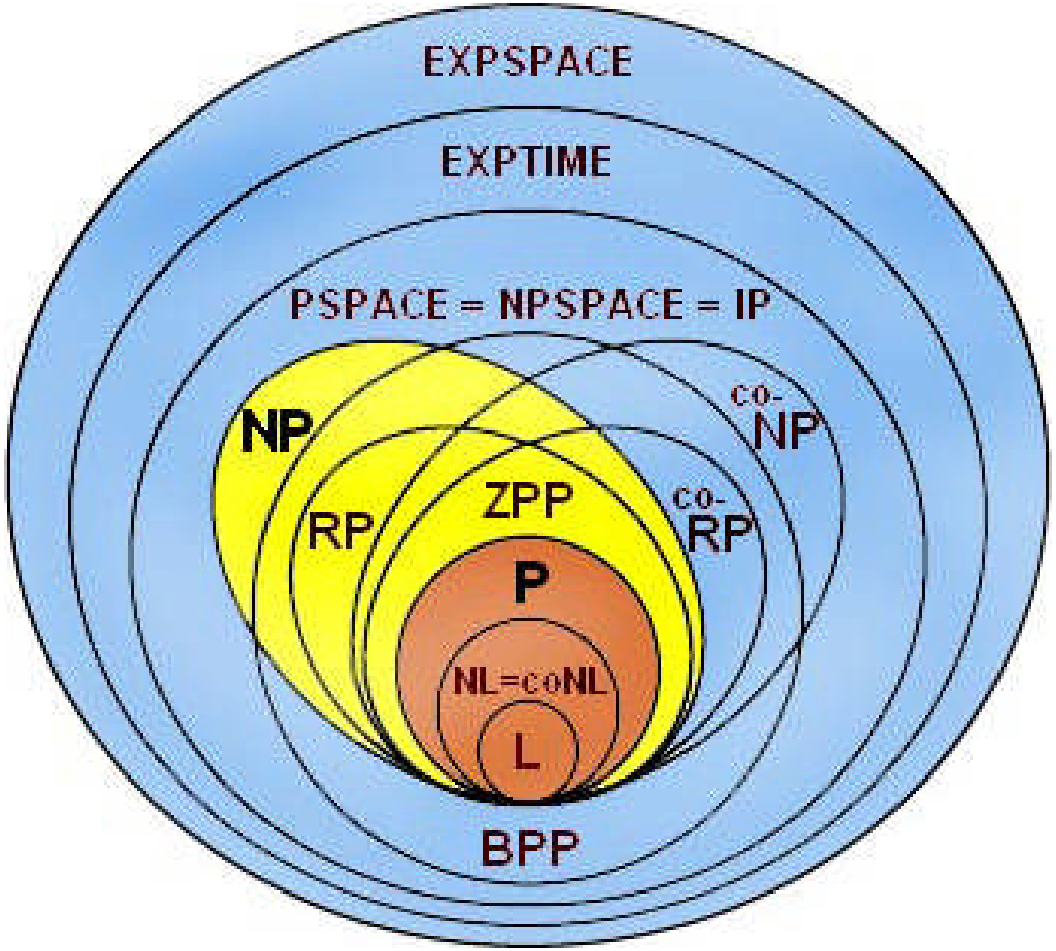
Complexity class	Model of computation	Resource constraint
DTIME($f(n)$)	Deterministic Turing machine	Time $f(n)$
P	Deterministic Turing machine	Time poly(n)
EXPTIME	Deterministic Turing machine	Time $2^{poly(n)}$
NTIME($f(n)$)	Non-deterministic Turing machine	Time $f(n)$
NP	Non-deterministic Turing machine	Time poly(n)
NEXPTIME	Non-deterministic Turing machine	Time $2^{poly(n)}$
DSPACE($f(n)$)	Deterministic Turing machine	**Space** $f(n)$
L	Deterministic Turing machine	**Space** O(**log** n)
PSPACE	Deterministic Turing machine	**Space** poly(n)
EXPSPACE	Deterministic Turing machine	**Space** $2^{poly(n)}$
NSPACE($f(n)$)	Non-deterministic Turing machine	**Space** $f(n)$
NL	Non-deterministic Turing machine	**Space** O(**log** n)
NPSPACE	Non-deterministic Turing machine	**Space** poly(n)
NEXPSPACE	Non-deterministic Turing machine	**Space** $2^{poly(n)}$

It turns out that PSPACE = NPSPACE and EXPSPACE = NEXPSPACE by Savitch's theorem.

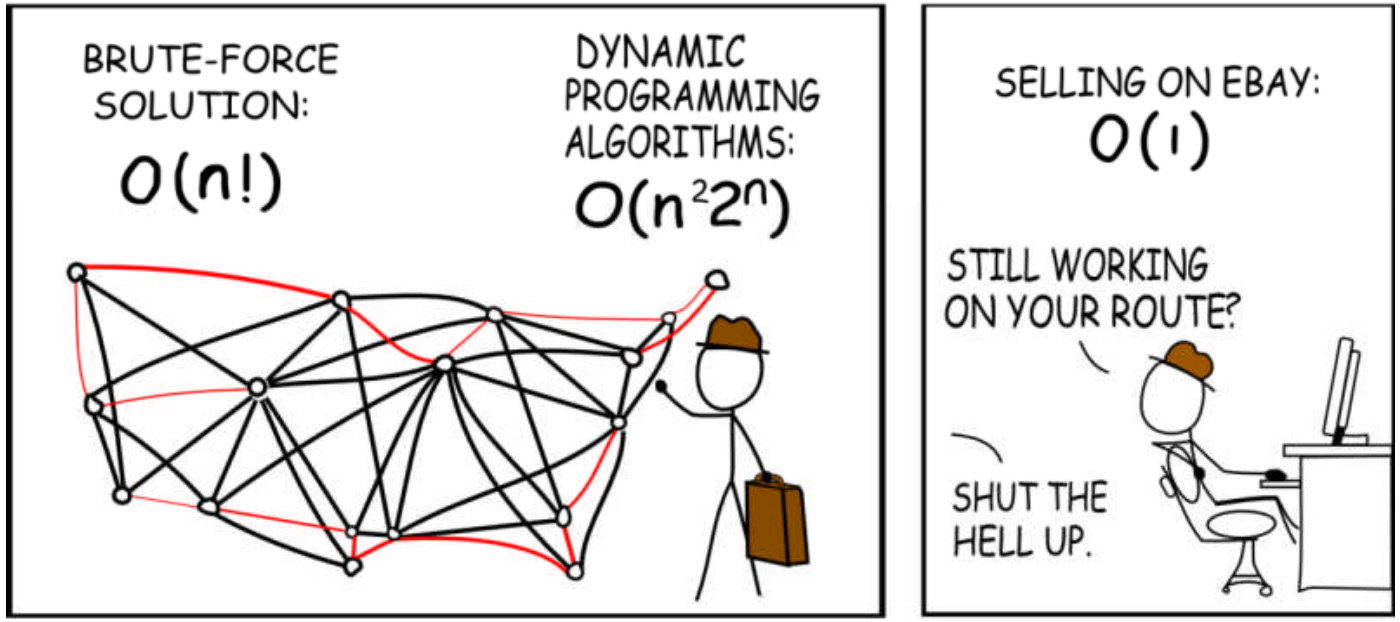
Space time hierarchy



Space time hierarchy



Complexity of Traveling salesman



xkcd.com

Basic math for algorithms

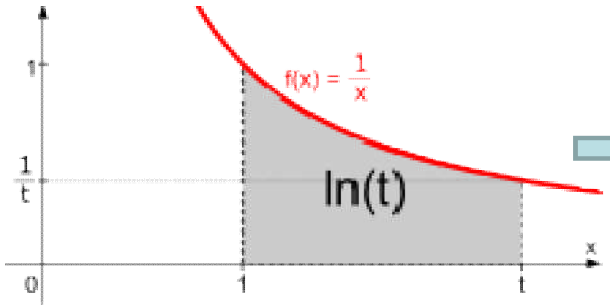
Logarithmic spiral

Logarithms – notation, simple facts

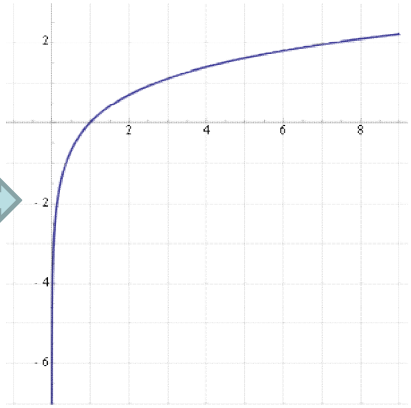
1. $\lg n = \log_2 n$ (logarithm of base 2)
2. $\ln n = \log_e n$ (natural logarithm to base e=2.7182818284590451)
3. $\log n = \log_{10} n$ (common logarithm to base 10)

$\lg^k n = (\lg n)^k$ *(notation)*

$\lg \lg n = \lg (\lg n)$ .. right associative.



Area under 1/x is log



log grows very slowly, measures number of digits in n.

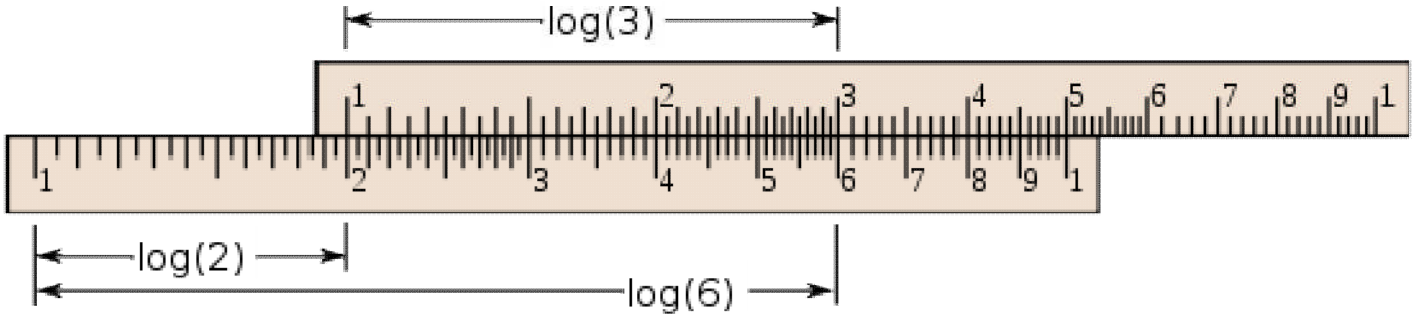
Logarithms – notation, simple facts

$a = b^{\log_b a}$

$\log_c (a*b) = \log_c a + \log_c b$ *.. Mult is addition of log*

$\log_c a^n = n * \log_c a$ .. Exp is mult of log

$\log_b (1/a) = -\log_b a$ *.. Reciprocal is negation of log*



1. Slide rule multiplies numbers by adding their logs, e.g. log(2)+log(3)=log(2*3)
2. Log tables are used for: 2*3 = antilog(log(2)+log(3)

Changing base of log

- $\log_a n$
- $= \log_b n * \log_a b$
- $= \log_b n / \log_b a$
- $\log_a b$ is a constant, when a, b are constants.
- $\log_{10} n = \ln n / \ln 10 = \ln n * 0.4343$

```
C:\> c:\python33\python.exe
>>> from numpy import *
>>> log10(e)
0.43429448190325182
```

Iterated log star* function

(very slowly growing function)

Defn: lg*n = number of times lg function must be applied to make n ≤ 1.

Table showing range of x, and the corresponding value of lg*(x), it is a very very slowly growing function

x	$\lg^* x$
$(-\infty, 1]$	0
$(1, 2]$	1
$(2, 4]$	2
$(4, 16]$	3
$(16, 65536]$	4
$(65536, 2^{65536}]$	5

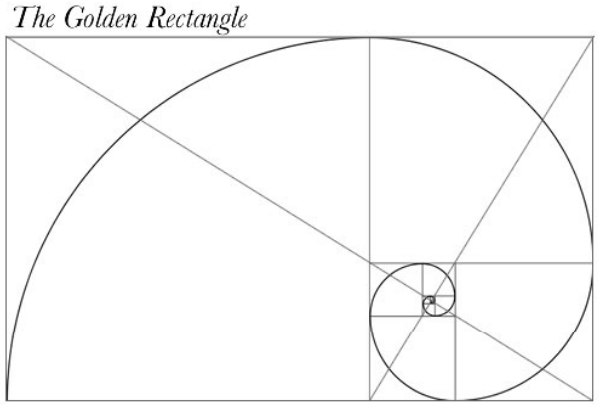
Floor and ceiling

- Floor(x) = int(x) = ⌊x⌋
- Ceiling(x)= int(x+0.5)=⌈x⌉
- Eg. ⌊2.5⌋ = 2 and ⌈2.5⌉ = 3
- Mod(x,y) = remainder(x/y) = x % y
- E.g. 7 mod 4 = 3, $7 \equiv_4 3$, $7 \equiv 3 \pmod 4$
- -2 mod 3 = 1, 7 mod 0 = undefined.
- x div y = int(x/y), e.g. 7 div 3 = 2,
- x div 0 = undefined
- x mod y = x – (x div y) * y

Exponents

- $A^0 = 1$
- $A^1 = A$
- $A^{-1} = 1/A$
- (A^m)^n = A^(mn) != A^(m^n) [non-associative]
- A^m*A^n = A^(m+n)
- e^(i*pi) = -1 .. from complex numbers.
- n! = n*(n-1)*…*1 ~= n^n

Fibonacci sequence



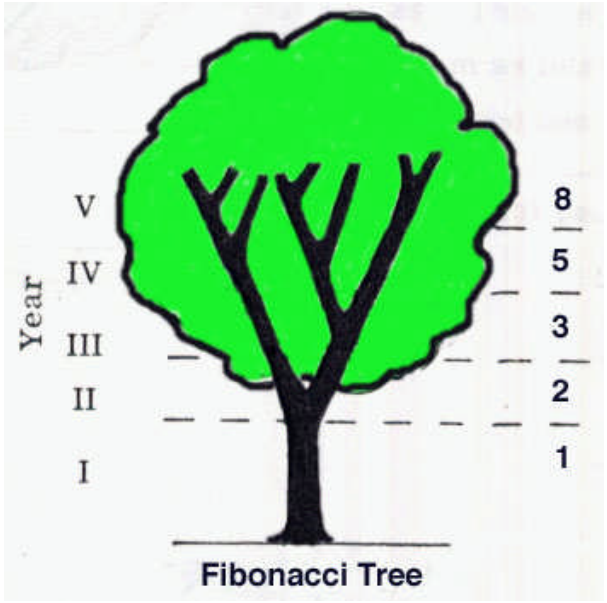
- F(0) = 0, F(1)=1,
- F(n)=F(n-1)+F(n-2)
- 0,1,1,2,3,5,8,13,21,34,55
- Golden ratio:

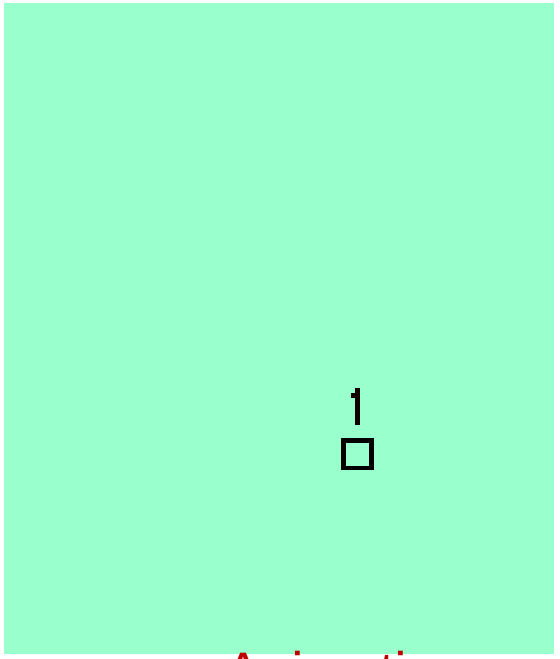
  phi = (1 + sqrt(5))/2 = 1.618

  phi' = (1 - sqrt(5))/2 = 0.618

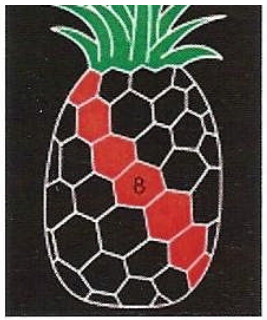
  $F(n) = (phi^n - phi'^n) / sqrt(5) \sim= phi^n / sqrt(5)$
- Fibonacci has exponential growth

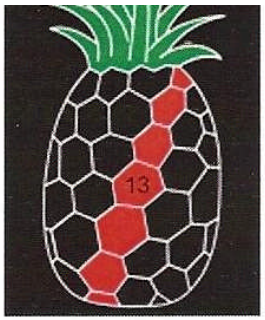
Fibonacci numbers in nature





Animation





References

1. Math for CS pdf, by Lehman, MIT.
2. Cormen 3^e.
3. Wikipedia.

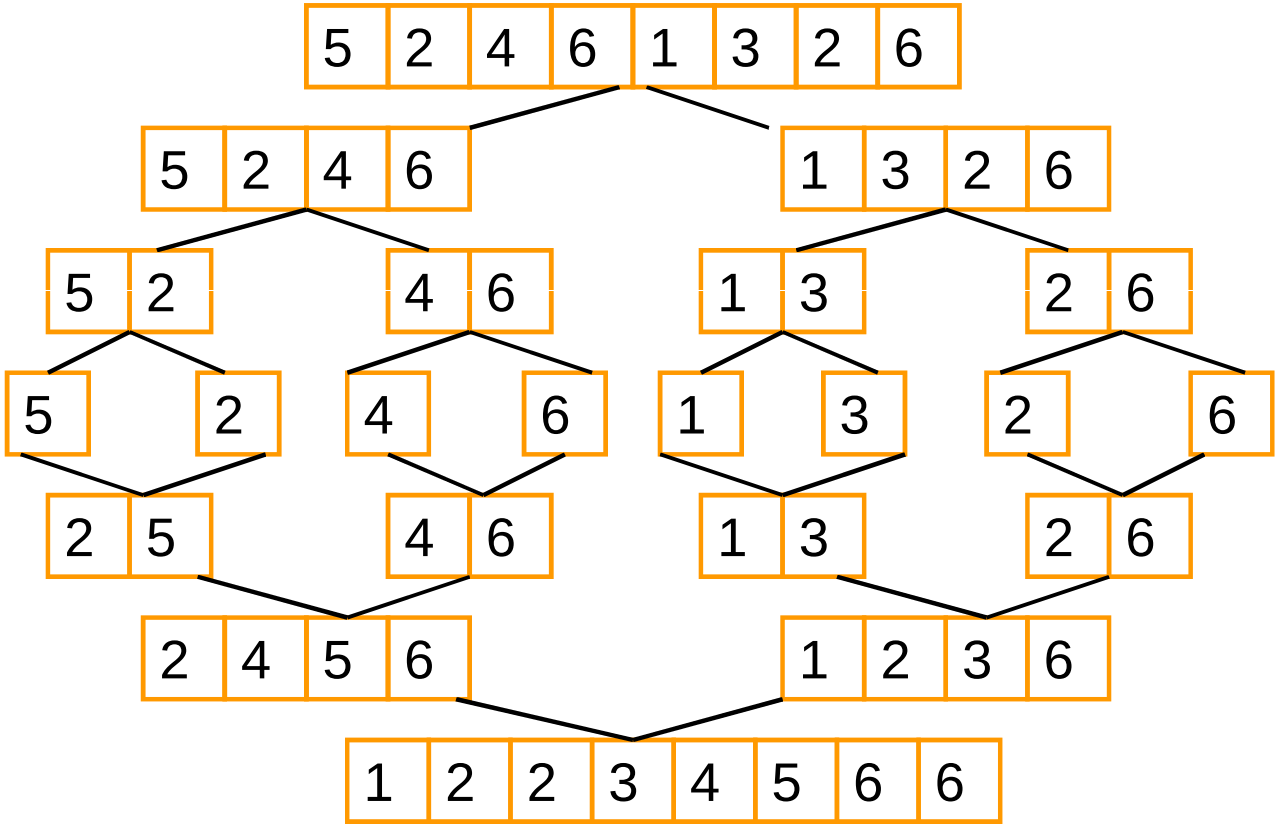
Master Method

Solving Recurrences in
Complexity of Algorithms

Example: Mergesort

- DIVIDE the input sequence in half
- RECURSIVELY sort the two halves
  - basis of the recursion is sequence with 1 key
- COMBINE the two sorted subsequences by merging them

Mergesort Example



Recurrence Relation for Mergesort

- Let $T(n)$ be worst case time on a sequence of n keys
- If $n = 1$, then $T(n) = \Theta(1)$ (constant)
- If $n > 1$, then $T(n) = 2\,T(n/2) + \Theta(n)$
  - two subproblems of size $n/2$ each that are solved recursively
  - $\Theta(n)$ time to do the merge

Recurrence Relations

How To Solve Recurrences

- Ad hoc method:
  - expand several times
  - guess the pattern
  - can verify with proof by induction
- Master theorem
  - general formula that works if recurrence has the form *T(n)*

    = $aT(n/b) + f(n)$
    - *a* is number of sub-problems
    - *n / b* is size of each subproblem
    - *f(n)* is cost of non-recursive part

Master Theorem

Consider a recurrence of the form

$$T(n) = a\,T(n / b) + f(n)$$

with $a \geq 1$, $b > 1$, and $f(n)$ eventually positive.

a) If $f(n) = O(n^{\log_b(a)-\varepsilon})$, then $T(n)=\Theta(n^{\log_b(a)})$.

b) If $f(n) = \Theta(n^{\log_b(a)})$, then $T(n)=\Theta(n^{\log_b(a)} * \log(n))$.

c) If $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ and f(n) is regular, then $T(n)=\Theta(f(n))$

Defn: f(n) is regular iff eventually $a*f(n/b) \leq c*f(n)$ for some constant $c<1$

Master method details

Essentially, the Master theorem compares the function f(n) with $g(n) = n^{\log_b(a)}$.

Roughly, the theorem says:

a) If f(n) « g(n) then $T(n)=\Theta(g(n))$.

b) If f(n) ≈ g(n) then $T(n)=\Theta(g(n) \log(n))$.

c) If f(n) » g(n) then $T(n)=\Theta(f(n))$.

Where: $T(n) = a\, T(n / b) + f(n)$

Master Theorem

Theorem 4.1 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Master method details (CLR)

Before applying the master theorem to some examples, let's spend a moment trying to understand what it says. In each of the three cases, we compare the function $f(n)$ with the function $n^{\log_b a}$. Intuitively, the larger of the two functions determines the solution to the recurrence. If, as in case 1, the function $n^{\log_b a}$ is the larger, then the solution is $T(n) = \Theta(n^{\log_b a})$. If, as in case 3, the function $f(n)$ is the larger, then the solution is $T(n) = \Theta(f(n))$. If, as in case 2, the two functions are the same size, we multiply by a logarithmic factor, and the solution is $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$.

Beyond this intuition, you need to be aware of some technicalities. In the first case, not only must $f(n)$ be smaller than $n^{\log_b a}$, it must be *polynomially* smaller.

Nothing is perfect…

The Master theorem does not cover all possible cases. For example, if

$$f(n) = \Theta(n^{\log_b(a)} \log n),$$

then we lie between cases 2) and 3), but the theorem does not apply.

There exist better versions of the Master theorem that cover more cases, but these are even harder to memorize.

Idea of the Proof

Let us iteratively substitute the recurrence:

$$
\begin{aligned}
T(n) &= aT(n/b) + f(n) \\
&= a(aT(n/b^2)) + f(n/b)) + bn \\
&= a^2T(n/b^2) + af(n/b) + f(n) \\
&= a^3T(n/b^3) + a^2 f(n/b^2) + af(n/b) + f(n) \\
&= \dots \\
&= a^{\log_b n}T(1) + \sum_{i=0}^{(\log_b n)-1} a^i f(n/b^i) \\
&= n^{\log_b a}T(1) + \sum_{i=0}^{(\log_b n)-1} a^i f(n/b^i)
\end{aligned}
$$

Idea of the Proof

Thus, we obtained

$$T(n) = n^{\log_b(a)}\, T(1) + \sum a^i\, f(n/b^i)$$

The proof proceeds by distinguishing three cases:

1) The first term in dominant: $f(n) = O(n^{\log_b(a)-\varepsilon})$
2) Each part of the summation is equally dominant: $f(n) = \Theta(n^{\log_b(a)})$
3) The summation can be bounded by a geometric series: $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ and the regularity of 'f' is key to make the argument work.

Example 2. Strassen's Matrix Multiplication

- Strassen found a way to get all the required information with only 7 matrix multiplications, instead of 8, using divide and conquer.
- Recurrence for new algorithm is
  - $T(n) = 7T(n/2) + \Theta(n^2)$

Solving the Recurrence Relation

Applying the Master Theorem to

$T(n) = a\,T(n/b) + f(n) = 7T(n/2) + \Theta(n^2)$

We have: a=7, b=2, and $f(n)=\Theta(n^2)$.

Since $f(n) = O(n^{\log_b(a)-\varepsilon}) = O(n^{\log_2(7)-\varepsilon})$,

case 'a' applies and we get

$T(n)= \Theta(n^{\log_b(a)}) = \Theta(n^{\log_2(7)}) = O(n^{2.81})$.

Example 3

- $T(n) = 9\ T(n/3)+n$,
- We have $a=9$, $b=3$, $f(n)=n$
- $n^{(\log_b a)}=n^{(\log_3 9)} =n^2$
- $f(n)=n= n^1=n^{(2-e)}= n^{((\log_b a)-e)}$, $e=1$
- Case 'a' of master theorem
- $T(n) = \Theta(n^2)$.

Example 3

- T(n)=T(2*n/3)+1,
- with a=1, b=3/2, f(n)=1
- $n^{(\log_b a)}=n^{(\log_{3/2} 1)} =n^0=1$
- Case '2' $f(n)= \Theta(n^{(\log_b a)})= \Theta(1)$
- Hence $T(n)= \Theta(\lg n)$

Example 4

- $T(n)=3T(n/4)+n \lg n$,
  with $a=3$, $b=4$, $f(n)=n \lg n$.
- $n^{(\log_b a)}=n^{(\log_4 3)} = O(n^{0.793})$
- $f(n)= \Omega(n^{(\log_4 3+e)})$, $e \sim 0.2$
- For suff. large n,
  $3(n/4)*\lg(n/4) <= \frac{3}{4}*n*\lg(n)=c\, f(n)$,
- With $c=3/4$, by case '3' $T(n)= \Theta(n \lg n)$

Example 5

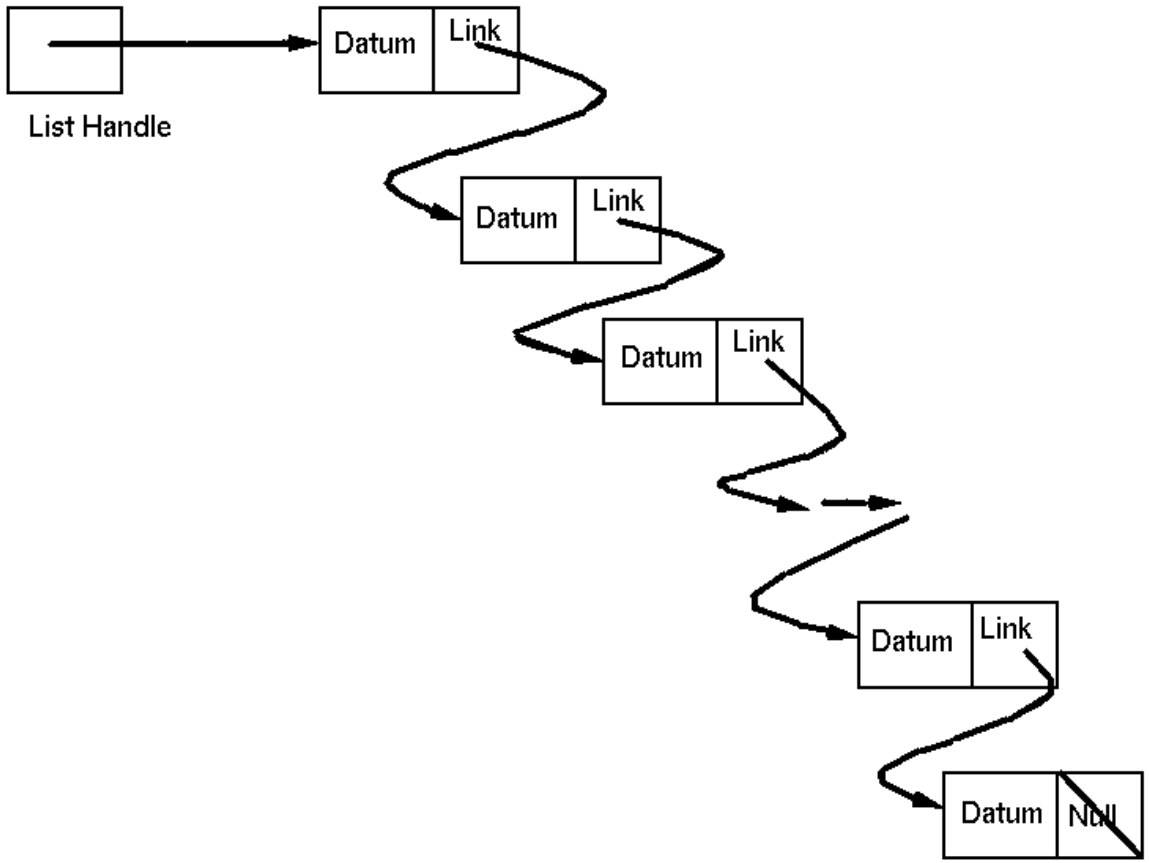
- T(n)=2T(n/2)+n lg n,

  with a=2, b=2, f(n)=n lg n.
- $n^{(\log_b a)}=n^{(\log_2 2)} =n^1=n$
- f(n)= n lg n, but case 3 does NOT apply, since n lg(n) is not polynomially larger than n
- $f(n)/n^{(\log_b a)}=n \lg n / n = \lg n$ is less than n^e, for all constant e.

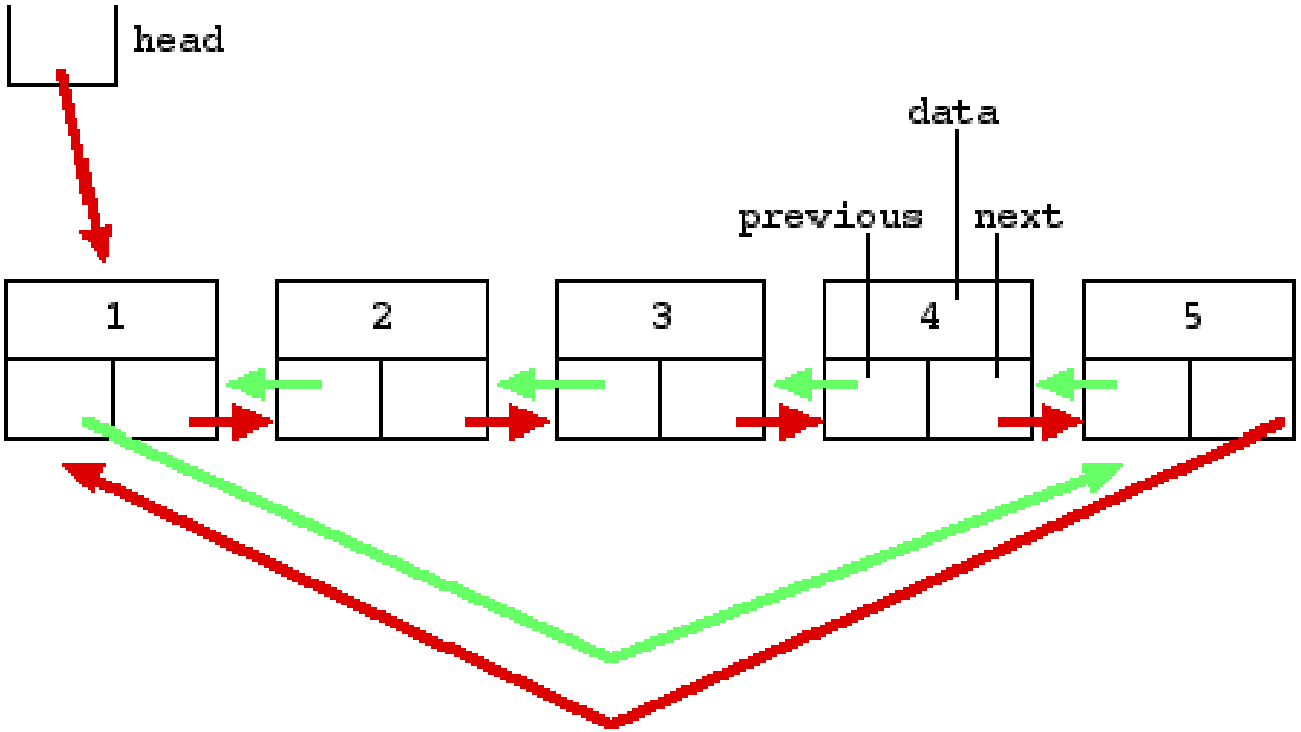
Simple Data Structures

Linked lists, Stacks, Queues

Linked list


List Handle
Datum
Link
Datum
Link
Datum
Link
Datum
Link
Datum
Null

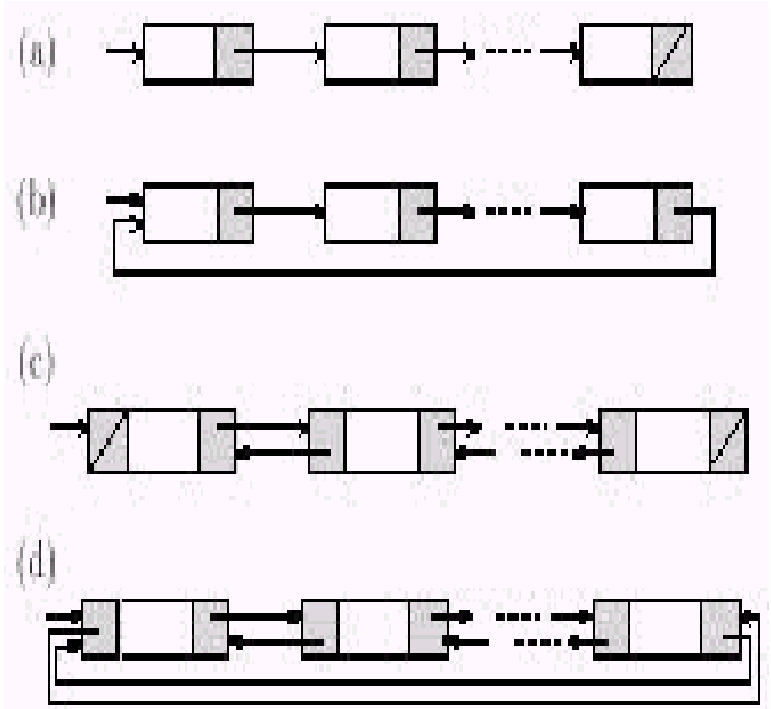
Circular doubly linked list



Properties of linked lists

1. Advantages of LL
2. Disadvantages of LL

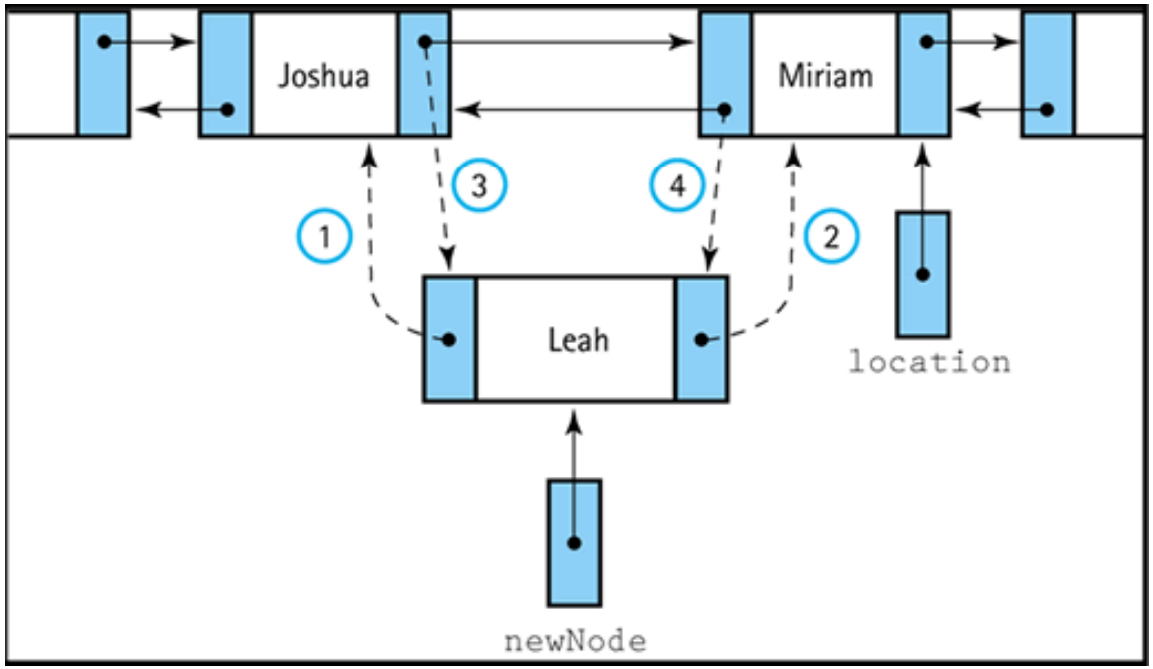
a) Single linked list
b) Circular singly linked list
c) Doubly linked list
d) Doubly circular linked list



Linked Lists

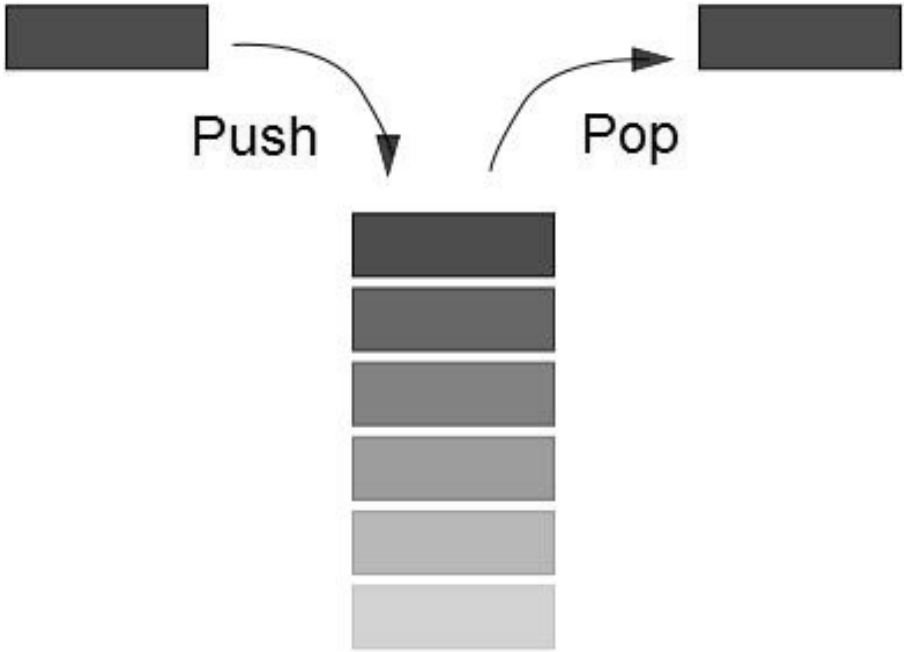
- A linked list consists of a finite sequence of elements or nodes that contain information plus (except possibly the last one) a link to another node.
- If node *x* points to node *y*, then *x* is called the *predecessor* of *y* and *y* the *successor* of *x*. There is a link to the first element called the *head* of the list.
- If there is a link from the last element to the first, the list is called *circular*.
- If in a linked list each node (except possibly the first one) points also to its *predecessor*, then the list is called a *doubly linked list*.
- If the first and the last nodes are also connected by a pair of links, then we have a *circular doubly linked list*.

The two primary operations on linked lists are insertion and deletion.
Insertion in doubly linked list

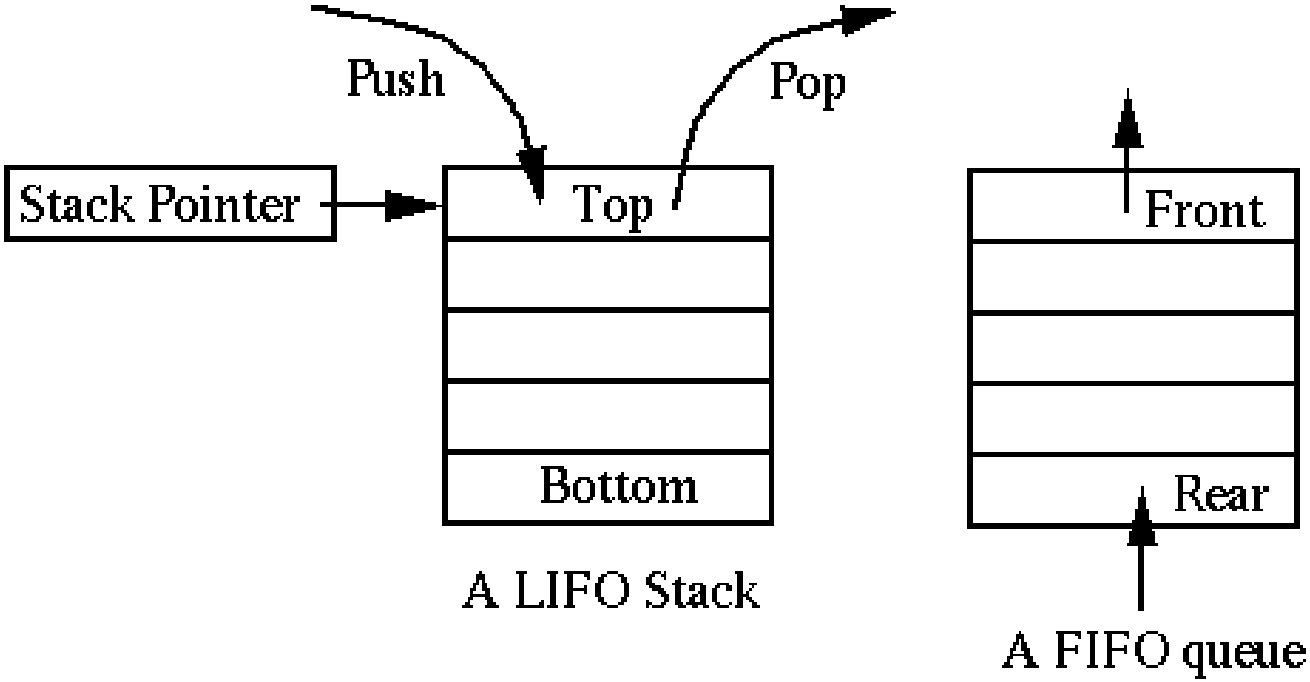


Stack

Stack of books



Stack pointer push and pop

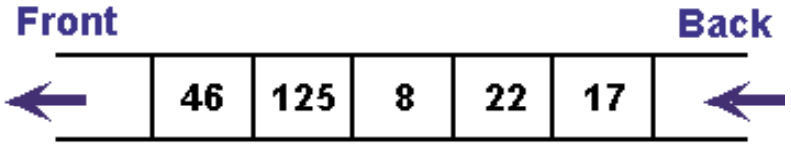


Stack

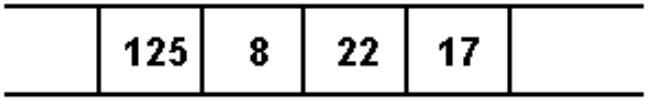
- A *stack* is a linked list in which insertions and deletions are permitted only at one end, called the *top* of the stack. The stack pointer (sp) points to the top of the stack.
- *stack* supports two basic operations:
  - *pop(S)* returns the top of the stack and removes it from S.
  - *push(S, x)* adds *x* to top of *S* and updates the sp to point to *x*.

Queues

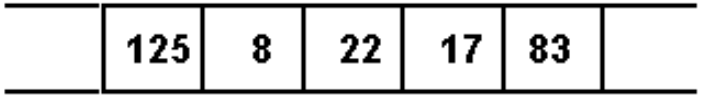
In a queue, all operations take place at one end of the queue or the other. The "enqueue" operation adds an item to the "back" of the queue. The "dequeue" operation removes the item at the "front" of the queue and returns it.



Items enter queue at back and leave from front.



After dequeue()

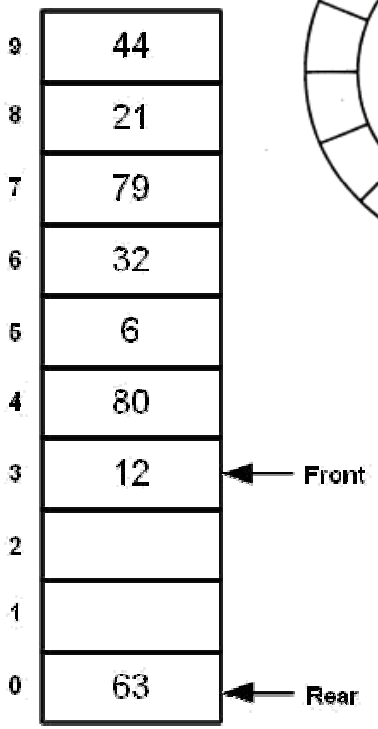


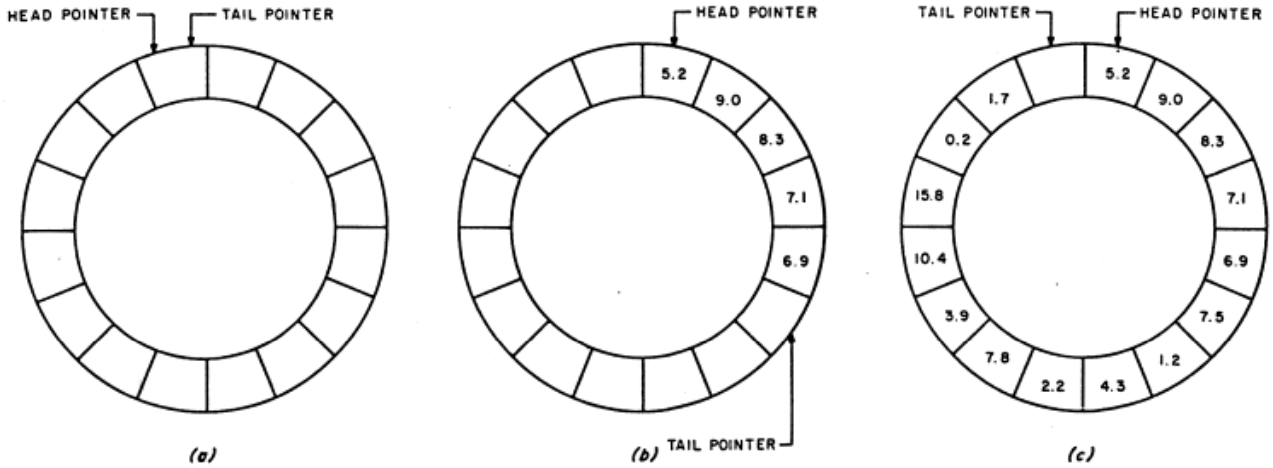
After enqueue(83)

Queue

- A *queue* is a list in which insertions are permitted only at one end of the list called its *rear,* and all deletions are from the other end called the *front* of the queue.
- The operations supported by queues are
  - *Add(Q,x)* adds x at the rear of the queue Q.
  - *Delete(Q,x)* delete x from the front of the queue Q.

Circular queue





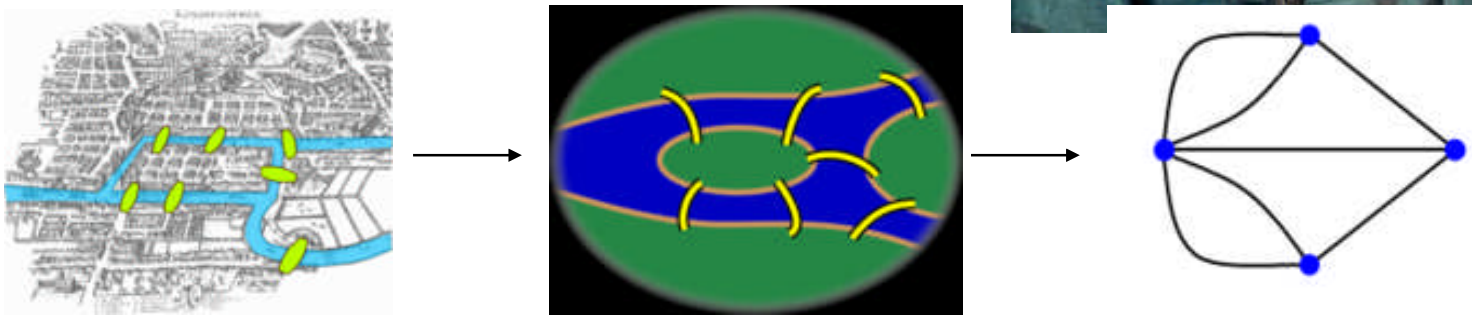
Graphs

Data Structure

Graph Theory - History

Leonhard Euler's paper on "*Seven Bridges of Königsberg*" ,

published in 1736.



Famous problems

- Euler circuit: travel on every edge once
- Hamiltonian path: visit every vertex once
  - NP complete
- Traveling salesman problem
  - Shortest hamiltonian path (Visit every vertex once and travel least amount of edges).
- Graph Isomorphism: Are two graphs identical?
- Graph coloring: Color the vertices, so that adjacent vertices have different colors
- 4 Color problem: Every map can be coloured with just 4 colours, and adjacent countries have different colors

4 color problem

In 1852 Francis Guthrie posed the “four color problem” which asks if it is possible to color, using only four colors, any map of countries in such a way as to prevent two bordering countries from having the same color.

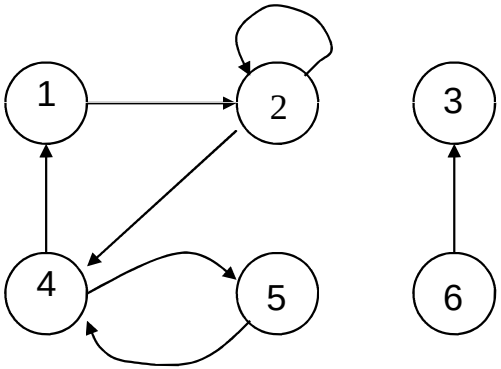
Proved in 1976 by Kenneth Appel and Wolfgang Haken, with heavy use of computers.

Uses

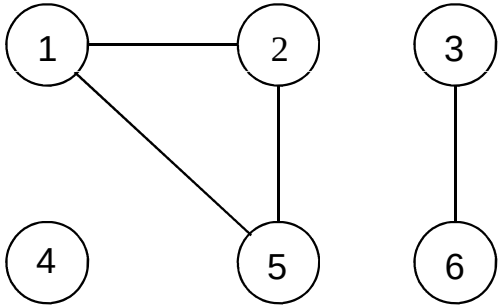
- Transportation: travel and routing
- Arranging wires on circuit boards
- Maximizing flow of liquids in network.
- Routing network packets
- Any more?

What is a Graph?

- G*raph* is a set of vertices (nodes) joined by a set of edges (lines or arrows).



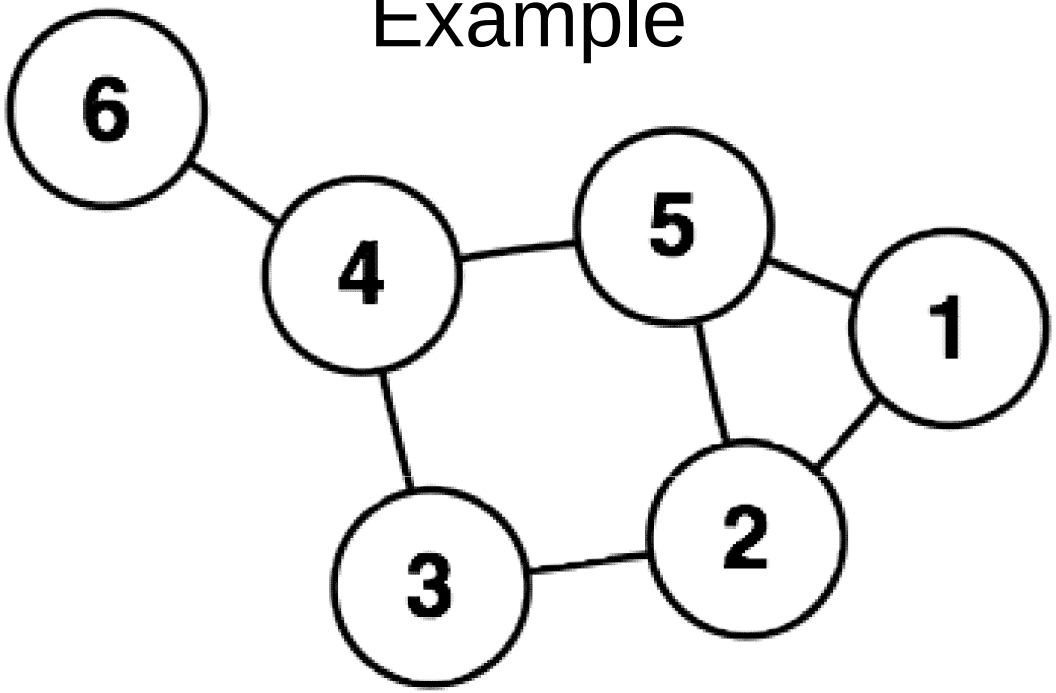
Directed graph



Graph

- **Graph.** A graph *G=(V ,E)* is
- Vertices*V* (nodes) and Edges (links) *E*.
- Edge e connects two nodes
- Edge e can have a weight (distance), or 1.
- Directed Graph if edges have direction.
- Size of Graph is measured by size of V and E

Example



- V:={1,2,3,4,5,6}
- E:={{1,2},{1,5},{2,3},{2,5},{3,4},{4,5},{4,6}}

Vertex degree

- ***Vertex degree*** **is**

  Undirected G:

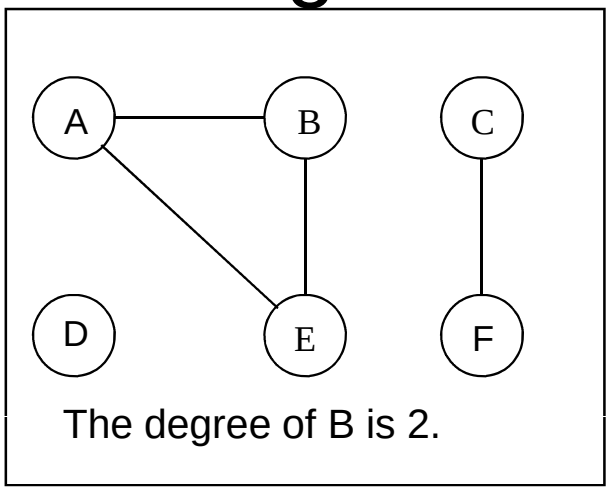
  number of vertices adjacent to the vertex**.**

  Directed G:

  In-degree*:* number of edges directed to *vi*

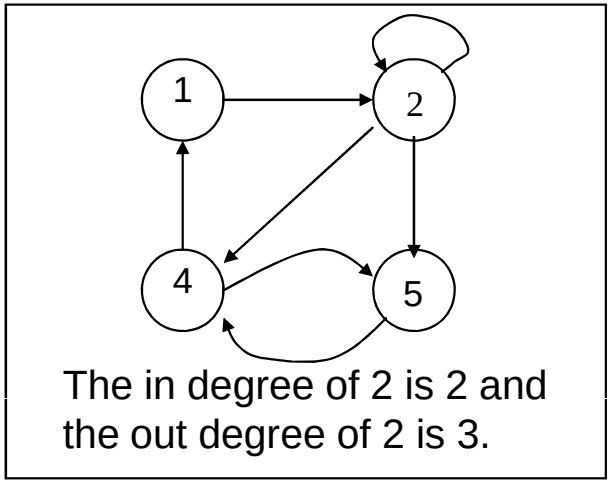
  Out-degree: number of edges directed out of to *vi*

Degree



- Number of edges incident on a node

Degree (Directed Graphs)



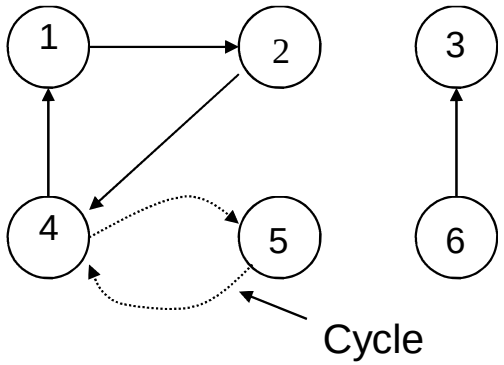
- In degree: Number of edges entering
- Out degree: Number of edges leaving

- Degree = indegree + outdegree

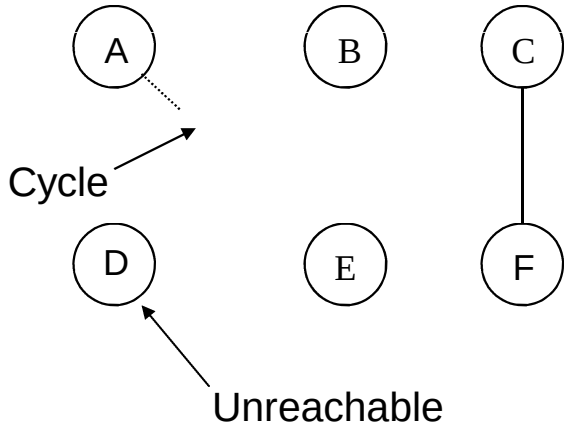
- **Path** ***in G from vertex v_1 to vertex v_k is a*** *sequence of vertices <v_1, v_2, …,v_k> with k-1 edges between adjacent v_i and v_i+1.*
  - ***length*** of a path is the number of edges in the path.
  - ***Simple path*** **if all its vertices are distinct.**
  - ***Cycle in a path*** **if *v*1 = *v*k.**

Path

- A *path* is a sequence of vertices such that there is an edge from each vertex to its successor.
- A path is ***simple*** if each vertex is distinct.



Simple path from 1 to 5
= [1, 2, 4, 5]
Our text's alternates the vertices and edges.



If there is path *p* from *u* to *v* then we say *v* is **reachable** from *u* via *p*.

Connectivity

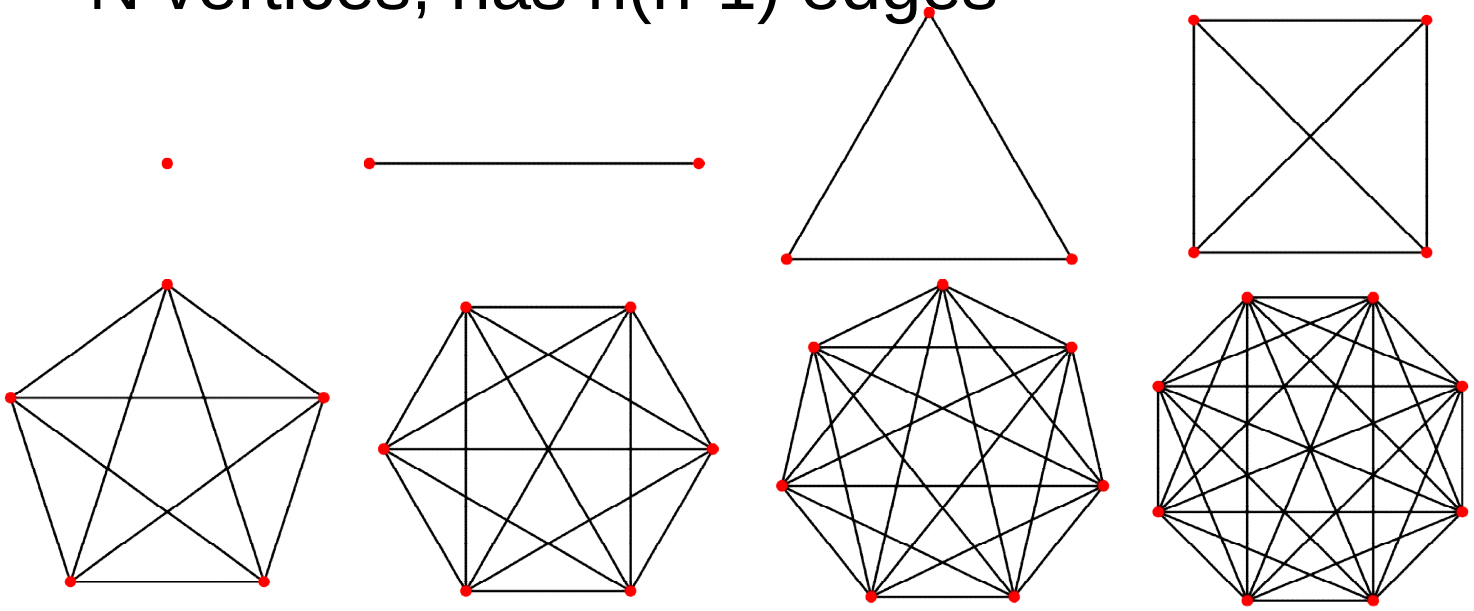
- G is ***connected*** if
  - you can get from any node to any other by following a sequence of edges, i.e.
  - any two nodes are connected by a path.
- ***G is strongly connected*** if
  - there is a *directed* path from any node to any other node.

Sparse/Dense Graphs

- A graph is ***sparse*** if $| E | \approx | V |$
- A graph is ***dense*** if $| E | \approx | V |^{2}$.

Complete Graph

- Denoted K_n
- Every pair of vertices are adjacent
- N vertices, has n(n-1) edges



Bipartite graph

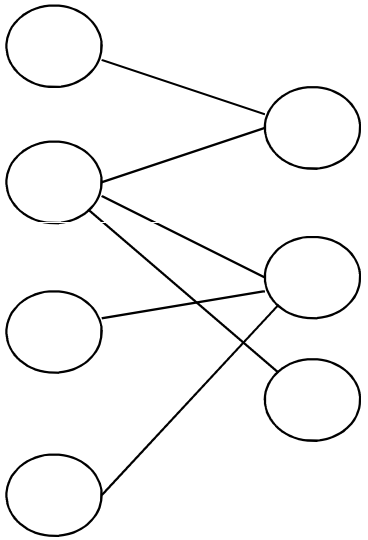
- V can be partitioned into 2 sets V_1 and V_2 such that $(u,v) \in E$ implies

  either

  $u \in V_1$ and $v \in V_2$

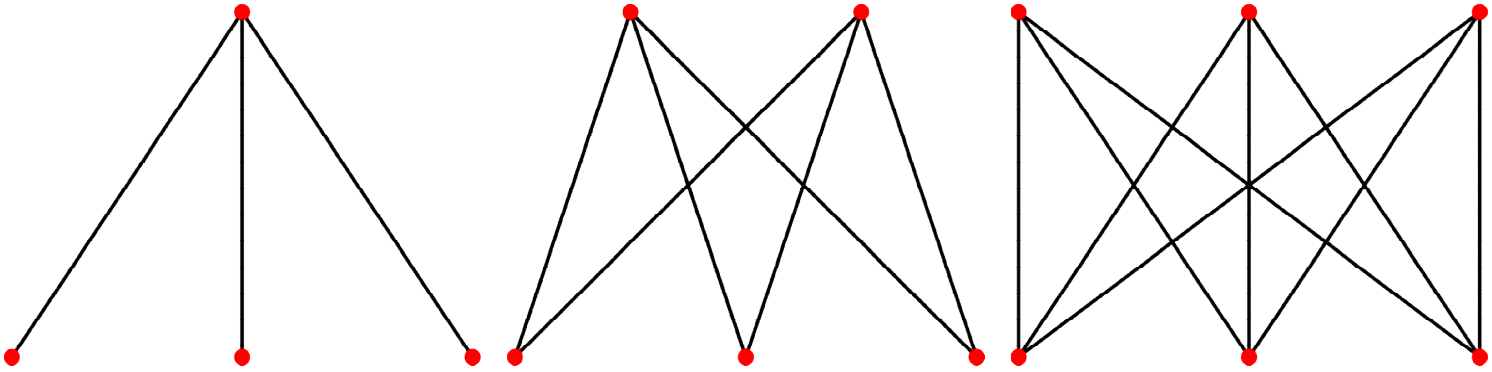
  OR

  $v \in V_1$ and $u \in V_2$.



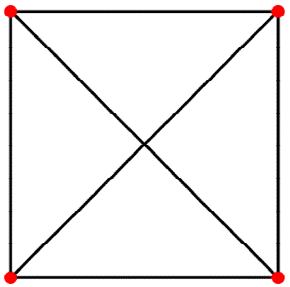
Complete Bipartite Graph

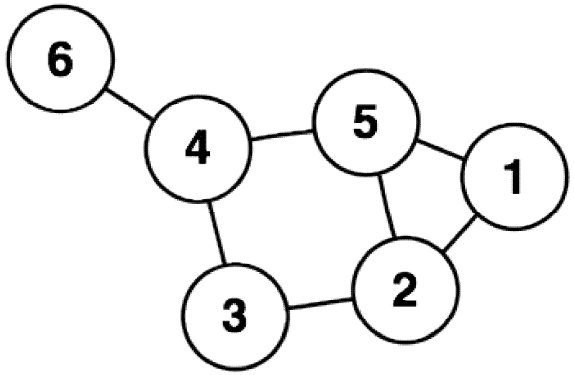
- Bipartite Variation of Complete Graph
- Every node of one set is connected to every other node on the other set



Planar Graph

K4





- Can be drawn on a plane such that no two edges intersect
- K_4 is the largest complete graph that is planar
- K5, K6, .. are not planar

Isomorphism Problem

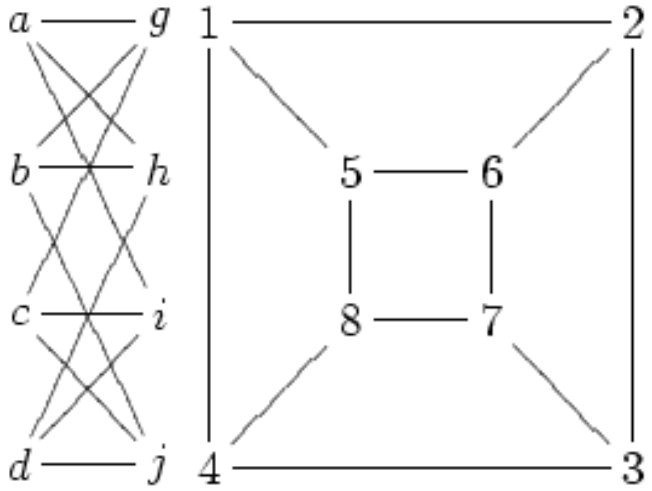
- Determining whether two graphs are isomorphic
- Although these graphs look very different, they are isomorphic
- One isomorphism between them is:

  f(a) = 1 f(b) = 6

  f(c) = 8 f(d) = 3

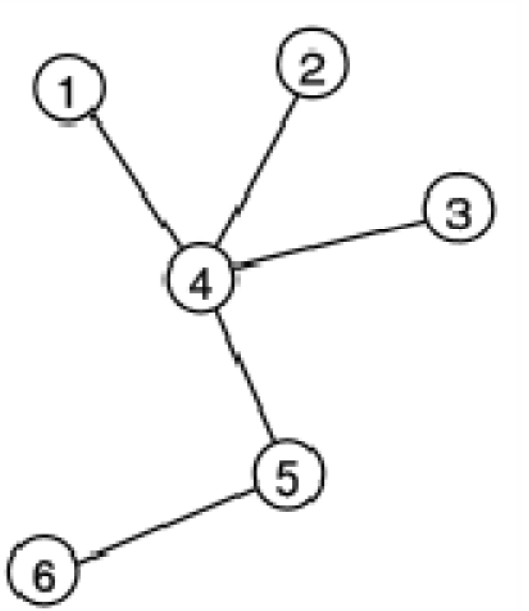
  f(g) = 5 f(h) = 2

  f(i) = 4 f(j) = 7



Tree

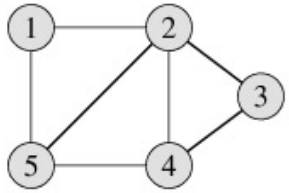
- Connected Acyclic Graph
- Two nodes have *exactly* one path between them



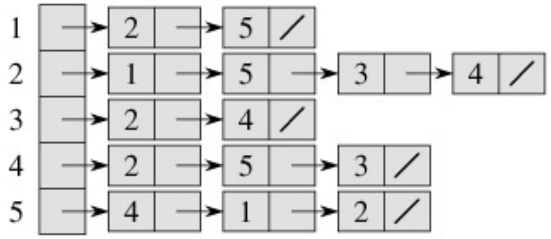
Graph Data Structures

Representations of graphs

Adjacency list adjacency matrix



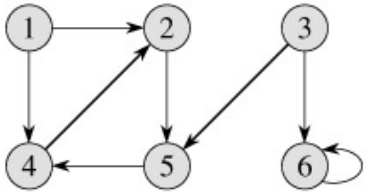
(a)



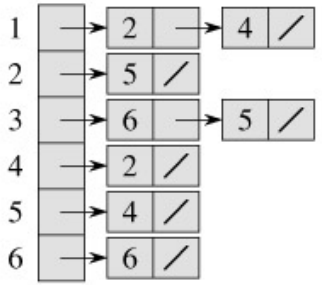
(b)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)



(a)



(b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

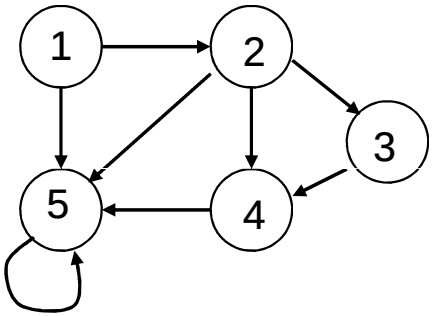
Adjacency lists

- $G = (V, E)$ an array of adjacencies: for each vertex $v \in V$, store a list of vertices adjacent to v
- *Storage requirements:*
  - directed graph, the sum of the lengths of all the adjacency lists is $|E|$,
  - undirected graph, the sum of the lengths of all the adjacency lists is $2\,|E|$,
- *Storage requirements: For both directed and undirected graphs*, amount of memory it requires is $\Theta(|V| + |E|)$.

Adjacency lists

- Advantage:
  - Saves space for sparse graphs. Most graphs are sparse.
  - Traverse all the edges that start at v, in θ(degree(v))
- Disadvantage:
  - Check for existence of an edge (v, u) in worst case time θ(degree(v))

Problem:
Show the adjacency-list of



1

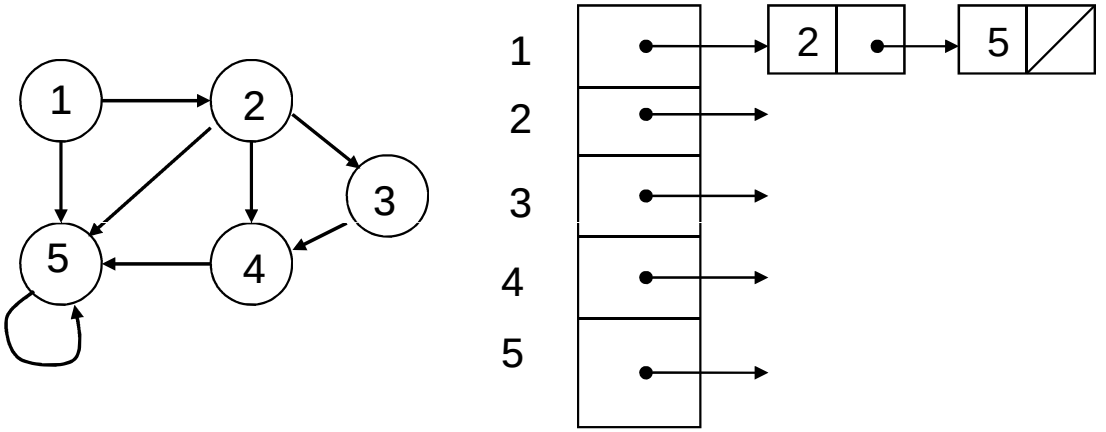
2

3

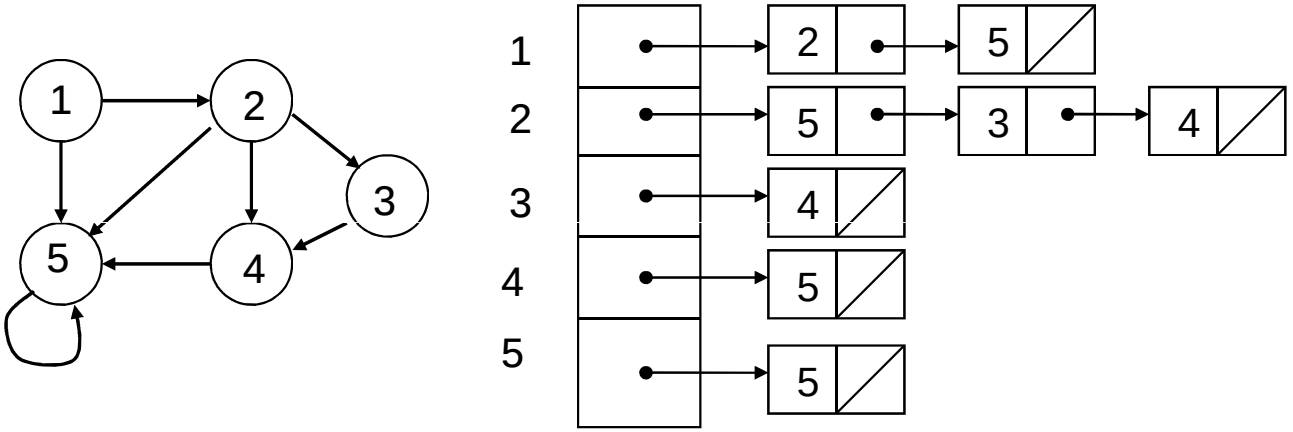
4

5

Problem: adjacency-list 2



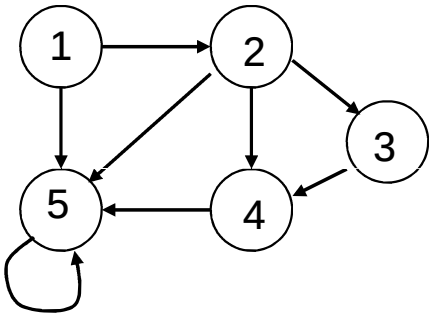
Problem: adjacency-list 3



Adjacency matrix

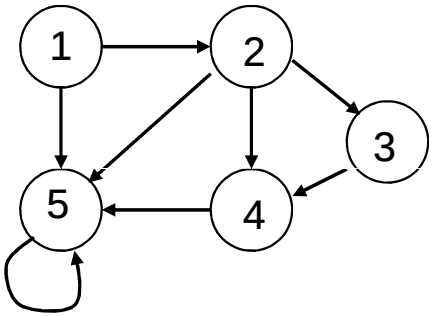
- Represents the graph as a *n* x *n* matrix A:
  - A[*i*, *j*] = 1 if edge (*i*, *j*) $\in$ E (*or weight of edge)*
    = 0 if edge (*i*, *j*) $\notin$ E
- Storage requirements: $O(V^2)$
- Efficient for small graphs
  - Especially if store just one bit/edge
  - Undirected graph: only need one diagonal of matrix

Problem: ***Adjacency matrix***



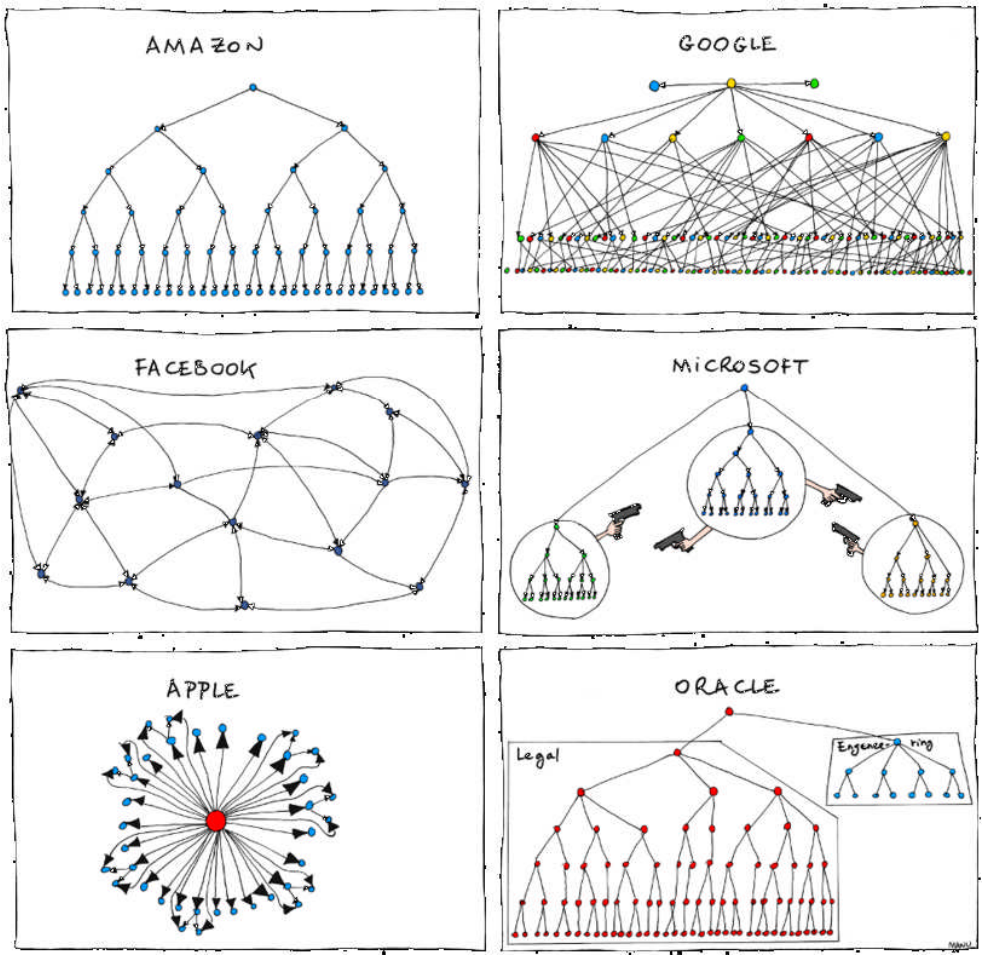
	1	2	3	4	5
1	x	x	x	x	x
2	x	x	x	x	x
3	x	x	x	x	x
4	x	x	x	x	x
5	x	x	x	x	x

Adjacency matrix 2



	1	2	3	4	5
1	0	1	0	0	1
2	0	0	1	1	1
3	0	0	0	1	0
4	0	0	0	0	1
5	0	0	0	0	1

Large graphs



Graph Problems

- Bipartite, matchings
- Shortest path
- Max flow
- Hamitonian path
- TSP
- Vertex cover
- Colouring
- Isomorphism

Searching algorithms

Linear and Binary Search

Searching algorithms

- Given a list, find a specific element in the list
- We will see two types
  - Linear search (a.k.a. sequential search)
  - Binary search

Linear search in unsorted list

- Given a list, find a specific element in the list.

procedure linear_search (x: int; a_1, a_2, …, a_n: int)
i := 1
while ($i \leq n$ and $x \neq a_i$)
  i := i + 1
if $i \leq n$ **then** *found* := i
else *found* := 0

Algorithm 2: Linear search, take 1

procedure linear_search (x: integer; a_1, a_2, …, a_n: integers)

$i := 1$

while ($i \leq n$ and $x \neq a_i$)

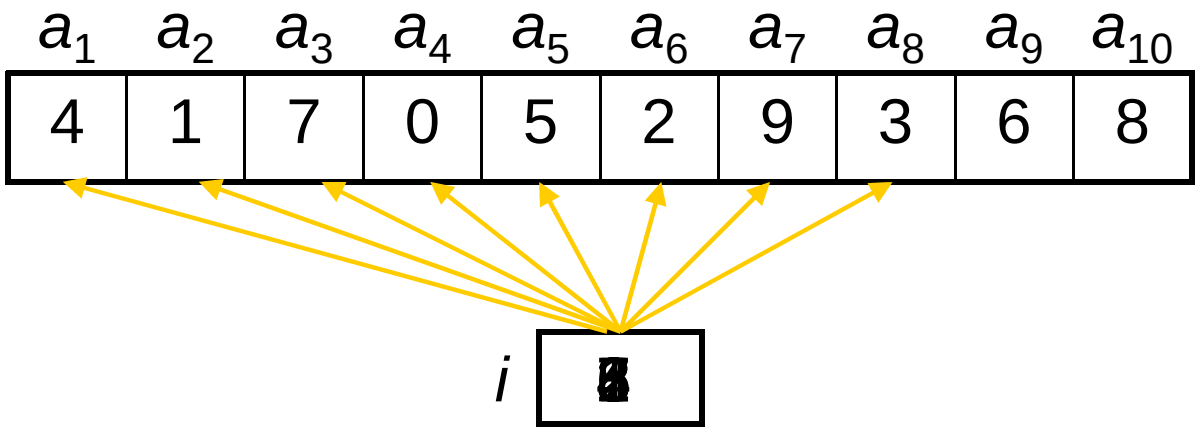
  $i := i + 1$

if $i \leq n$ **then** *location* := i

else *location* := 0

x	3

location	8



Algorithm 2: Linear search, take 2

procedure linear_search (x: integer; a_1, a_2, …, a_n: integers)

$i := 1$

while ($i \leq n$ and $x \neq a_i$)

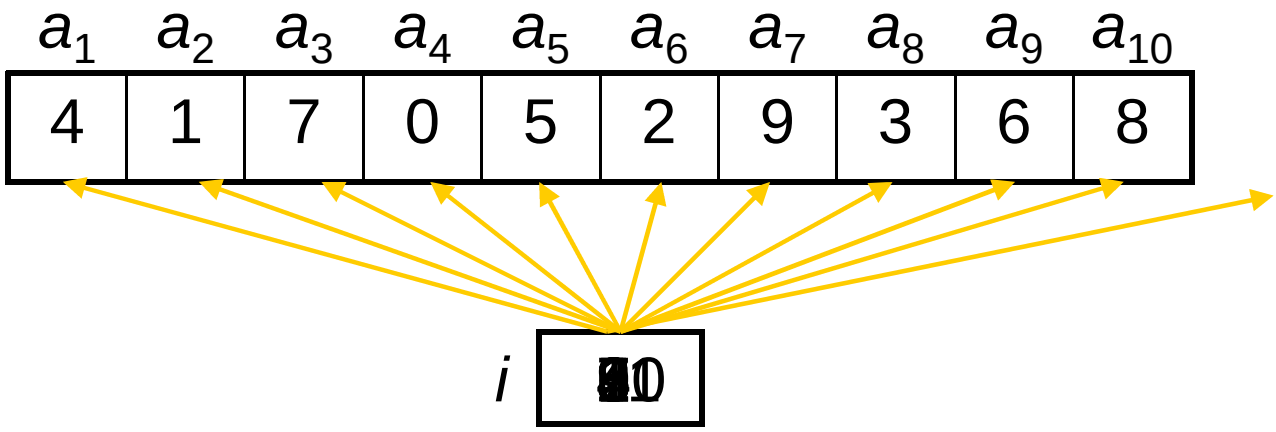
  $i := i + 1$

if $i \leq n$ **then** *location* := i

else *location* := 0

x	11

location	0



Linear search running time

- How long does this take?
- If the list has *n* elements, worst case scenario is that it takes *n* “steps”
  - Here, a step is considered a single step through the list

Algorithm 3: Binary search

- Given a list, find a specific element in the list
  - List MUST be sorted!
- Each time it iterates through, it cuts the list in half

procedure binary_search (x: integer; a_1, a_2, …, a_n: increasing integers)
i := 1 { i is left endpoint of search interval }
j := n { j is right endpoint of search interval }
while $i < j$
begin
  m := $\lfloor (i+j)/2 \rfloor$ { m is the point in the middle }
  if x > a_m **then** i := m+1
  else j := m
end
if $x = a_i$ **then** *location* := i
else *location* := 0

{*location* is the subscript of the term that equals x, or it is 0 if x is not found}

Algorithm 3: Binary search, take 1

procedure binary_search (x: integer; a_1, a_2, …, a_n: increasing integers)

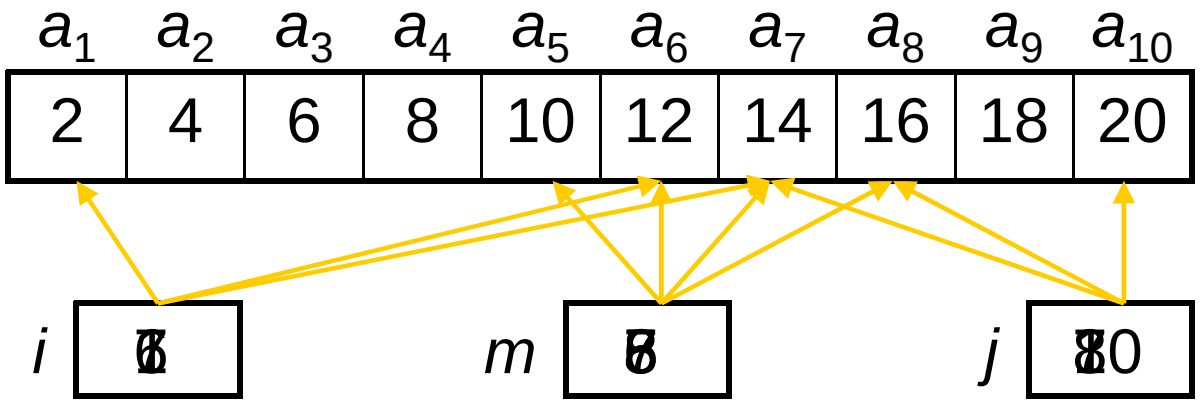
i := 1
j := *n*

while *i* < *j*
begin
  m := $\lfloor (i+j)/2 \rfloor$
  if x > a_m **then** *i* := *m*+1
  else *j* := *m*
end

if *x* = a_i **then** *location* := *i*
else *location* := 0

x 14

location 7



Binary Search

- Assumes list (array) is sorted
- Example of a recursive algorithm as algorithm applied to smaller list

Binary search in worst case takes

$O(\log_2 n)$ steps, where n = size of the list.

Binary search running time

- How long does this take (worst case)?

- If the list has 8 elements
  - It takes 3 steps
- If the list has 16 elements
  - It takes 4 steps
- If the list has 64 elements
  - It takes 6 steps

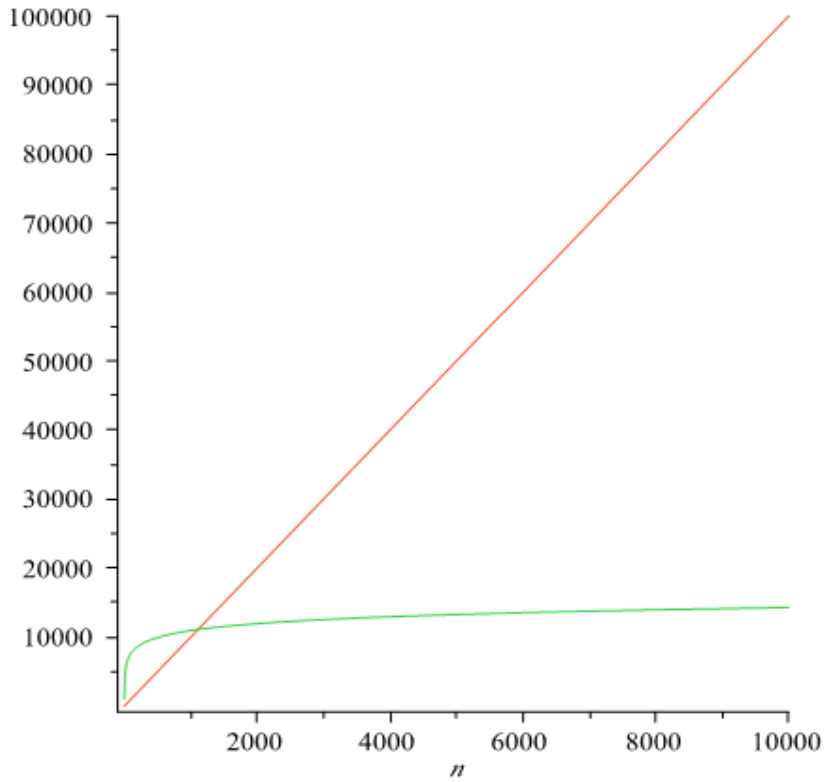
- If the list has *n* elements
  - It takes $\log_2 n$ steps

Comparing Sequential and Binary Search

Sequential search is worst case and average case **O(n)**

Binary search is worst case **O(log n)**

n	log n
$2 = 2^1$	1
$16 = 2^4$	4
$1024 = 2^{10}$	10
1 million = 2^{20}	20
1 billion = 2^{30}	30



Median in Linear Time

Selection in Linear time (divide and conquer)

Problem: Given an array *A* of *n* numbers = *A*[*1*..*n*] and a number *k* (*1<k<n),* find the k^{th} smallest number in *A*.

The median is the ceil(n/2)th smallest element

Example: If A= {7, 4, 8, 2, 4}, then |A| = 5 and ceil(5/2)=3rd the 3rd smallest element in A (and median) is 4.

Solutions for our problem:

1. Naive approach:
   1. Sort the array (heap,merge) → O(n log n)
   2. return the *k* th element. → O(1)
   3. Total Execution time → *O*(*n* log *n*)
2. There is a linear-time selection algorithm.

Strategy1: Partition-based (quick sort) selection

- Choose one element *p as pivot* from array A.
- Split input into three sets: A = [L | E | R]
  - LESS: elements from A that are smaller than p
  - EQUAL: elements from A that are equal to p
  - RIGHT: elements from A that are greater than p
- We have three cases:

  (a) k < |L| → the element we are looking for is also the k^{th} smallest number in L

  (b) k in |L| .. |L| + |E| → the element we are looking for is pivot p.

  (c) k > |L| + |E| → the element we are looking for is the (k - |L| - |E|)th smallest element in R.
- This takes O(n) in the best and average case,
- Unfortunatelly the running time can grow to $O(n^2)$ in the worst case with bad pivots, reduces size by 1 per step.

Linear Time selection algorithm

Procedure *select* (*A[1..n], low…high, k*)

1. If n is small, for example n<6, then, sort and return the kth smallest element (bounded by 7 steps).
2. If n>5, then partition the numbers into groups of 5. (time n/5)
3. Sort the elements in each group. Sort and select the middle elements of each group (medians) into set **M.** (time: 7*n/5)
4. Calculate the mm = median(**M**) = **median of the medians.**
5. Guarantees mm is in A[30% to 70%],
6. Use mm to Partition the (n-1) elements into 3 lists (L,E,R).
    1. L → A[i] < mm
    2. E → A[i] = mm
    3. R → A[i] > mm, (let r = size of R)
7. Find the k^{th} median recursively:
    1. k<r →return k^{th} smallest of set L. ***select* (L, *1..|L|*, *k*)**
    2. k=r →return mm .. DONE (Note. mm is the k^{th} smallest; k = r is size of R).
    3. k>r→ return $(k-r)^{th}$ smallest of set R. ***select* (R, *r+1..|R|*, k-|*L*|+*1*)**

Example

Input of 25 numbers, find the median at 13th

8, 33, 17, 51, 57, 49, 35, 11, 25, 37,14, 3,

2, 13, 52, 12, 6, 29, 32, 54, 5, 16, 22, 23, 7

The median is the kth element = k = 25/ 2 = 13th in sorted list

Step 1. divide into groups of 5
(8, 33, 17, 51, 57)
(49, 35, 11, 25, 37)
(14, 3, 2, 13, 52)
(12, 6, 29, 32, 54)
(5, 16, 22, 23, 7).

Step 2. Sort each group
(8, 17, 33, 51, 57)
(11, 25, 35, 37, 49)
(2, 3, 13, 14, 52)
(6, 12, 29, 32, 54)
(5, 7, 16, 22, 23).

Step 2. Note: sorting smaller arrays take less time than sorting whole array.
Step 3. Extract the median of each group ***M*** = {33, 35, 13, 29, 16}, sort **M**,
Step 4. Find median(M) = median of medians *mm* = 29 = {13,16,29,33,35}

Step 5. Partition *A* into three sequences using mm as the pivot:
L = {8, 17, 11, 25, 14, 3, 2, 13, 12, 6, 5, 16, 22, 23, 7} size=15
E = {29},
R = {33, 51, 57, 49, 35, 37, 52, 32, 54} size=9

Example continued

L = {8, 17, 11, 25, 14, 3, 2, 13, 12, 6, 5, 16, 22, 23, 7}

- We repeat the same procedure above with L as A.
- select (L, 1..|L|, k)
  - We divide the elements into 3 groups of 5 elements each: (8,17,11,25,14), (3,2,13,12,6), (5,16,22,23,7).
  - Sort each of the group, and find the new set of medians: *M* = {6, 14, 16}.
  - Find new mm=Median(M) = 14.
  - Next, partition A into three sequences with pivot mm=14

    L1 = {*8, 11, 3, 2, 13, 12, 6, 5,* 7}, |L1|=9

    E1 = {14}

    R1 = {*17, 25, 16, 22, 23*} .
  - Since k=25/2=13th > *(9+1)* = $|L1| + |E1|$, we set *A* = R1 and find the 3rd (3rd = 13th – *(9+1)*) element in R1.
  - The algorithm will return 3rd element of sorted R1[3] = 22.
- Thus, the median of the numbers in the given sequence is 22.

Notes and References

Notes:

- *Balanced-Quick-Sort* in O(n*log(n)) can use QuickSelect to pick guaranteed good pivots.
- Since the constants in complexity are large, we can use random pivot (like quicksort).

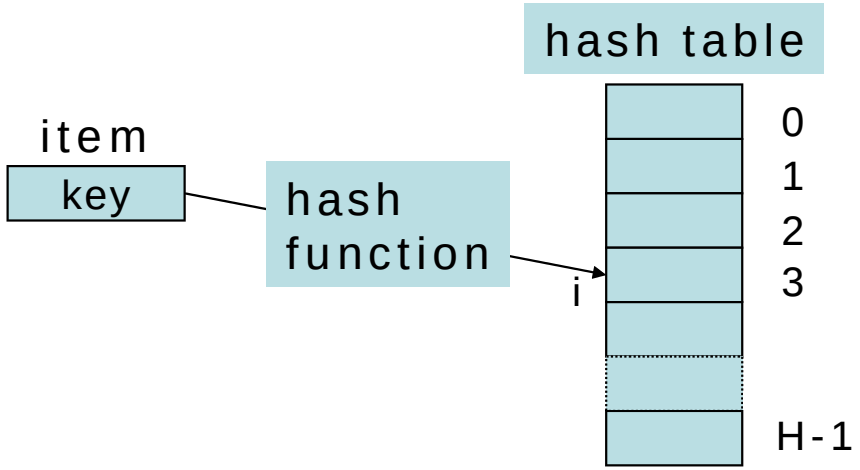
References:

1. https://en.wikipedia.org/wiki/Median_of_medians
2. BFPRT 1973 - Blum, Floyd, Pratt, Rivest, Tarjan. "Time bounds for selection" (PDF). *JCSS*. **7** (4) 448–461.

Hashing

Hash Tables

- Hashing is used for storing relatively large amounts of data in a table called a hash table.
- Hash table is usually fixed as H-size, which is larger than the amount of data that we want to store.
- We define the load factor (λ) to be the ratio of data to the size of the hash table.
- Hash function maps an *item* into an *index* in range of hash-table.



Hash Tables (2)

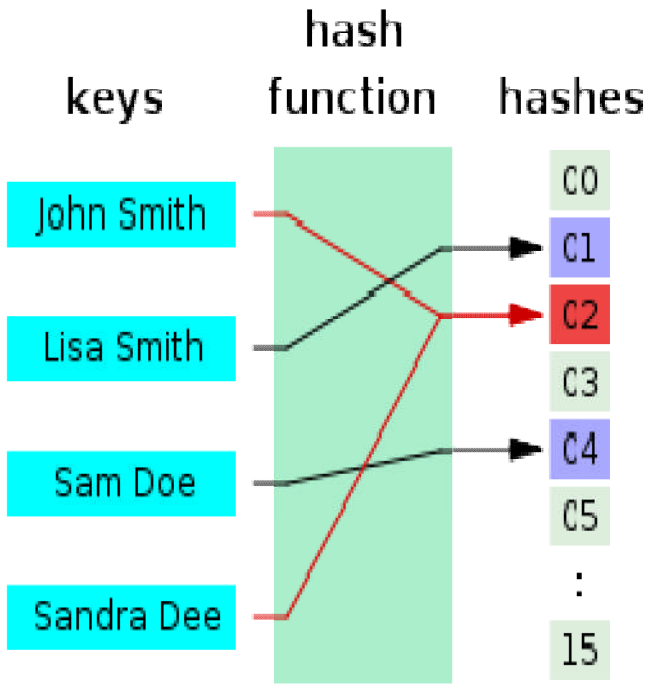
- Hashing is a technique used to perform insertions, deletions, and finds in constant average time.
- To insert or find a certain data, we assign a key to the elements and use a function to determine the location of the element within the table called hash function.
- Hash tables are arrays of cells with fixed size containing data or keys corresponding to data.
- For each key, we use the hashing function to map key into some number in the range [0 to H_size-1] using hashing function.

Hash Function

- Hashing function should have the following features:
  - Easy and fast to compute.
  - Two distinct key map to two different cells in array (Not true in general). Low collisons - why?
  - This can be achieved by using direct-address table where universal set of keys is reasonably small.
  - Distributes the keys evenly among cells.
- One simple hashing function is to use mod function with a prime number (i = x mod p).
- Any manipulation of digits, with less complexity and good distribution can be used.

Hash function

- A **hash function** is any *well-defined procedure or mathematical function* that converts a large, possibly variable-sized amount of data into a small data (usually an integer that may serve as an index to an array).



Example of hash function in Java

```
int hash(String key, int tableSize) {
    int hashVal = 0;
    for (int i=0; i < key.length(); i++)
        hashVal += key.charAt(i)
    return hashVal % tableSize;
}
```

Choosing a hash function

A good has function should satisfy two criteria:

1. It should be quick to compute
2. It should minimize the number of collisions

Hashing

Hash tables are used to store data as {<key,value> pairs }, accessible by the key.

Hashing is a method of inserting data into a table.

Tables can be implemented in many ways.

Hashing is used for faster data retrieval **: O(1)** on average for insert, lookup, and remove

Examples

1. fixed array (limiting number of elements),
2. array of linked lists (potentially unlimited number of elements)

Hash table (called Associative array)

1. **hash table** (also **hash map**) is a data structure used to implement an associative array, a structure that can map keys to values.
2. A hash table uses a hash function to compute an *index* into an array of *buckets* or *slots*, from which the correct value can be found.

Collisions

1. Ideally, the hash function should assign to each possible key to a unique bucket, but this ideal is rarely achievable in practice (unless the hash keys are fixed; i.e. new entries are never added to the table after it is created, e.g. in hardware).
2. Instead, most hash table designs assume that *hash collisions* will occur and must be accommodated in some way.

Collision resolution

- When two keys map into the same cell, we get a collision.
- We may have collision in insertion, and need to set a procedure (collision resolution) to resolve it.

Closed Hashing

- If collision, try to find alternative cells within table.
- *Closed hashing* also known as *open addressing*.
- For insertion, we try cells in sequence by using incremented function like:
  - $h_i(x)$ = (hash(x) + f(i)) mod table_size f(0) = 0
- Function f is used as collision resolution strategy.
- The table is bigger than the number of data items.
- Different methods to choose function f :
  - Linear probing
  - Quadratic probing
  - Double hashing

Issues

Other issues common to all closed hashing resolutions:

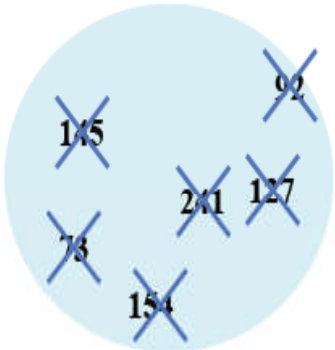
- Confusing after deletion.
- Simpler than open hashing function
- Good if we do not expect too many collisions.
- If search is unsuccessful, we may have to search the whole table.
- Use of a large table compared to size of input data.

1. Table size is usually prime to avoid aliasing
2. large table size means wasted space
3. small table size means more collisions.

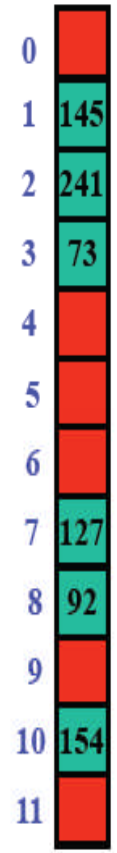
Example:

table_size = 12

- **145 mod 12 =1**
- **92 mod 12 = 8**
- **241 mod 12 = 1 // collision**
- **127 mod 12 = 7**
- **73 mod 12 =1 // collision**
- **145 mod 12 = 10**



154 mod 12 = 10



Hash table_size=12

Used in OS and Engg

- In many situations, hash tables turn out to be more efficient than search trees or any other table lookup structure.
- For this reason, they are widely used in OS (operating systems) and Engineering, particularly for associative arrays, database indexing, caches, and sets.

Hash Functions

- A *collision* is defined when multiple keys map onto the same table index.
- A good hash function has the following characteristics
  - avoids collisions
  - Spreads keys evenly in the array
  - Inexpensive to compute - must be O(1)
- Hash Functions for integers

  hash(int key) = abs(key) % table_size
- table_size should be a prime

Universal hashing

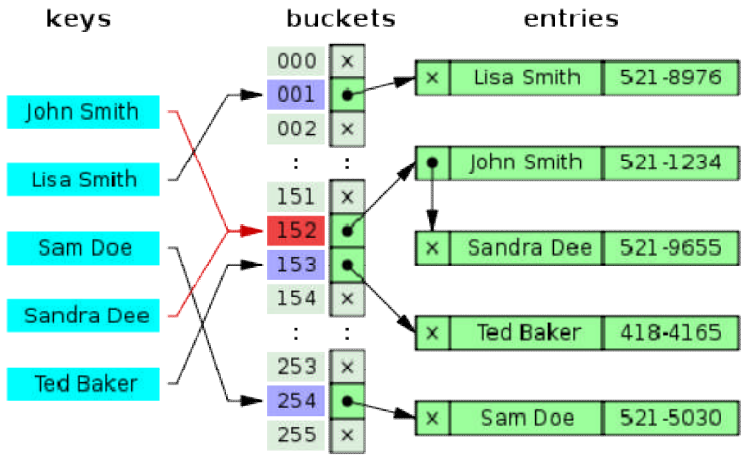
- **Universal hashing** (in a randomized algorithm or data structure) refers to selecting a hash function at random from a family of hash functions with a certain mathematical property.
- This guarantees a low number of collisions in expectation in worst case (the data is chosen by an adversary).

Perfect hashing

- If all keys are known at time of coding, a perfect hash function can be used to create a perfect hash table that has no collisions.
- minimal perfect hashing means every location in the hash table is used.
- Load_factor = data_size /hash_table_size
- Eg. Used by Db, compiler keyword table,

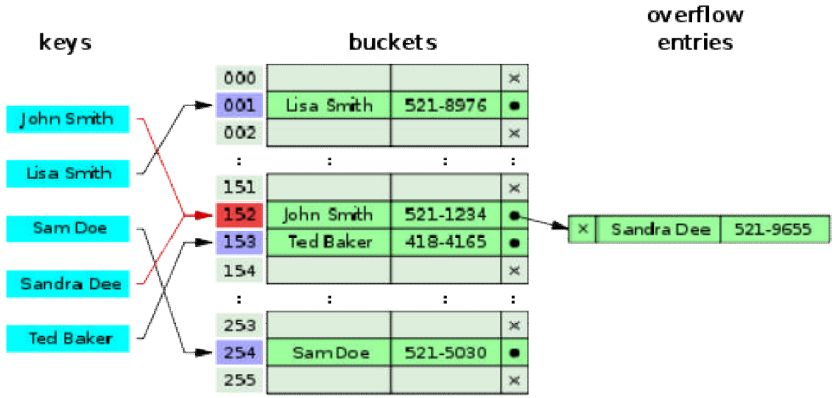
Collision resolution

- *separate chaining*, each bucket is independent, and has a list of entries with the same index.



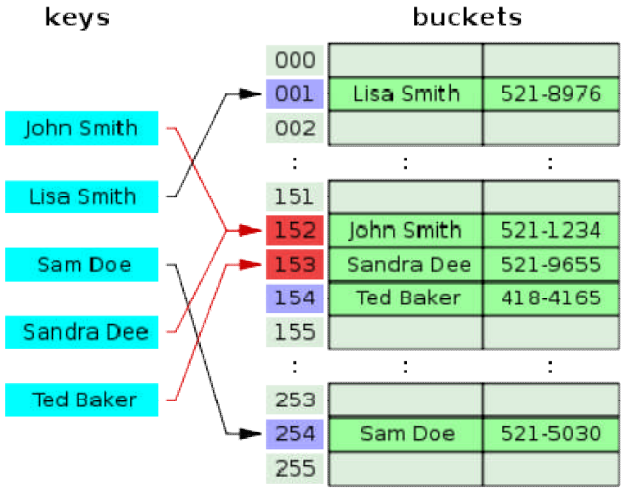
Separate chaining

- Some chaining implementations store the first record of each chain in the slot array itself. Future collisions are chained.



Open addressing

1. open addressing, all entry records are stored in the bucket array itself. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some *probe sequence*, until an unoccupied slot is found
2. Deletion is hard (causing gaps)



Probing sequence for empty slot

Well-known probe sequences include:

1. Linear probing, in which the interval between probes is fixed (usually 1)
2. Quadratic probing, in which the interval between probes is increased by adding the successive outputs of a quadratic polynomial to the starting value given by the original hash computation
3. Double hashing, in which the interval between probes is computed by another hash function

Practice

- **Associative array**, **hash**, **map**, or **dictionary** is an abstract data type composed of a collection of (key, value) pairs, such that each possible key appears at most once in the collection.
- Operations associated with this data type allow:
  1. addition of pairs to the collection
  2. removal of pairs from the collection
  3. modification of the values of existing pairs
  4. lookup of the value associated with a particular key

hash maps in languages

C: see glibc, next slide

C++:
```
#include <map>
std::map<std::string, std::string> mymap;
mymap[“mosh"] = “nitk";
```

Java:
```
Map<String, String> mymap = new HashMap<String, String>();
mymap.put(“mosh", “nitk");
```

Perl: `$mymap{'mosh'}='nitk';`

Php: `$mymap['mosh']='nitk';`

Python: `mymap = { } ; mymap['mosh']='nitk'`

See http://rosettacode.org/wiki/Associative_arrays

Hashing in C with gcc

- #include <search.h>
- First a hash table must be created using **hcreate**()
- The function **hdestroy**() frees the memory occupied by the hash table that was created by **hcreate**(). After calling **hdestroy**() a new hash table can be created using **hcreate**().
- The **hsearch**() function searches the hash table for an item with the same key as *item*
- On success, **hsearch**() returns a pointer to an entry in the hash table. **hsearch**() returns NULL on error, that is, if *action* is **ENTER** and the hash table is full, or *action* is **FIND** and *item* cannot be found in the hash table.
- Disadvantage: single global hash table in library.
- Advantage: Useful in small utility programs.
- See http://linux.die.net/man/3/hsearch

Hashing in g++

```
$ g++  -std=c++0x  hash_map4.cpp
$ a.exe
september -> 30
february  -> 28
```

```
1.  #include <iostream>
2.  #include <string>
3.  #include <unordered_map>
4.  using namespace std;
5.  int main() {
6.      unordered_map<string, int> months;
7.      months["february"] = 28;
8.      months["september"] = 30;
9.      cout << "september -> " << months["september"] << endl;
10.     cout << "february  -> " << months["february"] << endl;
11.     return 0;
12. }
```

Custom hashing function in g++

```
1.  typedef pair<string,string> Name;
2.  struct hash_name {
3.      size_t operator( )(const Name &name) const {
4.          return hash<string>( )(name.first)
5.                  ^ hash<string>( )(name.second);
6.      }
7.  } ;
```

```
1.  int main(int argc, char* argv[]) {
2.     unordered_map<Name,int,hash_name> ids;
3.      ids[Name("Amit", "Gupta")] = 234;
4.      ids[Name("Alok","Raj")] =  567;
5.      for ( auto ii = ids.begin() ; ii != ids.end() ; ii++ )
6.          cout << ii->first.first << " " << ii->first.second << " : " << ii->second << endl;
7.      return 0;
8.  }
```

```
1.#include <iostream>
2.#include <unordered_map>
3.#include <string>
4.#include <functional>
5.using namespace std;
```

```
$ g++  -std=c++0x  hash_map5.cpp
$  ./a.exe
Amit Gupta : 234
Alok Raj : 567
```

Perl hash tables

```
my %grade;     # declare grade to be a hash table
grade{'Ajay'} = ‘B’;
grade{'Ann'} = 90;  # mixed types, string and int.
 # Sort by name and print the grades
 for my $name (sort keys %grade) {
    print $name, $grade;
}
 grade{'Ann'}++;        # add one to grade of Ann.
 delete grade{'Ajay'}; # delete one entry.
```