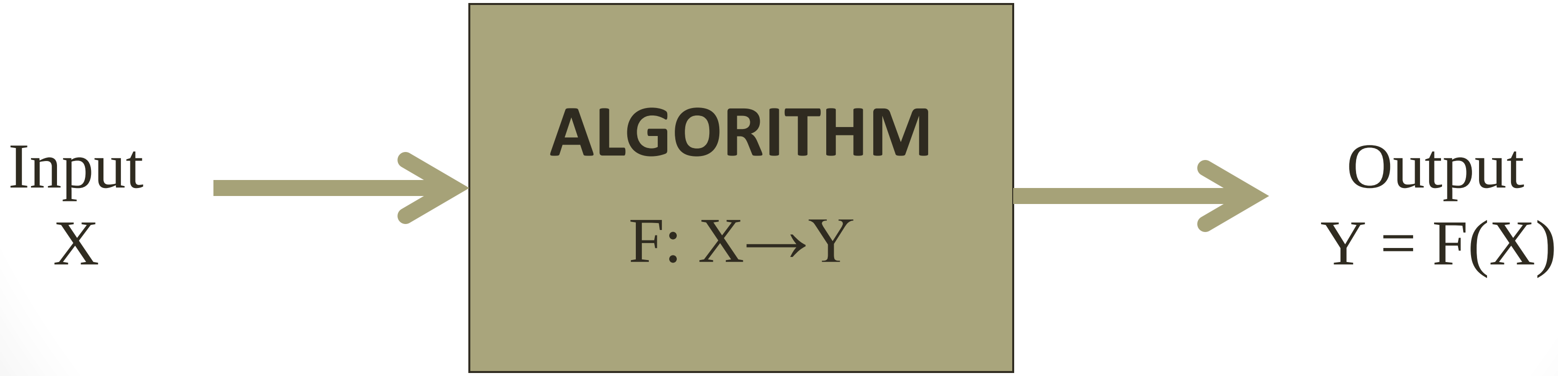
Design and Analysis of Algorithms

Algorithm Definition

- **An algorithm is a step-by-step procedure for solving a particular problem in a finite amount of time.**

- More generally, an algorithm is any well defined computational procedure that takes collection of elements as input and produces a collection of elements as output.



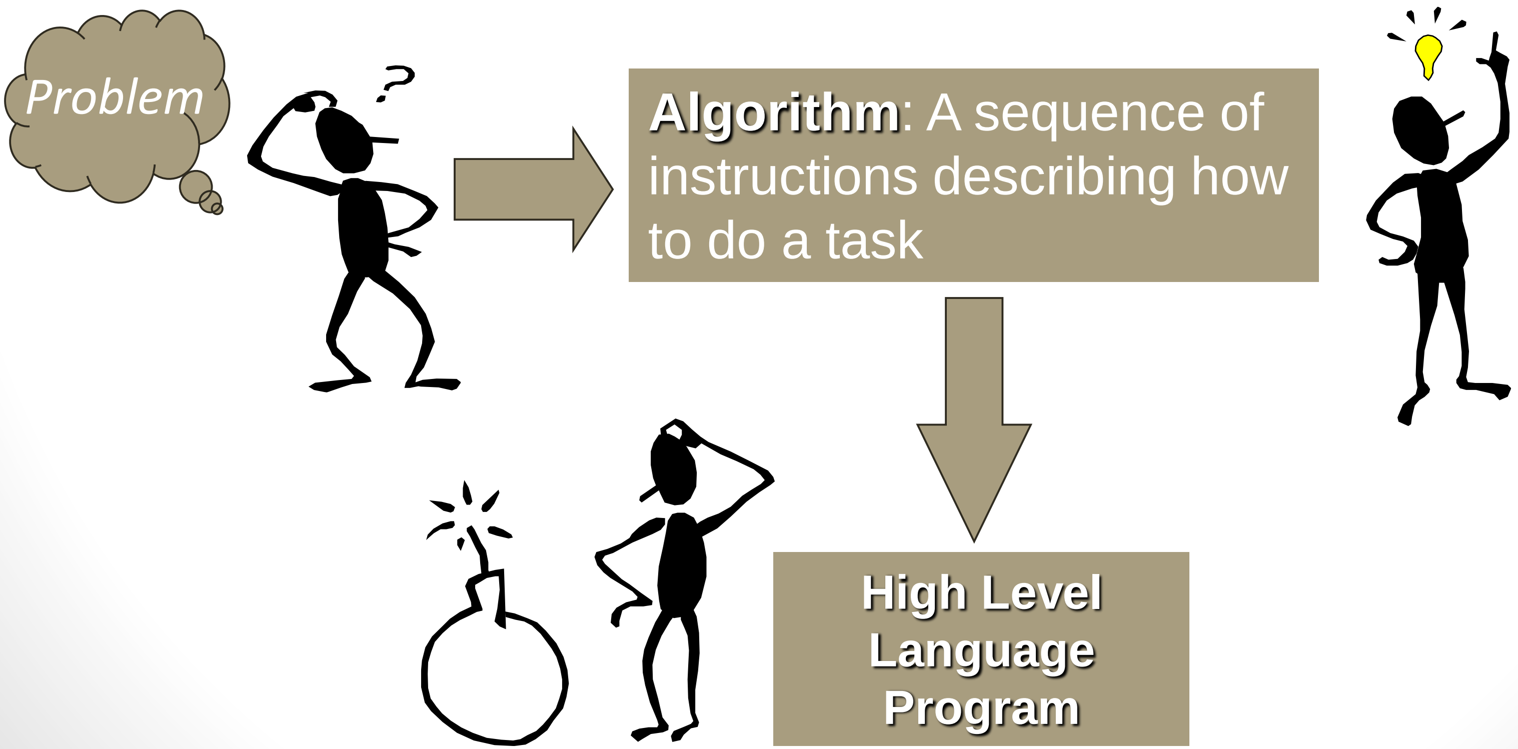
Algorithm -- Examples

- Repairing a lamp
- A cooking recipe
- Calling a friend on the phone
- The rules of how to play a game
- Directions for driving from A to B
- A car repair manual
- Human Brain Project
- Internet & Communication Links (Graph)
- Matrix Multiplication

Algorithm vs. Program

- **A computer program is an instance, or concrete representation, for an algorithm in some programming language**

- Set of instructions which the computer follows to solve a problem



Solving Problems (1)

When faced with a problem:

1. First clearly define the problem
2. Think of possible solutions
3. Select the one that seems the best under the prevailing circumstances
4. And then apply that solution
5. If the solution works as desired, fine; else go back to step 2

Solving Problems (2)

- It is quite common to first solve a problem for a particular case
- Then for another
- And, possibly another
- And watch for patterns and trends that emerge
- And to use the knowledge from these patterns and trends in coming up with a general solution
- And this general solution is called "Algorithm"

One Problem, Many Algorithms

Problem

- The statement of the problem specifies, in general terms, the desired input/output relationship.

Algorithm

- The algorithm describes a specific computational procedure for achieving input/output relationship.

Example

- Sorting a sequence of numbers into non-decreasing order.

Algorithms

- Various algorithms e.g. merge sort, quick sort, heap sorts etc.

Problem Instances

- An **input sequence** is called an **instance of a Problem**
- A problem has many particular *instances*
- An algorithm must **work correctly on all instances** of the problem it claims to solve
- Many interesting problems have infinitely many instances
  - Since computers are finite, we usually need to <u>limit the number and/or size of possible instances in this case</u>
  - This restriction doesn't prevent us from doing analysis in the abstract

Properties of Algorithms

- It must be composed of an ordered sequence of precise steps.
- It must have finite number of well-defined instructions /steps.
- The execution sequence of instructions should not be ambiguous.
- It must be correct.
- It must terminate.

Syntax & Semantics

An algorithm is "correct" if its:

- Semantics are correct
- Syntax is correct

Semantics:

an algorithm (the soul!)

Syntax:

- The actual representation of an algorithm (the body!)

WARNINGS:

1. An algorithm can be syntactically correct, yet semantically incorrect – dangerous situation!

2. Syntactic correctness is easier to check as compared to semantic correctness

Colorless green ideas sleep furiously!

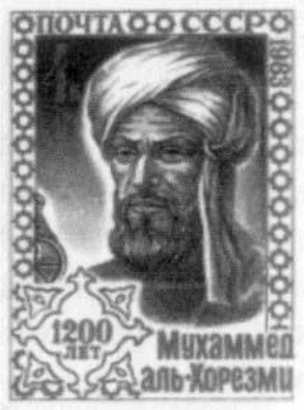
Algorithm Summary

- Problem Statement
  - Relationship b/w input and output
- Algorithm
  - Procedure to achieve the relationship
- Definition
  - A sequence of steps that transform the input to output
- Instance
  - The input needed to compute solution
- Correct Algorithm
  - for every input it halts with correct output

Brief History

- The study of algorithms **began with mathematicians** and was a significant area of work in the early years. The **goal of those early studies** was to find a single, general algorithm that could solve all problems of a **single type.**

- Named after 9th century Persian Muslim mathematician Abu Abdullah Muhammad ibn Musa al-Khwarizmi who lived in Baghdad and worked at the Dar al-Hikma



- Dar al-Hikma acquired & translated books on science & philosophy, particularly those in Greek, as well as publishing original research.

- The word *algorism* originally referred only to the **rules of performing arithmetic** using Hindu-Arabic numerals, but later evolved to include all definite procedures for solving problems.

Al-Khwarizmi's Golden Principle

All complex problems can be and must be solved
using the following simple steps:

1. Break down the problem into small, simple sub-problems
2. Arrange the sub-problems in such an order that each of them can be solved without effecting any other
3. Solve them separately, in the correct order
4. Combine the solutions of the sub-problems to form the solution of the original problem

Why Algorithms are Useful?

- Once we find an algorithm for solving a problem, we do not need to re-discover it the next time we are faced with that problem

- Once an algorithm is known, the task of solving the problem reduces to following (almost blindly and without thinking) the instructions precisely

- All the knowledge required for solving the problem is present in the algorithm

Why Write an Algorithm Down?

- For your own use in the future, so that you don't have spend the time for rethinking it

- Written form is easier to modify and improve

- Makes it easy when explaining the process to others

Designing of Algorithms

- Selecting the basic approaches to the solution of the problem
- Choosing data structures
- Putting the pieces of the puzzle together
- Expressing and implementing the algorithm
  - clearness, conciseness, effectiveness, etc.

Major Factors in Algorithms Design

- Correctness: An algorithm is said to be correct if for every input, it halts with correct output. An incorrect algorithm might not halt at all OR it might halt with an answer other than desired one. Correct algorithm solves a computational problem
- Algorithm Efficiency: Measuring efficiency of an algorithm
  - do its analysis i.e. growth rate.
  - Compare efficiencies of different algorithms for the same problem.

Designing of Algorithms

- Most basic and popular algorithms are search and sort algorithms
- Which algorithm is the best?
- Depends upon various factors, for example in case of sorting
  - The number of items to be sorted
  - The extent to which the items are already sorted
  - Possible restrictions on the item values
  - The kind of storage device to be used etc.

Important Designing Techniques

- **Brute Force**–Straightforward, naive approach–Mostly expensive
- **Divide-and-Conquer** –Divide into smaller sub-problems
  - e.g merge sort
- **Iterative Improvement**–Improve one change at a time.
  - e.g greedy algorithms
- **Decrease-and-Conquer**–Decrease instance size
  - e.g fibonacci sequence
- **Transform-and-Conquer**–Modify problem first and then solve it
  - e.g repeating numbers in an array
- **Dynamic programming**–Dependent sub-problems, reuse results

Algorithm Efficiency

- Several possible algorithms exist that can solve a particular problem
  - each algorithm has a given *efficiency*
  - compare efficiencies of different algorithms for the same problem
- The efficiency of an algorithm is a measure of the **amount of resources consumed** in solving a problem of size n
  - *Running time* (number of primitive steps that are executed)
  - *Memory/Space*
- Analysis in the context of algorithms is concerned with predicting the required resources
- There are always tradeoffs between these two efficiencies
  - allow one to decrease the running time of an algorithm solution by increasing space to store and vice-versa
- Time is the resource of most interest
- By analyzing several candidate algorithms, the most efficient one(s) can be identified

Algorithm Efficiency

Two goals for best design practices:

1. To design an algorithm that is easy to understand, code, debug.
2. To design an algorithm that makes efficient use of the computer's resources.

How do we improve the time efficiency of a program?

<u>The 90/10 Rule</u>

- 90% of the execution time of a program is spent in executing 10% of the code. So, how do we locate the critical 10%?
  - software metrics tools
  - global counters to locate bottlenecks (loop executions, function calls)

Time Efficiency improvement

- Good programming: Move code **out of loops** that does not belong there
- Remove any **unnecessary I/O** operations
- Replace an inefficient algorithm (best solution)

Moral - Choose the most appropriate algorithm(s) BEFORE program implementation

Analysis of Algorithms

- Two essential approaches to measuring algorithm efficiency:
- Empirical analysis:
  - Program the algorithm and measure its running time on example instances
- Theoretical analysis
  - Employ mathematical techniques to derive a *function* which relates the **running time to the *size of instance***

- **In this cousre our focus will be on Threoretical Analysis.**

Analysis of Algorithms

- Many criteria affect the running time of an algorithm, including
  - speed of CPU, bus and peripheral hardware
  - design time, programming time and debugging time
  - language used and coding efficiency of the programmer
  - quality of input (good, bad or average)
  - But
- Programs derived from two algorithms for solving the same problem should both be
  - Machine independent
  - Language independent
  - Amenable to mathematical study
  - Realistic

Analysis of Algorithms

- The following three cases are investigated in algorithm analysis:

- A) Worst case: The worst outcome for any possible input
  - We often concentrate on this for our analysis as it provides a **clear** ***upper bound*** of resources
  - an absolute guarantee

- B) Average case: Measures performance over the entire set of possible instances
    - Very useful, but treat with care: what is "average"?
    - Random (equally likely) inputs vs. real-life inputs

- C) Best Case: The best outcome for any possible input
  - provides lower bound of resources

Analysis of Algorithms

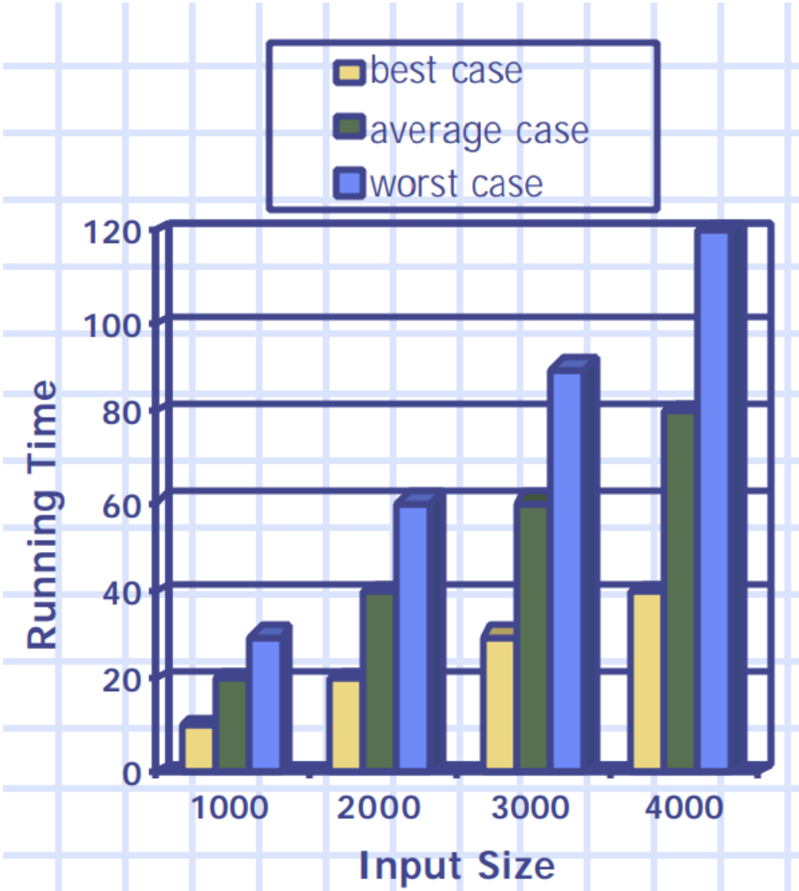
- An algorithm may perform very differently on different example instances. e.g: bubble sort algorithm might be presented with data:
    - already in order
    - in random order
    - in the exact reverse order of what is required

- Average case analysis can be difficult in practice
  - to do a realistic analysis we need to know the likely distribution of instances
  - However, it is often very useful and more relevant than worst case; for example *quicksort* has a catastrophic (extremly harmful) worst case, but in practice it is one of the best sorting algorithms known

- The average case uses the following concept in probability theory. Suppose the numbers n_1, n_2 , …, n_k occur with respective probabilities p_1, p_2,…..p_k. Then the expectation or average value E is given by $E = n_1p_1 + n_2p_2 + ...+ n_k.p_k$

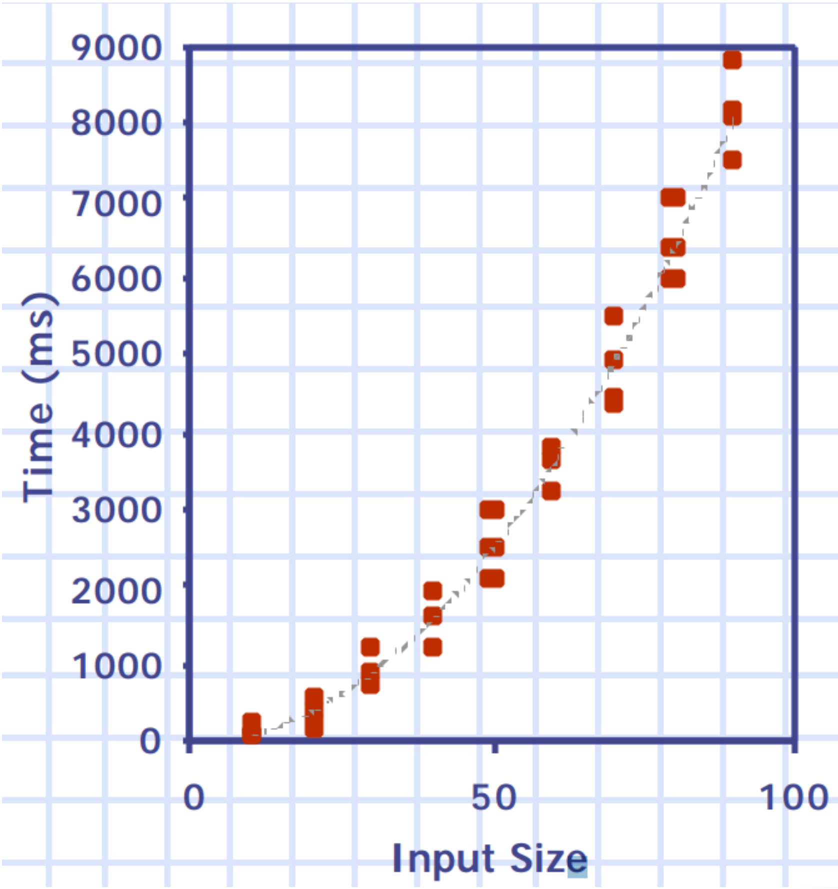
Empirical Analysis

- Most algorithms transform input objects into output objects
- The running time of an algorithm typically grows with the input size
- Average case time is often difficult to determine
- We focus on the worst case running time
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



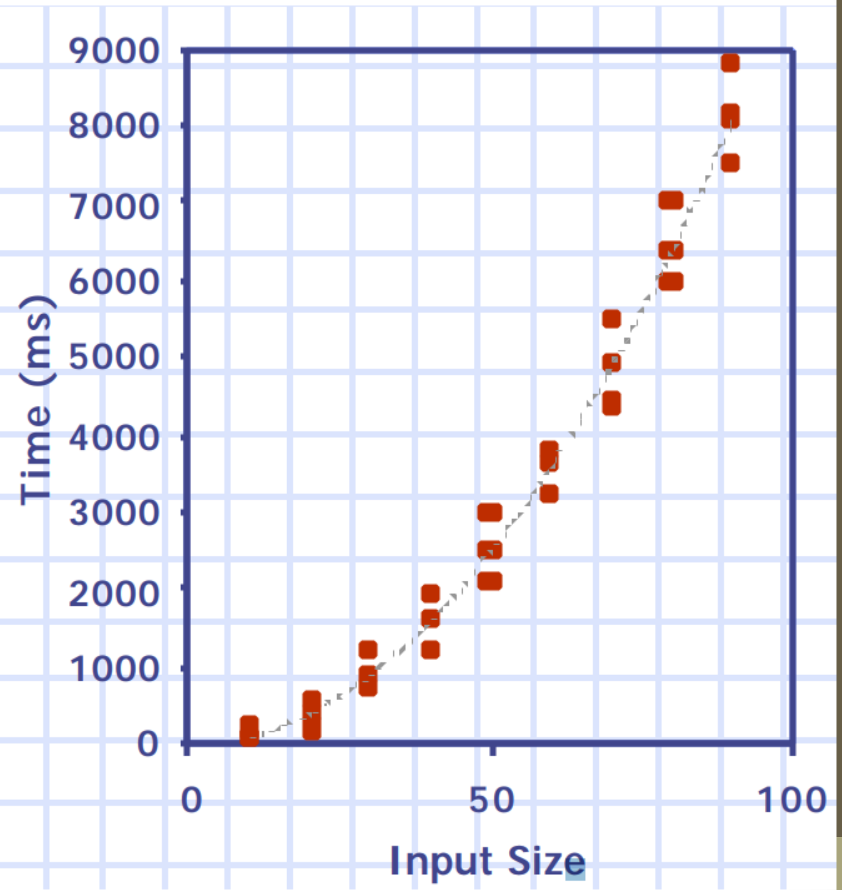
Empirical Analysis

- Write a program implementing the algorithm
- Run the program with inputs of varying size and compositions
- Use timing routines to get an accurate measure of the actual running time e.g. System.currentTimeMillis()
- Plot the results



Limitations of Empirical Analysis

- Implementation dependent
  - Execution time differ for different implementations of same program
- Platform dependent
  - Execution time differ on different architectures
- Data dependent
  - Execution time is sensitive to amount and type of data minipulated.
- Language dependent
  - Execution time differ for same code, coded in different languages



∴ **absolute measure for an algorithm is not appropriate**

Theorerical Analysis

- Data independent
  - Takes into account all possible inputs
- Platform independent
- Language independent
- Implementatiton independent
  - not dependent on skill of programmer
  - can save time of programming an inefficient solution
- Characterizes running time as a function of input size, n. Easy to extrapolate without risk

Why Analysis of Algorithms?

- For real-time problems, we would like to prove that an algorithm terminates in a given time.
- Algorithmics may indicate which is the best and fastest solution to a problem without having to code up and test different solutions
- Many problems are in a complexity class for which no practical algorithms are known
  - better to know this before wasting a lot of time trying to develop a ”perfect” solution: **verification**

But Computers are So Fast These Days??

- Do we need to bother with algorithmics and complexity any more?
    - computers are fast, compared to even 10 years ago...

- Many problems are so computationally demanding that **no growth in the power of computing will help very much.**

- Speed and efficiency are still important

Importance of Analyzing Algorithms

- Need to recognize limitations of various algorithms for solving a problem
- Need to understand relationship between problem size and running time
  - When is a running program not good enough?
- Need to learn how to analyze an algorithm's running time without coding it
- Need to learn techniques for writing more efficient code
- Need to recognize bottlenecks in code as well as which parts of code are easiest to optimize

Importance of Analyzing Algorithms

- An array-based list retrieve operation takes at most one operation, a linked-list-based list retrieve operation at most “n” operations.
- But insert and delete operations are much easier on a linked-list-based list implementation.
- When selecting the implementation of an Abstract Data Type (ADT), we have to consider how frequently particular ADT operations occur in a given application.
- For small problem size, we can ignore the algorithm’s efficiency.
- We have to weigh the trade-offs between an algorithm’s time requirement and memory requirements.

What do we analyze about Algorithms?

- Algorithms are analyzed to understand their behavior and to improve them if possible

- Correctness
  - Does the input/output relation match algorithm requirement?
- Amount of work done
  - Basic operations to do task
- Amount of space used
  - Memory used
- Simplicity, clarity
  - Verification and implementation.
- Optimality
  - Is it impossible to do better?

Problem Solving Process

- Problem
- Strategy
- Algorithm
  - Input
  - Output
  - Steps
- Analysis
  - Correctness
  - Time & Space Optimality
- Implementation
- Verification

Computation Model for Analysis

- To analyze an algorithm is **to determine the amount of resources** necessary to execute it. These resources include computational time, memory and communication bandwidth.

- Analysis of the algorithm is performed with respect to a computational model called RAM (Random Access Machine)

- A RAM is an idealized **uni-processor machine** with an **infinite large random-access memory**
  - Instruction are executed one by one
  - All memory equally expensive to access
  - No concurrent operations
  - Constant word size
  - All reasonable instructions (basic operations) take unit time

Complexity of an Algorithm

- The complexity of an algorithm is the amount of work the algorithm performs to complete its task. It is the **level of difficulty** in solving mathematically posed problems as measured by:
  - Time (time complexity)
  - No. of steps or arithmetic operations (computational complexity)
  - Memory space required (space complexity)

- Complexity is a function T(n) which yields the time (or space) required to execute the algorithm of a problem of size ‘n’.

Pseudocode

- High-level description of an algorithm
- More structured than English prose but Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

```
ArrayMax(A, n)
   Input: Array A of n integers
   Output: maximum element of A

   currentMax ← A[0];
   for i = 1 to n-1 do
      if A[i] > currentMax then
                currentMax ← A[i]
   return currentMax;
```

Pseudocode

- **Indentation** indicates block structure. e.g body of loop
- **Looping Constructs** while, for and the conditional if-then-else
- The symbol // indicates that the reminder of the line is a **comment.**
- **Arithmetic & logical expressions:** (+, -,*,/,) (*and, or* and *not*)
- **Assignment & swap statements**: **$a \leftarrow b$, $a \leftarrow b \leftarrow c$, $a \leftrightarrow b$**
- **Return/Exit/End:** termination of an algorithm or block

```
ArrayMax(A, n)
   Input: Array A of n integers
   Output: maximum element of A

currentMax ← A[0];
for i = 1 to n-1 do
   if A[i] > currentMax then
          currentMax ← A[i]
return currentMax;
```

Pseudocode

- **Local variables** mostly used unless global variable explicitly defined
- If A is a **structure** then |A| is size of structure. If A is an **Array** then n =length[A], upper bound of array. All **Array elements** are accessed by name followed by index in square brackets A[i].
- **Parameters** are passed to a procedure by values
- Semicolons used for multiple short statement written on one line

```
ArrayMax(A, n)
   Input: Array A of n integers
   Output: maximum element of A

currentMax ← A[0]
for i = 1 to n-1 do
   if A[i] > currentMax then
          currentMax ← A[i]
return currentMax
```

Elementary Operations

- An elementary operation is an operation which takes **constant time** regardless of problem size.
- The **running time** of an algorithm on a particular input is determined by the number of "**Elementary Operations**" executed.
  - Theoretical analysis on paper from a description of an algorithm
- Defining elementary operations is a little trickier than it appears
  - We want elementary operations to be relatively machine and language independent, but still as informative and easy to define as possible
- Example of elementary operations include
  - variable assignment
  - arithmetic operations (+, -, x, /) on integers
  - comparison operations (a < b)
  - boolean operations
  - accessing an element of an array
- We will measure number of steps taken in term of size of input

Components of an Algorithm

- Variables and values
- Instructions
- Sequences
- Selections
- Repetitions

Instruction and Sequence

- A linear sequence of elementary operations is also **performed in constant time**.
- More generally, given two program fragments P_1 and P_2 which run sequentially in times t_1 and t_2
  - use the **maximum rule** which states that the larger time dominates
  - complexity will be $\max(t_1,t_2)$

e.g. Assignment Statements

x=a 1

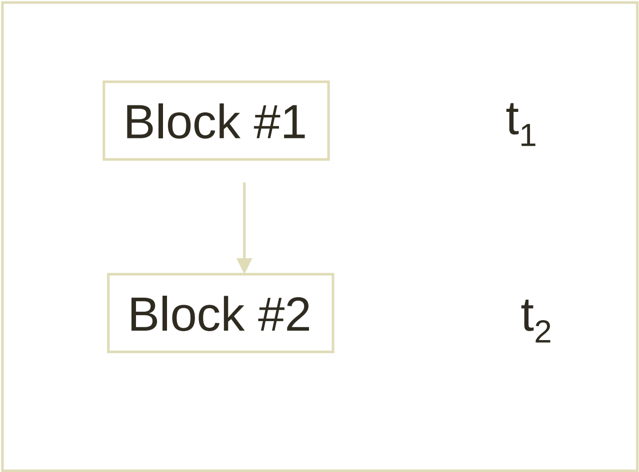
x= a+b*c/h-u 1

a>b 1

T(n) = 1+1+1
T(n) = 3

Sequences

- Analysing a group of consecutive statements
- The statement taking the maximum time will be the one counted
  - use the **maximum rule**



$$T(n) = \max(t_1, t_2)$$

- e.g. a fragment with single for-loop followed by double for- loop
  - $\mathbf{T(n) = n^2}$
- Always analyze function calls first

Selection

- If *<test>* then P_1 *else* P_2 structures are a little harder; **conditional loops**.
- The maximum rule can be applied here too:
  - max(t_1, t_2), assuming t_1, t_2 are times of P_1, P_2
- However, the maximum rule may prove too conservative
  - if <test> is always true the time is t_1
  - if <test> is always false the time is t_2

e.g. if (a>b) then 1

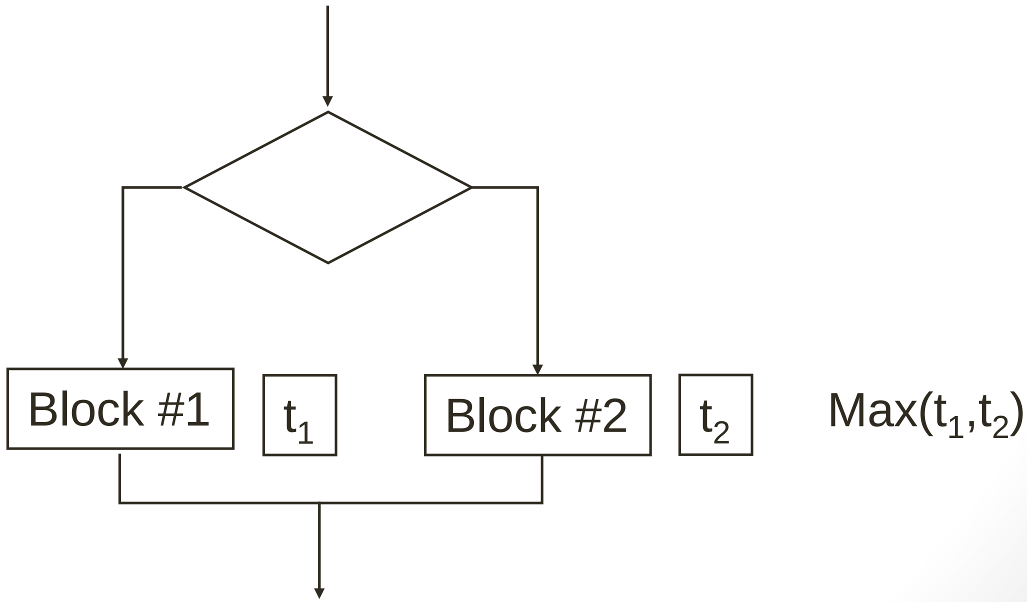
a=2......................1

b=b+3*t 1

else

x=x+2*3 1

T(n)= 1+max(2,1) = 3



Repetition (Loops)

- Analyzing loops: Any loop has two parts:
  - How many iterations are performed?
  - How many steps per iteration?

for *i* = 1 to *n* do
    P(*i*);

- Assume that P(*i*) takes time *t*, where *t* is independent of *i*
- Running time of a for-loop is at most the running time of the statements inside the for-loop times number of iterations

$$T(n) = nt$$

- This approach is reasonable, provided n is positive
- If *n* is zero or negative the relationship $T(n) = nt$ is not valid

Repetition (Loops)

Analysing Nested Loops

for i = 0 to n do

for j = 0 to m do

P(j);

- Assume that P(j) takes time t, where t is independent of i *and* j
- Start with outer loop:
  - How many iterations? n
  - How much time per iteration? Need to evaluate inner loop
- Analyze inside-out. Total running time is running time of the statement multiplied by product of the sizes of all the for-loops

T(n) = nmt

Repetition (Loops)

Analysing Nested Loops

for i = 0 to n do

for j = 0 to i do

P(j);

- Assume that P(j) takes time t, where t is independent of i *and* j
- How do we analyze the running time of an algorithm that has complex nested loop?
- The answer is we write out the loops as summations and then solve the summations. To convert loops into summations, **we work from inside-out.**

$$T(n) = n + \sum_{i=0}^{n} r_i + t \sum_{i=0}^{n} r_i$$

$$= n + n(n+1)/2 + tn(n+1)/2$$

Analysis Example

Algorithm:	Number of times executed
1. n = read input from user	1
2. sum = 0	1
3. i = 0	1
4. while i < n	n
5. number = read input from user	n or $\sum_{i=0}^{n-1} 1$
6. sum = sum + number	n or $\sum_{i=0}^{n-1} 1$
7. i = i + 1	n or $\sum_{i=0}^{n-1} 1$
8. mean = sum / n	1

The computing time for this algorithm in terms on input size n is:

$T(n) = 1 + 1 + 1 + n + n + n + n + 1$

$T(n) = 4n + 4$

Another Analysis Example

```
i=1 ..............................1
while (i < n)................n-1
          a=2+g...............n-1
          i=i+1 ................n-1
  if (i<=n)......................1
          a=2 ....................1
  else
          a=3.....................1
```

$T(n) = 1 + 3(n-1) + 1 + 1$

$= 3n$

Another Analysis Example

i=1................................?	1
while (i<=10)................?	10
i=i+1..................?	10
i=1 ....?	1
while (i<=n)...................?	n
a=2+g ....?	n
i=i+1 ....?	n
if (i<=n)........................ ?	1
a=2 .... ?	1
else	
a=3..................... ?	1

T (n) = ? **T (n) = 3n +24**

Asymptotic Growth Rate

- Changing the hardware/software environment
  - Affects T(n) by constant factor, but does not alter the growth rate of T(n)
- Algorithm complexity is usually very complex. The **growth** of the complexity functions is what is more important for the analysis and is a <u>suitable measure for the comparison of algorithms</u> with increasing input size n.
- Asymptotic notations like big-O, big-Omega, and big-Theta are used to compute the complexity because different implementations of algorithm may differ in efficiency.
- The big-Oh notation gives an upper bound on the growth rate of a function.
- The statement "f(n) is O(g(n))" means that the growth rate of f(n) is no more than the growth rate of g(n).
- We can use the big-Oh notation to rank functions according to their growth rate.

Asymptotic Growth Rate

Two reasons why we are interested in asymptotic growth rates:

- ***Practical purposes:*** For large problems, when we expect to have big computational requirements
- ***Theoretical purposes:*** concentrating on growth rates **frees us** from some important issues:
  - **fixed costs** (e.g. switching the computer on!), which may dominate for a small problem size but be largely irrelevant
  - **machine and implementation details**
  - The growth rate will be a compact and **easy to understand the function**

Properties of Growth-Rate Functions

Example: 5n + 3

Estimated running time for different values of n:

n = 10	=> 53 steps
n = 100	=> 503 steps
n = 1,000	=> 5003 steps
n = 1,000,000	=> 5,000,003 steps

As "n" grows, the number of steps grow in *linear* proportion to **n** for this function *"Sum"*

What about the "+3" and "5" in 5n+3?

- As n gets large, the +3 becomes insignificant
- 5 is inaccurate, as different operations require varying amounts of time and also **does not have any significant importance**

What is fundamental is that the time is *linear* in **n**.

Asymptotic Algorithm Analysis

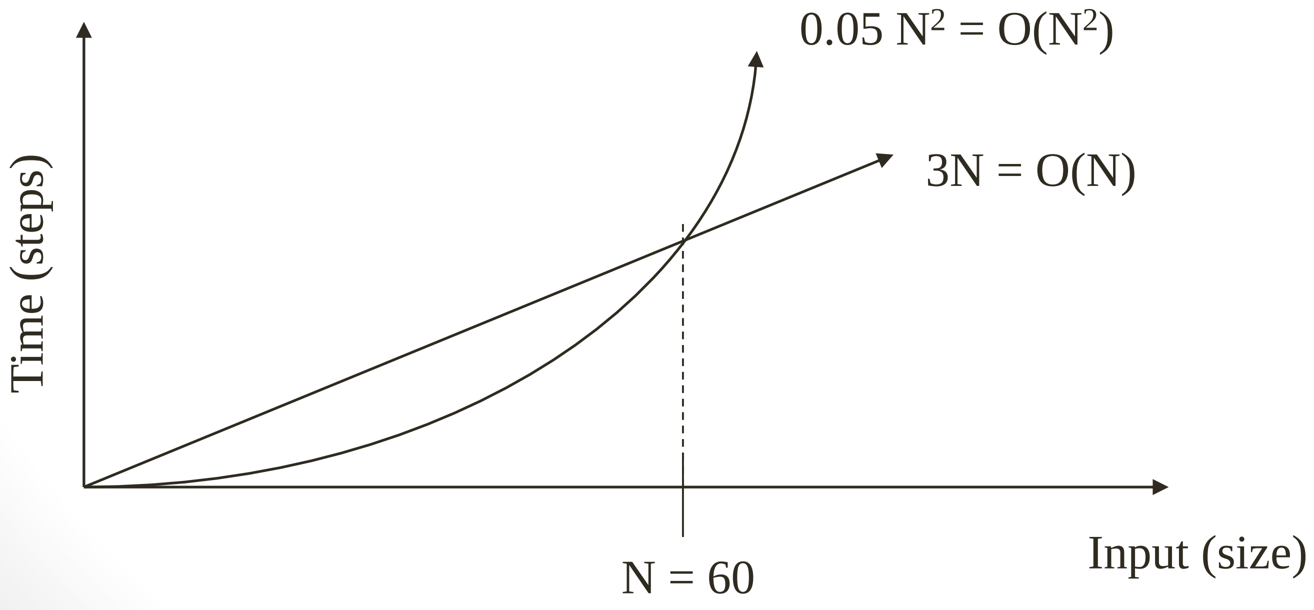
- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size n
  - We express this function with big-Oh notation
- Example: An algorithm executes $T(n) = 2n^2 + n$ elementary operations. We say that the algorithm runs in $O(n^2)$ time
- Growth rate is not affected by constant factors or lower-order terms so these terms can be dropped
- The $2n^2 + n$ time bound is said to **"grow asymptotically"** like n^2
- This gives us an approximation of the complexity of the algorithm
- Ignores lots of (machine dependent) details

Algorithm Efficiency

Measuring efficiency of an algorithm

- do its analysis i.e. growth rate.
- Compare efficiencies of different algorithms for the same problem.

As inputs get larger, any algorithm of a smaller order will be more efficient than an algorithm of a larger order



Important Functions

These functions often appear in algorithm analysis:

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	$N \log N$
N^2	Quadratic
N^3	Cubic
2^N	Exponential
$N!$	
N^N	

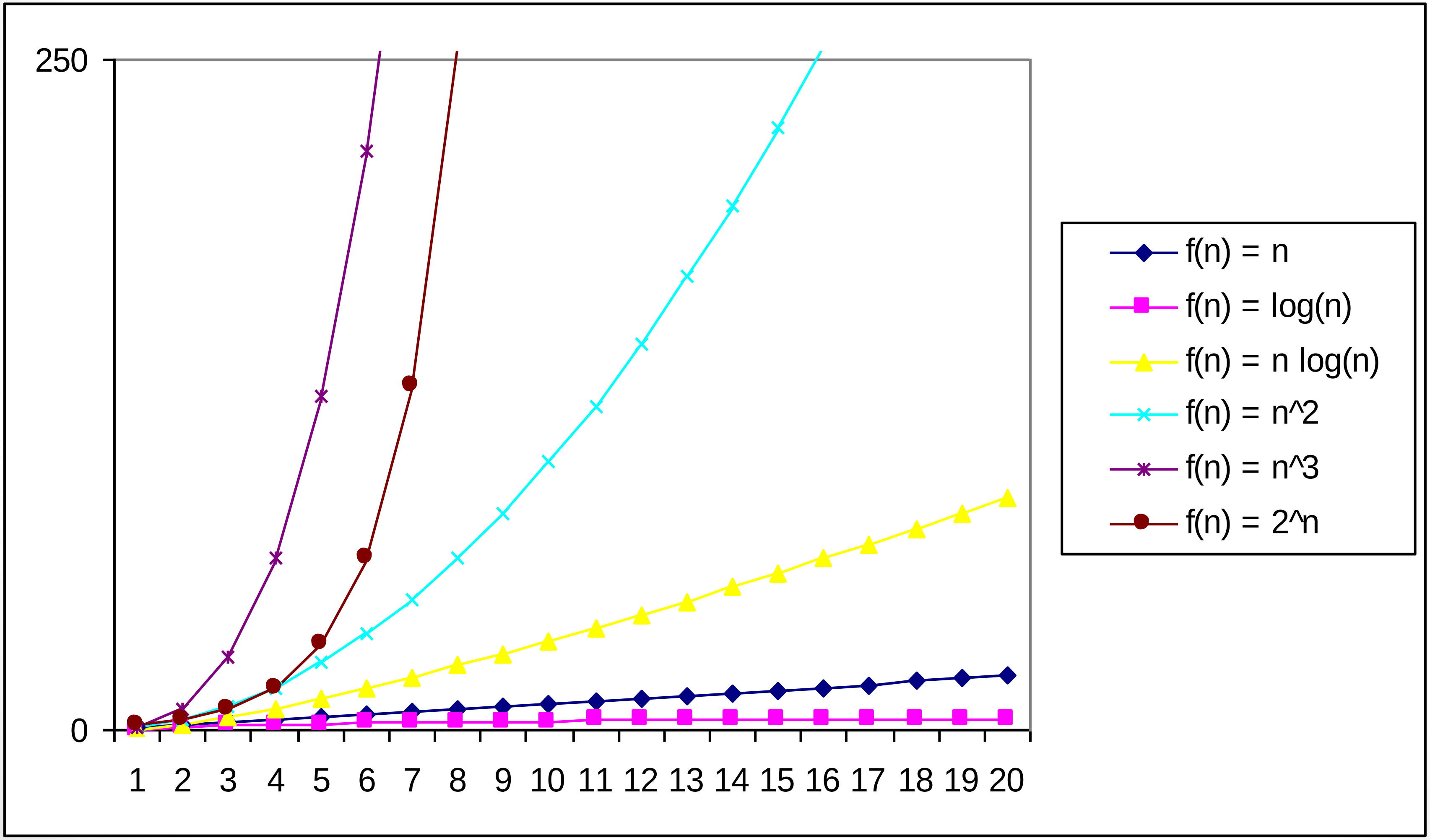
Functions in order of increasing growth rate

A comparison of Growth-Rate Functions

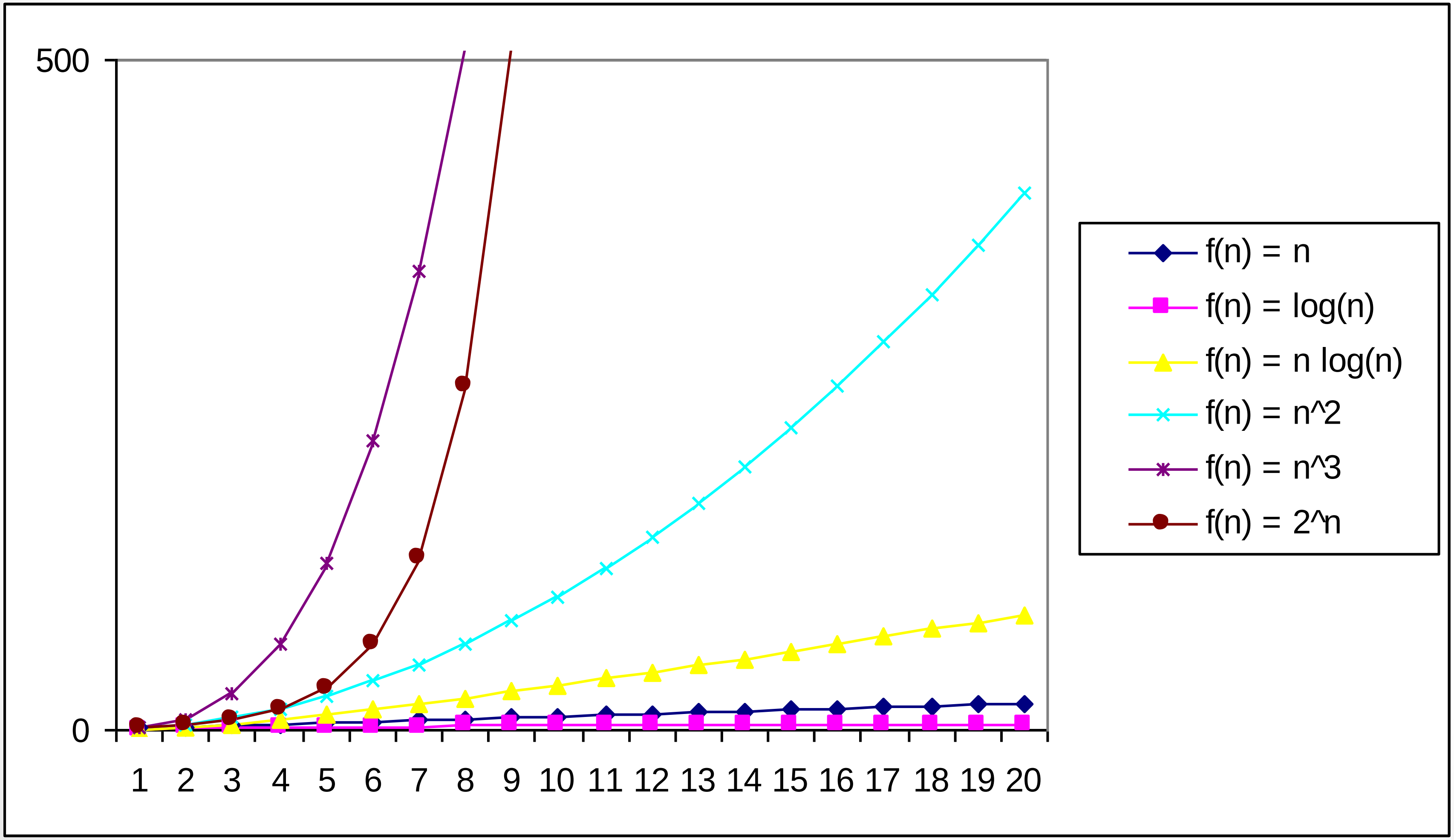
Size does Matter:
What happens if we increase the input size N?

Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

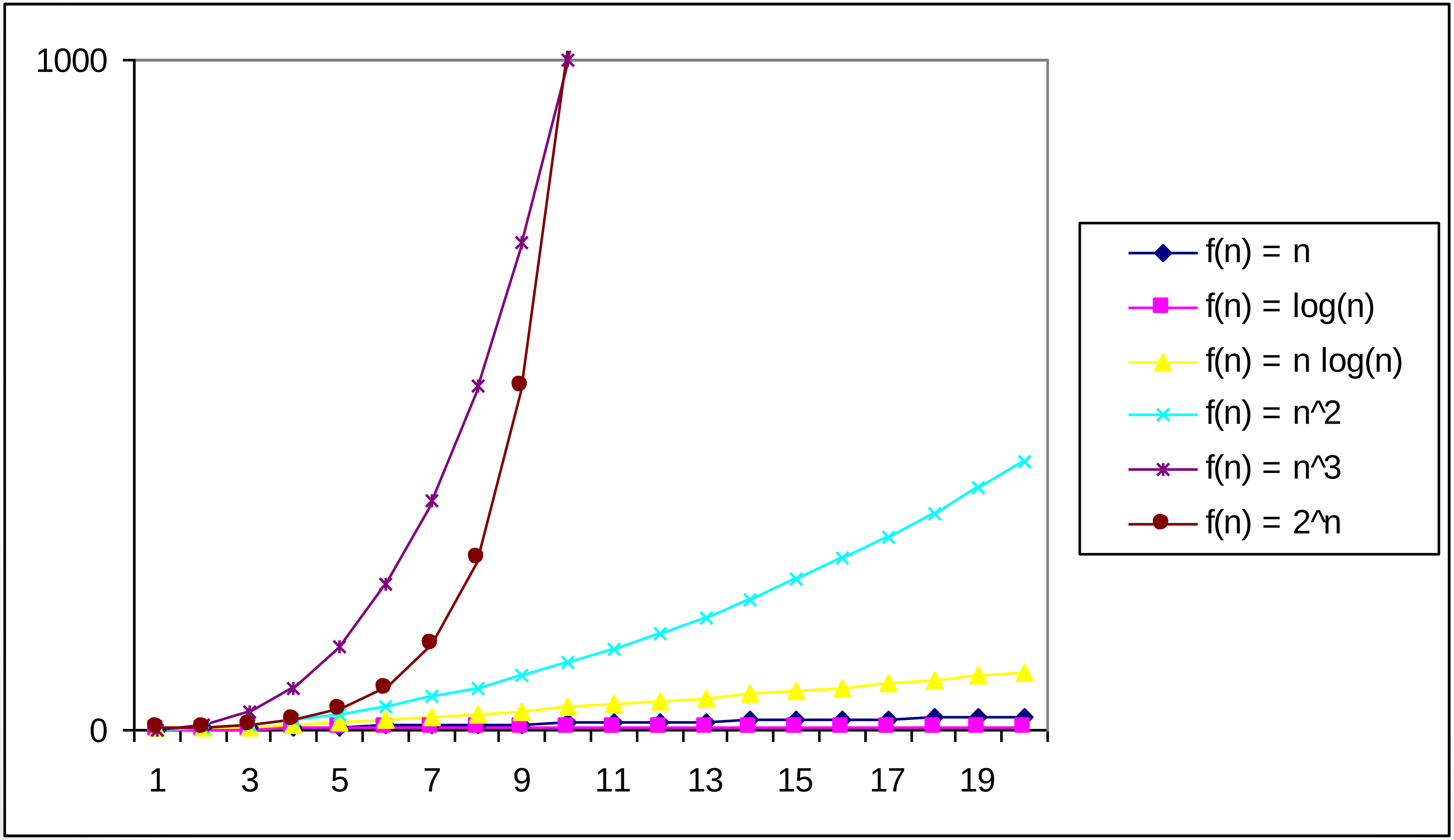
A comparison of Growth-Rate Functions



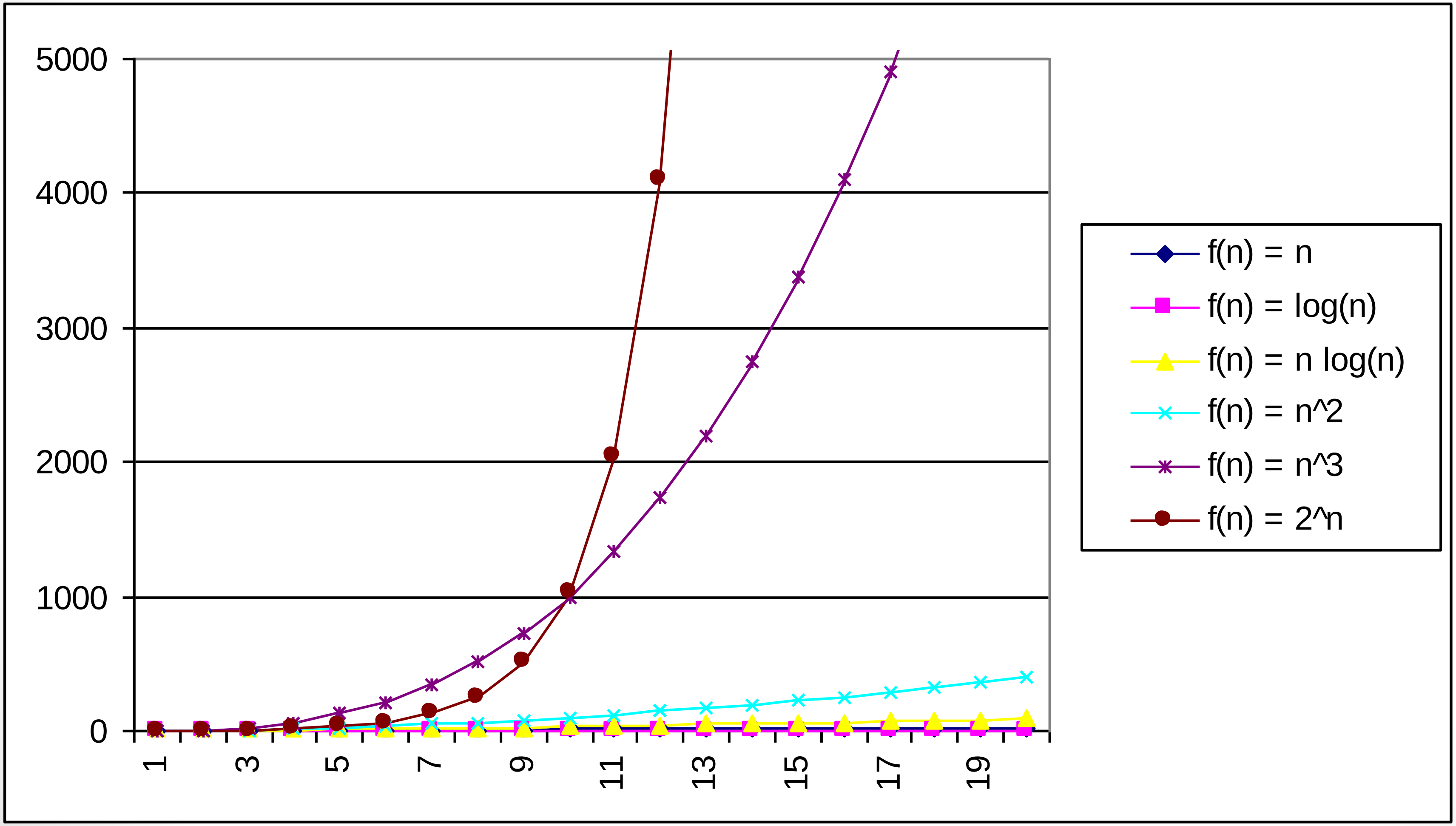
A comparison of Growth-Rate Functions



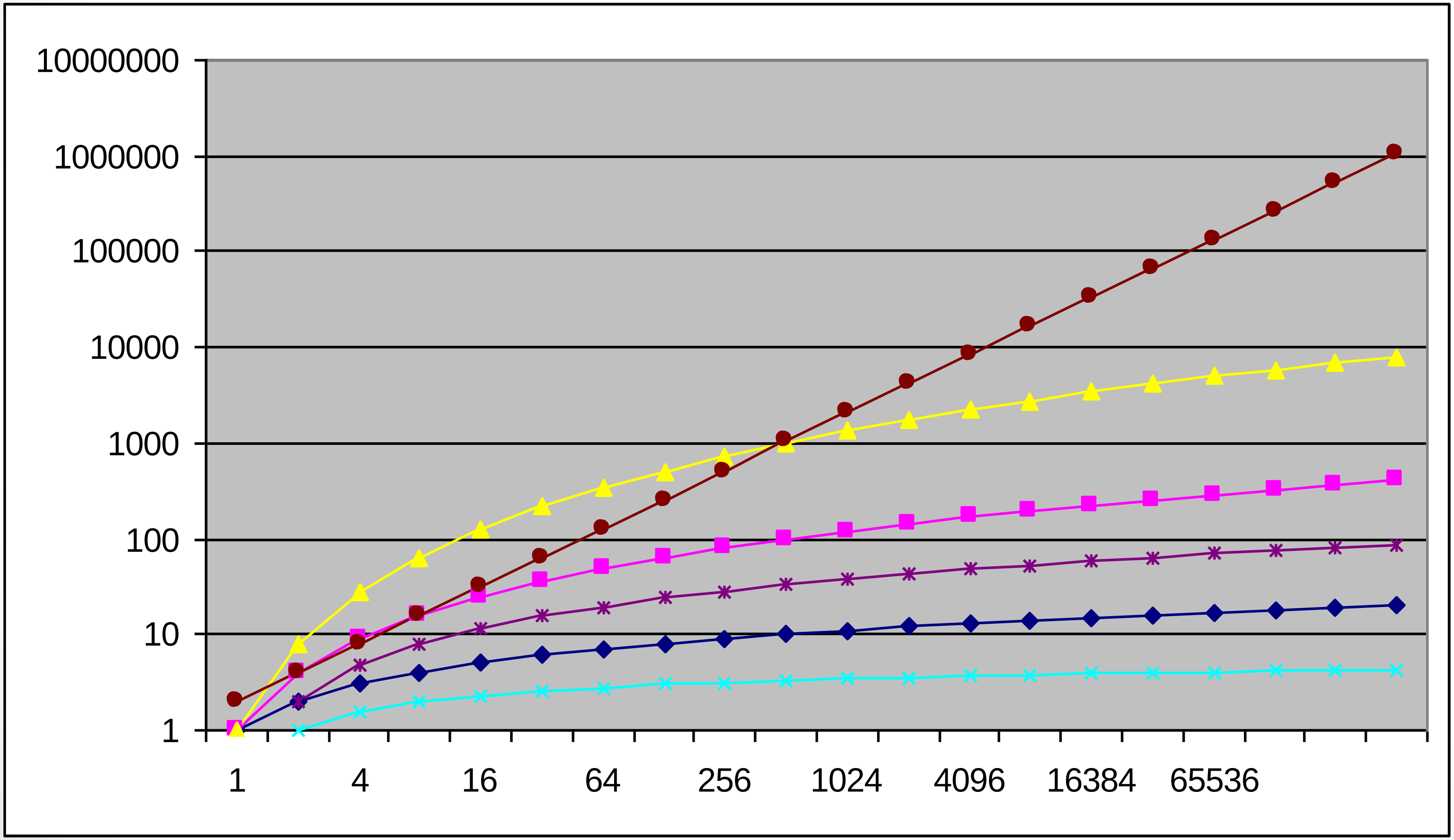
A comparison of Growth-Rate Functions



A comparison of Growth-Rate Functions



A comparison of Growth-Rate Functions



Performance Classification

f(*n*)	Classification
1	*Constant*: **run time is fixed, and does not depend upon n**. Most instructions are executed once, or only a few times, regardless of the amount of information being processed
`log n`	*Logarithmic*: **when *n* increases, so does run time, but much slower**. Common in programs which solve large problems by transforming them into smaller problems.
n	*Linear*: **run time varies directly with *n***. *Typically, a small amount of processing is done on each element.*
`n log n`	**When *n* doubles, run time slightly more than doubles**. *Common in programs which break a problem down into smaller sub-problems, solves them independently, then combines solutions*
n^2	***Quadratic*: when *n* doubles, runtime increases fourfold.** Practical only for small problems; typically the program processes all pairs of input (e.g. in a double nested loop).
n^3	***Cubic*: when n doubles, runtime increases eightfold**
2^n	*Exponential*: **when n doubles, run time squares**. This is often the result of a natural, “brute force” solution.

Running Time vs. Time Complexity

- Running time is how long it takes a program to run.
- **Time complexity is a description of the asymptotic behavior of running time as input size tends to infinity.**
- The exact running time might be $2036.n^2 + 17453.n + 18464$ but you can say that the running time "is" $O(n^2)$, because that's the formal(idiomatic) way to describe complexity classes and big-O notation.
- Infact, the running time is not a complexity class, IT'S EITHER A DURATION, OR A FUNCTION WHICH GIVES YOU THE DURATION. "Being $O(n^2)$" is a mathematical property of that function, not a full characterization of it.

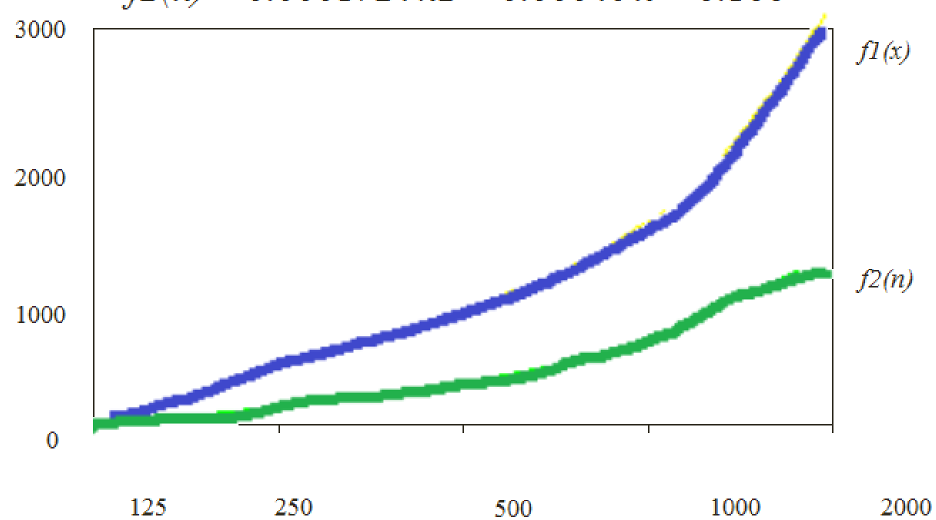
Example:
Running Time to Sort Array of 2000 Integers

Computer Type	Desktop	Server	Mainframe	Supercomputer
Time (sec)	51.915	11.508	0.431	0.087

Array Size	Desktop	Server
125	12.5	2.8
250	49.3	11.0
500	195.8	43.4
1000	780.3	172.9
2000	3114.9	690.5

$$f1(n) = 0.0007772\,n2 + 0.00305\,n + 0.001$$

$$f2(n) = 0.0001724\,n2 + 0.00040\,n + 0.100$$



Analysis of Results

$f(n) = a\,n^2 + b\,n + c$
where a = 0.0001724, b = 0.0004 and c = 0.1

n	**f(n)**	**$a\,n^2$**	**% of n^2**
125	2.8	2.7	94.7
250	11.0	10.8	98.2
500	43.4	43.1	99.3
1000	172.9	172.4	99.7
2000	690.5	689.6	99.9

Model of Computation

Drawbacks:

- poor assumption that each basic operation takes constant time
  - Adding, Multiplying, Comparing etc.

Finally what about Our Model?

- With all these weaknesses, our model is not so bad because
  - We have to give the comparison, not absolute analysis of any algorithm.
  - We have to deal with large inputs not with the small size

- Model seems to work well describing computational power of modern nonparallel machines

Can we do Exact Measure of Efficiency ?

- Exact, not asymptotic, measure of efficiency can be sometimes computed but it usually requires certain assumptions concerning implementation

Complexity Examples

What does the following algorithm compute?

procedure who_knows($a_1, a_2, \ldots, a_n$: integers)
m := 0
for i := 1 to n-1
    for j := i + 1 to n
        if $|a_i - a_j| > m$ **then** m := $|a_i - a_j|$

{m is the maximum difference between any two numbers in the input sequence}

Comparisons: n-1 + n-2 + n-3 + … + 1

$$= n*(n-1)/2 = 0.5n^2 - 0.5n$$

Time complexity is $O(n^2)$.

Complexity Examples

Another algorithm solving the same problem:

procedure max_diff(a_1, a_2, …, a_n: integers)
min := a1
max := a1
for i := 2 to n
    if a_i < min **then** min := a_i
    else if a_i > max **then** max := a_i
m := max - min

Comparisons: 2n + 2

Time complexity is O(n).

Prerequisite Review :
Mathematics
Data Structures

Notations

- Floor $\lfloor x \rfloor$ and Ceiling $\lceil x \rceil$ ➔ $\lfloor 3.4 \rfloor = 3$ and $\lceil 3.4 \rceil = 4$
- open interval (a, b) is $\{x \in \Re \mid a < x < b\}$
- closed interval [a, b] is $\{x \in \Re \mid a \leq x \leq b\}$
- Set is a collection of not ordered, not repeated elements e.g {a, b, c}
  - Operations: union, intersection, difference, complement
- Membership: $x \in X$ ➔ x is a member of X e.g. $a \in \{a, b, c\}$
- Existential Quantifier (∃) ➔ $\exists\, x\ (x \geq x^2)$ is true since x=0 is a solution
- Universal Quantifier (∀) ➔ $\forall\, x\ (x^2 \geq 0)$ is true for all values of x

Exponent Review

$$x^a x^b = x^{a+b}$$

$$\frac{x^a}{x^b} = x^{a-b}$$

$$(x^a)^b = x^{ab}$$

$$x^0 = 1$$

$$x^n + x^n = 2x^n$$

$$2^n + 2^n = 2 * 2^n = 2^{n+1}$$

$$(xy)^a = x^a y^a$$

$$x^{-m} = \frac{1}{x^m}$$

Logarithm Review

$$x = \log_b y \quad \Rightarrow \quad b^x = y$$

$$\log_a b = \frac{\log_c b}{\log_c a}$$

$$\log ab = \log a + \log b$$

$$\log \frac{a}{b} = \log a - \log b$$

$$\log(a^b) = b \log a$$

$$\lg 1 = 0, \lg 2 = 1, \lg 1024 = 10$$

$$x^{\log_y n} = n^{\log_y x}$$

$$b = a^{\log_a b}$$

$$b^c = a^{c*\log_a b}$$

$$\log_x 1 = 0$$

Prove that $3^{\log_4^n} = n^{\log_4^3}$

Proof

$$3^{\log_4^n} = n^{\log_4^3}$$

$$\Leftrightarrow \quad \log_3(3^{\log_4^n}) = \log_3(n^{\log_4^3})$$

$$\Leftrightarrow \quad \log_4^n \cdot \log_3^3 = \log_4^3 \cdot \log_3^n$$

$$\Leftrightarrow \quad \log_4^n = \log_4^3 \cdot \log_3^n$$

$$\Leftrightarrow \quad \log_4^n = \log_4^n \qquad \because \log_a^b \cdot \log_b^c = \log_a^c$$

Summation Algebra

$$\sum_{i=1}^{N} x_i$$ The sum of numbers from 1 to N; e.g 1 + 2 + 3 … + N

$$\sum_{i=1}^{5} x_i^2$$ Suppose our list has 5 number, and they are 1, 3, 2, 5, 6.

Then resulting summation will be $1^2 + 3^2 + 2^2 + 5^2 + 6^2 = 75$

$$\sum_{i=x}^{y} a = (y - x + 1)a \quad \text{or} \quad \sum_{i=1}^{N} a = Na$$ **The First constant Rule**

$$\sum_{i=1}^{N} ax_i = a\sum_{i=1}^{N} x_i$$ **The Second constant Rule**

$$\sum_{i=1}^{N} (x_i \pm y_i) = \sum_{i=1}^{N} x_i \pm \sum_{i=1}^{N} y_i$$ **The Distributive Rule**

$$\sum_{i=1}^{3}\sum_{j=1}^{3} x_{ij} = x_{11} + x_{12} + x_{13} + x_{21} + x_{22} + x_{23} + x_{31} + x_{32} + x_{33}$$

Double Summation

Summation Algebra: Practice Questions

$$\sum_{i=0}^{6} 2 \Longrightarrow 14$$

$$\sum_{i=1}^{N} 6x_i y \Longrightarrow 6y\sum_{i=1}^{N} x_i$$

$$\sum_{i=0}^{2}\sum_{j=0}^{3} 2i+3j \Longrightarrow 78$$

Arithmetic Series/Sequence

- A sequence in which **the difference between one term and the next is a constant**. (add a fixed value each time ... on to infinity)
  - Generalized form: {a, a+d, a+2d, a+3d, ... }
  - where **a** is the first term, and **d** is the **common difference** between the two terms

- **Example:** 1, 4, 7, 10, 13, 16, 19, 22, 25, ...
- N^{th} term would be $x_n = a + d(n-1)$
- The summation of Arithmetic Sequence is $S_N = \frac{n}{2}[2a_1 + (n-1)d]$

- $\mathbf{S_n}$ is the sum of the first n terms in a sequence
- $\mathbf{a_1}$ is the first term in the sequence
- **d** is the common difference in the arithmetic sequence
- **n** is the number of terms you are adding up

Geometric Series/Sequence

- A sequence in which each term is found by **multiplying the previous term by a constant**.
  - Generalized form: {a, ar, ar^2, ar^3, ... }
  - where **a** is the first term, and **r** is the **common ratio (r ≠ 0)**

- **Example:** 2, 4, 8, 16, 32, 64, 128, 256, ...
- N^{th} term would be $x_n = ar^{(n-1)}$
- The summation of geometric Sequence is $S_N = a_1 \frac{[1 - r^n]}{1 - r}$

- $\mathbf{S_n}$ is the sum of the first n terms in a sequence
- $\mathbf{a_1}$ is the first term in the sequence
- **r** is the common ratio in the geometric sequence
- **n** is the number of terms you are adding up

Series Review

Arithmetic Series Formulas:

$$a_n = a_1 + (n-1)d$$

$$a_i = \frac{a_{i-1} + a_{i+1}}{2}$$

$$S_n = \frac{a_1 + a_n}{2} \cdot n$$

$$S_n = \frac{2a_1 + (n-1)d}{2} \cdot n$$

Geometric Series Formulas:

$$a_n = a_1 \cdot q^{n-1}$$

$$a_i = \sqrt{a_{i-1} \cdot a_{i+1}}$$

$$S_n = \frac{a_n q - a_1}{q-1}$$

$$S_n = \frac{a_1(q^n - 1)}{q-1}$$

$$S = \frac{a_1}{1-q} \qquad for \quad -1 < q < 1$$

Series Examples

$$1+2+3+\ldots+n=\sum_{i=1}^{n} i=\frac{n(n+1)}{2}$$

$$1+4+9+\ldots+n^2=\sum_{i=1}^{n} i^2=\frac{n(n+1)(2n+1)}{6}$$

$$1+x+x^2+\cdots+x^n=\frac{1-x^{n+1}}{1-x} \quad \text{for } x \neq 1$$

$$1+x+x^2+\cdots=\frac{1}{1-x} \quad \text{for } |x| < 1$$

Permutation and Combination

Permutation

Set of n elements is an arrangement of the elements in given order

e.g. Permutation for elements a, b, c are **abc, acb, bac, bca, cab, cba**

- n! permutation exist for a set of elements

  5! = 120 permutation for 5 elements

Combination

Set of n elements is an arrangement of the elements in any order

e.g. Combination for elements a, b, c is **abc**

	Order	Number
Permutation	matters	$P(n,k) = \frac{n!}{(n-k)!}$
Combination	does not matter	$C(n,k) = \frac{n!}{(n-k)!k!}$

Sample Space

A set **"S"** consisting of all possible outcomes that can result from a random experiment (real or conceptual), can be defined as the sample space for that experiment.

- Each possible outcome is called a sample point in that space.

Example: The sample space for an experiment of tossing a coin is expressed as S = {H, T}, as two possible outcomes are possible: a head (H) or a tail (T). 'H' and 'T' are the two sample points.

Example: The sample space for tossing two coins at once (or tossing a coin twice) will contain four possible outcomes and is denoted by S = {HH, HT, TH, TT}.
In this example, clearly, S is the Cartesian product $A \times A$, where A = {H, T}.

Events

Any subset of a sample space S of a random experiment, is called an event. In other words, an event is an individual outcome or any number of outcomes (sample points) of a random experiment.

Simple event is an event that contains exactly one sample point.

Compound event is an event that contains more than one sample point, and is produced by the union of simple events.

Occurrence of an event: An event A is said to occur if and only if the outcome of the experiment corresponds to some element of A.

Example: The occurrence of a 6 when a die is thrown, is a simple event, while the occurrence of a sum of 10 with a pair of dice, is a compound event, as it can be decomposed into three simple events (4, 6), (5, 5) and (6, 4).

Example: Suppose an event A={2, 4, 6} represents occurrence of an even number when the dice is thrown. If a dice shows '2', '4' or '6', we say that the event A of our interest has occurred.

Events

Complementary Event is the event "not-A", denoted by $\bar{A}$ or A^c, and is called the negation of A.

Example: If we toss a coin once, then the complement of "heads" is "tails". If we toss a coin four times, then the complement of "at least one head" is "no heads".

A sample space consisting of n sample points can produce 2^n different subsets of simple and compound events.

Example: Consider a sample space S containing 3 sample points, i.e. $S = \{a, b, c\}$. Then the $2^3 = 8$ possible subsets are:

$$\phi,\ \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}.$$

Each of these subsets is an event.

The subset $\{a, b, c\}$ is the sample space itself and is also an event. It always occurs and is known as the certain or sure event.

The empty set ϕ is also an event, sometimes known as impossible event, because it can never occur.

Events

Mutually Exclusive Events Two events A and B of a single experiment are said to be mutually exclusive or disjoint if and only if they cannot both occur at the same time; i.e. they have no points in common.

Example: When we toss a coin, we get either a head or a tail, but not both at the same time. The two events head and tail are therefore mutually exclusive.

Probability Theory

If a random experiment can produce n mutually exclusive and equally likely outcomes, and if m out to these outcomes are considered favorable to the occurrence of a certain event A, then the probability of the event A, denoted by P(A), is defined as the ratio m/n.

$$P(A) = \frac{m}{n} = \frac{Number\ of\ \ favourable outcomes}{Total\ number of\ \ possible\, outcomes}$$

Formal Definition: Let S be a sample space with the sample points E_1, E_2, … E_i, …E_n. Each sample point is assigned a real number, denoted by the symbol $P(E_i)$, and called the probability of E_i, that must satisfy the following basic axioms:

- Axiom 1: For any event E_i, $0 \leq P(E_i) \leq 1$.
- Axiom 2: P(S) = 1 for the sure event S.
- Axiom 3: If A and B are mutually exclusive events (subsets of S), then P $(A \cup B) = P(A) + P(B)$.

Probability Theory

Example: If a card is drawn from an ordinary deck of 52 playing cards, find the probability that

i) the card is a red card,

ii) the card is a 10.

Solution: The total number of possible outcomes is 13+13+13+13 = 52, and we assume that all possible outcomes are equally likely.

(It is well-known that an ordinary deck of cards contains 13 cards of diamonds, 13 cards of hearts, 13 cards of clubs, and 13 cards of spades.)

(i) Let A represent the event that the card drawn is a red card. Then the number of outcomes favorable to the event A is 26 (since the 13 cards of diamonds and the 13 cards of hearts are red).

$$P(A) = \frac{m}{n} = \frac{Number\ of\ \ favourable\ outcomes}{Total\ \ number\ of\ \ possible\ outcomes} = \frac{26}{52} = \frac{1}{2}$$

(ii) Let B denote the event that the card drawn is a 10. Then the number of outcomes favorable to B is 4 as there are four 10's

$$P(B) = \frac{4}{52} = \frac{1}{13}.$$

Probability Theory

Example: A fair coin is tossed three times. What is the probability that at least one head appears?

Solution: The sample space for this experiment is:

S = {HHH, HHT, HTH, THH, HTT, THT, TTH, TTT}

and thus the total number of sample points is n(S) = 8.

Let A denote the event that at least one head appears. Then:

A = {HHH, HHT, HTH, THH, HTT, THT, TTH}

and therefore n(A) = 7.

$$P(A) = \frac{n(A)}{n(S)} = \frac{7}{8}.$$

Probability Theory

Example: Four items are taken at random from a box of 12 items and inspected. The box is rejected if more than 1 item is found to be faulty. If there are 3 faulty items in the box, find the probability that the box is accepted.

Solution: The sample space for this experiment is number of possible combinations for selecting 4 out of 12 items from the box.

$$\binom{12}{4} = \frac{n!}{(n-k)!k!} = \frac{12!}{(8)!4!} = 495$$

The box contains 3 faulty and 9 good items. The box is accepted if there is (i) no faulty items, or (ii) one faulty item in the sample of 4 items selected.

Let A denote the event the number of faulty items chosen is 0 or 1. Then:

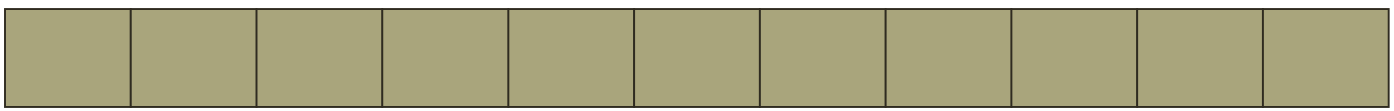
$$n(A) \quad = \binom{3}{0}\binom{9}{4} + \binom{3}{1}\binom{9}{3} = 126 + 252 = 378 \ \ sample \ \ po\text{int}\, s.$$

$$\therefore \quad P(A) = \frac{m}{n} = \frac{378}{495} = 0.76$$

Hence the probability that the box is accepted is 76%.

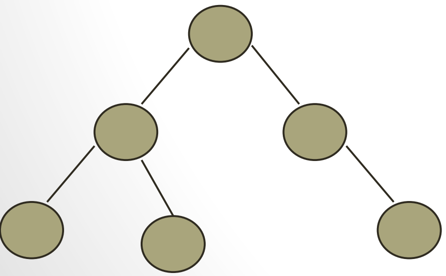
Data Structures Review

- Data structure is the logical or mathematical model of a particular organization of data.
- Data structures let the input and output be represented in a way that can be handled efficiently and effectively.
- Data may be organized in different ways.

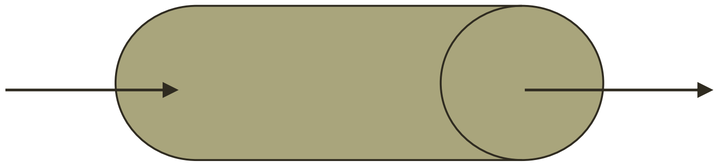


Array

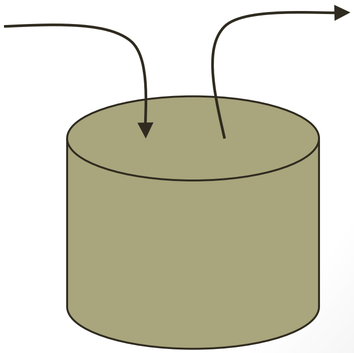
Linked list



Graph/Tree



Queue



Stack

Arrays

Customer
Jamal
Sana
Saeed
Farooq
Salman
Danial

Linear Arrays

	Customer	Salesperson
1	Jamal	Tony
2	Sana	Tony
3	Saeed	Nadia
4	Farooq	Owais
5	Salman	Owais
6	Danial	Nadia

Two Dimensional Arrays

Example: Linear Search Algorithm

- Given a linear array **A** containing **n** elements, locate the position of an Item '**x**' or indicate that '**x**' does not appear in **A**.
- The linear search algorithm solves this problem by comparing '**x**', one by one, with each element in **A**. That is, we compare ITEM with A[1], then A[2], and so on, until we find the location of '**x**'.

LinearSearch(A, x)	**Number of times executed**	
$i \leftarrow 1$	1	
while ($i \leq n$ and $A[i] \neq x$)	n	
$i \leftarrow i+1$	n	
if $i \leq n$	1	**T(n) = 2n+3**
return true	1	
else		
return false	1	

Best/Worst Case

Best case: **'x'** is located in the first location of the array and loop executes only once

$$T(n) = 1 + n + n + 1 + 1$$
$$= 1+1+0+1+1$$
$$= O(1)$$

Worst case: **'x'** is located in last location of the array or is not there at all.

$$T(n) = 1 + n + n + 1 + 1$$
$$= 2n + 3$$
$$= O(n)$$

Average case

Average Case: Assume that it is equally likely for 'x' to appear at any position in array **A**,. Accordingly, the number of comparisons can be any of the numbers 1,2,3,..., n, and each number occurs with probability p = 1/n.

T(n) = 1.1/n + 2. 1/n +……….+ n.1/n

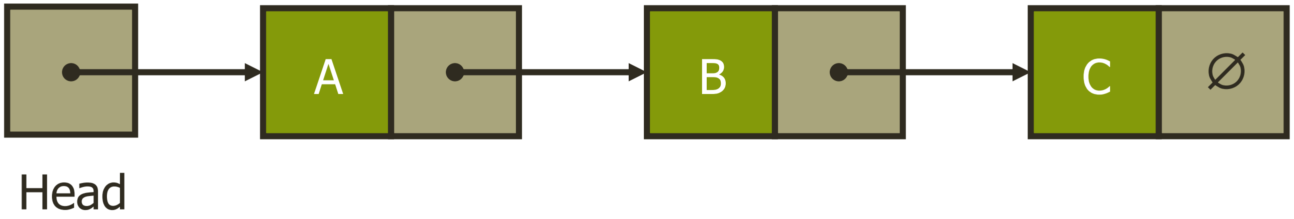
= (1+2+3+……+n).1/n

= [n(n+1)/2] 1/n = n+1/2

= O(n)

This agrees with our intuitive feeling that the average number of comparisons needed to find the location of 'x' is approximately equal to half the number of elements in the **A** list.

Linked List

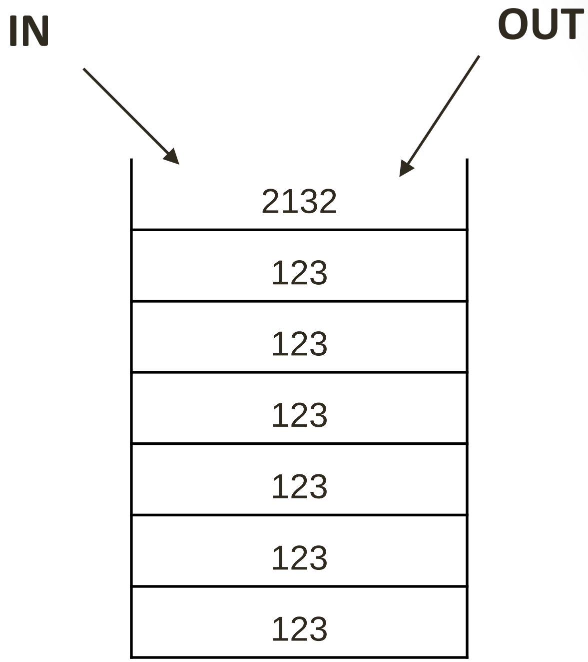


- A series of connected nodes
  - Each node contains a piece of data and a pointer to the next node

Operations	Average Case	Worst Case
Insert	O(1)	O(1)
Delete	O(1)	O(1)
Search	O(N)	O(N)

Stack

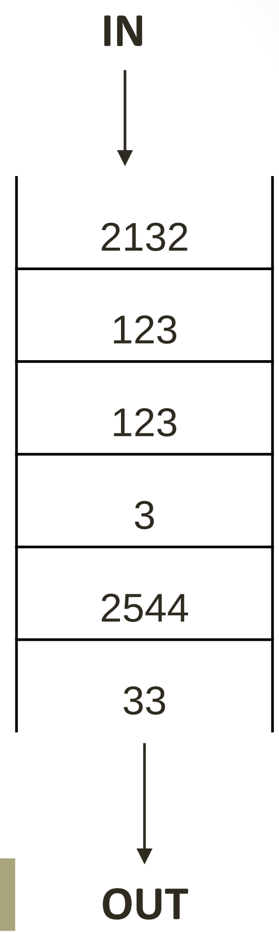
- LIFO
  - Implemented using linked-list or arrays



Operations	Average Case	Worst Case
Push	O(1)	O(1)
Pop	O(1)	O(1)
IsEmpty	O(1)	O(1)

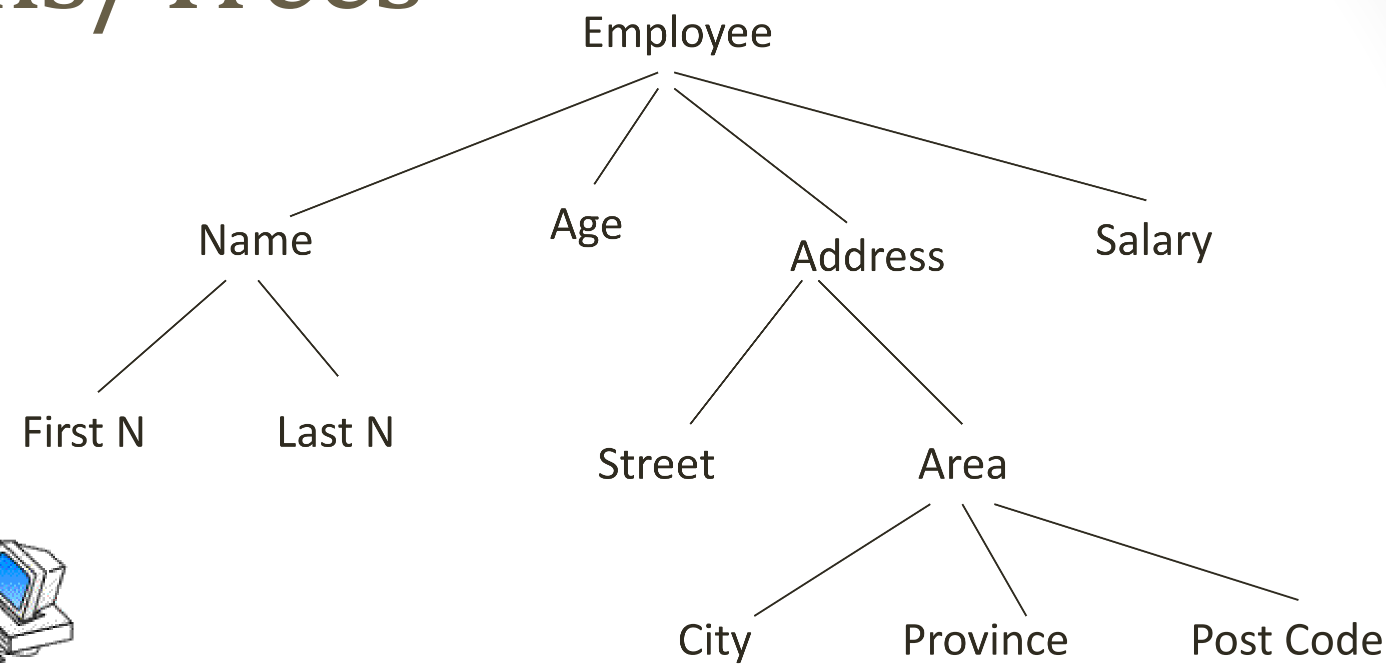
Queue

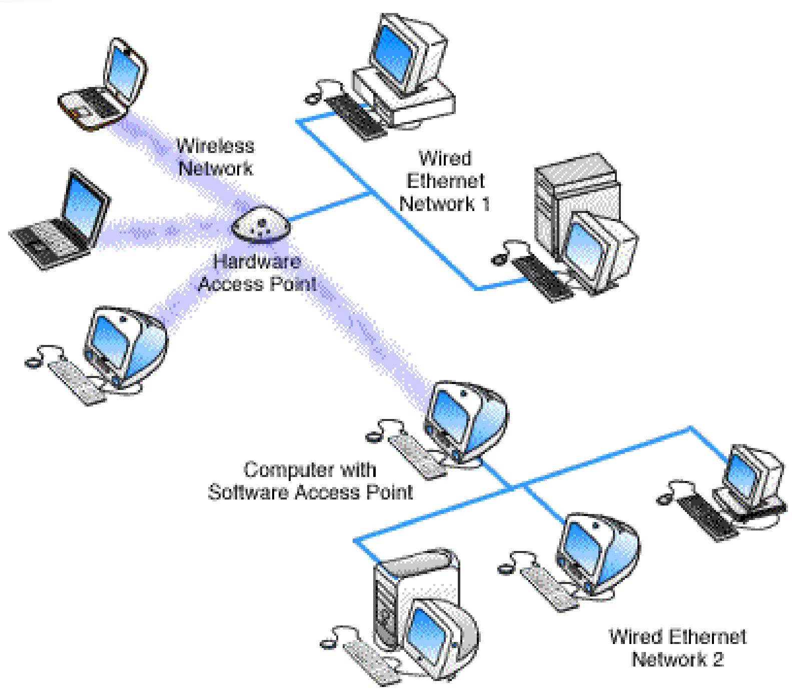
- FIFO
  - Implemented using linked-list or arrays



Operations	Average Case	Worst Case
Enqueue	O(1)	O(1)
Dequeue	O(N)	O(N)

Graphs/Trees





Approaches to Algorithms Analysis (Nested Loops)

i) Top-Down Approach

ii) Bottom-Up Approach

Example: Top-down vs. Bottom-up

for i → 1 to n do
    for j → 1 to m do
        a→b

Bottom-up Solution: Inner Loop

for j → 1 to m do .... m
    a→b .... m
Inner(m)= m+m = O(m)

for i → 1 to n do .... n
    Inner(m) .... m

T(n) = n+Inner(m)(n) ➔ = n+O(m)(n)
    = O(nm)
    = $O(n^2)$ if m>=n

Top-down Solution:

for i → 1 to n do .... n
    for j → 1 to m do .... nm
        a→b .... nm

T(n) = n + 2mn
    = O(nm)
    = $O(n^2)$ if m>=n

Example: LOOP Analysis (Bottom Up Approach)

```
NESTED-LOOPS()
1   for i ← 1 to n
2   do
3       for j ← 1 to 2i
4       do k = j ...
5           while (k ≥ 0)
6           do k = k − 1 ...
```

Step-1: Bottom while Loop (line 5 and 6)

while(j) = $\sum_{k=0}^{j} 1 = j+1$

Step-2: Inner For Loop (line 3 and 4)

for(i) = $\sum_{j=1}^{2i} while(j) = \sum_{j=1}^{2i} j+1$

= $\sum_{j=1}^{2i} j + \sum_{j=1}^{2i} 1$

= (2i(2i+1)/2) + 2i ➔ $2i^2 + 3i$

Step-3: Outer For Loop (Line 1)

T(n) = $\sum_{i=1}^{n} for(i)$ = $\sum_{i=1}^{n} (2i^2 + 3i)$

= $2\sum_{i=1}^{n} i^2 + 3\sum_{i=1}^{n} i$

= 2 [($2n^3+3n^2+n$)/6] + 3 [n(n+1)/2]

= O (n^3)

Quadratic Series

$$\sum_{i=1}^{n} i^2 = 1 + 4 + 9 + \ldots + n^2$$

$$\sum_{k=1}^{N} k^2 = \frac{N(N+1)(2N+1)}{6}$$

$$= \frac{2n^3 + 3n^2 + n}{6} = \Theta(n^3)$$