

Design & Analysis of Algorithms

Design

Why is design important?

Divide & conquer

Branch & bound

Graph theoretic

Heuristic

Greedy

Optimal

Exact

What is Algorithm Analysis For

- § Foundations of Algorithm Analysis and Data Structures.
- § Analysis:
 - How to predict an algorithm's performance
 - How well an algorithm scales up
 - How to compare different algorithms for a problem
- § Data Structures
 - How to efficiently store, access, manage data
 - Data structures effect algorithm's performance

HOW TO ANALYZE PROGRAMS

There are many criteria upon which we can judge a program, for instance:

- (1) Does it do what we want it to do?
- (2) Does it work correctly according to the original specifications of the task?
- (3) Is there documentation that describes how to use it and how it works?
- (4) Are subroutines created in such a way that they perform logical sub-functions?
- (5) Is the code readable?

The above criteria are all vitally important when it comes to writing software, most especially for large systems.

Other criteria which have a more direct relationship to performance,

- computing time
- storage requirements of the algorithms

Performance evaluation phases

- apriori estimates
- a posteriori testing

a priori estimation

Consider the statement

$x = x + 1$

Important to determine two numbers for this statement

- the amount of time a single execution will take
- number of times it is executed

The product of these numbers will be the total time taken by this statement

- To determine exactly how much time it takes to execute any command the following information are required

- (1) the machine we are executing on;
- (2) its machine language instruction set;
- (3) the time required by each machine instruction;
- (4) the translation a compiler will make from the source to the machine language

Second statistics is called the frequency count

- This may vary from data set to data set
- The hardest task in estimating frequency counts is to choose adequate samples of data.

To determine these figures

- Choose a real machine and an existing compiler or
- Define a hypothetical machine (with imaginary execution times)
 - make the times reasonably close to those of existing Hardware so that resulting figures would be representative

In both cases

- exact times would not apply to many machines or to any machine
- There would be the problem of the compiler, which could vary from machine to machine
- It is often difficult to get reliable timing figures because of
 - clock limitations
 - multi-programming or time sharing environment
- The difficulty of learning another machine language outweighs the advantage of finding "exact" fictitious times

a priori analysis

Case 1

$x = x + 1$

end

Case 2

for i = 1 to n do
 $x = x + 1$

end

Case 3

for i = 1 to n do
 for j = 1 to n do
 $x = x + 1$

end

end

Case 1

- The statement $x = x + 1$ is not contained within any loop either explicit or implicit
- Its frequency count is one

Case 2

- The same statement will be executed n times

Case 3

- The same statement will be executed n^2 times (assuming $n \geq 1$)

- 1, n, and n^2 are said to be in increasing orders of magnitude just as 1, 10, 100 would be if $n = 10$
- Concern is chiefly with determining the order of magnitude of an algorithm
- This means determining those statements which may have the greatest frequency count

Formulae such as

$$\sum_{1 \leq i \leq n} 1, \quad \sum_{1 \leq i \leq n} i, \quad \sum_{1 \leq i \leq n} i^2 \quad \text{often occur}$$

In the program segment of figure (c) the statement $x \leftarrow x + 1$ is executed

$$\sum_{1 \leq i \leq n} \sum_{1 \leq j \leq n} 1 = \sum_{1 \leq i \leq n} n = n^2 \text{ times}$$

Simple forms for the above three formulas are well known, namely

$$n, \quad \frac{n(n+1)}{2}, \quad \frac{n(n+1)(2n+1)}{6}$$

In general

$$\sum_{1 \leq i \leq n} i^k = \frac{n^{k+1}}{k+1} + \text{terms of lower degree}$$

Example Algorithms

- Two algorithms for computing the Factorial
- Which one is better?

```
int factorial (int n) {  
    if (n <= 1) return 1;  
    else return n * factorial(n-1);  
}
```

```
int factorial (int n) {  
    if (n<=1) return 1;  
    else {  
        fact = 1;  
        for (k=2; k<=n; k++)  
            fact *= k;  
        return fact;  
    }  
}
```

Examples of famous algorithms

- § Constructions of Euclid
- § Newton's root finding
- § Fast Fourier Transform (signal processing)
- § Compression (Huffman, Lempel-Ziv, GIF, MPEG)
- § DES, RSA encryption (network security)
- § Simplex algorithm for linear programming (optimization)
- § Shortest Path Algorithms (Dijkstra, Bellman-Ford)
- § Error correcting codes (CDs, DVDs)
- § TCP congestion control, IP routing (computer networks)
- § Pattern matching (Genomics)
- § Search Engines (www)

Role of Algorithms in Modern World

- s Enormous amount of data
 - Network traffic (telecom billing, monitoring)
 - Database transactions (Sales, inventory)
 - Scientific measurements (astrophysics, geology)
 - Sensor networks. RFID tags
 - **Radio frequency identification (RFID)** is a method of remotely storing and retrieving data using devices called **RFID tags**.
 - Bioinformatics (genome, protein bank)

A real-world Problem

- § Communication in the Internet
- § Message (email, ftp) broken down into IP packets.
- § Sender/receiver identified by IP address.
- § The packets are routed through the Internet by special computers called Routers.
- § Each packet is stamped with its destination address, but not the route.
- § Because the Internet topology and network load is constantly changing, routers must discover routes dynamically.
- § What should the Routing Table look like?

IP Prefixes and Routing

- § Each router is really a switch: it receives packets at several input ports, and appropriately sends them out to output ports.
- § Thus, for each packet, the router needs to transfer the packet to that output port that gets it closer to its destination.
- § Should each router keep a table: IP address x Output Port?
- § How big is this table?
- § When a link or router fails, how much information would need to be modified?
- § A router typically forwards several million packets/sec!

Data Structures

- § The IP packet forwarding is a Data Structure problem!
- § Efficiency, scalability is very important.
- § Similarly, how does Google find the documents matching your query so fast?
- § Uses sophisticated algorithms to create index structures, which are just data structures.
- § Algorithms and data structures are ubiquitous.
- § With the data glut created by the new technologies, the need to organize, search, and update MASSIVE amounts of information FAST is more severe than ever before.

Algorithms to Process these Data

- s Which are the top K sellers?
- s Correlation between time spent at a web site and purchase amount?
- s Which flows at a router account for $> 1\%$ traffic?
- s Did source S send a packet in last s seconds?
- s Send an alarm if any international arrival matches a profile in the database
- s Similarity matches against genome databases
- s Etc.

Algorithm Analysis

- To compare different algorithms for the same task
- To predict performance in a new environment
- To set values of algorithm parameters

Algorithm analysis

- Important factors in a precise analysis are usually outside a given programmer's domain of influence.
 - C programs translated into m/c code for a given computer
 - Many programs are extremely sensitive to input data, performance might fluctuate wildly depending on the input.
 - Two programs might not be comparable at all – one may run much more efficiently on one particular kind of I/P, the other runs efficiently under other circumstances

Why Efficient Algorithms Matter

- § Suppose $N = 10^6$
- § A PC can read/process N records in 1 sec.
- § But if some algorithm does $N*N$ computation, then it takes 1M seconds = 11 days!!!
- § 100 City **Traveling Salesman Problem**.
 - A supercomputer checking 100 billion tours/sec still requires 10^{100} years!
- § Fast **factoring** algorithms can break encryption schemes. Algorithms research determines what is safe code length. (> 100 digits)

How to Measure Algorithm Performance

- § What metric should be used to judge algorithms?
 - Length of the program (lines of code)
 - Ease of programming (bugs, maintenance)
 - Memory required
 - Running time

- § **Running time is the dominant standard.**
 - Quantifiable and easy to compare
 - Often the critical bottleneck

Analysis questions

- Forms of analysis
 - concrete /emprical
 - theoretical
- Criteria for analysis
 - running time
 - memory load
 - external factors

Concrete analysis

- Experimental: limited to test cases
- Must have implemented the object
- Heavily dependent on environment
- Analyses not necessarily comparable
- Good for “live” trials
- Fits user perspective

Theoretical analysis

- Covers all possible inputs
- Can analyze pseudo-implementations
- Environment independent
- Analyses are easily comparable
- No “real world” guarantees
- Detached from user perspective

- Theoretical over concrete
 - machines change, theory doesn't
 - must cover all inputs
- Consider concrete analysis
 - when discussing tradeoffs
 - given a Java implementation

Abstraction

- § An algorithm may run differently depending on:
 - the hardware platform (PC, Cray, Sun)
 - the programming language (C, Java, C++)
 - the programmer (you, me, Bill Joy)
- § While different in detail, all hardware and programming models are equivalent in some sense: **Turing machines.**
- § It suffices to count basic operations.
- § Crude but valuable measure of algorithm's performance *as a function of input size.*

Running time analysis

- Algorithms & data structure operations
- Use asymptotic notation
- Analyze pseudocode, not Java
- Round off per abstraction
- Assume all primitive operations run in constant time

Memory load analysis

- Data structure contents & algorithm overhead
- Use asymptotic notation
- Analyze pseudocode, not C or Java
- Round off per abstraction
- Assume all primitive types take up constant memory

External factors

- Applies only to concrete analysis
- Example factors
 - specific usage requirements
 - input conditions
 - financial considerations
 - reusability, adaptability, upgradeability, reliability, any other –ability...

Analysis styles

- Average-case analysis
 - common in concrete analysis
- Best-case analysis
 - used in concrete and theoretical analysis
 - results not very useful
- Worst-case analysis
 - used in concrete and theoretical analysis
 - free of all statistical theory
 - accounts for all possible inputs
 - worst case often apparent or common
 - offers a “guarantee”
 - can be misleading (in theoretical analysis)

Average, Best, and Worst-Case

- § On which input instances should the algorithm's performance be judged?
- § Average case:
 - Real world distributions difficult to predict
- § Best case:
 - Seems unrealistic
- § **Worst case:**
 - Gives an absolute guarantee
 - **We will use the worst-case measure.**

Examples

• Vector addition $\mathbf{Z} = \mathbf{A} + \mathbf{B}$

```
for (int i=0; i<n; i++)
```

```
     $Z[i] = A[i] + B[i];$ 
```

$T(n) = c n$

• Vector (inner) multiplication $z = \mathbf{A} * \mathbf{B}$

```
z = 0;
```

```
for (int i=0; i<n; i++)
```

```
     $z = z + A[i] * B[i];$ 
```

$T(n) = c' + c_1 n$

Examples

- § Vector (outer) multiplication $Z = A \cdot B^T$

```
for (int i=0; i<n; i++)  
    for (int j=0; j<n; j++)  
        Z[i,j] = A[i] * B[j];
```

$$T(n) = c_2 n^2;$$

- § A program does all the above

$$T(n) = c_0 + c_1 n + c_2 n^2;$$

Simplifying the Bound

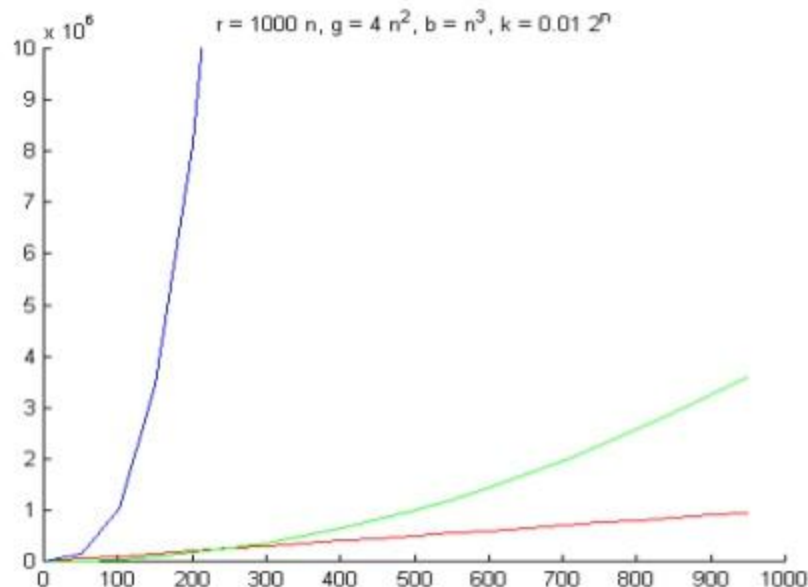
- s $T(n) = c_k n^k + c_{k-1} n^{k-1} + c_{k-2} n^{k-2} + \dots + c_1 n + c_0$
 - too complicated
 - too many terms
 - Difficult to compare two expressions, each with 10 or 20 terms
- s Do we really need that many terms?

Simplifications

- § Keep just one term!
 - the fastest growing term (dominates the runtime)
- § No constant coefficients are kept
 - Constant coefficients affected by machines, languages, etc.
- § **Asymtotic behavior** (as n gets large) is determined entirely by the **leading** term.
 - Example. $T(n) = 10 n^3 + n^2 + 40n + 800$
 - If $n = 1,000$, then $T(n) = 10,001,040,800$
 - error is 0.01% if we drop all but the n^3 term
 - In an assembly line the slowest worker determines the throughput rate

Simplification

- Drop the constant coefficient
 - Does not effect the relative order



Simplification

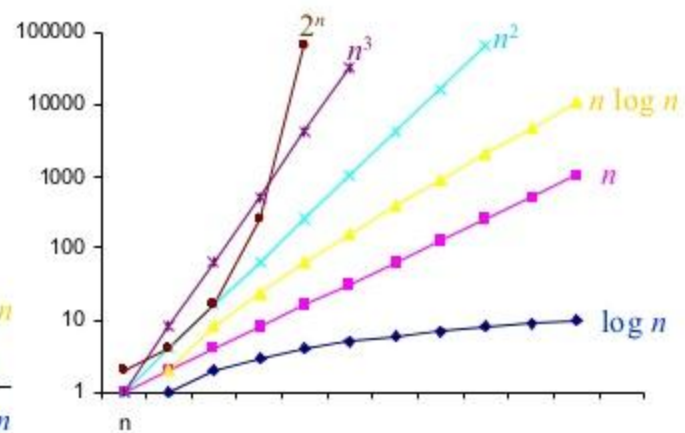
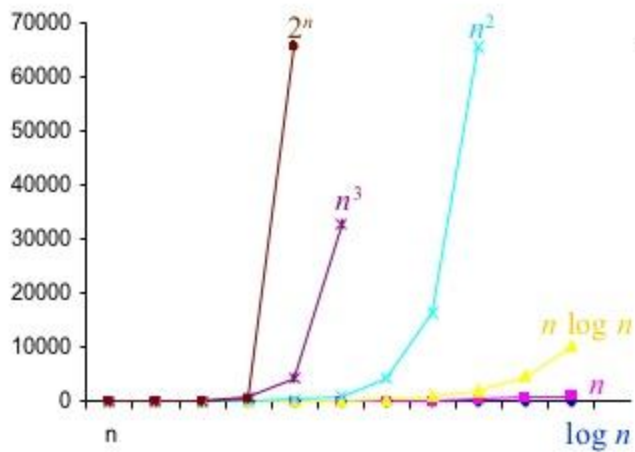
- § The *faster* growing term (such as 2^n) *eventually* will outgrow the slower growing terms (e.g., $1000n$) no matter what their coefficients!
- § Put another way, given a certain increase in allocated time, a higher order algorithm will not reap the benefit by solving much larger problem

Complexity and Tractability

	$T(n)$						
n	n	$n \log n$	n^2	n^3	n^4	n^{10}	2^n
10	.01 μ s	.03 μ s	.1 μ s	1 μ s	10 μ s	10s	1 μ s
20	.02 μ s	.09 μ s	.4 μ s	8 μ s	160 μ s	2.84h	1ms
30	.03 μ s	.15 μ s	.9 μ s	27 μ s	810 μ s	6.83d	1s
40	.04 μ s	.21 μ s	1.6 μ s	64 μ s	2.56ms	121d	18m
50	.05 μ s	.28 μ s	2.5 μ s	125 μ s	6.25ms	3.1y	13d
100	.1 μ s	.66 μ s	10 μ s	1ms	100ms	3171y	4×10^{13} y
10^3	1 μ s	9.96 μ s	1ms	1s	16.67m	3.17×10^{13} y	32×10^{283} y
10^4	10 μ s	130 μ s	100ms	16.67m	115.7d	3.17×10^{23} y	
10^5	100 μ s	1.66ms	10s	11.57d	3171y	3.17×10^{33} y	
10^6	1ms	19.92ms	16.67m	31.71y	3.17×10^7 y	3.17×10^{43} y	

Assume the computer does 1 billion ops per sec.

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65,536
5	32	160	1,024	32,768	4,294,967,296



Another View

- More resources (time and/or processing power) translate into large problems solved if complexity is low

$T(n)$	Problem size solved in 10^3 sec	Problem size solved in 10^4 sec	Increase in Problem size
$100n$	10	100	10
$1000n$	1	10	10
$5n^2$	14	45	3.2
N^3	10	22	2.2
2^n	10	13	1.3

Asymptotics

$T(n)$	keep one	drop coef
$3n^2+4n+1$	$3n^2$	n^2
$101n^2+102$	$101n^2$	n^2
$15n^2+6n$	$15n^2$	n^2
an^2+bn+c	an^2	n^2

§ They all have the same "growth" rate

Asymptotic efficiency of algorithms

- How does the running time of an algorithm increase with the size of the input in the limit, as the size of the input increases without bound.
- Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.

Asymptotic notation

$O(f(n))$ Big-oh

Asymptotic less than or equal to

$\Omega(f(n))$ Big-omega

Asymptotic greater than or equal to

$\Theta(f(n))$ Big-theta

Asymptotic equal to

For a given function $g(n)$, we denote by $\mathcal{O}(g(n))$ the set of functions

$$\mathcal{O}(g(n)) = \{f(n): \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}.$$

A function $f(n)$ belongs to the set $\mathcal{O}(g(n))$ if there exist positive constants c_1 and c_2 such that it can be “sandwiched” between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large n .

Although $\mathcal{O}(g(n))$ is a set, we write “ $f(n) \in \mathcal{O}(g(n))$ ” to indicate that $f(n)$ is a member of $\mathcal{O}(g(n))$, or “ $f(n) \in \mathcal{O}(g(n))$.”

Figure gives an intuitive picture of functions $f(n)$ and $g(n)$, where $f(n) = \Theta(g(n))$. For all values of n to the right of n_0 , the value of $f(n)$ lies at or above $c_1g(n)$ and at or below $c_2g(n)$.

In other words, for all $n \geq n_0$, the function $f(n)$ is equal to $g(n)$ to within a constant factor.

We say that $g(n)$ is an **asymptotically tight bound** for $f(n)$.

The definition of $\Theta(g(n))$ requires that every member of $\Theta(g(n))$ be **asymptotically nonnegative**,

$f(n)$ be nonnegative whenever n is sufficiently large.

Consequently, the function $g(n)$ itself must be asymptotically nonnegative, or else the set $\Theta(g(n))$ is empty.

We shall therefore assume that every function used within Θ -notation is asymptotically non-negative.

Introduction of an informal notion of Θ -notation amounts to throwing away lower-order terms and ignoring the leading coefficient of the highest-order term. Let us briefly justify this intuition by using the formal definition to show that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. To do so, we must determine positive constants c_1 , c_2 and n_0 such that

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

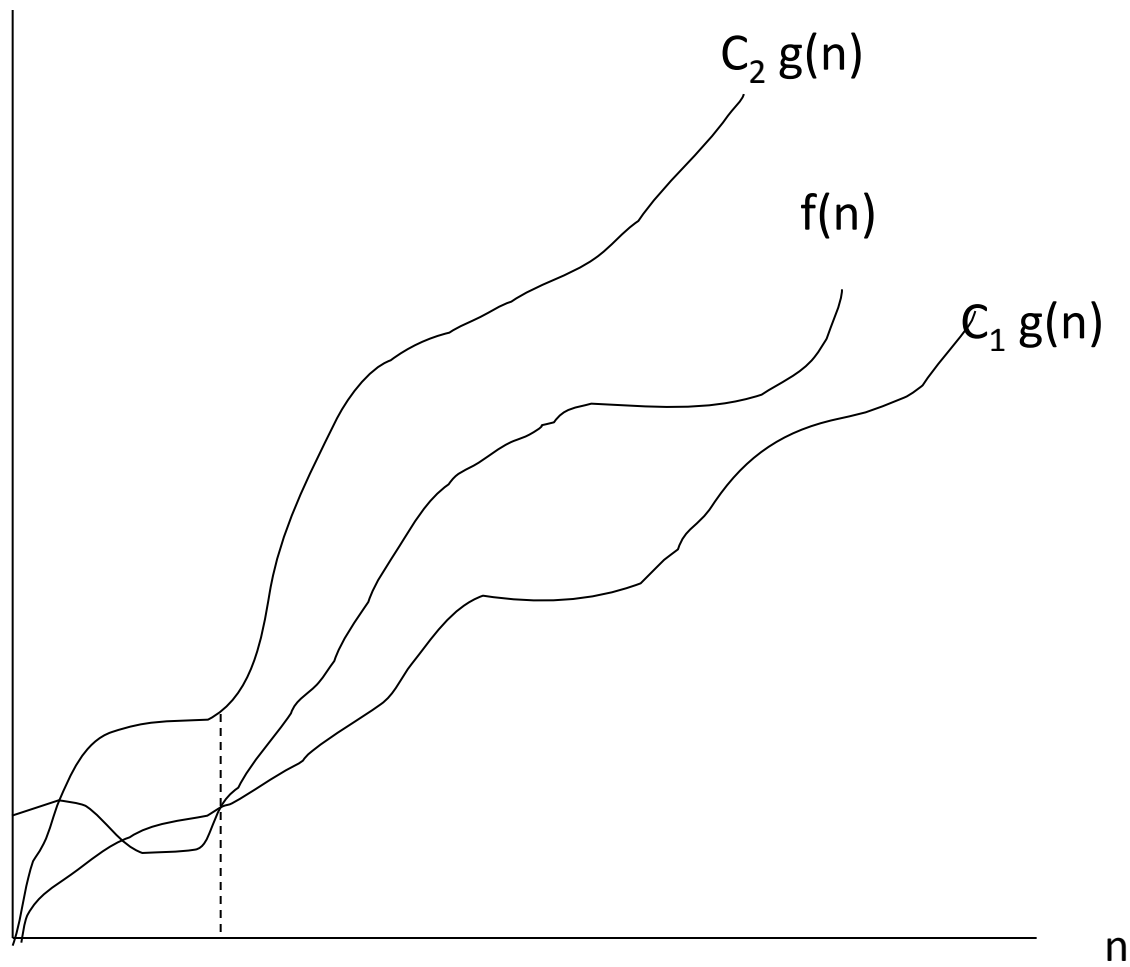
For all $n > n_0$. Dividing by n^2 yields

$$c_1 \leq \frac{\frac{1}{2} - \frac{3}{n}}{1} \leq c_2$$

The right-hand inequality can be made to hold for any value of $n \geq 1$ by choosing $c_2 \geq \frac{1}{2}$. Likewise, the left-hand inequality can be made to hold for any value of $n \geq 7$ by choosing $c_1 \leq 1/14$. Thus, by choosing $c_1 = 1/14$, $c_2 = \frac{1}{2}$ and $n_0 = 7$, we can verify that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Certainly, other choices for the constants exist, but the important thing is that some choice exists. Note that these constants depend on the function $\frac{1}{2}n^2 - 3n$; a different function belonging to $\Theta(n^2)$ would usually require different constants.

Intuitively, the lower-order terms of an asymptotically positive function can be ignored in determining asymptotically tight bounds because they are insignificant for large n . A tiny fraction of the highest-order term is enough to dominate the lower-order terms. Thus, setting c_1 to a value that is slightly smaller than the coefficient of the highest-order term and setting c_2 to a value that is slightly larger permits the inequalities in the definition of Θ -notation to be satisfied. The coefficient of the highest-order term can likewise be ignored, since it only changes c_1 and c_2 by a constant factor equal to the coefficient.

Since any constant is a degree-0 polynomial, we can express any constant function as $\mathcal{O}(n^0)$, or $\mathcal{O}(1)$. This latter notation is a minor abuse, however, because it is not clear what variable is tending to infinity.¹ We shall often use the notation $\mathcal{O}(1)$ to mean either a constant or a constant function with respect to some variable.



n_0

$$f(n) = \Theta(g(n))$$

O-notation

The Θ -notation asymptotically bounds a function from above and below. When we have only an asymptotic upper bound, we use O notation. For a given function $g(n)$, we denote by $O(g(n))$ the set of functions

$O(g(n)) = \{f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$
 $0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}.$

We use O -notation to give an upper bound on a function, to within a constant factor. Figure shows the intuition behind O -notation. For all values n to the right of n_0 , the value of the function $f(n)$ is on or below $g(n)$.

To indicate that a function $f(n)$ is a member of $O(g(n))$, we write $f(n) = O(g(n))$. Note that $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$, since O -notation is a stronger notion than Θ -notation. Written set-theoretically, we have $\Theta(g(n)) \subseteq O(g(n))$. Thus, our proof that any quadratic function an^2+bn+c , where $a>0$, is in $\Theta(n^2)$ also shows that any quadratic function is in $O(n^2)$. What may be more surprising is that any linear function $an+b$ is in $O(n^2)$, which is easily verified by taking $c=a+|b|$ and $n_0 = 1$.

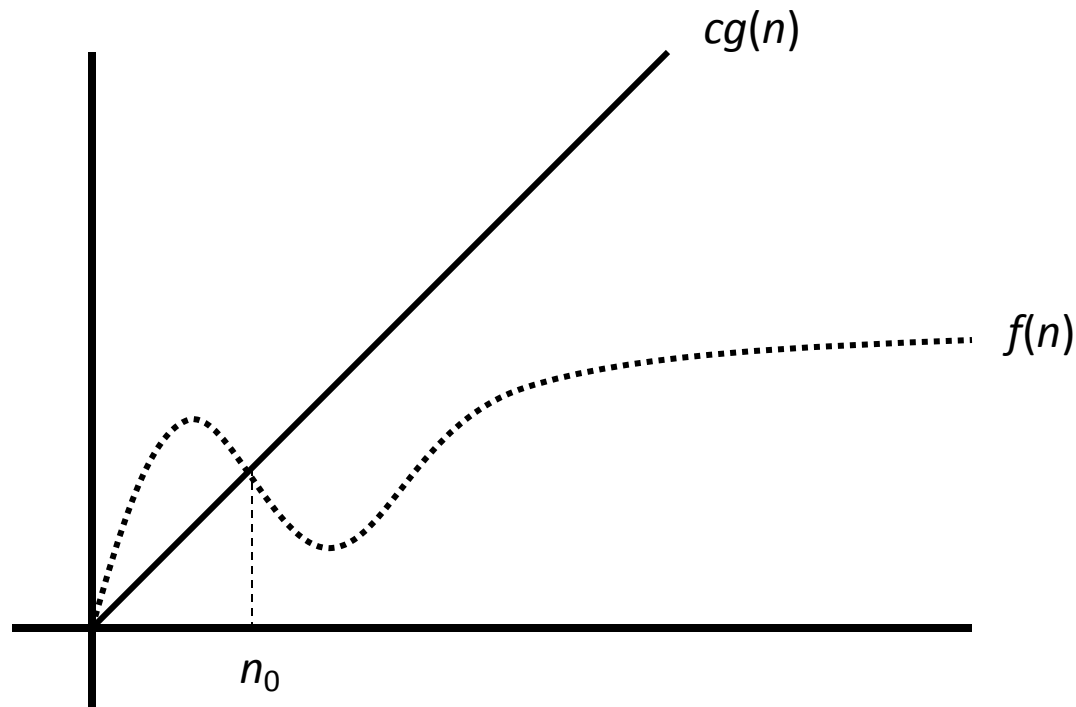
Using O -notation, we can often describe the running time of an algorithm merely by inspecting the algorithm's overall structure. For example, the doubly nested loop structure of the insertion sort algorithm immediately yields an $O(n^2)$ upper bound on the worst-case running time: the cost of the inner loop is bounded from above by $O(1)$ (constant), the indices i and j are both at most n , and the inner loop is executed at most once for each of the n^2 pairs of values for i and j .

Since O -notation describes an upper bound, when we use it to bound the worst-case running time of an algorithm, by implication we also bound the running time of the algorithm on arbitrary inputs as well. Thus, the $O(n^2)$ bound on worst-case running time of insertion sort also applies to its running time on every input. The $\Theta(n^2)$ bound on the worst-case running time of insertion sort, however, does not imply a $\Theta(n^2)$ bound on the running time of insertion sort on every input.

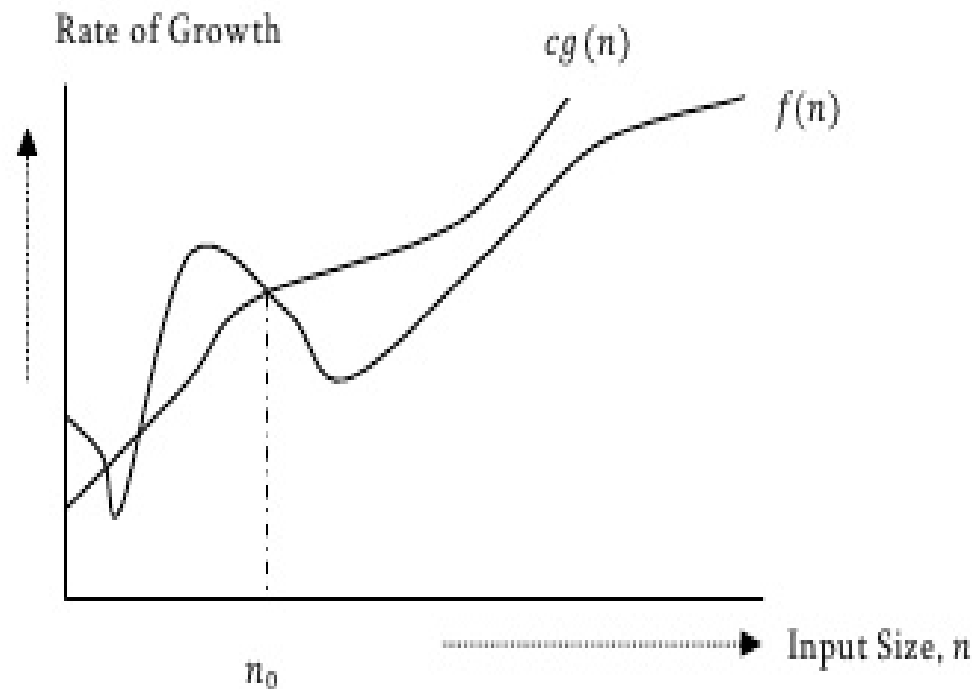
Technically, it is an abuse to say that the running time of insertion sort is $O(n^2)$, since for a given n , the actual running time depends on the particular input of size n . That is, the running time is not really a function of n . What we mean when we say “the running time is $\Theta(n^2)$ ” is that the worst-case running time (which is a function of n) is $O(n^2)$, or equivalently, no matter what particular input of size n is chosen for each value of n , the running time on that set of inputs is $O(n^2)$.

Visual description (Big-oh)

- $f(n)$ is $O(g(n))$



In general, we do not consider lower values of n . That means the rate of growth at lower values of n is not important. In the below figure, n_0 is the point from which we consider the rate of growths for a given algorithm. Below n_0 the rate of growths may be different.



Function Orders

A function $f(n)$ is $O(g(n))$ if “increase” of $f(n)$ is not faster than that of $g(n)$.

A function $f(n)$ is $O(g(n))$ if there exists a number n_0 and a nonnegative c such that for all $n \geq n_0$, $0 \leq f(n) \leq cg(n)$.

If $\lim_{n \rightarrow \infty} f(n)/g(n)$ exists and is finite, then $f(n)$ is $O(g(n))$

Big-O examples

Example-1 Find upper bound for $f(n) = 3n + 8$

Solution: $3n + 8 \leq 4n$, for all $n \geq 1$

$$\therefore 3n + 8 = O(n) \text{ with } c = 4 \text{ and } n_0 = 8$$

Example-2 Find upper bound for $f(n) = n^2 + 1$

Solution: $n^2 + 1 \leq 2n^2$, for all $n \geq 1$

$$\therefore n^2 + 1 = O(n^2) \text{ with } c = 2 \text{ and } n_0 = 1$$

Example-3 Find upper bound for $f(n) = n^4 + 100n^2 + 50$

Solution: $n^4 + 100n^2 + 50 \leq 2n^4$, for all $n \geq 1$

$$\therefore n^4 + 100n^2 + 50 = O(n^4) \text{ with } c = 2 \text{ and } n_0 = 100$$

Example-4 Find upper bound for $f(n) = 2n^3 - 2n^2$

Solution: $2n^3 - 2n^2 \leq 2n^3$, for all $n \geq 1$

$$\therefore 2n^3 - 2n^2 = O(2n^3) \text{ with } c = 2 \text{ and } n_0 = 1$$

Example-5 Find upper bound for $f(n) = n$

Solution: $n \leq n^2$, for all $n \geq 1$

$\therefore n = O(n^2)$ with $c = 1$ and $n_0 = 1$

Example-6 Find upper bound for $f(n) = 410$

Solution: $410 \leq 410$, for all $n \geq 1$

$\therefore 100 = O(1)$ with $c = 1$ and $n_0 = 1$

No uniqueness?

There is no unique set of values for n_0 and c in proving the asymptotic bounds. Let us consider, $100n + 5 = O(n^2)$. For this function there are multiple n_0 and c .

Solution1: $100n + 5 \leq 100n + n = 101n \leq 101n^2$ for all $n \geq 5, n_0 = 5$ and $c = 101$ is a solution.

Solution2: $100n + 5 \leq 100n + 5n = 105n \leq 105n^2$ for all $n \geq 1, n_0 = 1$ and $c = 105$ is also a solution.

Time descriptions

- If $f(n)$ is $O(g(n))$ and $g(n)$ is
 - c constant
 - $\log n$ logarithmic
 - n linear
 - n^2 quadratic
 - n^k for $k \geq 1$ polynomial
 - a^n for $a > 1$ exponential

Simple heuristics

- Dominant term is asymptotic upper bound
 - ex. $6n^3 - 2\frac{1}{2}n + \log(n - 1)$ is $O(n^3)$
- Asymptotic addition
 - ex. $O(n^3) + 2O(n^2) - \frac{1}{2}O(n)$ is $O(n^3)$
- Asymptotic multiplication
 - ex. $O(n) \cdot 2O(n) \cdot \frac{1}{2}O(\log n)$ is $O(n^2)$

Example Functions

$\text{sqrt}(n)$, n , $2n$, $\ln n$, $\exp(n)$, $n + \text{sqrt}(n)$, $n + n^2$

$$\lim_{n \rightarrow \infty} \text{sqrt}(n) / n = 0,$$

$\text{sqrt}(n)$ is $O(n)$

$$\lim_{n \rightarrow \infty} n / \text{sqrt}(n) = \text{infinity},$$

n is not $O(\text{sqrt}(n))$

$$\lim_{n \rightarrow \infty} n / 2n = 1/2,$$

n is $O(2n)$

$$\lim_{n \rightarrow \infty} 2n / n = 2,$$

$2n$ is $O(n)$

$$\lim_{n \rightarrow \infty} \ln(n) / n = 0,$$

$\ln(n)$ is $O(n)$

$$\lim_{n \rightarrow \infty} n / \ln(n) = \text{infinity},$$

n is not $O(\ln(n))$

$$\lim_{n \rightarrow \infty} \exp(n) / n = \text{infinity},$$

$\exp(n)$ is not $O(n)$

$$\lim_{n \rightarrow \infty} n / \exp(n) = 0,$$

n is $O(\exp(n))$

$$\lim_{n \rightarrow \infty} (n + \sqrt{n}) / n = 1,$$

$n + \sqrt{n}$ is $O(n)$

$$\lim_{n \rightarrow \infty} n / (\sqrt{n} + n) = 1,$$

n is $O(n + \sqrt{n})$

$$\lim_{n \rightarrow \infty} (n + n^2) / n = \text{infinity},$$

$n + n^2$ is not $O(n)$

$$\lim_{n \rightarrow \infty} n / (n + n^2) = 0,$$

n is $O(n + n^2)$

Commonly used Rate of Growths

Below is the list of rate of growths which come across.

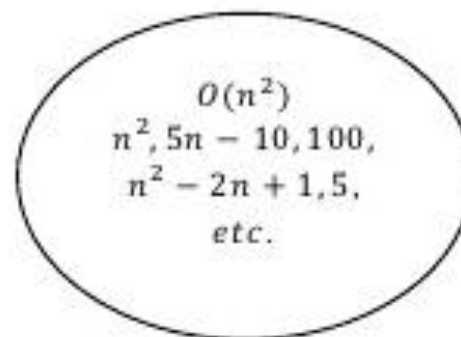
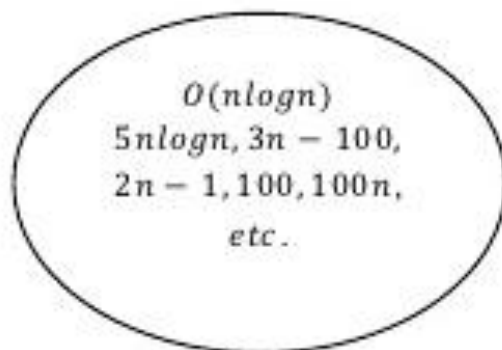
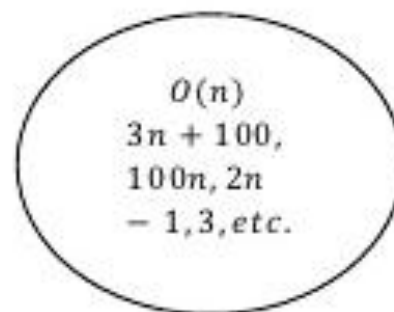
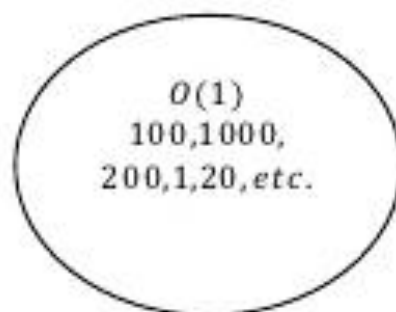
Time complexity	Name	Example
$O(1)$	Constant	Adding an element to the front of a linked list
$O(\log n)$	Logarithmic	Finding an element in a sorted array
$O(n)$	Linear	Finding an element in an unsorted array
$O(n \log n)$	Linear Logarithmic	Sorting n items by 'divide-and-conquer'-Mergesort
$O(n^2)$	Quadratic	Shortest path between two nodes in a graph
$O(n^3)$	Cubic	Matrix Multiplication
$O(2^n)$	Exponential	The Towers of Hanoi problem

Note: Analyze the algorithms at larger values of n only. What this means is, below n_0 we do not care for rates of growth.

n_0

Asymptotically, n

Big-O Visualization



$O(g(n))$ is the set of functions with smaller or same order of growth as $g(n)$. For example, $O(n^2)$ includes $O(1), O(n), O(n \log n)$ etc..

Implication of O notation

Suppose we know that our algorithm uses at most $O(f(n))$ basic steps for any n inputs, and n is sufficiently large, then we know that our algorithm will terminate after executing at most constant times $f(n)$ basic steps.

We know that a basic step takes a constant time in a machine.

Hence, our algorithm will terminate in a constant times $f(n)$ units of time, for all large n .

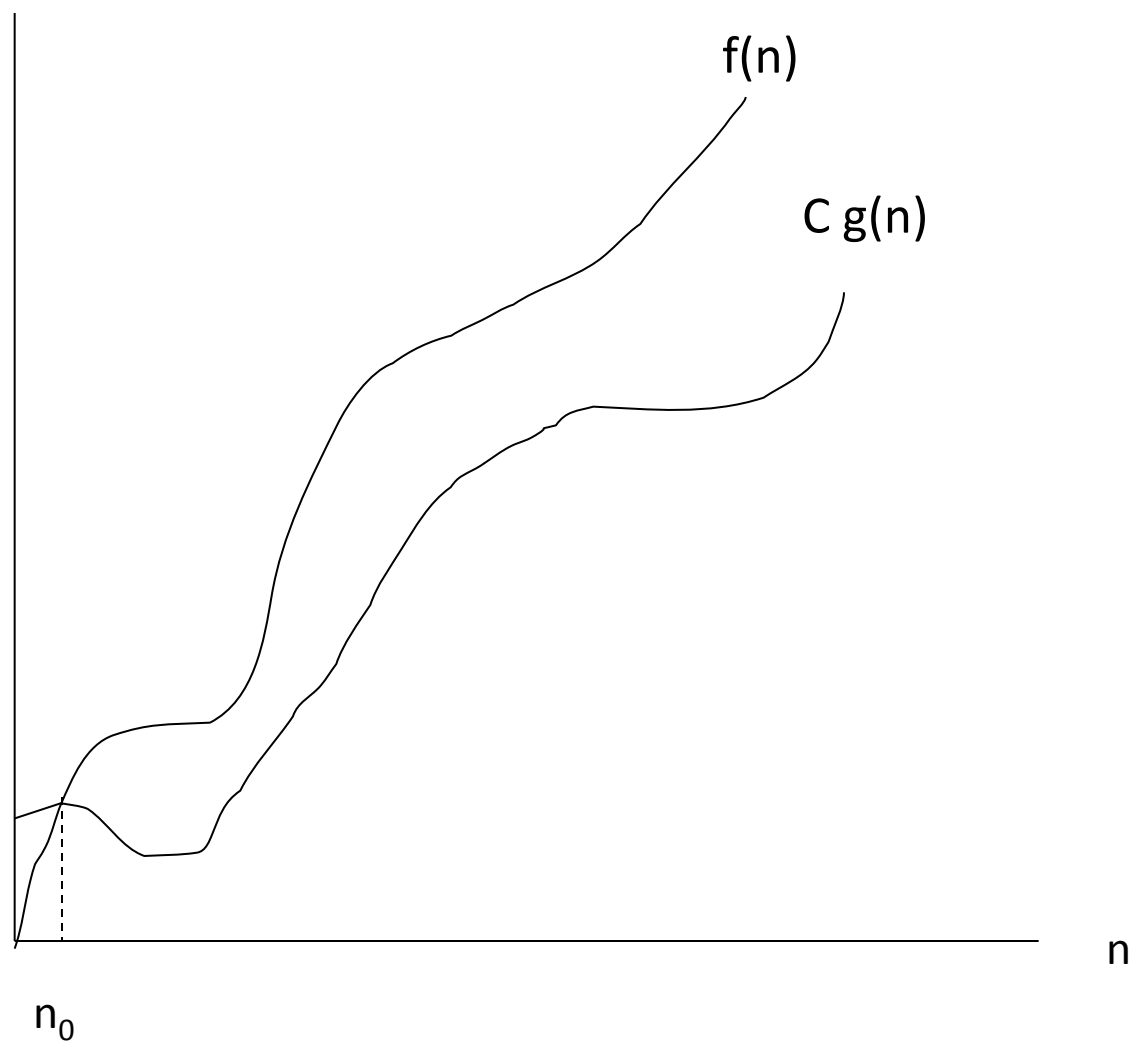
Ω -notation

Just as O -notation provides an asymptotic upper bound on a function, Ω -notation provides an asymptotic lower bound. For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions

$$\Omega(g(n)) = \{f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

The intuition behind Ω -notation is shown in Figure 2.1©. For all values n to the right of n_0 , the value of $f(n)$ is on or above $g(n)$.

From the definitions of the asymptotic notations we have seen thus far, it is easy to prove the following important theorem

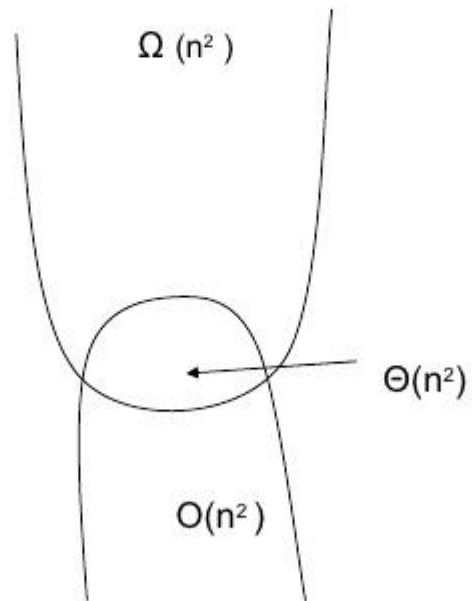


$$f(n) = \Omega(g(n))$$

Asymptotic notation (cont.)

- $o(f(n))$ Little-oh
 - Asymptotic less than
- $\omega(f(n))$ Little-omega
 - Asymptotic greater than

Mappings for n^2



Mr. Sachin Sharma Bhandari

Be wary of asymptotic notation

- Hidden constants may conceal poor experimental performance
- Mathematical “trickery” may lead to good asymptotic performance that is (partially) unwarranted
- But, asymptotic notation is still very useful and well-regarded

Emprical Analysis

- Run both and see which takes longer
- Monitor performance by careful implementation on typical input . We get performance results which are direct indicator of efficiency.
- Challenges
 - Develop a correct complete implementation.
 - To determine the nature of I/P data & other factors that have direct influence on the experiments to be performed.

Empirical Analysis

3 Choices

- actual data – enable us to truly measure the cost of the program in use
- Random data – assures us that our experiments test the algorithm not the data.
- Perverse data - Assures us that our programs can handle any I/P presented to them

Problem of determining which I/P data to use to compare algorithms

Common Mistakes

- Easy to make mistakes when we make implementation. Particularly if differing machine compilers or systems are involved or with ill-specified I/Ps
- One implementation may be more carefully coded than the other
- Ignorance of performance characteristics
- Faster algorithms are more complicated than brute force solutions.
- Implementers – willing to accept a slower algorithm to avoid having to deal with the complexity.
- When we are dealing with huge problem sizes, we may have no choices but to seek a better algorithm

Common Mistakes

- Paying too much attention to performance characteristics
- Improving the running time of a program by a factor of 10 is inconsequential if the program takes only a few micro second
- Even if a program takes a few second , the effort required to implement & debug an improved algorithm may be substantiated
- The program is run only a few time

Mathematical Tools to Analyze algorithm

- To identify the abstract options on which is algorithms is based, to separate the analysis from the implementation
- This separation allows us to compare algorithms that are independent of particular implementations or of particular computers
- The analysis amounts to determining the frequency of execution of a few fundamental options.

Mathematical Tools to Analyze algorithm

Study the data model that might be presented in the algorithm

- If i/p is random study the average case performance of the algorithm.
 - Studying the average case performance of the algorithms is attractive because it allows us to make predictions about the running time of the programs.
 - Difficulty with average case analysis – The i/p model may not accurately characterize the inputs encountered in practice or there may be no natural input model at all.
 - The process of characterizing random inputs is difficult for many algorithms, but for many others it is straight forward
 - Randomly scramble the input before attempting to process it the the assumption that i/p is random is accurate

Mathematical Tools to Analyze algorithm

- If the i/p is perverse, study the worst case performance
 - Worst case running time of an algorithm – Upper bound on the running time for any i/p guarantee that the algorithm will never take any longer
 - Worst case analysis using the O notation frees the analyst from considering the details of particular machine characteristics
 - Independent of input
 - Separates analysis from implementation
 - Is attractive because it allows us to make guarantees about the running time of programs
 - Algorithms with good worst case performance are significantly more complicated than other algorithms
 - And practical data is not perverse most of the times

```
int i, j, k, count = 0;
```

```
for(i=0; i<N; i++)
```

```
for(j=0; j<N; j++)
```

```
for(k=0; k<N; k++)
```

```
    count ++;
```

```
N = 10;
```

Initialize count once

Initialize i once

Increment i \rightarrow N times

Compare i and N \rightarrow N times

Loop on i \rightarrow N Times

Initialize $j \rightarrow N$ times

Increment $j \rightarrow N^2$ times

Compare j and $N \rightarrow N^2$ times

Loop on $j \rightarrow N^2$ times

Initialize $k \rightarrow N^2$ times

Increment $k \rightarrow N^3$ times

Compare j and $N \rightarrow N^3$ times

Loop on $j \rightarrow N^3$ times

Increment count $\rightarrow N^3$ times

Totally $4N^3 + 4N^2 + 4N + 2$

Runtime Analysis

• Useful rules

- simple statements (read, write, assign)
 - $O(1)$ (constant)
- simple operations (+ - * / == > >= < <=)
 - $O(1)$
- sequence of simple statements/operations
 - rule of sums
- for, do, while loops
 - rules of products

Runtime Analysis (cont.)

```
if (cond) then       $O(1)$   
    body1           $T_1(n)$   
else  
    body2           $T_2(n)$   
endif
```

$$T(n) = O(\max(T_1(n), T_2(n)))$$

Runtime Analysis (cont.)

- Method calls

- ☐ A calls B
- ☐ B calls C
- ☐ etc.

- A sequence of operations when call sequences are flattened

$$T(n) = \max(T_A(n), T_B(n), T_C(n))$$

Example

```
for (i=1; i<n; i++)  
    if A(i) > maxVal then  
        maxVal= A(i);  
        maxPos= i;
```

Asymptotic Complexity: $O(n)$

Example

```
for (i=1; i<n-1; i++)  
  for (j=n; j>= i+1; j--)  
    if (A(j-1) > A(j)) then  
      temp = A(j-1);  
      A(j-1) = A(j);  
      A(j) = tmp;  
    endif  
  endfor  
endfor
```

§ Asymptotic Complexity is $O(n^2)$

The notation $f(n) = O(g(n))$ has a precise mathematical definition.

Definition: $f(n) = O(g(n))$ if there exist two constants c and n_0 such that $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$.

- $f(n)$ will normally represent the computing time of some algorithm.
- If the computing time of an algorithm is $O(g(n))$ it means that its execution takes no more than a constant times $g(n)$.
- n is a parameter which characterizes the inputs and / or outputs.
- For example n might be the number of inputs or the number of outputs or their sum or the magnitude of one of them.
- For the fibonacci program n represents the magnitude of the input and the time for this program is written as
 $T(\text{FIBONACCI}) = O(n)$

- $O(1)$ means a constant computing time
- $O(n)$ is called linear,
- $O(n^2)$ is called quadratic,
- $O(n^3)$ is called cubic,
- $O(2^n)$ is called exponential.
- If an algorithm takes time $O(\log n)$, it is faster, for sufficiently large n , than if it had taken $O(n)$.

These seven computing times, $O(1)$, $O(\log n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, and $O(2^n)$ are the ones encountered most often

Two algorithms which perform the same task --

- The first has a computing time $O(n)$ and second $O(n^2)$,
- Usually, the first is superior.
- The reason -- as n increases, the time for the second algorithm will get far worse than the time for the first

For example, if the constant for algorithms one and two are 10 and $\frac{1}{2}$ respectively, then the following table indicates the computing times:

n	$10n$	$N^2/2$
1	10	$\frac{1}{2}$
5	50	$12\frac{1}{2}$
10	100	50
15	150	$112\frac{1}{2}$
20	200	200
25	250	$312\frac{1}{2}$
30	300	450

- For $n \leq 20$, **algorithm two** had a **smaller** computing time but **once past** that point **algorithm one** became **better**.
- This shows why the algorithm with the smaller order of magnitude is chosen, but this is not the whole story.
- For small data sets, the respective constants must be carefully determined.

Growth of Functions

Asymptotic efficiency of an algorithm

- We are concerned with how the run time of an algorithm increases with the size of the input in the limit as the size of the input increases with the bound
- Usually an algorithm that is asymptotically more efficient will be the best choice for all but very small i/p s.
- Examples $\rightarrow 2^n, n^3$ for $n = 2$

$$2^3, 3^3$$

$$2^4, 3^4 \rightarrow 2^4, 2^6$$

$$2^8, 8^3 \rightarrow 2^8, 2^9$$

$$2^{16}, 16^3 \rightarrow 2^{16}, (2^4)^3 \rightarrow 2^{16}, 2^{12}$$

$$\text{Lt}_{n \rightarrow \infty} f(n)/g(n) = \text{const}$$

Growth of Functions

- Primary parameter that affects the running time most significantly
- An abstract measure of the size of the problem being considered
- Most often directly proportional to the size of the data set being processed
- When there is more than one such parameter.
- Reduce analysis to one parameter by expressing one of the parameters as a function of the other or by considering one parameter at a time
- Therefore we can restrict ourselves to considering a single parameter N without loss of generality

Growth of Functions

Objectives

- To express the resource required of our programs (most often running time) in terms of N , using mathematical formulae that are simple as possible & accurate for large values of the parameter.

Typically proportional to one of the following functions:

1

- Most instructions are executed once or at most a few times
- Constant running time

Log N

- running time of a program is logarithmic.
- This running time commonly occurs in programs that solve a big problem by transformation into a series of smaller problems cutting the problem size by some constant fraction at each step

$$N = 1000$$

$$\log N = 3 \text{ (base 10)}$$

$$\approx 10 \text{ (base 2)}$$

$$N = 1,000,000 \rightarrow \log N \text{ only doubles}$$

Whenever N doubles, $\log N$ increases by a constant.

But $\log N$ does not double until N increases to N^2 .

N

- The running time of a program is linear
- Small amount of processing is done on each i/p element

$N \rightarrow 1\text{m}$, so the running time

$N \rightarrow \text{doubles}$, so does the running time.

N Log N

- Arises, when algorithm solve a problem by breaking it up into smaller subprograms, solving them independently then combining the solutions

$N = 1\text{m}$, $N \log N \rightarrow 20$ million perhaps

When N doubles, running time more than doubles
(but not much more)

N^2

- Running time algorithm is quadratic
- Quadratic response times typically arise in algorithms that process all pairs of data items. Perhaps a double nested loops

$N = 1000$

$R.T \rightarrow 1m$

N – doubles, running time incremented 4 fold

N^3

- Cubic runtime

2^N

- Exponential running time
- Brute force solutions

$N = 20$, $RT \rightarrow 1m$

$N \rightarrow$ doubles, runtime squares

- In practice these constants depend on many factors,
 - the language
 - the machine used.
- Thus, usually the establishment of the constant is postponed until after the program has been written.
- Then a performance profile can be gathered using real time calculation.
- Figures show how the computing times (counts) grow with a constant equal to one.
 - The times $O(n)$ and $O(n \log n)$ grow much more slowly than the others.
 - For large data sets, algorithms with a complexity greater than $O(n \log n)$ are often impractical.
 - An algorithm which is exponential will work only for very small inputs.
 - For exponential algorithms, even if the constant is improved, say by $\frac{1}{2}$ or $\frac{1}{3}$, the amount of data that can be handled is not improve very much.

$\log_2 n$	$n \log_2 n$	N^2	N^3	2^n	
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	2,147,483,648

Given an algorithm, analyze the frequency count of each statement and total the sum.
This may give a polynomial

$$P(n) = c_k n^k + c_{k-1} n^{k-1} + \dots + c_1 n + c_0$$

Where the c_i are constants, $c_k \neq 0$ and n is a parameter.

Using big-oh notation, $p(n) = O(n^k)$.

On the other hand, if any step is executed 2^n times or more, the expression

$$C2^n + p(n) = O(2^n).$$

Another valid performance measure of an algorithm is the space it requires.

Often one can trade space for time, getting a faster algorithm but using more space.

Binary Search: An Example

For concreteness,

assume that the telephone book contains a million names $X_0, X_1, \dots, X_{999999}$, $N = 1,000,000$

The first comparison

- not between Y and first or last name in L
- but between Y and the middle name X_{500000} (if the list is of odd length)
- between Y and the last name in the first half of the list X_{500000} (if the list is of even length)

if

the compared names turn out to be unequal

two possibilities: **(the list is sorted alphabetically)**

(1) Y precedes X_{500000} in alphabetic order

(2) X_{500000} precedes Y

if (1) is the case &

if Y appears in the list at all

it has to be in the first half

if (2) is the case

it must appear in the second half

Restrict successive search to the appropriate half of the list

Accordingly

The next comparison will be
between Y & the middle element of that half:
 X_{250000} in case (1) and X_{750000} in case (2)

Again, the result of this comparison
narrows the possibilities down to half of this new shorter list
to a list whose length is a quarter of the original.

This process continues,
reducing the length of the list by half at each step

In more general terms,

the size of the problem reduced by half at each step,
until either Y is found,
in which case the procedure terminates, reporting success,

or the trivial empty list is reached,
in which case it terminates, reporting failure

This procedure is called binary search
- an application of the divide-and-conquer paradigm

What is the time complexity of binary search?

- Let us first count comparisons -
- How many comparisons will the binary search algorithm require?
- In the worst case on one-million-name telephone book.
- The naive search would require a million comparisons
- Here, in the very worst case
- (what is an example of one?)
- how many worst cases are there?)
- The algorithm will require only 20 comparisons!

The larger N becomes, the more impressive the improvement.

- An international telephone book containing, say, a billion names, would require at most 30 comparisons, instead of a billion!
- each comparison reduces the length of the input list L by half, the process terminates when, or before the list becomes empty
- the worst-case number of comparisons is obtained by figuring out how many times a number N can be repeatedly divided by 2 before it is reduced to 0
(The rules of the game are to ignore fractions)

- This number is termed the base-2 logarithm of N , denoted by $\log_2 N$.
- Actually $\log_2 N$ counts the number required to reduce N to 1, not 0,
- so the number we are really after is $1 + \log_2 N$.
- The algorithm takes $O(\log N)$ comparisons in the worst case.

The algorithm takes $O(\log N)$ comparisons in the worst case

The following table plots several values of N against the number of comparisons required by binary search in the worst case:

N		$1 + \log_2 N$
10		4
100		7
1000		10
a million	20	
a billion	30	
a billion billions	60	

To complete the complexity analysis of binary search

Why is It Enough Count Comparisons?

To show::

If we are happy with a big-O estimate,
it suffices to count comparisons only

Why?

In binary search there are considerably more instructions executed than just comparisons

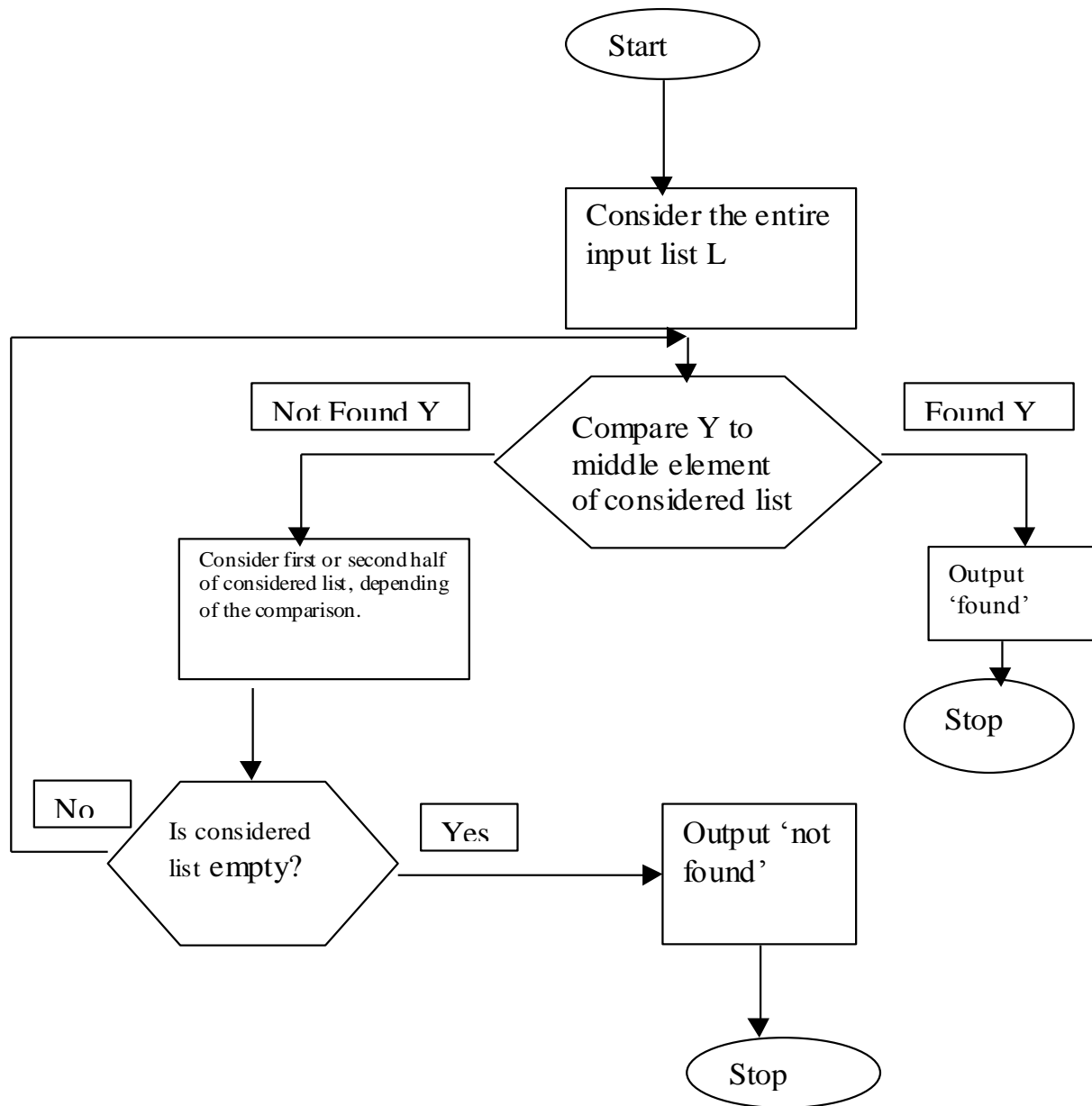
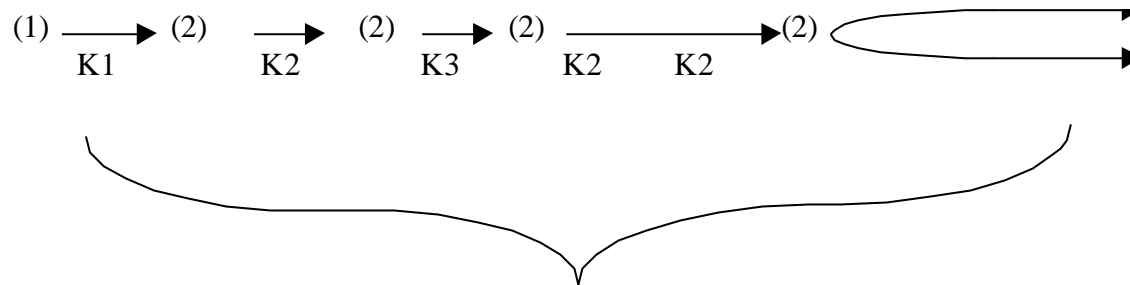


Figure 6.3 shows the schematic flowchart of Figure 1 with checkpoints included, and it illustrates the possible local paths, or "hops," between them

- The checkpoints cut all loops
 - there is only one in this case
- so that the four local paths are loop-free
- it is important because
- loop-free and subroutine-free segments take at most fixed amounts of time
- The presence of conditional statements in such a segment might yield several different ways of traversing it
- However, the absence of loops (and of recursive routines) guarantees that there is a bound on this number
- Hence the total number of instructions carried out in any single execution of such a segment is no greater than some constant.
- This often facilitates easy calculation of the time taken by the algorithm.

In Figure



- The constants K_1 through K are associated with the four possible local paths
- This means that any single execution of the local path from checkpoint (2) back to itself is assumed to involve no more than k_2 instructions.
- (By convention, this includes the comparison made at the beginning of the segment, but not the one made at its end)

Consider a typical execution of binary search

Checkpoint (2) is the only place in the text of the algorithm involving a comparison,
It is reached (in the worst case) precisely $1 + \log_2 N$ times,
(this is the total number of comparisons made)

All but the last of these comparisons
result in the processor going around the loop again,
hence executing at most another k_2 instructions

This means that the total time cost of
all these $\log_2 N$ traversals of the (2) - (2) segment is $k_2 \times \log_2 N$

To complete the analysis,
the total number of all instructions carried out that are not part of
(2) - (2) segments
is either $k_1 + k_3$ or $K_1 + k_4$,
depending on whether the algorithm halts at checkpoint (3) or (4)
In both cases it is a constant that does not depend on N .

If K = the maximum of these two sums,
conclusion -->

The total number of instructions carried out by the binary search algorithm on a list of N
is bounded by
 $K + (k_2 \times \log_2 N)$
using the order-of-magnitude notation
the constants K and k_2 can be "buried" beneath the big-O &
binary search runs in time $O(\log_2 N)$ in the worst case.

The Robustness of the Big-O Notation

Big Os hide constant factors.

- In the worst-case analysis of binary search there was no need to be more specific about the details of the particular instructions that make up the algorithm.
- It was sufficient to analyze the algorithm's loop structure between any two comparisons
- There is only a constant number of instructions in the worst case.

**Surely the time complexity of binary search
does depend on the elementary instructions allowed**

Run Time for Recursive Programs

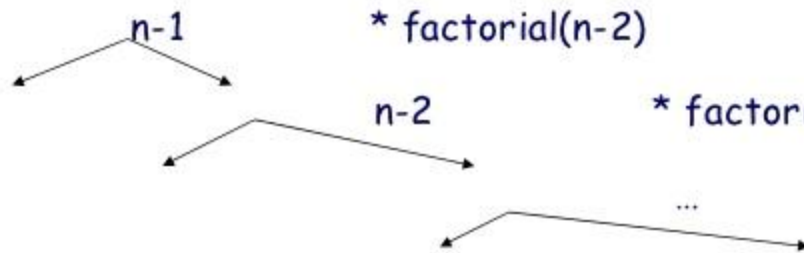
- § $T(n)$ is defined recursively in terms of $T(k)$, $k < n$
- § The **recurrence relations** allow $T(n)$ to be "unwound" recursively into some base cases (e.g., $T(0)$ or $T(1)$).
- § Examples:
 - Factorial
 - Hanoi towers

Example: Factorial

```
int factorial (int n) {
    if (n<=1) return 1;
    else return n * factorial(n-1);
}
```

$\text{factorial}(n) = n * n-1 * n-2 * \dots * 1$

$n \quad * \text{factorial}(n-1)$



$* \text{factorial}(n-2)$

$* \text{factorial}(n-3)$

$2 \quad * \text{factorial}(1)$



$T(1)$

$T(n)$

$= T(n-1) + d$

$= T(n-2) + d + d$

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

$= \dots$

$= T(1) + (n-1) * d$

$= c + (n-1) * d$

$= O(n)$

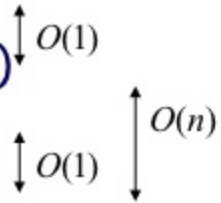
$T(n)$

$T(n-1)$

$T(n-2)$

Example: Factorial (cont.)

```
int factorial1(int n) {  
    if (n<=1) return 1;  
    else {  
        fact = 1;  
        for (k=2;k<=n;k++)  
            fact *= k;  
        return fact;  
    }  
}
```



§ Both algorithms are $O(n)$.

Example: Hanoi Towers

§ $\text{Hanoi}(n, A, B, C) =$

§ $\text{Hanoi}(n-1, A, C, B) + \text{Hanoi}(1, A, B, C) + \text{Hanoi}(n-1, C, B, A)$

$$T(n)$$

$$= 2T(n-1) + c$$

$$= 2^2 T(n-2) + 2c + c$$

$$= 2^3 T(n-3) + 2^2 c + 2c + c$$

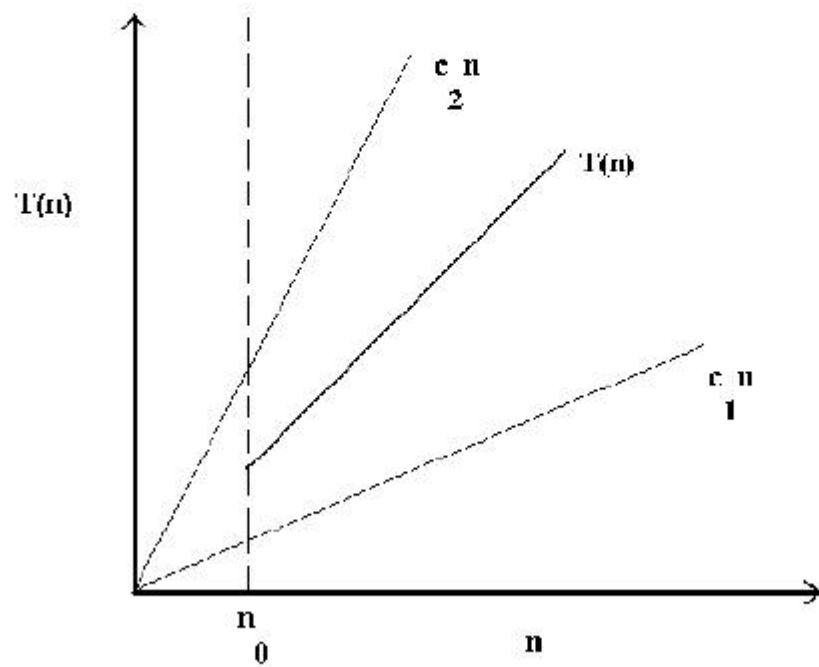
$$= \dots$$

$$= 2^{n-1} T(1) + (2^{n-2} + \dots + 2 + 1)c$$

$$= (2^{n-1} + 2^{n-2} + \dots + 2 + 1)c$$

$$= O(2^n)$$

Bounds of a Function



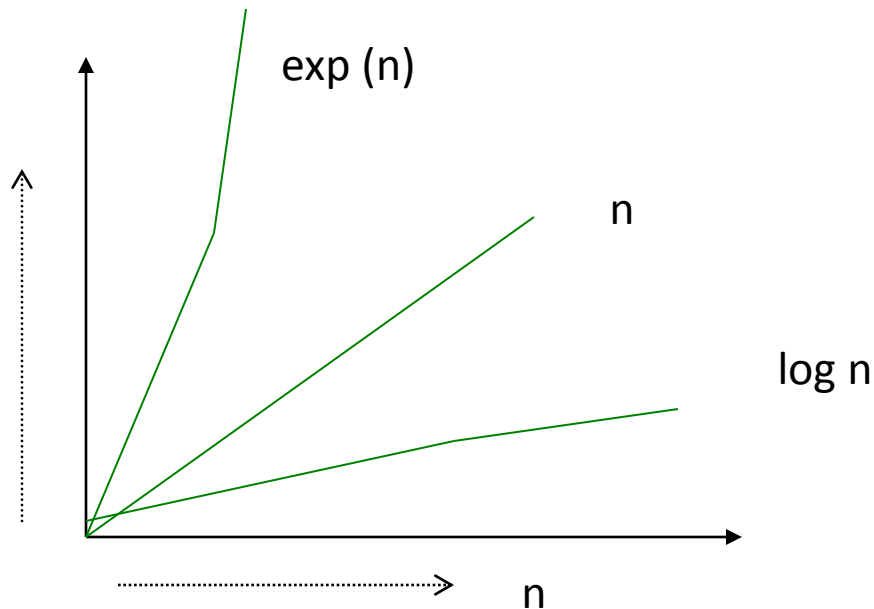
Mr. Sachin Sharma Bhandari

Order of Increase

We worry about the speed of our algorithms for large input sizes.

Note that for large n , $\log(n)/n$, and $n/\exp(n)$ are very small.

However, $n/2^n$ is a constant for all n .



If we were to allow the instruction:

search List L for item Y

The algorithm would consist simply of a single instruction

Hence its time complexity would be $O(1)$

That is, a constant number not depending on N at all

The details of the instructions are unimportant?

The time complexity of an algorithm is indeed a relative concept

It makes sense only in conjunction with an agreed on set of elementary instructions

Nevertheless, standard kinds of problems are usually thought of in the context of standard kinds of elementary instructions

- For example, searching and sorting problems usually involve comparisons, index updates, and end-of-list tests
Hence, complexity analysis is carried out with respect to these
- Moreover, if a particular programming language is fixed in advance then a particular set of elementary instructions has also been fixed for better or for worse

What makes these observations significant is
the fact that for most standard kinds of elementary instructions
It is affordable to be quite vague when it comes to order-of-magnitude complexity

Writing an algorithm in a particular programming language or using a particular compiler can obviously make a difference in the final running time. However, if the algorithm uses conventional basic instructions, these differences will consist of only a constant factor per basic instruction. This means that the big-O complexity is invariant under such implementational fluctuations.

In other words,

- as long as the basic set of allowed elementary instructions is agreed on
- as long as any shortcuts taken in high-level descriptions do not hide unbounded iterations of such instructions, but merely represent finite clusters of them,

big-O-time estimates are robust

The logarithmic base is of no importance either.

For any fixed K ,

the number $\log_2 N$ and $\log_K N$ differ only by a constant factor,

hence this difference can also be hidden under the big-O notation

Consequently,

logarithmic-time performance is referred to as $O(\log N)$, not $O(\log_2 N)$

The robustness of the big-O notation

constitutes both its strength and its weakness

Two algorithms

- i) logarithmic-time algorithm A,
- ii) algorithm B runs in linear time

B runs faster on sample inputs than A!

The reason is in the hidden constants

- Taken into account the constant number of elementary instructions between checkpoints,
- the programming language in which the algorithm is coded,
- the compiler which translates it downwards,
- the basic machine instructions used by the computer running the machine code &
- the very speed of that computer.

- Algorithm A performs in time bounded by $K \log_2 N$ microseconds
B performs within $J \times N$ microseconds,
 - but with K being 1000 and J being 10 this means, that every input of length less than a thousand (actually, the precise number is 996), algorithm B is superior to A
 - Only when inputs of size 1000 or more are reached does the difference between N and $\log_2 N$ become apparent,
 - As already hinted, when the difference starts paying off, it does so very handsomely:
 - By the time inputs of size of million are reached, algorithm A becomes upwards of 500 times more efficient than algorithm B
 - For inputs of size a billion the improvement is over 330,000-fold!
-
- A user who is interested only in inputs of length less than a thousand should definitely adopt algorithm B, despite A's order-of-magnitude superiority

- In most cases, however, the constant factors are not quite as far apart as 10 and 1000,
- Hence big-O estimates are usually far more realistic than in this contrived example

The moral of the story is

- to first search for a good and efficient algorithm stressing big-O performance,
- then to try improving it by tricks of the kind used earlier to decrease the constant factors involved

In any event,

- since big-O efficiency can be misleading,
- candidate algorithms should be run experimentally and
- their time performances for various typically occurring kinds of inputs should be tabulated

The robustness of big-O estimates coupled with the fact that in the majority of cases algorithms that are better in the big-O sense are also better in practice, renders the study of order-of-magnitude time complexity the most interesting to computer scientists

Big-O estimates and similar notions are used although they may hide issues of possible practical importance, such as constant factors.

Time Analysis of Nested Loops

Complicated algorithms that involve many intertwined control structures & contain possibly recursive subroutines can become quite difficult to analyze

The bubblesort algorithm consists of two loops nested as follows:

- (1) do the following $N - 1$ times:
 - (1.1) do the following $N-1$ times:

The inner loop is executed $N-1$ times for each of the $N-1$ times the outer loop is executed

As before, everything else is constant;

Hence the total time performance of bubblesort is on the order of

$$(N-1) \times (N-1) = N^2 - 2N + 1$$

N^2 is the dominant term of the expression meaning that the other parts, namely, the $- 2N$ and the $+1$, get "swallowed" by the N^2 when the big-O notation is used

Consequently, bubblesort is an $O(N^2)$, or quadratic time, algorithm.

Improved bubblesort version, allowed for traversing smaller and smaller portions of the input list

The first traversal is of $N-1$ elements,

the next is $N-2$ elements, and so on

The time analysis, therefore, results in the sum:

$$(N-1) + (N-2) + (N-3) + \dots + 3 + 2 + 1$$

Max Subsequence Problem

- Given a sequence of integers A_1, A_2, \dots, A_n , find the maximum possible value of a **subsequence** A_i, \dots, A_j .
- Numbers can be negative.
- You want a **contiguous** chunk with largest sum.
- Example: $-2, 11, -4, 13, -5, -2$
- The answer is 20 (subseq. A_2 through A_4).
- We will discuss **4 different algorithms**, with time complexities $O(n^3)$, $O(n^2)$, $O(n \log n)$, and $O(n)$.
- With $n = 10^6$, algorithm 1 may take > 10 years; algorithm 4 will take a fraction of a second!

Algorithm 1 for Max Subsequence Sum

- Given A_1, \dots, A_n , find the maximum value of $A_i + A_{i+1} + \dots + A_j$
0 if the max value is negative

```
int maxSum = 0;
for( int i = 0; i < a.size(); i++ )
for( int j = i; j < a.size(); j++ )
{
    int thisSum = 0;
    for( int k = i; k <= j; k++ )
        thisSum += a[ k ];
    if( thisSum > maxSum )
        maxSum = thisSum;
}
return maxSum;
```

$\updownarrow O(1)$

$\updownarrow O(1)$

$\updownarrow O(1)$

$\updownarrow O(1)$

$\updownarrow O(j-i)$

$$O\left(\sum_{j=i}^{n-1} (j-i)\right) \quad O\left(\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j-i)\right)$$

- Time complexity: $O(n^3)$

Algorithm 2

- ⌘ Idea: Given sum from i to $j-1$, we can compute the sum from i to j in constant time.
- ⌘ This eliminates one nested loop, and reduces the running time to $O(n^2)$.

```
int maxSum = 0;

for( int i = 0; i < a.size(); i++ )
    int thisSum = 0;
    for( int j = i; j < a.size(); j++ )
    {
        thisSum += a[ j ];
        if( thisSum > maxSum )
            maxSum = thisSum;
    }
return maxSum;
```

Algorithm 3

- § This algorithm uses divide-and-conquer paradigm.
- § Suppose we split the input sequence at midpoint.
- § The max subsequence is entirely in the **left half**, entirely in the **right half**, or it **straddles the midpoint**.
- § Example:

left half		right half
4 -3 5 -2		-1 2 6 -2

- § Max in left is 6 (A1 through A3); max in right is 8 (A6 through A7). **But straddling max is 11 (A1 thru A7).**

Algorithm 3 (cont.)

§ Example:

left half		right half
4 -3 5 -2		-1 2 6 -2

§ Max subsequences in each half found by recursion.

§ How do we find the straddling max subsequence?

§ **Key Observation:**

- Left half of the straddling sequence is the max subsequence ending with -2.
- Right half is the max subsequence beginning with -1.

§ A linear scan lets us compute these in $O(n)$ time.

Algorithm 3: Analysis

- § The divide and conquer is best analyzed through recurrence:

$$T(1) = 1$$

$$T(n) = 2T(n/2) + O(n)$$

- § This recurrence solves to $T(n) = O(n \log n)$.

Algorithm 4

2, 3, -2, 1, -5, 4, 1, -3, 4, -1, 2

```
int maxSum = 0, thisSum = 0;

for( int j = 0; j < a.size(); j++ )
{
    thisSum += a[ j ];

    if ( thisSum > maxSum )
        maxSum = thisSum;
    else if ( thisSum < 0 )
        thisSum = 0;
}
return maxSum;
}
```

- Time complexity clearly $O(n)$
- But why does it work? I.e. proof of correctness.

Proof of Correctness

- § Max subsequence cannot **start or end** at a negative A_i .
- § More generally, the max subsequence cannot have a prefix with a negative sum.

Ex: -2 11 -4 13 -5 -2

- § Thus, if we ever find that A_i through A_j sums to < 0 , then we can advance i to $j+1$
 - Proof. Suppose j is the first index after i when the sum becomes < 0
 - The max subsequence cannot start at any p between i and j . Because A_i through A_{p-1} is positive, so starting at i would have been even better.

Algorithm 4

```
int maxSum = 0, thisSum = 0;
for( int j = 0; j < a.size( ); j++ )
{
    thisSum += a[ j ];

    if ( thisSum > maxSum )
        maxSum = thisSum;
    else if ( thisSum < 0 )
        thisSum = 0;
}
return maxSum
```

- The algorithm resets whenever prefix is < 0 . Otherwise, it forms new sums and updates maxSum in one pass.