IE304 Algorithms

By

Rajeev Wankar

wankarcs@uohyd.ac.in

School of Computer and Information Sciences
University of Hyderabad, Hyderabad

What to learn?

- **Design Paradigms**
- **Algorithm Analysis**
- **Theory of NP-Completeness**

What to know?

- **Syllabus**
  - Download from the course web site
- **Books**
  - Fundamental of Computer Algorithm by E. Horowitz and S. Sahni
  - Introduction to Algorithm, Cormen et al.
  - Or any other referenced book

Syllabus

- **UNIT-I:**
  - Analysis of Algorithms: Asymptotic Notation; Best, worst and average case analysis of algorithms;
  - Solving recurrence relations using substitution method, generating functions, Master's theorem etc. (Basic Akra-Bazzi Theorem)
  - Warm-up to complexity analysis: Heap data structure, priority queue application, Best, worst and average case analysis of a few sorting algorithms like heap sort, insertion, bubble, selection, counting and radix sort algorithms.
  - Strategies for problem solving

Syllabus

- **UNIT-II:**
  - Divide and Conquer strategy: Time complexity analysis for Merge Sort and Quick Sort Algorithms
- **UNIT-III:**
  - Greedy strategy: Theoretical foundation of greedy strategy:
  - Matroids Algorithms for solving problems like Knapsack Problem (Fractional), Minimum Spanning Tree problem;
  - Shortest Paths, Job Scheduling, Huffman's code etc., along with proofs of corrections and complexity analysis

Syllabus

- **UNIT-IV:**
  - Dynamic Programming strategy: Identify situations in which greedy and divide and conquer strategies may not work.
  - Understanding of optimality principle.
  - Technique of memorization. Applications to problems like Coin change, 0/1 and 0/n-Knapsack, Shortest Paths, Optimal Binary Search Tree (OBST), Chained Matrix Multiplication, Traveling Salesperson Problem (TSP) etc.

Syllabus

- **UNIT-V:**
  - Backtracking and Branch & Bound strategies: State space tree construction, traversal techniques and solving problems like 0/1 and 0/n knapsack, TSP, Applications of Depth First Search:
  - Topological sorting, Finding strongly connected components and game problems.

Syllabus

- **UNIT-VI:**
  - Theory of NP-Completeness: Complexity classes of P, NP, NP-Hard, NP-Complete, Polynomial reductions, Cook's theorem.
  - Discussion of problems: Satisfiability(SAT), CNF-SAT, Min-Vertex Cover, Max-Clique, Graph Coloring, NP-Completeness proofs.

What to know?

- **Course Material (will be updated soon)**

  - **http://scis.uohyd.ac.in/~wankarcs/algo-25.html**
  - **https://rajeevwankar.wixsite.com/mysite/algorithms**

Algorithms

Abu Ja'far Mohammed ibn Musa al Khowarizmi

- Why do we need algorithms?
    - To solve problems

- What is a "problem"?
    - A task to be performed
    - We can think of a "problem" in terms of inputs and matching outputs

What is an algorithm

Concise Oxford dictionary: “Process or rule for calculation”.

Webster dictionary: “Any special method for solving certain kind of problem”.

Computer Science: “Precise method usable by the computer for the solution of a problem”.

An **algorithm** is composed of finite number of steps, each of which may require one or more operation to be performed. These operations must be-

Definite: it must be clear that what is to be done, “add 6 or 3 to a is not permitted”.

Effective: each step is such that it can, in principle, be done by a person using pencil and paper in a finite amount of time.

Terminate: it must terminate after a finite amount of time.

Study of algorithm

How to device algorithm?

To design an algorithm that is easy to understand, code and debug

To design an algorithm that makes efficient use of the computer's resources

- »Space (main memory)
- »Time
- »Secondary Storage
- »Networks & Energy

Algorithm design paradigms-

- Divide and conquer
- Greedy method
- Basic search and traversal techniques
- Dynamic programming
- Backtracking
- Branch and bound

How to express algorithms: Structured programming

How to validate algorithms: To show that the algorithm works correctly for all possible legal inputs.

- **Validation** checks whether an algorithm **solves the intended problem**.
- It's typically done through **testing**, **simulation**, and **analysis of behavior** on a variety of inputs.

- **Key Points:**

- Often **empirical** (based on running the algorithm).
- Involves designing test cases to **observe** outcomes.
- Can find bugs but **cannot guarantee correctness for all inputs**.

- Answers the question:

“Does this algorithm appear to work as intended?”

- **Example:**
- Run a sorting algorithm on 1,000 test arrays and verify all outputs are sorted.

- **Verification** is about **internal correctness** **Proof of correctness:**

- A **formal mathematical proof** that the algorithm is correct for **all valid inputs**.
- Based on **induction**, **loop invariants**, **recurrence relations**, etc.

- **Key Points:**
- It is **theoretical and rigorous**.
- It shows that **no matter the input**, the algorithm will **always** produce the correct result.

- Typically involves:
  - Proving partial correctness (if it terminates, it's correct)
  - Proving total correctness (it always terminates **and** gives correct result)

- **Example:**
- Using **induction** to prove that Merge Sort always returns a sorted array and uses correct comparisons.

Category	Example Algorithm(s)	Typical Proof Technique
Divide & Conquer	Merge Sort, Quick Sort	Induction
Dynamic Programming	LCS, Knapsack, Floyd-Warshall	Induction + Optimal Substructure
Greedy Algorithms	Kruskal, Prim, Dijkstra	Greedy-choice & Optimal Substructure proof
Graph Traversal	BFS, DFS	Loop Invariant, Reachability
Shortest Path	Dijkstra, Bellman-Ford	Invariants + Induction
Minimum Spanning Tree	Kruskal, Prim	Cut Property + Greedy choice
Backtracking	N-Queens, Sudoku Solver	Recursion correctness
Binary Search	Search in sorted array	Loop Invariant + Termination
Sorting (Comparison-based)	Bubble, Insertion, Selection	Loop Invariant or Induction
Union-Find (Disjoint Sets)	Union by Rank, Path Compression	Data structure correctness
String Matching	KMP, Rabin-Karp	Pattern preservation
Recursion-based Algorithms	Tower of Hanoi, DFS	Induction on recursion depth

How to express algorithms: Structured programming

How to validate algorithms: To show that the algorithm works correctly for all possible legal inputs.

How to analyze algorithms: The process of computing. How much computing time and storage an algorithm will require is called as analysis of an algorithm.

Basics of Algorithm Analysis

- How to measure efficiency
- Running time of an algorithm
- Asymptotic algorithm analysis
    - Growth rate
    - Upper bounds of growth rate
    - Lower bounds of growth rate
    - Θ Notation

- Space complexity
- Tradeoffs of implementations
- Analyzing Problems – Optimal solution

How to test algorithms: It consists of two phases:

1. Debugging
2. Profiling

Debugging is a process of executing programs on data set and to determining if faulty results occur, and if so, to correct them.

“The proof of the correctness is much more valuable than thousands of tests, since it guarantees that the program works correctly for all possible inputs”

Profiling is the process of executing a correct program on data sets and measuring time and space it takes to compute results.

Analysis of algorithm

- Why people analyze algorithms?
- Analyzing an algorithm is an intellectual activity, it is a fun.
- Prediction about algorithm is gratifying activity when we succeed.
- To device new ways to do certain task even faster.

- In conventional computers, instructions are carried out one at a time and major cost of the algorithm depends on the operations it perform.

- Given an algorithm to be analyzed the ***first task*** is:

  - ❖ to determine the operations to be performed and what their relative cost is

- the ***second task*** is

  - ❖ to determine sufficient set of data which cause algorithm to exhibit all patterns of behavior.

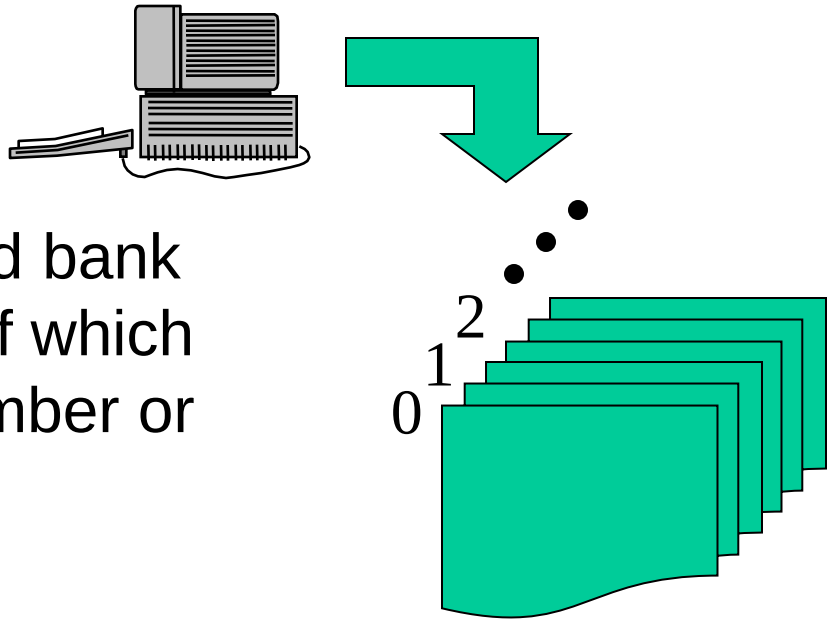
- In producing the complete analysis of the algorithm we distinguish between two phases- priory analysis, posteriori analysis.

- ***Priori analysis:*** Obtain a function of relevant parameter which bounds the computing time of the algorithm.

- ***Posteriori analysis:*** We collect actual statistics about the algorithm consumption of time and space it requires when executing.

The Random Access Machine (RAM) Model

- A **CPU**



- An potentially unbounded bank of **memory** cells, each of which can hold an arbitrary number or character

- Memory cells are numbered, and accessing any cell in memory takes unit time.

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model

- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

Suppose there is a statement x := x + y; we want to determine total time it requires- we have two items of information:

1. Statements frequency.
2. Time for one execution.

Consider the following program segments

```
x := x+y;
```

1

```
for i:= 1 to n do
    x := x+y
```

n

```
for i := 1 to n do
    for j := 1 to n do
        x := x+y;
```

n^2

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

Algorithm ***arrayMax***(A, n)	# operations
$currentMax \leftarrow A[0]$	1
for $i \leftarrow 1$ **to** $n - 1$ **do**	$n - 1$
if $A[i] > currentMax$ **then**	$(n - 1)$
$currentMax \leftarrow A[i]$	$(n - 1)$
{ increment counter i }	$(n - 1)$
return $currentMax$	1
Total	$4n - 1$

Estimating Running Time

- Algorithm ***arrayMax*** executes $4n - 1$ primitive operations in the worst case. Define:

  a = Time taken by the fastest primitive operation

  b = Time taken by the slowest primitive operation

- Let $T(n)$ be worst-case time of ***arrayMax.*** Then

$$a\,(4n - 1) \leq T(n) \leq b\,(4n - 1)$$

- Hence, the running time $T(n)$ is bounded by two linear functions

Growth Rates

- Suppose we are plotting a function like:

  $f(n)=a \cdot n^k$

- Taking logarithm on both sides:

  $\log(f(n)) = \log(a) + k \cdot \log(n)$

- This is in the form:

  $Y=C + kX$ where:

  $Y=\log(f(n))$

  $X=\log(n)$

  $C=\log(a)$

- k is the slope

Growth Rates

- What does it Mean?
- When we plot log(f(n)) vs log(n):
  - We get a straight line if f(n) is a power function (like n^2, n^3, etc.).
  - The slope of that line is k, which tells you how fast the function grows.

Growth Rates

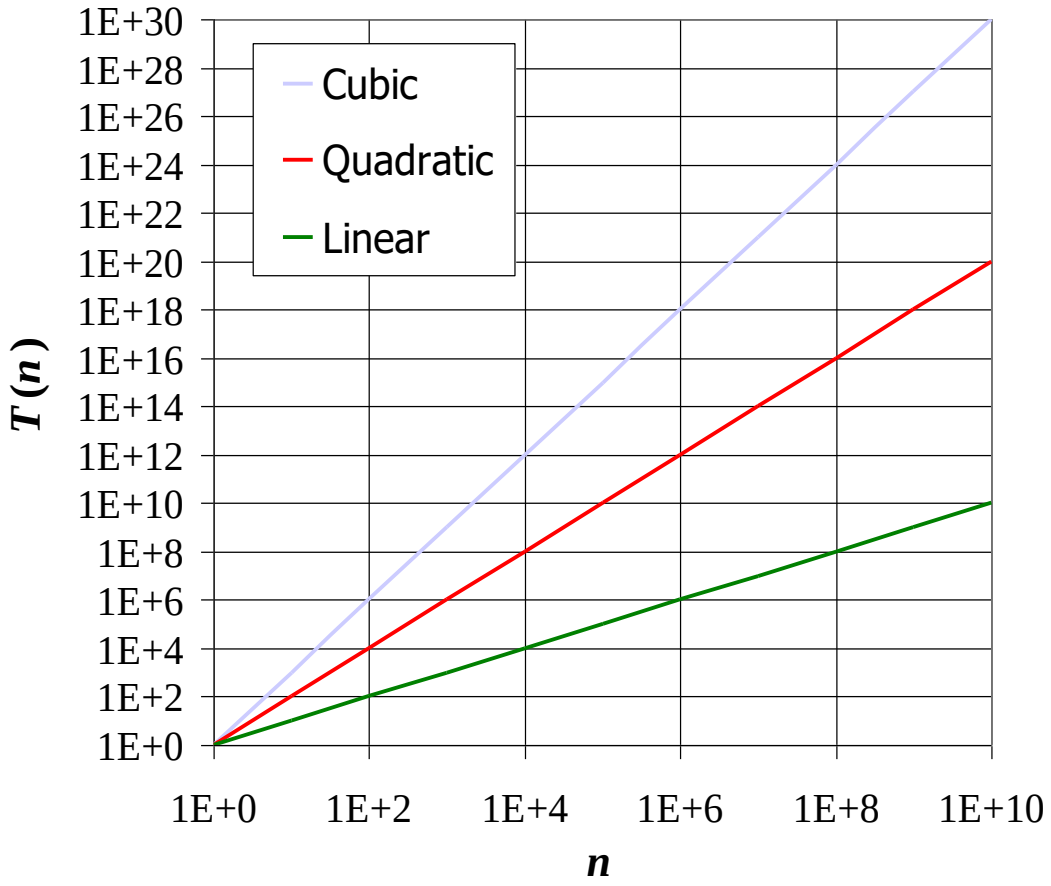
Function f(n)	Log-Log Plot Slope	Growth Rate Description
f(n) = n	1	Linear growth
f(n) = n^2	2	Quadratic growth
f(n) = n^3	3	Cubic growth
f(n) = logn	0 (flattened line)	Sublinear

In Simple Terms:

- A **steeper slope → faster-growing function**.

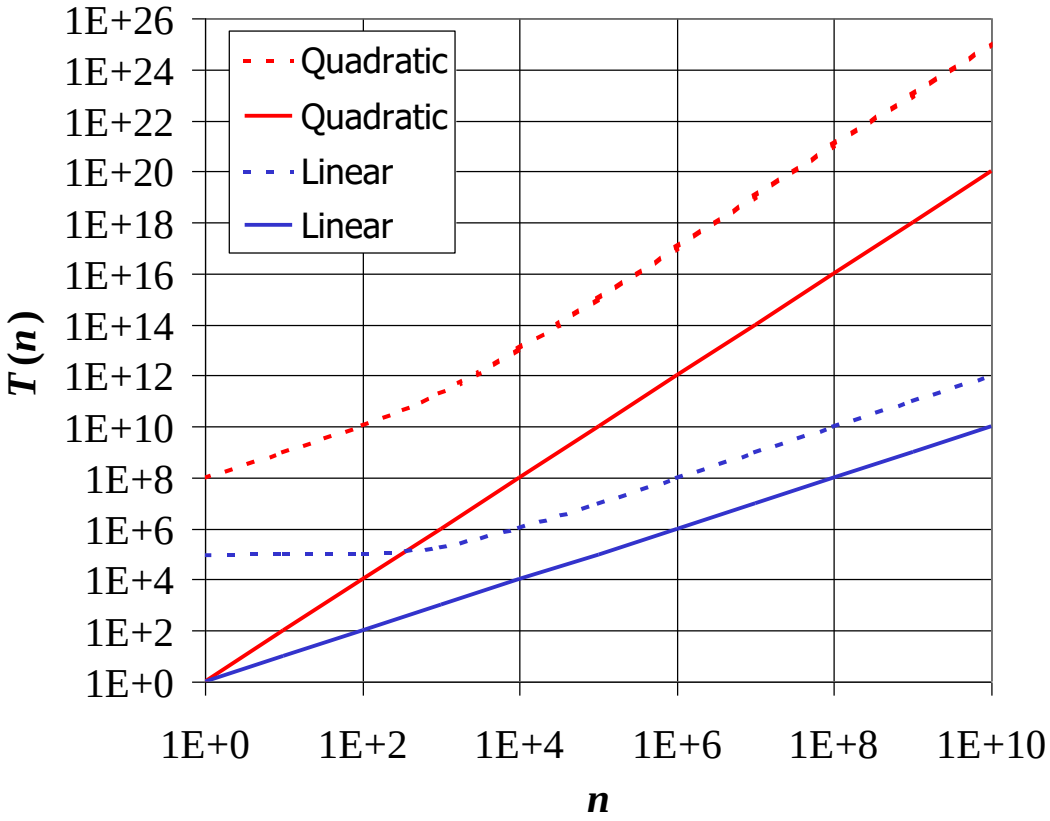
Growth Rates

- Growth rates of functions:
  - Linear ≈ n
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
- In a log-log chart, the slope of the line corresponds to the growth rate of the function



Constant Factors

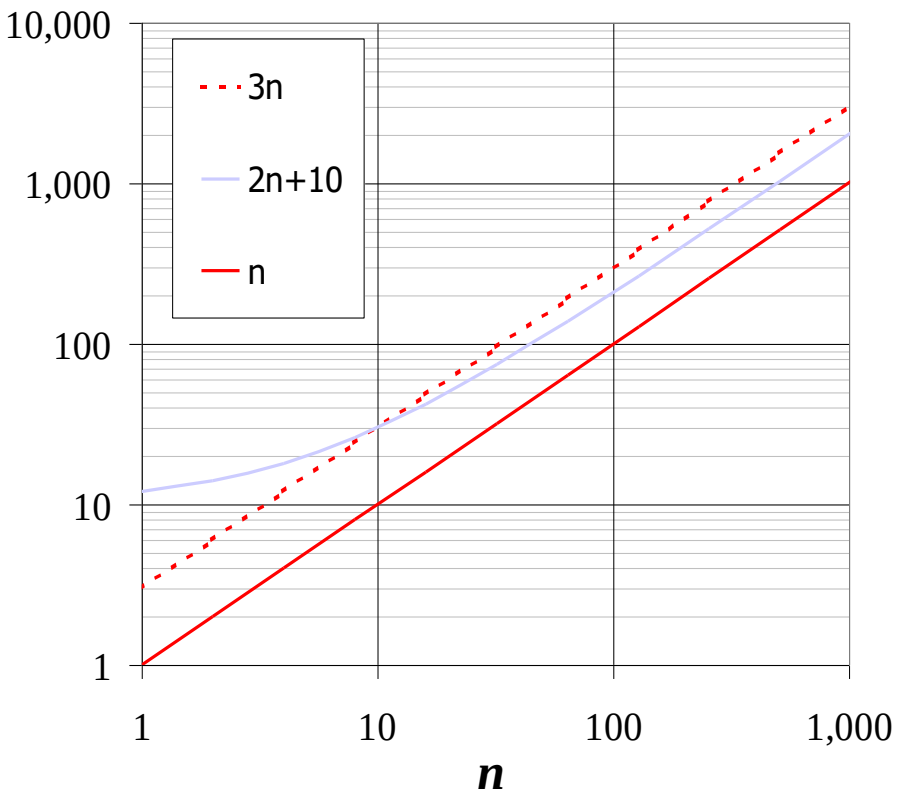
- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2 n + 10^5$ is a linear function
  - $10^5 n^2 + 10^8 n$ is a quadratic function



Big-Oh Notation

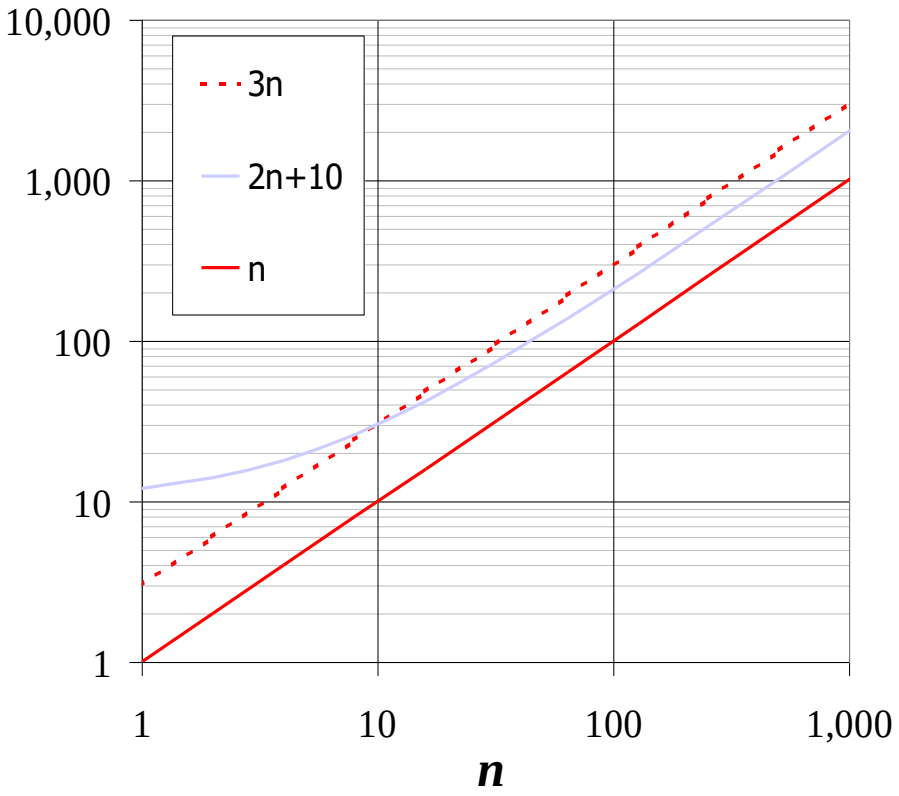
- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

  $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$
- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$



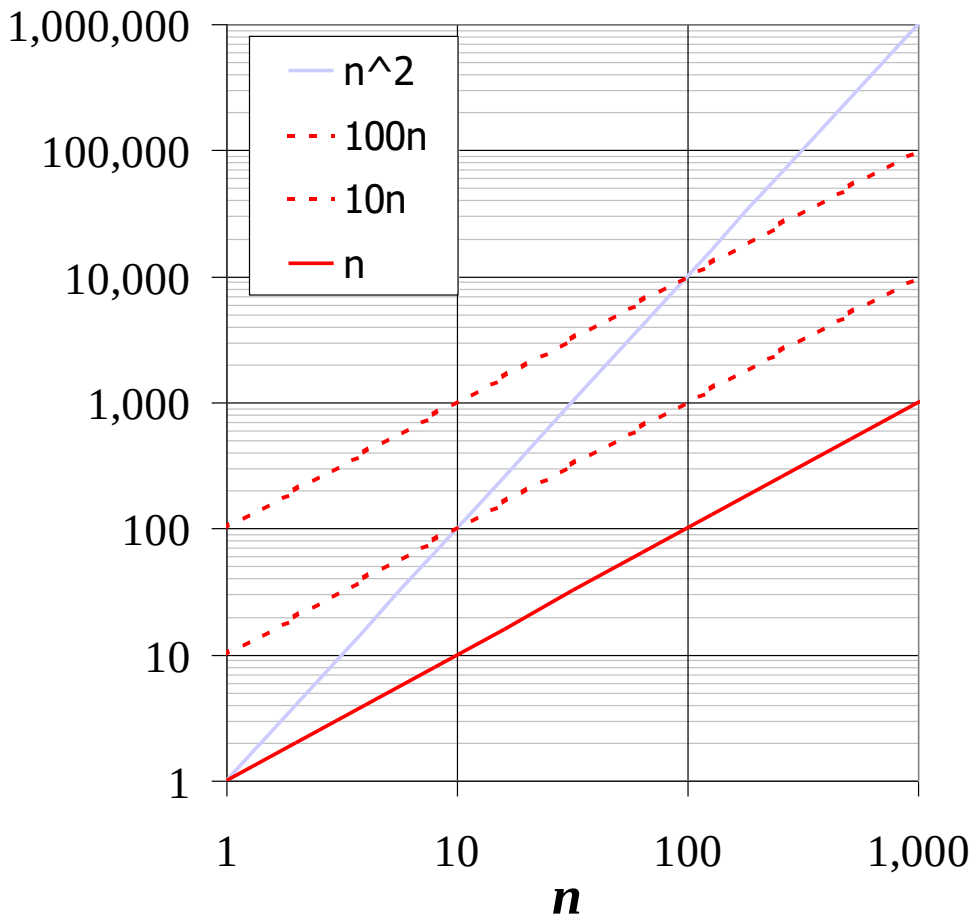
Big-Oh Notation

- Big-Oh gives an upper bound on the growth of a function.
- It tells us:
- "The algorithm will not grow faster than this."



Big-Oh Example

- Example: the function n^2 is **not** $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - The above inequality cannot be satisfied since c must be a constant



What is the time complexity of the algorithm requiring n^2+8n operations?

n	n^2		8n
1	1	<	8
2	4	<	16
3	9	<	24
4	16	<	32
5	25	<	40
6	36	<	48
7	49	<	56
8	64	=	64
9	81	>	72

What if c = 2 ?

$2n^2$
2
8
18
32
50
72
98
128
162

What is the time complexity of the algorithm requiring n^2+8n operations?

For all $n \geq 4(n_0)$ and $c = 2$, $T(n) = O(n^2)$

More Big-Oh Examples

- $7n-2$

  $7n-2$ is $O(n)$

  need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \bullet n$ for $n \geq n_0$

  this is true for $c = 7$ and $n_0 = 1$

- $3n^3 + 20n^2 + 5$

  $3n^3 + 20n^2 + 5$ is $O(n^3)$

  need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \bullet n^3$ for $n \geq n_0$

  this is true for $c = 4$ and $n_0 = 21$

- $3 \log n + \log \log n$

  $3 \log n + \log \log n$ is $O(\log n)$

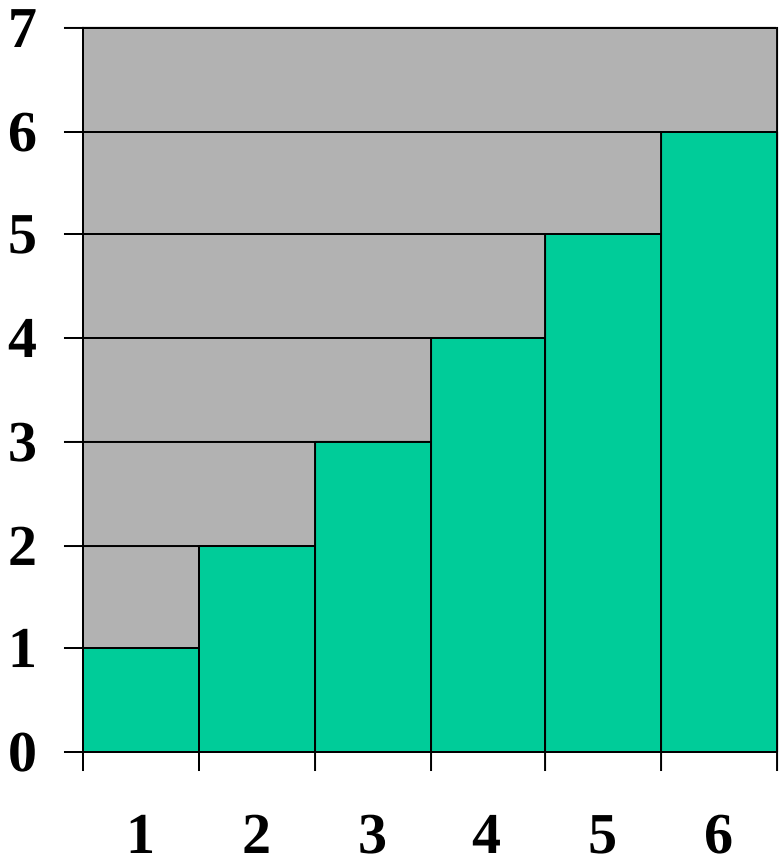
  need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + \log \log n \leq c \bullet \log n$ for $n \geq n_0$ this is true for $c = 4$ and $n_0 = 2$

Big-Oh Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Arithmetic Progression

- The running time of ***prefixAverages1*** is ***O***(1 + 2 + …+ ***n***)
- The sum of the first ***n*** integers is ***n***(***n*** + 1) ***/*** 2
  - There is a simple visual proof of this fact
- Thus, algorithm ***prefixAverages1*** runs in ***O***(n^2) time



Prefix Averages

- The following algorithm computes prefix averages in linear time by keeping a running sum

```
Algorithm prefixAverages2(X, n)
    Input array X of n integers
    Output array A of prefix averages of X                #operations
    A ← new array of n integers                           n
    s ← 0                                                 1
    for i ← 0 to n − 1 do                                 n
        s ← s + X[i]                                      n
        A[i] ← s / (i + 1)                                n
    return A                                              1
```

- Algorithm ***prefixAverages2*** runs in $O(n)$ time

Theorem: If $A(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots\ldots\ldots + a_1 n + a_0$ is a polynomial of degree m then $A(n) = O(n^m)$.

Proof is left as an exercise

Most common computing time for the algorithms are:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

Ω -notation: *f(n)* = Ω (g(n)) {read as f of n equals omega of g of n} iff there exist two positive constants *c* and n_o such that-

$$|f(n)| \geq c|g(n)| \quad \text{for all} \quad n > n_0$$

- Meaning
  - –Algorithm has lower bound to its growth rate of *f*(*n*)

- Meaning
  - –Algorithm has lower bound to its growth rate of $f(n)$
  - –Big-Omega gives a lower bound on growth.
  - –It tells us:
  - –"The algorithm will run at least this fast."

- Theta notation - Θ
- Definition : $f(n) = \Theta(g(n))$ iff there exist positive constants c_1, c_2 and n_o such that

  $$c_1|g(n)| \le |f(n)| \le c_2|g(n)| \quad \text{for all } n > n_0$$

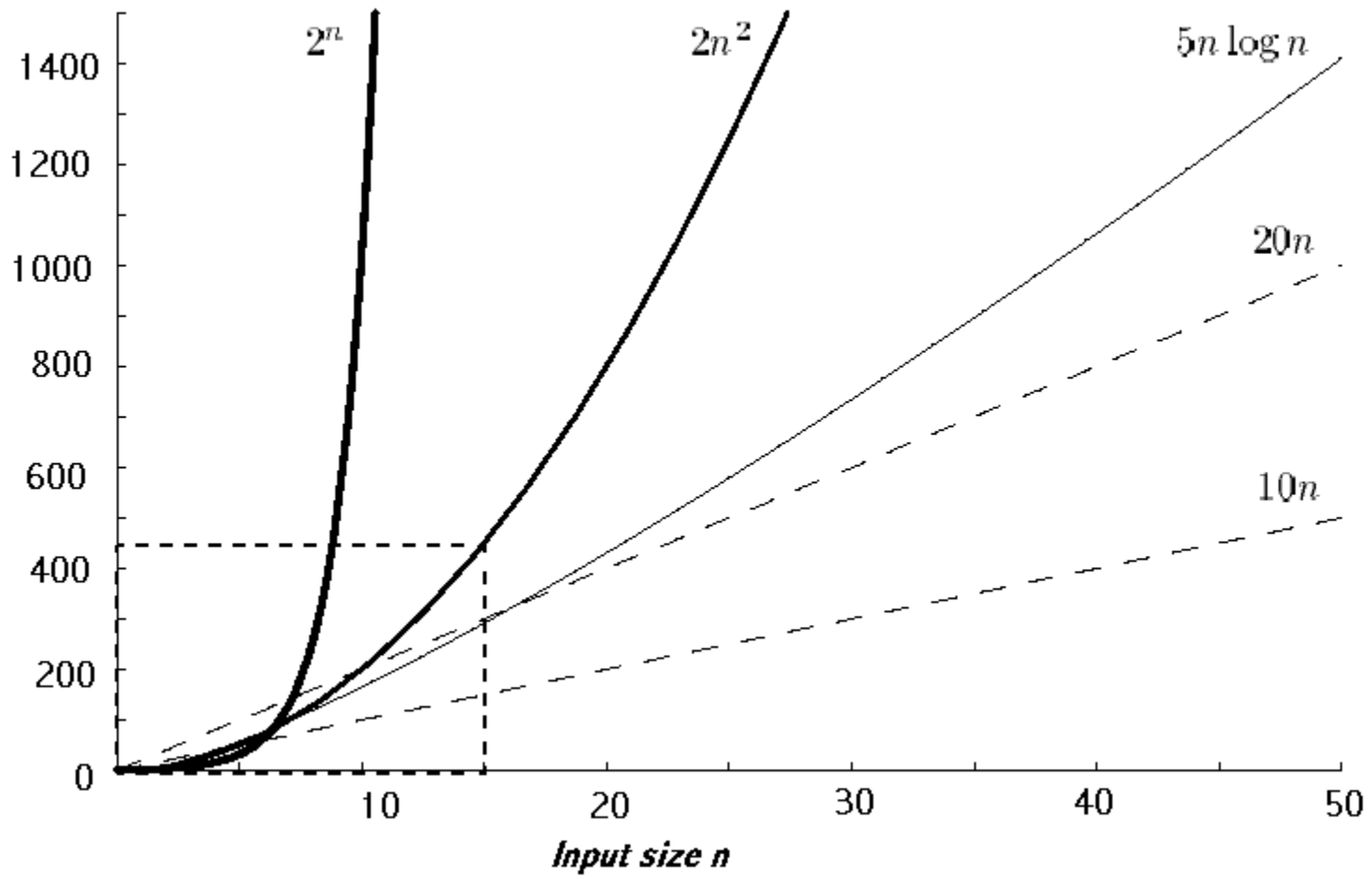
- Meaning
  - ***Big-Theta gives a tight bound — both upper and lower.***
  - ***It tells us:***
  - **"The algorithm runs exactly this fast, asymptotically."**

Asymptotic: $f(n) = \sim o(g(n))$ iff $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 1$

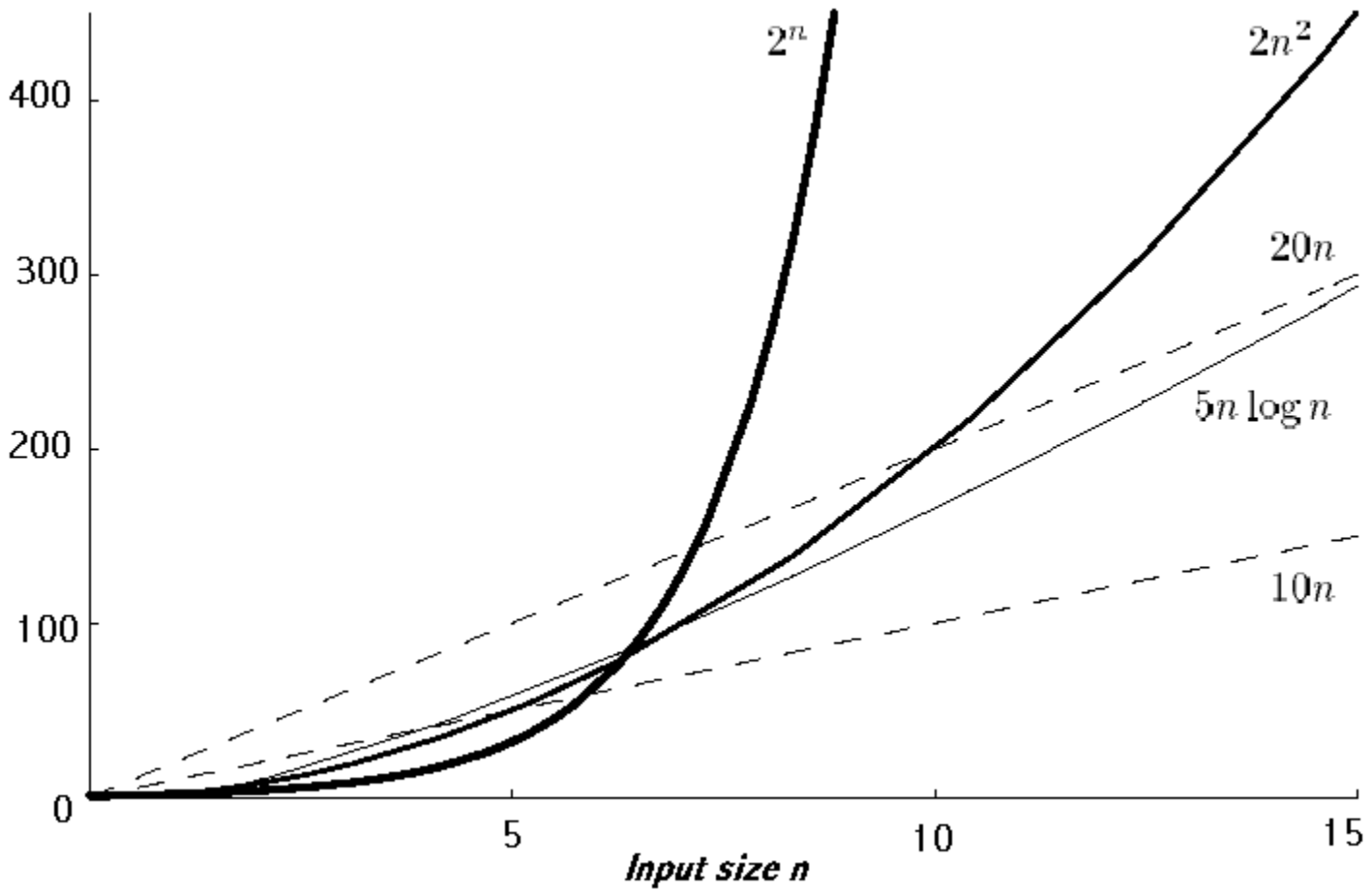
Compare Values of n and T(n)

n	$100\log_2 n$	$20n + 5$	$3n^2 + 7$	2^n
2	100	45	19	4
5	232	105	82	32
10	332	205	307	1024
20	432	405	1207	1048576
100	632	2005	30007	$1.27*10^{30}$

Growth Rate Graph (1)



Growth Rate Graph(2)



Math you need to Review

- Summations
- Logarithms and Exponents

- **properties of logarithms:**

  $\log_b(xy) = \log_b x + \log_b y$

  $\log_b (x/y) = \log_b x - \log_b y$

  $\log_b xa = a\log_b x$

  $\log_b a = \log_x a/\log_x b$

- **properties of exponentials**:

  $a^{(b+c)} = a^b a^c$

  $a^{bc} = (a^b)^c$

  $a^b / a^c = a^{(b-c)}$

  $b = a^{\log_a b}$

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

- Proof techniques
- Basic probability