19CSE302 Design and Analysis of Algorithms

Course Overview

This course aims to provide the fundamentals of algorithm design and analysis specifically in terms of algorithm design techniques, application of these design techniques for real-world problem solving and analysis of complexity and correctness of algorithms

Course Outcomes

CO-1 Evaluate the correctness and analyze complexity of algorithms.
CO-2 Understand and implement various algorithmic design techniques and solve classical problems.
CO-3 Design solutions for real world problems by identifying, applying and implementing appropriate design techniques.
CO-4 Design solutions for real world problem by mapping to classical problems.
CO-5 Analyze the impact of various implementation choices on the algorithm complexity.

Textbook:

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest and Clifford Stein. Introduction to Algorithms. Third Edition, Prentice Hall of India Private Limited; 2009.

References:

1. Michael T Goodrich and Roberto Tamassia. Algorithm Design Foundations - Analysis and Internet Examples. John Wiley and Sons, 2007.

2. Dasgupta S, Papadimitriou C and Vazirani U. Algorithms. Tata McGraw-Hill; 2009.

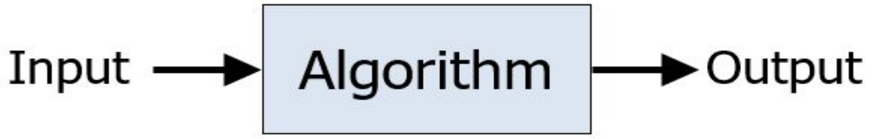
3. Jon Kleinberg, Eva Tardos. Algorithm Design. First Edition, Pearson Education India; 2013.

Key concepts

- Problem
  - Can the problem be solved?
  - How difficult is the problem?
  - Real-life problems to algorithmic problems
- Algorithm
  - How to find suitable algorithm?
  - How to make it efficient?
- Instance
  - Upper and lower limits

What is algorithm?

- We need a well-specified problem that tells what needs to be achieved
- Algorithm solves the problem
- It consists of a sequence of commands that takes an input and gives output



Algorithm principles

- Sequence
  - One command at a time
  - Parallel and distributed computing
- Condition
  - IF
  - CASE
- Loops
  - FOR
  - WHILE
  - REPEAT

Analyzing Algorithms:

Algorithms are most important and durable part of computer science because they can be studied in a language- and machine independent way

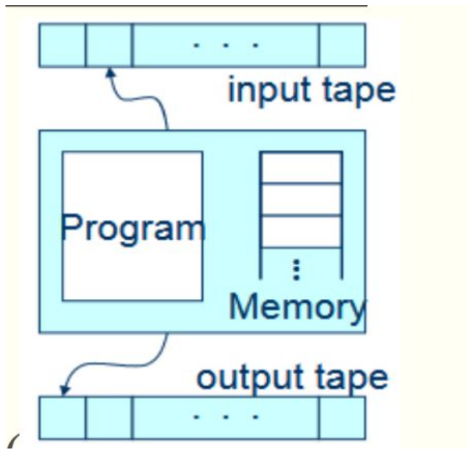
Efficiency of algorithms :

- The RAM model of computation and,
- The asymptotic analysis of worst-case complexity

RAM Model

- Random Access Machine (not R.A. Memory)
- An idealized notion of how the computer works Each "simple" operation (+, -, =, if) takes exactly 1 step.
- Each memory access takes exactly 1 step Loops and method calls are not simple operations, but depend upon the size of the data and the contents of the method.
- Measure the run time of an algorithm by counting the number of steps.
- The program for RAM is not stored in memory.
- Thus we are assuming that the program does not modify itself.

RAM Model Consists of:



- a fixed program
- an unbounded memory
- a read-only input tape
- a write-only output tape
- Each memory register can hold an arbitrary integer ().
- Each tape cell can hold a single symbol from a finite alphabet s

Factors that determine running time of a program

- problem size: n
- basic algorithm / actual processing *
- memory access speed
- CPU/processor speed
- # of processors?
- compiler/linker optimization?

*Primitive operation unit of operation that can be identified in the pseudo-code

Factors that determine running time of a program

- amount of input: n   min. linear increase
- basic algorithm / actual processing depends on algorithm!
- memory access speed by a factor
- CPU/processor speed by a factor
- # of processors? yes, if multi-threading or multiple processes are used.
- compiler/linker optimization? ~20%

- Alg.: MIN (a[1], …, a[n])

```
m ← a[1];
for i ← 2 to n
    if a[i] < m
     then m ← a[i];
```

- **Running time**:
  - the number of primitive operations (steps) executed before termination

T(n) =1 [first step] + (n) [for loop] + (n-1) [if condition] + (n-1) [the assignment in then] = 3n - 1

- Order (rate) of growth:
  - The leading term of the formula

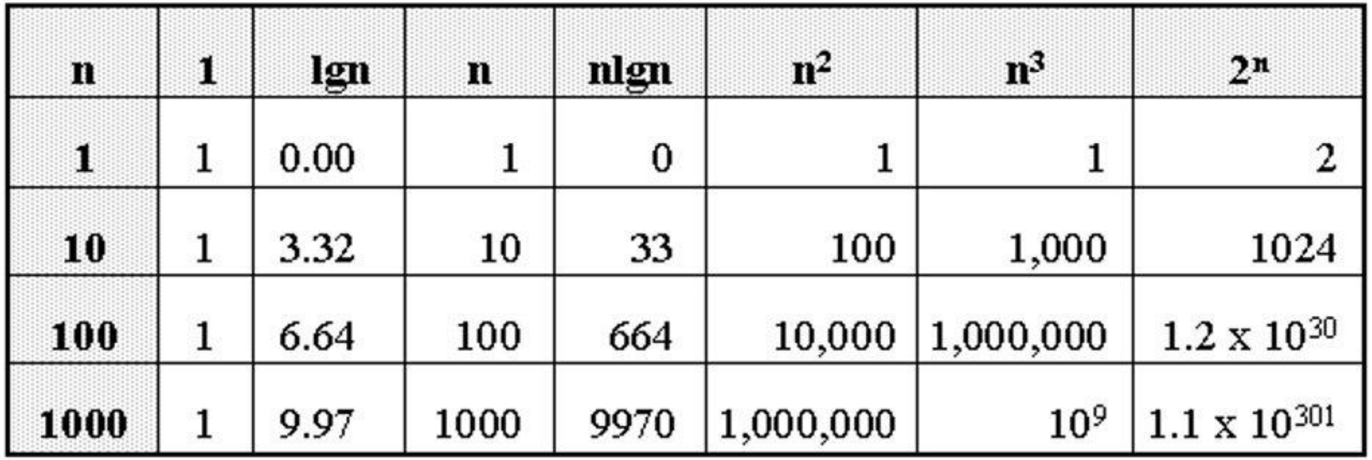
Typical Running Time Functions

- 1 (constant running time):
  - Instructions are executed once or a few times
- logN (logarithmic)
  - A big problem is solved by cutting the original problem in smaller sizes, by a constant fraction at each step
- N (linear)
  - A small amount of processing is done on each input element
- N logN
  - A problem is solved by dividing it into smaller problems, solving them independently and combining the solution

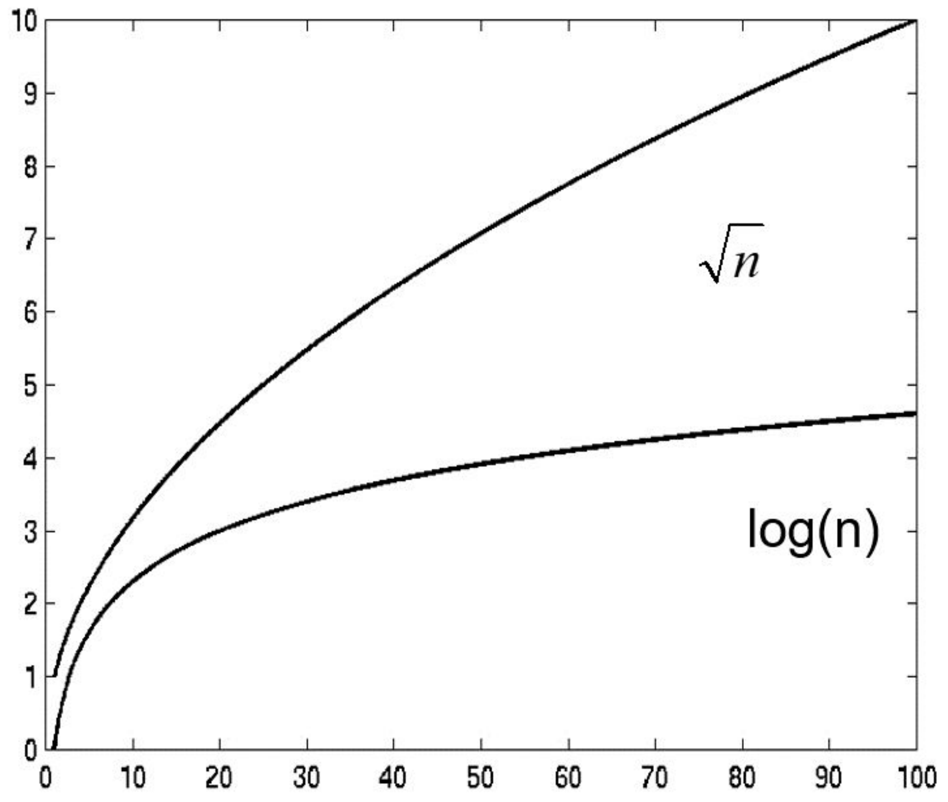
Typical Running Time Functions

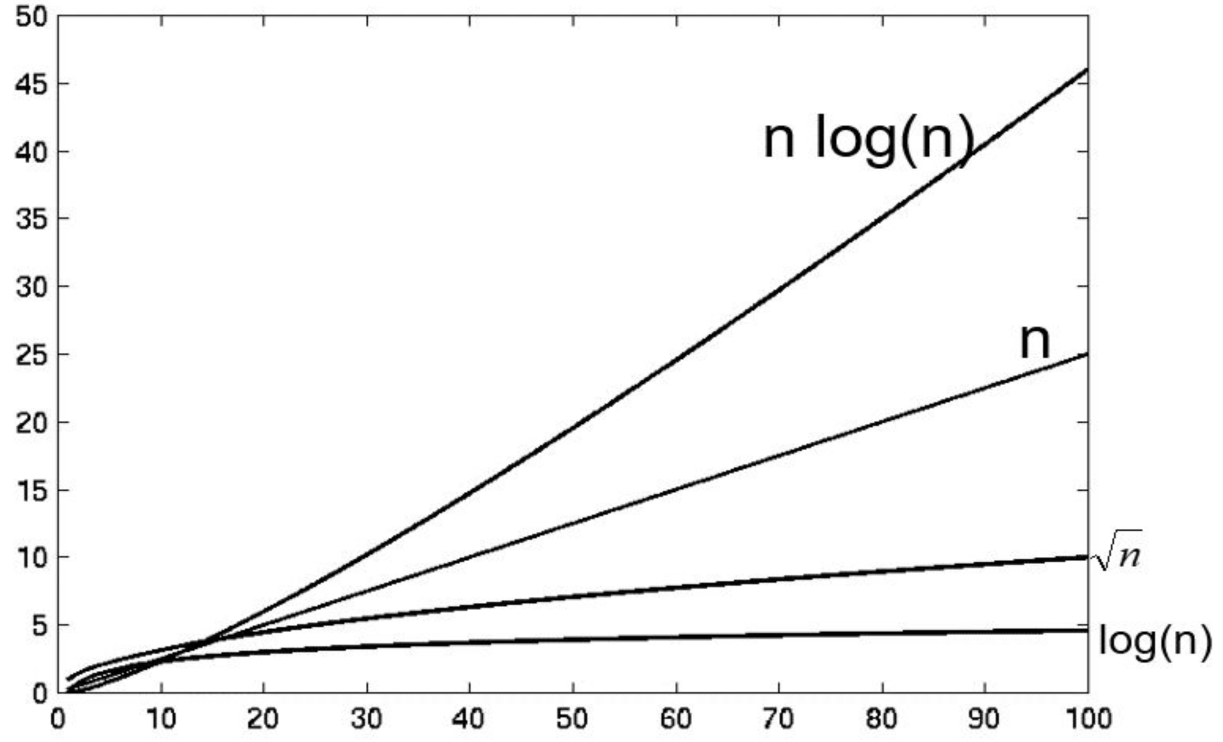
- N^2 (quadratic)
  - Typical for algorithms that process all pairs of data items (double nested loops)
- N^3 (cubic)
  - Processing of triples of data (triple nested loops)
- N^K (polynomial)
- 2^N (exponential)
  - Few exponential algorithms are appropriate for practical use

Growth of Functions

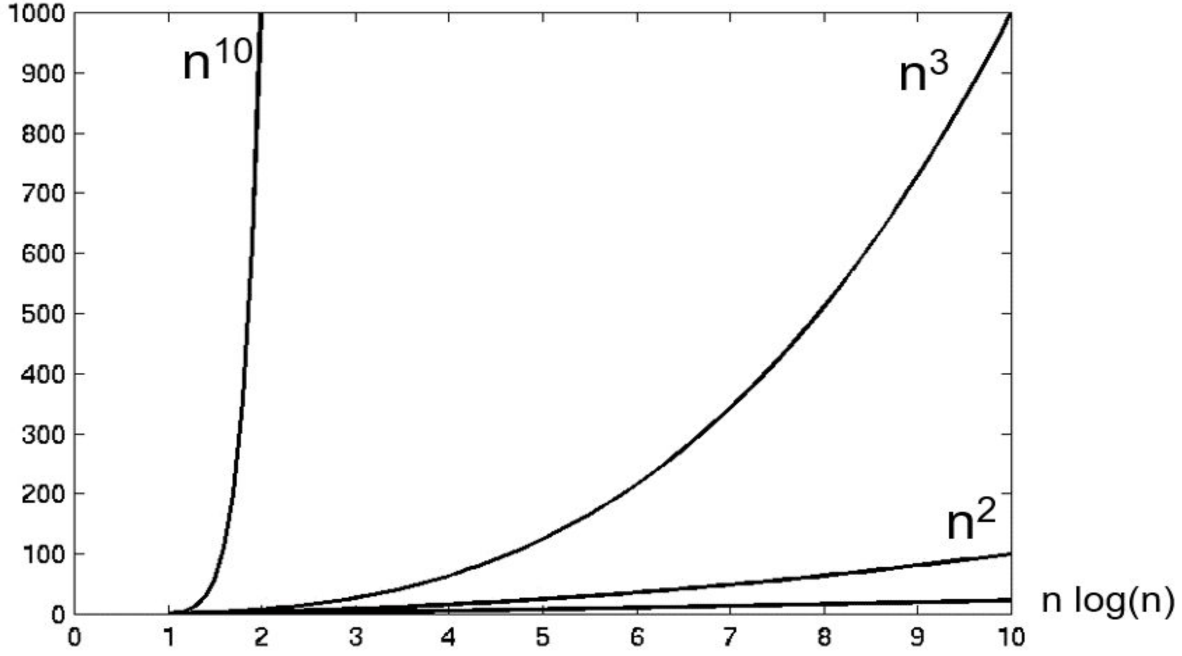


n	1	lgn	n	nlgn	n^2	n^3	2^n
1	1	0.00	1	0	1	1	2
10	1	3.32	10	33	100	1,000	1024
100	1	6.64	100	664	10,000	1,000,000	1.2×10^{30}
1000	1	9.97	1000	9970	1,000,000	10^9	1.1×10^{301}

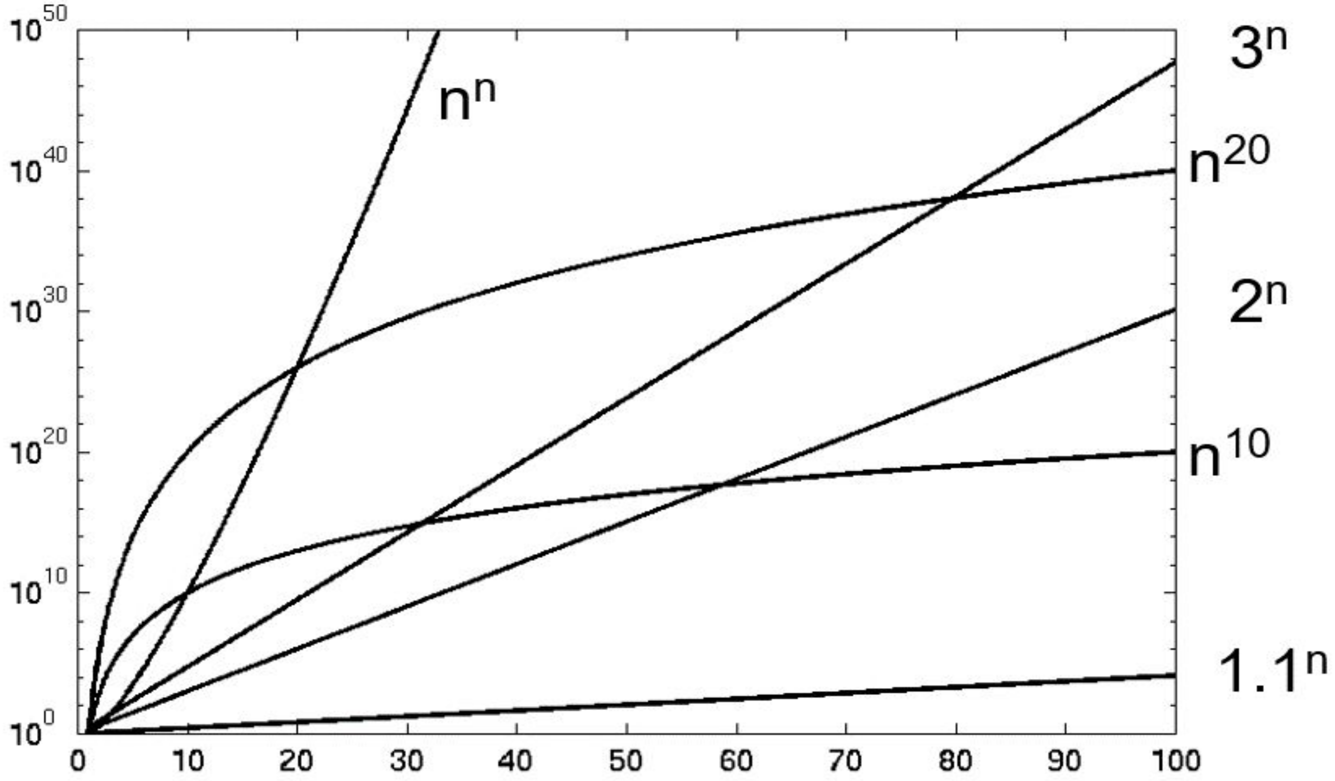

$\sqrt{n}$
log(n)
0
1
2
3
4
5
6
7
8
9
10
0
10
20
30
40
50
60
70
80
90
100



n log(n)
n
$\sqrt{n}$
log(n)
0
5
10
15
20
25
30
35
40
45
50
0
10
20
30
40
50
60
70
80
90
100



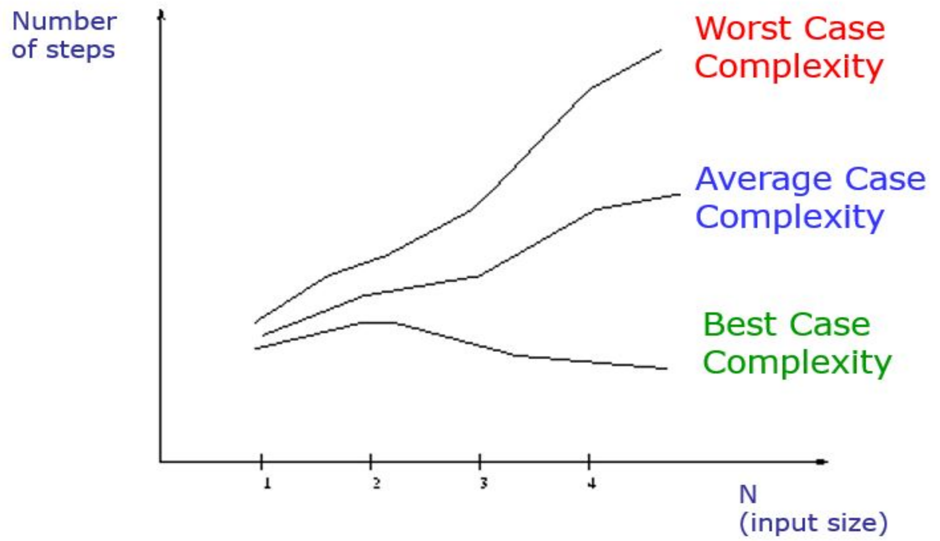
1000
900
800
700
600
500
400
300
200
100
0
0
1
2
3
4
5
6
7
8
9
10
n^{10}
n^3
n^2
n log(n)


10^{50}
10^{40}
10^{30}
10^{20}
10^{10}
10^{0}
0
10
20
30
40
50
60
70
80
90
100
n^n
3^n
n^{20}
2^n
n^{10}
1.1^n

Algorithm Complexity

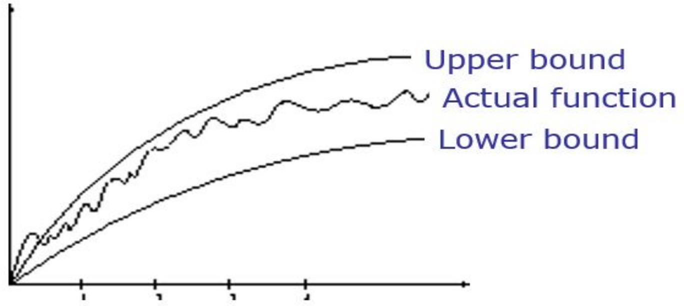
- Worst Case Complexity:
  - the function defined by the *maximum* number of steps taken on any instance of size *n*
- Best Case Complexity:
  - the function defined by the *minimum* number of steps taken on any instance of size *n*
- Average Case Complexity:
  - the function defined by the *average* number of steps taken on any instance of size *n*

Best, Worst, and Average Case Complexity



Doing the Analysis

- It's hard to estimate the running time exactly
  - Best case depends on the input
  - Average case is difficult to compute
  - So we usually focus on worst case analysis
    - Easier to compute
    - Usually close to the actual running time
- Strategy: find a function (an equation) that, for large n, is an upper bound to the actual function (actual number of steps, memory usage, etc.)



Constant Time:

- Constant time means there is some constant k such that this operation always takes k nanoseconds
- A Java statement takes constant time if:
- ❖ It does not include a loop
- ❖ It does not include calling a method whose time is unknown or is not a constant
- If a statement involves a choice (if or switch) among operations, each of which takes constant time, we consider the statement to take constant time
- This is consistent with worst-case analysis

Linear Time

- We may not be able to predict to the nanosecond how long a Java program will take, but do know *some* things about timing:

  -
    ```
    for (i = 0, j = 1; i < n; i++) {
        j = j * i;
    }
    ```
  - This loop takes time k*n + c, for some constants k and c
  - k : How long it takes to go through the loop once
    (the time for j = j * i, plus loop overhead)
  - n : The number of times through the loop
    (we can use this as the “size” of the problem)
  - c : The time it takes to initialize the loop
  - The total time k*n + c is *linear in n*

Constant time is (usually)better than linear time

- Suppose we have two algorithms to solve a task:
  - Algorithm A takes `5000` time units
  - Algorithm B takes `100*n` time units
- Which is better?
  - Clearly, algorithm B is better if our problem size is small, that is, if `n < 50`
  - Algorithm A is better for larger problems, with `n > 50`
  - So B is better on small problems that are quick anyway
  - But A is better for large problems, *where it matters more*
- We usually care most about very large problems
  - But not always!