IIT ROORKEE

FREE ONLINE EDUCATION swayam शिक्षित भारत, उन्नत भारत

NPTEL

NPTEL ONLINE CERTIFICATION COURSE

VLSI Physical Design with Timing Analysis

Lecture – 3: Complexity Analysis for Algorithms

Bishnu Prasad Das

Department of Electronics and Communication Engineering

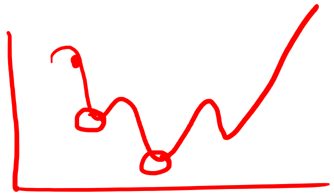
Contents

- Algorithms
- Data Structures
- Complexity Issues: Asymptotic Notations
- NP-Hardness
  - Polynomial Time (P) Algorithms
  - Non-deterministic polynomial time (NP) Algorithms

Algorithms

- **Algorithm**
  - A finite set of instructions to solve a problem.
  - Produces an output within a finite timeframe.
  - Transforms input into the desired output.

- **Some basic algorithmic techniques are**
  - Greedy Algorithms



  - Divide and Conquer Algorithms
  - Dynamic Programming Algorithms
  - Linear/Integer Programming Techniques

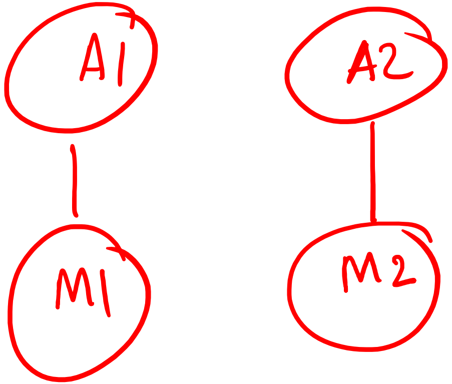
Data Structures

- **Data Structure**
  - Method for storing and organizing data.
  - Aims to simplify data access and modifications.
- **Basic data structures:**
  - Stack, Linked List, Queue, Tree, Graph…..

- **Data structures related to VLSI Physical Design:**
  - Linked List of Blocks
  - Bin-Based Method
  - Neighbor Pointers
  - Corner Stitching

Why study Time and Space complexity?

- Studying the time and space complexity of algorithms is essential for several reasons:
  - Performance Evaluation
  - Algorithm Selection
  - Resource Management
  - Optimization



Running time of an algorithm

- Depends on the type of input
- Depends on the machine
- Depends on the programming language

Example

Linear_search(X, n, key)

1. **for** i = 0 to n-1 do → (n + 1)
2. if(X[i] == key) → n
3. return i → 1
4. return -1 → 1

- In this algorithm, each statement takes a different time to execute.
- Ignore the actual costs and assume each statement takes the same time to execute.

O(n)

So the total time to execute the above algorithm is (2n + 3)

Order of growth

- Remove the constant factors.
  - Now, running time becomes 2n.
- The coefficient can also be ignored.
  - Now, running time becomes n.
- The remaining term is called the **order of growth(n)**.

O (n)

- Examples:
  - $70n\log n \rightarrow O(n\log n)$
  - $8n^2 + 2n + 10 \rightarrow \boxed{O(n^2)}$
  - $2n^3\log n + 10 \rightarrow O(n^3\log n)$

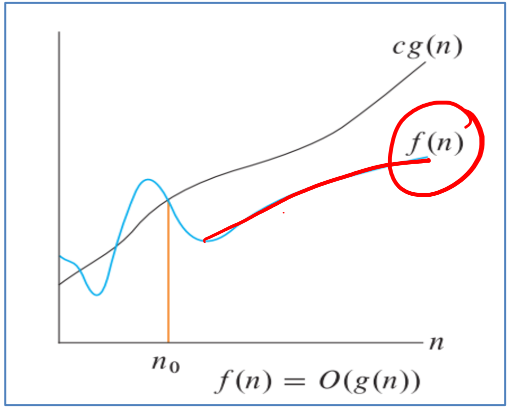
Asymptotic notation

- Analyses the algorithm's running time as the input grows.
- Algorithms need not be implemented in any programming language.
    - Makes analysis faster.
- Measures efficiency of algorithms that won't depend on machine-specific constants.

O(big oh) – Notation

1

- O – Notation characterizes an ***upper bound*** on the asymptotic behavior of a function.



- $f(n) = O(g(n))$ if there exists positive constants c and n_0 such that

c

n_0

$$\mathbf{0 \leq f(n) \leq cg(n)} \text{ for all } \mathbf{n \geq n_0}.$$

Source :Cormen, Leiserson, Rivest, Stein "Introduction to Algorithms"

O – Notation

- Example:
  - $f(n) = 7n^3 + 6n^2 + 100n + 20$
  - $g(n) = n^3$
  - We need to find positive constants c and n_0 such that

$$\underbrace{7n^3 + 6n^2 + 100n + 20}_{f(n)} \le \underbrace{cn^3}_{c\,g(n)} \rightarrow 7 + \frac{6}{n} + \frac{100}{n^2} + \frac{20}{n^3} \le c$$

  - This inequality is satisfied for many choices of c and n_0.

$$\therefore \underline{f(n) = O(g(n)) = O(n^3)}.$$

  - For example

n_0	c
1	133
10	8.62

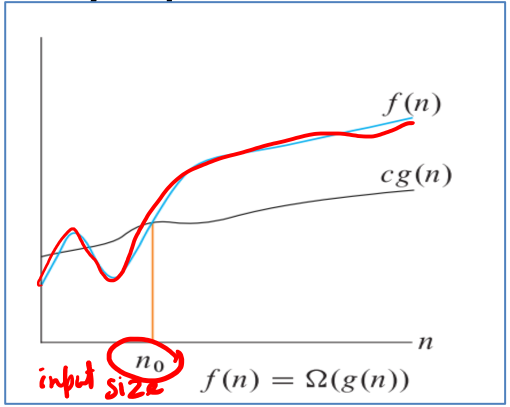
Ω(Omega) – Notation

- Ω - notation characterizes a ***lower bound*** on the asymptotic behavior of a function.

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

if there exists positive constants c and n_0 such that

$$\mathbf{0 \leq cg(n) \leq f(n)} \text{ for all } \mathbf{n \geq n_0}.$$



Source :Cormen, Leiserson, Rivest, Stein “Introduction to Algorithms”

Ω – Notation

- Example:
  - $f(n) = 7n^3 + 6n^2 + 100n + 20$
  - $g(n) = n^3$
  - We need to find positive constants c and n_0 such that

$$7n^3 + 6n^2 + 100n + 20 \geq cn^3 \rightarrow 7 + \frac{6}{n} + \frac{100}{n^2} + \frac{20}{n^3} \geq c$$

  - This inequality holds when $n_0 \geq 1$ and $c = 7$.

    We can also check n_0 for other values of c.

$f(n) = \Omega(g(n)) = \Omega(n^3)$.

Θ(Theta) – Notation

3

- Θ - notation characterizes a ***tight bound*** on the asymptotic behavior of a function.
- f(n) = Θ(g(n)) iff
  - f(n) = O(g(n)) and
  - f(n) = Ω(g(n))

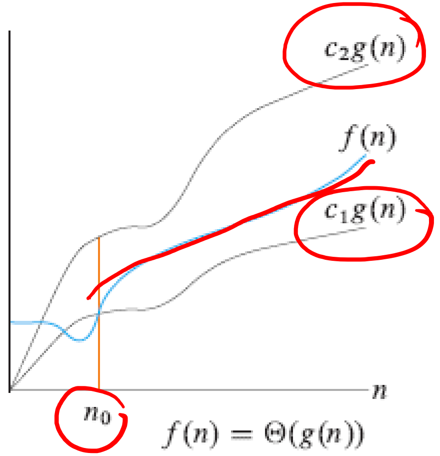
Θ(Theta) – Notation

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

- if there exists

positive constants c_1, c_2 and n_0 such that

$$0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 .$$



Source :Cormen, Leiserson, Rivest, Stein “Introduction to Algorithms”

Example

- Example:
  - $f(n) = 7n^3 + 6n^2 + 100n + 20$
  - $g(n) = n^3$
  - From previous examples we have, $f(n) = O(g(n)) = \Omega(g(n))$
  - So, we can say **$\underline{f(n) = \Theta(g(n)) = \Theta(n^3)}$**

or

Example

- We need to find positive constants c_1, c_2 and n_0 such that

$$\boxed{\mathbf{0 \le c_1 n^3 \le 7n^3 + 6n^2 + 100n + 20 \le c_2 n^3}}$$

$$\rightarrow \mathbf{0 \le c_1 \le 7 + \frac{6}{n} + \frac{100}{n^2} + \frac{20}{n^3} \le c_2}$$

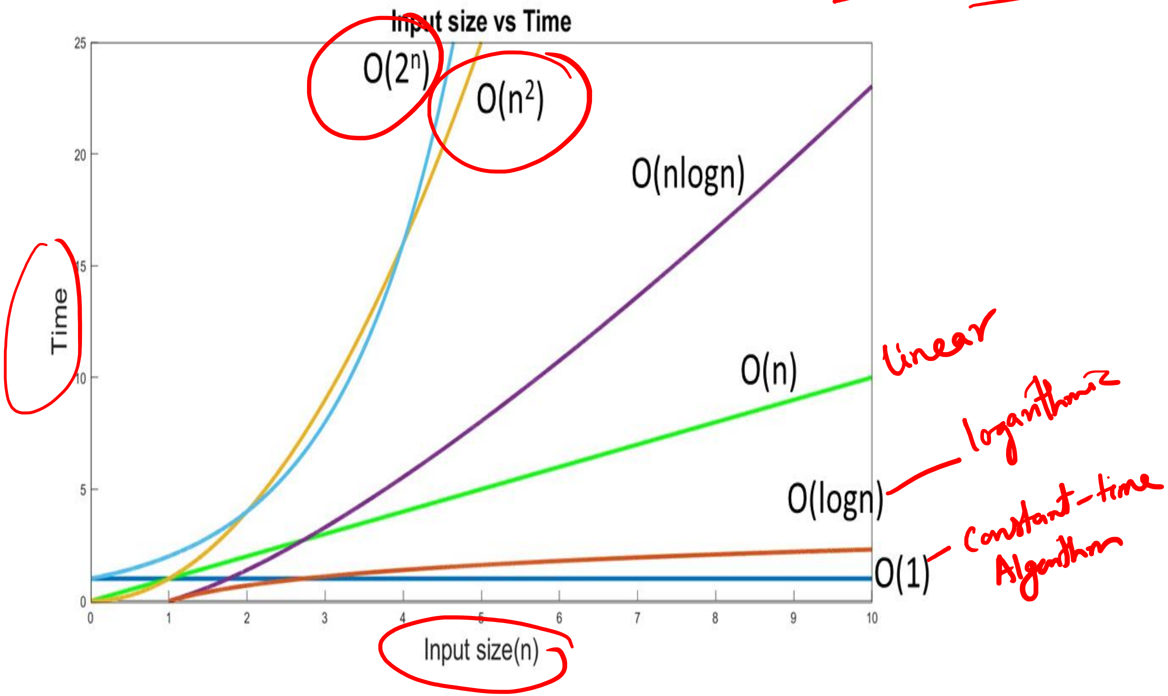
- The above inequality can be satisfied for many choices of c_1, c_2, and n_0. For example,

$$\boxed{c_1 = 7,\ c_2 = 133,\ n_0 = 1}$$

- So, we can say <u>**f(n) = Θ(g(n)) = Θ(n³)**</u>

Different Order of Time Complexities

$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) \boxed{< O(2^n)}$



Different order of Time Complexities

- Examples of different order of time complexities.
  - $O(1)$
  - $O(\log n)$ – Binary Search
  - $O(n)$ – Linear Search
  - $O(n\log n)$ – Merge sort, Insertion sort
  - $O(n^2)$ – Bubble sort
  - $O(2^n)$ – SAT problem, Knapsack Problem

Example - 1

- **Sum of 'n' natural numbers**:
  - **Input**: An integer n
  - **Output**: Sum of first n natural numbers

1. a = 0 → 1
2. **for** i = 1 to n **do** → n + 1
3. a = a + X[i] → n
4. return a → 1

$f(n) = 2n + 3$

$\boxed{f(n) = O(n)}$

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

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

Example - 1

- A better approach

'n'

1. a = [n(n+1)] / 2 → 1

2. return a → 1

$f(n) = 2$

$f(n) = O(1)$

$f(n) = \Omega(1)$

$f(n) = \Theta(1)$

Example - 2

- **Searching Algorithms**:
  - **Input**: An n-element array X of sorted integers 4n
  - **Output**: The index at which the key present

```
Linear_search(X, n, key)
    for i = 0 to n-1 do
        if(X[i] == key)
            return i
    return -1
```

Example - 2

Linear Search

0	1	2	3	4	5	6	7	8	9
15	**23**	**3**	**12**	**45**	**31**	**56**	**9**	**55**	**76**

Key: 56

Example - 2

Linear Search

0	1	2	3	4	5	6	7	8	9
15	**23**	**3**	**12**	**45**	**31**	**56**	**9**	**55**	**76**
X									

Key: 56

Example - 2

Linear Search

0	1	2	3	4	5	6	7	8	9
15	**23**	**3**	**12**	**45**	**31**	**56**	**9**	**55**	**76**
✗	✗								

Key: 56

Example - 2

Linear Search

0	1	2	3	4	5	6	7	8	9
15	**23**	**3**	**12**	**45**	**31**	**56**	**9**	**55**	**76**
X	X	X							

Key: 56

Example - 2

Linear Search

0	1	2	3	4	5	6	7	8	9
15	**23**	**3**	**12**	**45**	**31**	**56**	**9**	**55**	**76**
✗	✗	✗	✗						

Key: 56

Example - 2

Linear Search

0	1	2	3	4	5	6	7	8	9
15	**23**	**3**	**12**	**45**	**31**	**56**	**9**	**55**	**76**
✗	✗	✗	✗	✗					

Key: 56

Example - 2

Linear Search

0	1	2	3	4	5	6	7	8	9
15	**23**	**3**	**12**	**45**	**31**	**56**	**9**	**55**	**76**
X	X	X	X	X	X				

Key: 56

Example - 2

Linear Search

0	1	2	3	4	5	6	7	8	9
15	**23**	**3**	**12**	**45**	**31**	**56**	**9**	**55**	**76**
X	X	X	X	X	X	✓			

Key: 56

The key 56 is found at index i = 6

Example - 2

Binary_search(X, low, high, key)

```
for i = 0 to n-1 do
    mid = low + (high - low) / 2
    if(X[i] == key)
        return m
    if(X[i] < key)
        low = mid + 1
    else
        high = mid – 1
```

Example - 2

Binary Search

0	1	2	3	4	5	6	7	8	9	
3	**9**	**12**	**15**	**23**	**31**	**45**	**55**	**56**	**76**	Sorted Array

Key: 56

Example - 2

Binary Search

0	1	2	3	4	5	6	7	8	9
3	**9**	**12**	**15**	**23**	**31**	**45**	**55**	**56**	**76**

Sorted Array

low

mid

high

56 > 23 → low = mid + 1 = 5

mid = 5 + (9-5)/2 = 7

Key: 56

Example - 2

Binary Search

0	1	2	3	4	5	6	7	8	9
3	**9**	**12**	**15**	**23**	**31**	**45**	**55**	**56**	**76**

Sorted Array

X

Low

mid

high

56 > 55 → low = mid + 1 = 8

mid = 8 + (9-8)/2 = 8

Key: 56

$O(\log_2 n)$

Example - 2

Binary Search

0	1	2	3	4	5	6	7	8	9
3	**9**	**12**	**15**	**23**	**31**	**45**	**55**	**56**	**76**

Sorted Array

Low high

mid

Key: 56

The key 56 is found at index i = 6

Classes of Algorithms

- Algorithms are classified into different complexity classes to categorize their computational complexity.
- Helps us understand their efficiency and solvability.

P	We can efficiently find a solution
NP	Verifying solutions is easier than finding them
NP-complete	a benchmark for the difficulty of other problems within NP
NP-hard	at least as difficult as the hardest problems in NP

P (Polynomial Time) Algorithms

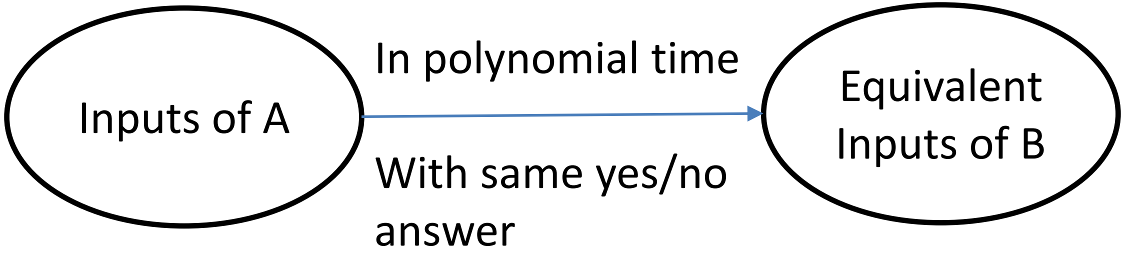
- P = {Problems solvable in **polynomial time**}
- Polynomial time: $O(n^k)$ where k is some constant and n is the input size

NP(Non-deterministic polynomial time) Algorithms

- NP = {Problems solvable in **Non-deterministic** polynomial time}
- Non-deterministic: guessing the correct answer or solution from many options in polynomial time.
- NP can also be considered a class of problems "whose solutions are verifiable in polynomial time."

Reduction

- A problem ‘A’ can be **reduced** to another problem ‘B’ if any instance of ‘A’ can be rephrased to an instance of ‘B’ so that solving the instance of ‘B’ also solves the instance of ‘A’.



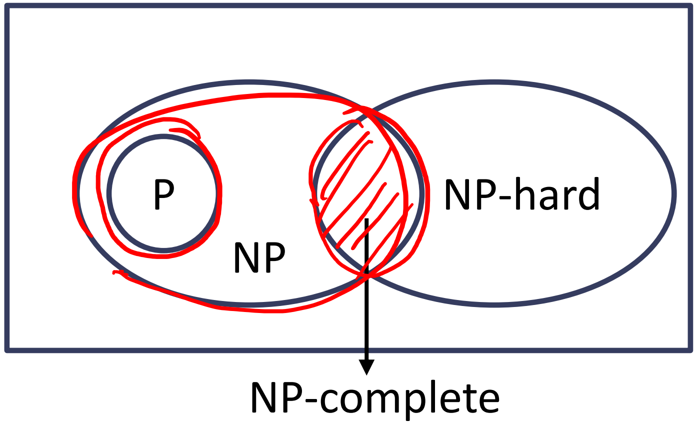
NP-hard Algorithms

- A problem X is NP-hard if every problem Y ϵ NP reduces to X.

- Simply, A problem is NP-hard if all problems in NP are **reducible** to it in polynomial time.

NP-Complete

- A problem X is NP-complete if X ϵ NP and X ϵ NP-hard.



How to prove a problem X is NP-complete?

- Prove X ϵ NP
  - By guessing and verifying or
  - By giving non-deterministic algorithm
- Reduce from known NP-complete problem Y to X.

Algorithms for NP-hard problems

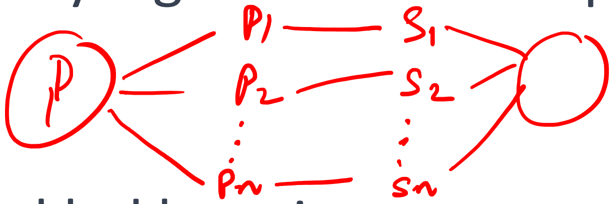
- Most optimization problems in physical design are NP-hard.
- Hence, a polynomial time algorithm won't exist for these problems.
- But a solution is needed even if it is not optimal due to the practical nature of Physical Design Automation.

Algorithms for NP-hard problems

- There are four choices of algorithms to solve NP-hard or NP-complete problems:
  - Exponential Algorithms
  - Special Case Algorithms
  - Approximation Algorithms
  - Heuristic Algorithms

Exponential Algorithms

- Exponential time complexity algorithms can be practical for small input sizes.



- Large problems can be tackled by using exponential-time algorithms for smaller sub-cases and combining results with other techniques for a global solution.

Special Case Algorithms

- Simplifying a general problem by imposing restrictions can lead to polynomial-time solvability in many cases.
- For instance, the graph coloring problem is NP-complete for general graphs,
  - But it can be solved in polynomial time for specific graph classes relevant to physical design.

$k=2$

Approximation Algorithms

- If near-optimality suffices, designers use approximation algorithms instead of strict optimality.
- Often in physical design algorithms, near-optimality is good enough.

hspice — more time, soln is accurate
finesim — less time, soln is not accurate

- The performance ratio of an algorithm is defined as $\gamma = \frac{\Phi}{\Phi^*}$, Where

  Φ is the solution produced by the algorithm

  $\Phi*$ is the optimal solution for the problem.

Heuristic Algorithms

- Heuristic algorithms are often used for NP-complete problems, offering solutions without guaranteeing optimality.
- Effective heuristics should have,
  - low time and space complexity.
  - produce optimal or near-optimal solutions in most realistic cases.
  - exhibit good average case performance.

Thank You