Design & Analysis of Algorithms

17/07/2023

Circullum : Chapters

1. Analysis of algorithms
   - algorithm concept & lifecycle
   - methodologies & types of analysis
   - Asymptotic Notations.
   - framework for analysing Recursive algorithms
   - Apriori Analysis
   - Space Complexity.
   - Analysing loops (loop complexity).

2. Design Strategies / Divide & Conquer
   - General method
   - Max-Min Problem
   - Merge Sort
   - Binary Search
   - Quick Sort
   - Matrix Multiplication, Long integer multiplication (Lim)

   Case Studies.

3. Greedy method.
   - General method, Knapsack problem
   - Job sequencing with deadlines
   - Optimal merge patterns
     - Huffman Coding
   - Minimum Cost Spanning trees
     - Prims Algo
     - Kruskals Algorithm
   - Dykstras Shortest path.

④ Dynamic Programming
- General problem/method
- Difference btw DP, Greedy & Divide & Conquer.
- Multi Staye Graphs,
- Travelling Sales Problems
- Bellman Ford Single Source Shortest path.
- Optimal Cost binary ... etc

⑤ Greedy Algorithms
- Representation of Graphs
- Graph Traversals
  - DFS
  - BFS

⑥ Heap Algorithms
- Operations : Create, Insert, Delete
- ~~Appli~~ Applications : Heap sort.

⑦ Sets
- Representations & Operations.

⑧ Sorting Algorithms
Methods : Bubble, Selection, Insertion, Radix

⑨ Backtracking : & Branch And Bound
(Not in GATE, but for coverage additional)

Pre Requisite
- Programming and data Structures.
- Mathematics:
  - functions
  - Series
  - Matrices
  - Logarithms
  - Exponents
  - Calculus
  - Probability.

Analysis of Algorithms

17/7/2023

Algorithm :- Consist of finite set of steps/statements to solve a given problem.

→ Each step/statement of the algorithm consist of one or more fundamental operations.

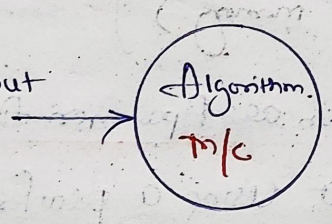
eg: (x ← 0+7)
(addition assignment.

(every operation must be definite (clear)
every operation must be effective (finite time).

⇒ Every algorithm may take zero/more inputs.

→ Every algorithm must produce one output

Algorithm ~ Abstract machine / Processing Logic



→ Every algorithm must terminate in finite amount of time.

Algorithm Lifecycle Steps

1. Problem Definition
2. Requirement/Conditions [SRS]
3. Logic/Design Strategy
4. Develop Algorithm
5. Validation [Prove Correctness]
6. Analysis
7. Implementation [Program development]
8. Testing and debugging

18/7/2023

Need for Analysis

1. To determine resource consumption (Time, Space, Registers, cost, etc).

2 Performance comparision to find an efficient solution / algorithm

Methodology of Analysis (Time Complexity)

eg: [x ← y+z] — Time is dependent upon:

- language
- O.S
- Hardware (CPU + memory)

} Platform / Environment

Aposteriori Analysis

→ Platform dependent analysis is known as Aposteriori Analysis.

→ analysis which is carried out using a platform having software and hardware is Aposteriori Analysis.

Aposteriori Analysis

Advantages

- Gives exact values in real units.

Drawbacks:

- It is difficult to carry out / manual work (machine dependent)
- Cannot consider for all cases of inputs.
- Non-uniformity: different output times on different machine (platform dependent).
- Performance comparision is difficult.

Apriori Analysis (Platform Independent)

Analysis is performed prior to running it on a specific system.

Analytic Framework

* Take into account all possible inputs.
* Allows us to calculate the relative efficiency (performance) of two algorithms in a way that is independent of platform.
* Carried out by studying the high level description of algorithm without actual implementation.
* Easy to carry out.

Drawback: Will not give real/actual values in unit.
It gives estimate/approximate values.

Components of Analytic Framework

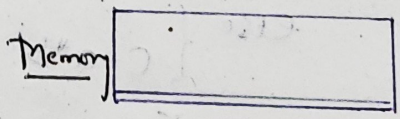
1. A language (Pseudo): to describing algorithm steps.
2. A computation model: that the algorithm executes within it.
3. A metric for measuring algorithm running time
4. Approach/Notation to characterize running time.

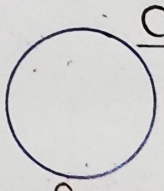
eg: Algorithm Test

```
{
1. x ← y+2;                          → 2 units (Constant).
2. for i ← 1 to n,                   → [1+(n+1)+n+n+n] = 4n+2
       a ← b+c;
3. for j ← 1 to n                    → 1+(n+1)+n+n
   for k ← 1 to n                    +(n+1).n+n.n+n.n
       l ← l+m;                      n*n+n.n
}                                    → 4n^2+4n+2
```

Time = 2+(4n+2)+(4n^2+4n+2) → 4n^2+8n+6 Units

Ram model of Computation





* Each fundamental Operation takes one unit of time.

20/7/2023

→ Algorithm Test

```
{ 1. x ← y+z;          → 2 units
  2. for i ← 1 to n    → 1+(n+1)+n
     { a ← b+c; }      → +n+n
  3. for i ← 1 to n    → 1+(n+1)+n
       for j ← 1 to n  → n+n(n+1)+n.n
         k ← k+z; }    → n.n+n.n
```

$4n^2+8n+6$

Step Count method

Time = $2+(4n+2)+(4n^2+4n+2)$

$\Rightarrow 4n^2+8n+6$

Order $O(n^2)$

→ To determine the time of algorithm under Apriori Analysis (Order of magnitude).

→ Order of magnitude of a statement/step of the algorithm refers to the frequency/count of the fundamental operations in the statement/step.

→ Algorithm Sum(a,b,c)

```
{ integer a,b,c;
  1. read (a,b); — 1
  2. if (a<b); — 1
     { c = a+b; } — 1
     else { c = a+b; } — 1        } 1+1
  3. Print(c); } — 1
```

$= 1+1+1+1 = 4$.

Constant $O(1)$

→ Algorithm DO_it(A,n)

```
{ integer n, Arr[n];
  integer i, Sum=0;
  1. for i ← 1 to n
     { Sum = Sum+Arr[i]; — n.
     }
  2. Print(Sum); — 1
```

∴ Time = n+1

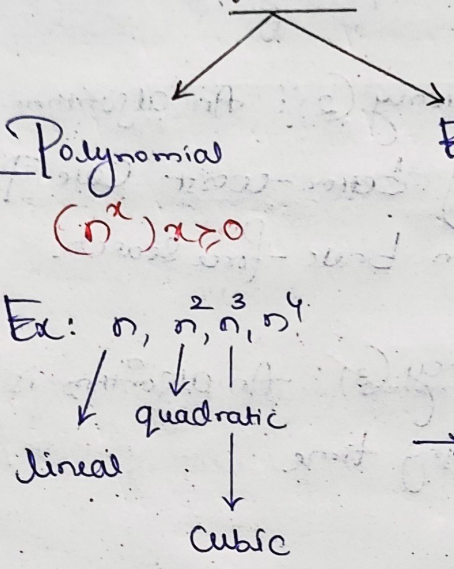
→ $O(n)$

→ The basic objective of Apriori Analysis is to represent (obtain) the time complexity (Running time) of the algorithm by means of mathematical functions with respect to input size.

eg: $T(n) = 1 + n + n^2$ $(4n^2 + 8n + 6)$ (Rate of growth of time)

$T(n) = n + 1$

Time α function



→ Exponential functions have higher rate of growth. (Takes more time).

→ Polynomial Functions have lesser rate of growth. (Takes lesser time).

Illustration:

T1 : A1 T2 : A2

n	n^2	2^n
2	4	4
3	9	8
4	16	16
5	25	32
6	36	64
7	49	128

→ The objective is always to develop algorithm for problems having polynomial time complexity (Efficient).

Apriori Analysis

To determine the running time with respect to increasing input size (n).

To observe the behaviour of the algorithm for a fixed input of size 'n'. (Behaviour / types of analysis)

Analysis of Algorithms

* Algorithms analysis involves thinking about how their resource requirements – the amount of time and space they use – will scale with increasing input size.

* Proposed definition of efficiency (1): An algorithm is efficient if, when implemented, it runs quickly on real input instances.

* Proposed definition of efficiency (2): An algorithm is efficient if it achieves qualitatively better-worst case performance, at an analytical level, than brute-force search.

* Proposed definition of efficiency (3): An algorithm is efficient if it has polynomial running time

eg:

```
Algorithm LS (A, n, x)
{  integer n, A[n];
   int i;
   for i ← 1 to n
   {   if (x = A[i])
       {  print (i);
          exit; }
   }
   print ("Element not found");
}
```

Time ∝ comparison.

1.) Best Case:

1+1+1 = 3 – O(1)

2). Worst Case:

= n.

→ O(n)

Time Complexity ~ worst case

(1) **Worst Case**: The input class for which algorithm takes maximum time, in worst case input and corresponding time is worst case time.

(2) **Best Case**: The input class for which the algorithm takes minimum time in best case input and best case time.

(3) **Average Case**: It is derived in 3 steps

(i) Enumerates all input classes. $(i_1, i_2 \ldots i_k)$.

(ii) Determine the time for $(t_1, t_2, \ldots t_k)$ for each input class.

$$A(n) = \sum_{i=1}^{k} p_i * t_i$$

(iii) Associated with each input class the probability of function (p_i).

eg: Linear Search $A(n)$

(1) Best case : 1 $O(1)$

(2) Worst case : n. $O(n)$

(3) Average case : (for a successful linear search).

→ No. of comparisions : $(1+2+3 \ldots n)$

$$\Rightarrow \frac{n(n+1)}{2} \Rightarrow \frac{1+n}{2} \Rightarrow \underline{\underline{O(n)}}$$

I. Best case Time : $B(n)$

II. Worst case Time : $W(n)$

III. Average case time : $A(n)$

$$\boxed{B(n) \leq A(n) \leq W(n)}$$

a. $B(n) = A(n) = W(n)$ → Merge sort, H.S

b. $B(n) < A(n) = W(n)$ → Linear search

c. $B(n) = A(n) < W(n)$ → Quick sort

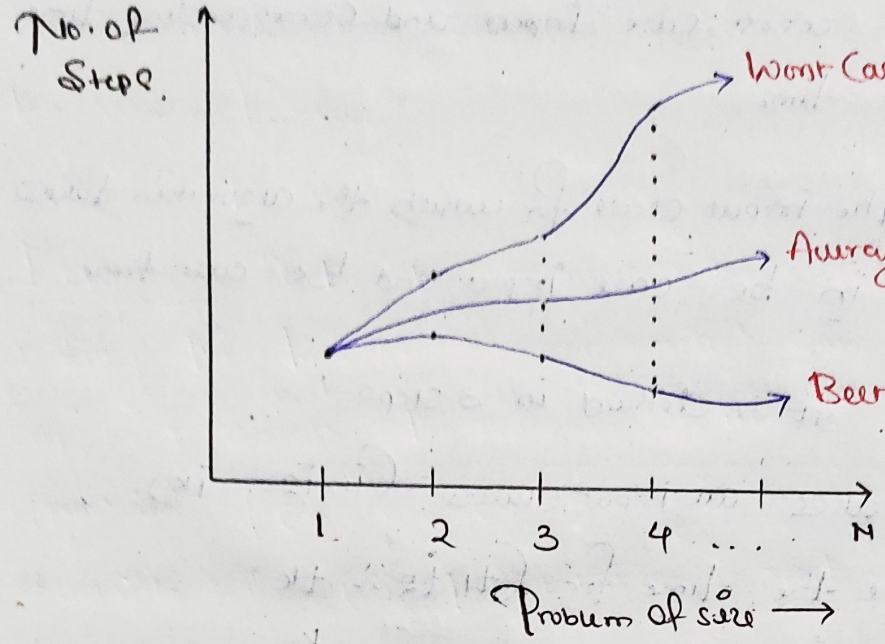
Homework

Search function of behaviour

$B(n) < A(n) < W(n)$

Time Complexity Graph

21/7/2023



Asymptotic Notations (ASN)

→ It is basically a mathematical tool to obtain/represent bounds of functions.

→ Representations of Time and Space Complexity of algorithm by functions

(ASN)

- UB. (Max)
- LB (Min)

Bounds of functions

- Upper bound <Max>
- Lower Bound <Min>
- Tight bound

ASN

Big notations

1. Big-oh : O (Upper bound)
2. Big Omega : Ω (Lower bound)
3. Theta : Θ (Tight bound)

Small Notations/little

1. Little-oh : o → Proper upper bound
2. Little Omega : ω → Proper lower bound

→ Big O notation is a mathematical notation that describes the limiting behaviour of a function when argument tends towards a particular value or infinity.

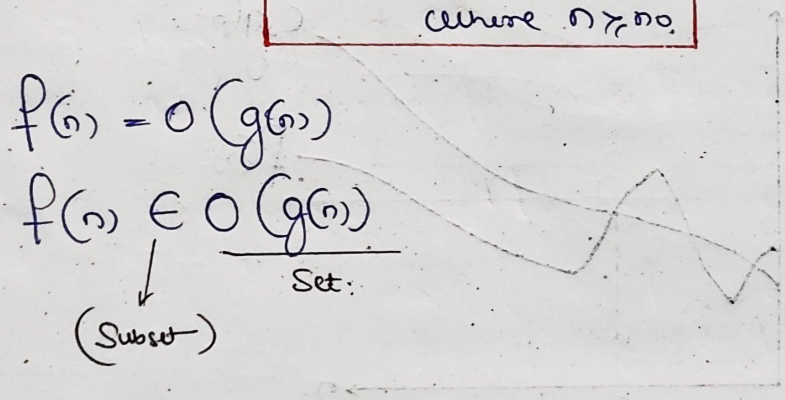
→ Big O is a member of a family of notations invented by Paul Bachmann, Edmund Landau and others, collectively called as Bachman–Landau notation or asymptotic notation.

The letter O was chosen by Bachmann to stand for Ordnung, meaning the order of approximation.

→ Let 'f' and 'g' be the functions from the set of integers/reals to Real numbers.

① Big-oh (O) : upper bound

– $f(n)$ is $O(g(n))$ if there exists some constants $(C>0$ and $n_0>0)$ such that $f(n) \leq C \cdot g(n)$ where $n \geq n_0$.



$f(n) = O(g(n))$

$f(n) \in O(g(n))$ — Set.

(Subset)

eg: $f(n) = (1+n+n^2) \rightarrow O(n^2)$

$1+n+n^2 \leq n^2+n^2+n^2$.

$f(n) \leq 3 \cdot n^2 \cdot n \geq 1$

$C \quad g \quad n_0$

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

Closest upper bound!

$f(n)$ is $O(n^2)$ ✓

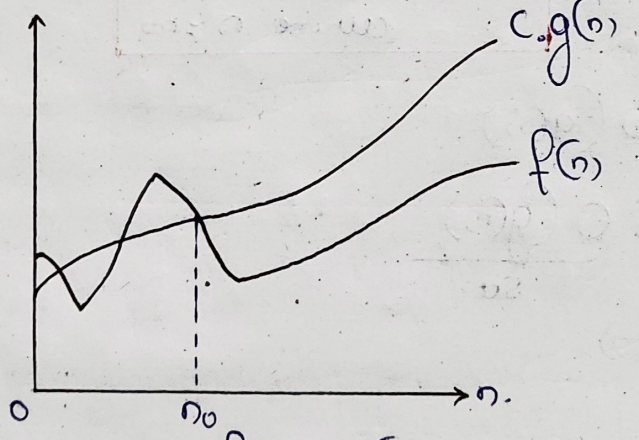
$f(n)$ is $O(n^3)$ ✓

$f(n)$ is $O(n^4)$ ✓

→ Whenever we determine the upper bound and lower bound, we should find that function 'g' which is closest to the given function.

Definition: A theoretical measure of the execution of an algorithm, usually the time or memory needed, given the problem size n, which is usually the number of items. Informally saying some equation $f(n) = O(g(n))$ means it is less than some constant multiple of $g(n)$. The notation is read, "f of n is big-oh of g of n".

Formal definition: $f(n) = O(g(n))$ means there are positive constants 'c' and 'K' such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq k$. The values of c and k must be fixed for the function f and must not depend on n.



$f(n) = O(g(n))$.

$f(n) = n^2$.

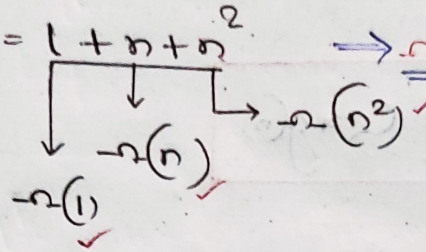
$g(n) = 2^n$.

n	f	g.
1	1	2
2	4	4
3	9	8
4	16	16
5	25	32
6	36	64

② Big-omega (Ω) : Lower bound

→ $f(n)$ is $\Omega(g(n))$ if there exists constants 'c' and 'n_0' such that $\boxed{f(n) \geq c \cdot g(n), \text{ whenever } n \geq n_0}$

eg: $f(n) = 1 + n + n^2$



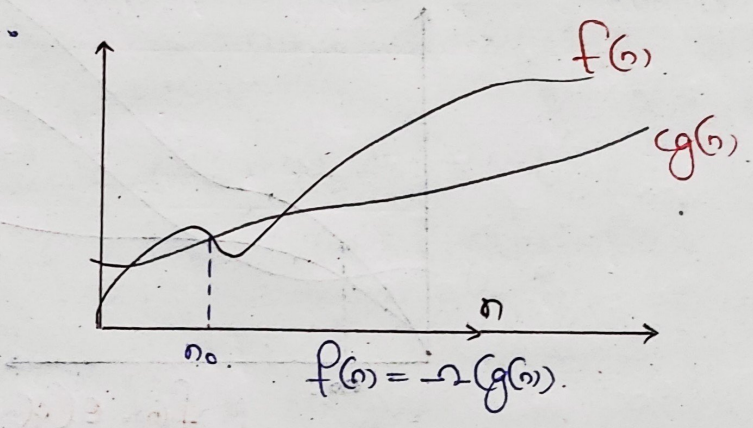
$1 + n + n^2 > 1.1$, $n \geq 1$

$1 + n + n^2 > 1.n$, $n > 1$

$\boxed{1 + n + n^2 \geq 1.n^2, \quad n \geq 1}$

c g

→ Similar to big O notation, big Omega (Ω) function is used in Computer Science to describe the performance or complexity of an algorithm. If a running time is $\Omega(f(n))$ then for large enough n, the running time is atleast $k.f(n)$ for some Constant k.



The formal definations associated with Big notations are as follows:

→ $f(n) = O(g(n))$ means $c.g(n)$ is upper bound on $f(n)$. Thus there exist some constant c such that $f(n)$ is always $\leq c.g(n)$ for large enough n. (i.e $n \geq n_0$ for some constant n_0)

→ $f(n) = \Omega(g(n))$ means $c.g(n)$ is a lower bound on $f(n)$. Thus there exists some constant c such that $f(n)$ is always $\geq c.g(n)$ for all $n \geq n_0$.

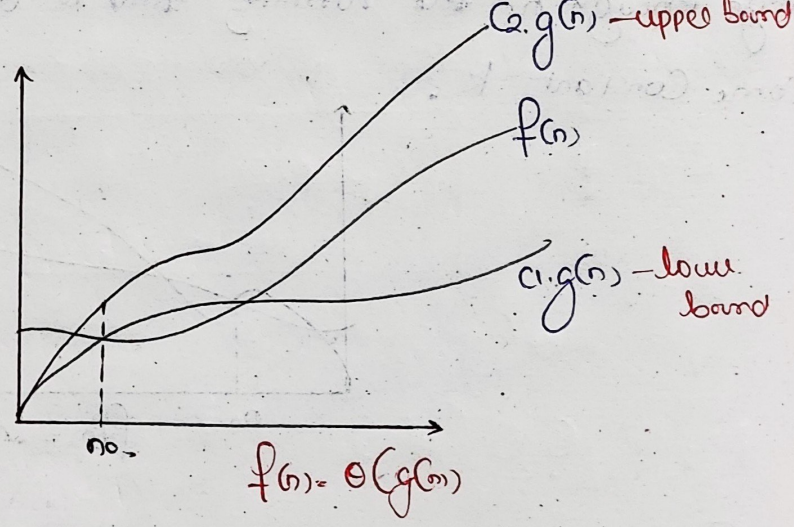
(3) Theta (Θ) : Tight bound :

$f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$

eg. $1+n+n^2$ (f) $< O(n^2)$, $\Omega(n^2)$ $\therefore \Theta(n^2)$

$$C_1 * g(n) \leq f(n) \leq C_2 * g(n)$$

→ $f(n) = \Theta(g(n))$ means $C_1 . g(n)$ is an upper bound on $f(n)$ and $C_2 . g(n)$ is a lower bound on $f(n)$, for all $n > n_0$. Thus there exist constants C_1 and C_2 such that $f(n) \leq C_1 . g(n)$ and $f(n) \geq C_2 . g(n)$. This means that $g(n)$ provides a nice, tight bound on $f(n)$.



→ For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Example Questions :

1) $f(n) = 1+n+n^2$ $< O(n^2)$, $\Omega(n^2)$ $\therefore \Theta(n^2)$

2) $f(n) = n$ $< O(n)$, $\Omega(n)$ $\therefore \Theta(n)$

$n \leq C.n \quad n > 1$

$n \leq 2.n, \ n > 1$

$n \geq 1.n, \ n > 1$

$\therefore 1/2$

22/7/23

3> $f(n) = c^{100}$ ↓ c. — $O(1)$, $\Omega(1)$ $\therefore \Theta(1)$ $\quad 2^{100} \leq 3^{100}$.

4> $f(n) = n + \log n$ — $O(n)$, $\Omega(n)$ $\therefore \Theta(n)$

5> $f(n) = \sqrt{n} + \log n$ — $O(\sqrt{n})$, $\Omega(\sqrt{n})$ $\therefore \Theta(\sqrt{n})$

$n + \log n \leq n + n$

$\leq 2.n$

↓

Constant.

$\log \sqrt{n}$

↳ $\log (n)^{1/2}$

$\frac{1}{2} \log(n)$

$\frac{1}{2} \log(n) > \log(\log(n))$

Note:

U.B ← 1. Big-O: $f(n)$ is $O(g(n)) \Rightarrow f(n) \leq c.g(n)$ whenever $n \geq n_0$;

L.B ← 2. Big-omega (Ω): $f(n)$ is $\Omega(g(n))$ if $f(n) \geq c.g(n)$, $n \geq n_0$.

T.B ← 3. Theta (Θ): $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ & $f(n)$ is $\Omega(g(n))$

* Smaller functions are in order of Bigger functions.
* Bigger functions are in the Omega of Smaller functions.
* If the rate of growth of both the functions are equal then they are theta of each other.

For all real $a > 0$, m, n

1. $a^0 = 1$
2. $a^1 = a$
3. $a^{-1} = \frac{1}{a}$
4. $(a^m)^n = a^{mn}$
5. $(a^m)^n = (a^n)^m$
6. $a^m \cdot a^n = a^{m+n}$

Note:

- $\log x^y = y \log x$
- $\log x.y = \log x + \log y$
- $\log \log n = \log(\log n)$
- $a^{(\log_b^x)} = x^{(\log_b^a)}$
- $\log n = \log_{10}^n$
- $\log^k n = (\log n)^k$
- $\log \frac{x}{y} = \log x - \log y$
- $\log_b^x = \dots = \frac{\log_a x}{\log_a b}$

Properties

1. $a = b^{\log_b a}$
2. $\log_c(ab) = \log_c a + \log_c b$
3. $\log_b a^n = n . \log_b a$
4. $\log_b a = \frac{\log_c a}{\log_c a}$
5. $\log_b \frac{1}{a} = -\log_b a$
6. $\log_b a = \frac{1}{\log_a b}$
7. $a^{\log_b c} = c^{\log_b a}$

Summation Series

Arithmetic Series :→ $\sum_{k=1}^{n} k = 1+2+\dots+n = \frac{n(n+1)}{2}$

Geometric Series :→ $\sum_{k=0}^{n} x^k = 1+x+x^2\dots+x^n = \frac{x^{n+1}-1}{x-1}\ (x\neq 1)$

Harmonic Series :→ $\sum_{k=1}^{n} \frac{1}{k} = 1+\frac{1}{2}+\dots\frac{1}{n} \approx \log n$

① The geometric sum formula for the finite terms is given as:

$$\sum_{i=0}^{n} x^i \left[(n+1)\text{ terms}\right]$$

→ if $r=1$, $S_n = n^*a$

→ if $|r|<1$, $S_n = \frac{a(1-r^n)}{1-r}$

→ if $|r|>1$, $S_n = \frac{a(r^n-1)}{r-1}$

$$\sum_{i=1}^{n} \tfrac{1}{2}^i = \frac{\frac{1}{2}\left(1-\frac{1}{2^n}\right)}{1-\frac{1}{2}} = 1-\frac{1}{2^n}$$

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

where :
- a is the first term
- r is the common ratio
- n is the no. of terms

② The geometric sum formula for infinite terms is given as:

→ if $|r|<1$, $S_\infty = \frac{a}{1-r}$

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x} \quad x<1$$

→ if $|r|>1$, the series does not converge and it has no sum.

1) $f(n) = 2^{\log_2 n} = O(n)$

$n^{\log_2 2} = n^1$.

2) $f(n) = 2^{\sqrt{n}\log_2 n} = O(n^{\sqrt{n}})$

$= 2^{\log_2 n^{\sqrt{n}}}$

$= n^{\sqrt{n}(\log_2 2)} = n^{\sqrt{n}}$

3) $f(n) = 2^{2n} = (2^2)^n$

$\neq O(2^n)$

2^n is $O(4^n)$ ✓

4^n is $\Omega(2^n)$ ✓

4^n ~~is~~ $O(2^n)$

$2^{n+1} = O(2^n)$

$2^n \cdot 2^1 \leq 4 \cdot 2^n$

c g

$n = 2^{\log_2 n}$

$f(n) = n^{1/\log_2 n}$

$= (2^{\log_2 n})^{\frac{1}{\log n}}$

$= 2^1 = 2(1)$

$= O(1)$

$f(n) = n^{1/\log n} = x$

$= \log n^{1/\log n} = \log_2 x$

$= \frac{1}{\log n} \cdot \log n = \log_2 x$

$\log_2 x = 1$

$= x = 2^1$

$O(1)$

→ eg: $f(n) = \log_2 n < g(n) = \sqrt{n}$

$\log_2 \log_2 n$. $\log \sqrt{n}$

$\log n^{1/2}$

$\log \log n < \frac{1}{2} \log(n)$

n^2	2^n
4	4
9	8

* $a < b$

$\log a < \log b$

---

* $a > b$

$\log a > \log b$

→ eg: $f(n) = \frac{1}{n}$ ↓ decreasing function

$g(n) = \log(n)$ (↳ increasing function)

assume $n = 8$

$\frac{1}{8} < \log_2 8 = 3$.

$\therefore f(n) < g(n)$

→ eg: $f(n) = n^2$. $g(n) = n^3$.

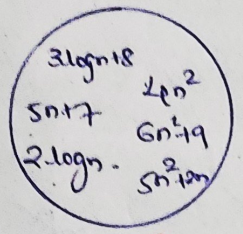
$2 \log n$ $3 \log n$.

$O(1)$ $O(1)$

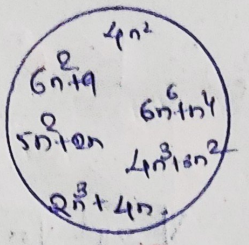
$2 < 3$.

$\therefore f(n) < g(n)$.

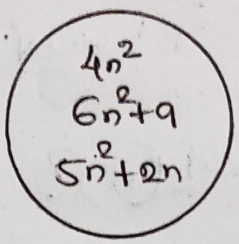
* even the order is $O(1)$ for both, we can't ignore the values. Compare the values



$O(n^2)$



$\Omega(n^2)$

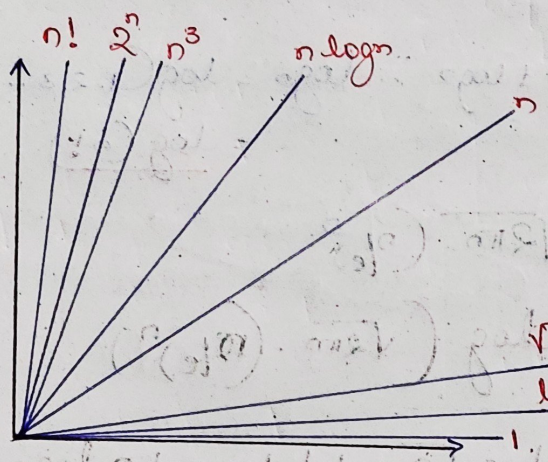


$\Theta(n^2) = O(n^2) \cap \Omega(n^2)$

Dominance Relation

Constants < logarithms < Polynomials < Exponentials.

$$c < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 \ldots < 2^n < 4^n < n^n < n^{n^2}$$



1. $f(n) = \sum_{i=1}^{n} 1 = n = O(n)$

2. $f(n) = \sum_{i=1}^{n} n = n \cdot \sum_{i=1}^{n} 1 = n \cdot n = O(n^2)$

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

4. $f(n) = \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = O(n^3)$

5. $f(n) = \sum_{i=1}^{n} i^3 = \left[\frac{n(n+1)}{2}\right]^2 = O(n^4)$

6. $f(n) = \sum_{i=1}^{n} 2^i = (2^{n+1} - 2) = O(2^n)$

7. $f(n) = \sum_{i=1}^{n} \frac{1}{2^i} = \left(1 - \frac{1}{2^n}\right) = O(1)$

8. $f(n) = \sum_{i=1}^{n} i \cdot 2^i = (n-1)\cdot 2^{n+1} + 2$

$[n \cdot 2^{n+1} - 2^{n+1} + 2]$ → $O(n \cdot 2^n)$

9. $f(n) = \prod_{i=1}^{n} 1 = O(1)$

10. $f(n) = \prod_{i=1}^{n} i = (1.2.3. \ldots n) = n!$

$= n.(n-1).(n-2) \ldots 1.$

$n! < 1.n^n \Rightarrow O(n^n)$ loose bound

Stirlings Approximation

$$n! \sim \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n \quad < n^n$$

Q.1. Is $n! = \Omega(n^n)$?

$[n \cdot (n-1)(n-2)\ldots 1] \times [n \times n \times n \ldots]$

$$n! \neq \Omega(n^n)$$

$$\sqrt{2\pi} \cdot \sqrt{n} \cdot \frac{n^n}{e^n}$$

$$C \cdot \sqrt{n} < e^n$$

$n!$ is $\Omega(n)$ ✓

$n!$ is $\Omega(n^2)$ ✓

Q2. $f(n) = \sum_{i=1}^{n} \log i = \log 1 + \log 2 \ldots \log n = \log(1.2.3 \ldots n)$

$= \log(n!)$

By Stirlings Approximation:

$n! \sim \sqrt{2\pi n} \cdot (n/e)^n$

(Tm1) $\log n! = \log\left(\sqrt{2\pi n} \cdot (n/e)^n\right)$

$= \log\sqrt{2\pi n} + \frac{1}{2}\log n + n\log n + \log\frac{1}{e^n}$

$= \frac{1}{2}\log 2\pi$

$= [C + \frac{1}{2}\log n + n\log n - n\log e]$

$O(n\log n)$ $\quad \Omega(n\log n)$

$$\log n! = \Theta(n\log n)$$

$$n! \text{ is } O(n^n)$$

By Integral Method: (Tm2)

$\rightarrow \log n! = \sum_{i=1}^{n} \log i \sim \int_1^n \log x \, dx = [x(\log x - 1) + C]_1^n$

$\sum_{i=1}^{n} \log i = [n \cdot \log n - n + C]$

$O(n\log n)$ $\quad \Omega(n \cdot \log n)$

$$\log n! = O(n \cdot \log n)$$

Generic Method (Pg 3)

$$\sum_{i=1}^{n} \log i = \log 1 + \log 2 \ldots \log n.$$

$$\log n + \log n-1 + \log n-2 < \log n + \log n \ldots \log n$$

$$< n.\log n.$$

$$O(n.\log n)$$

$$\log n + \log n-1 + \log \ldots + i > (\log n/2 + \log n/2 \ldots)$$

$$\boxed{\log n! > \frac{n}{2} \log n/2} \quad \Omega(n.\log n)$$

$$\therefore \log n! = \Theta(n \log n)$$

$$\boxed{n! < n^n \Rightarrow O(n^n) \\ \log n! = \Theta(n \log n)}$$

Homework

$$f(n) = \sum_{i=1}^{n} i^{1/2} = O(\underline{\quad ? \quad})$$

26/7/2023

$$f(n) = \sum_{i=1}^{n} \sqrt{i} = O(\quad) = [\sqrt{1} + \sqrt{2} + \sqrt{3} \ldots + \sqrt{n}]$$

$$\sum_{i=1}^{n} \sim \int \frac{1}{x} dx. \qquad \int_{i=1}^{n} i^{1/2} di = \left[\frac{i^{3/2}}{3/2}\right]_1^n$$

$$[\log x]_1^n \qquad \boxed{f(n) = n^{3/2} \times C}$$

$$= O(\log n) \qquad \Rightarrow O(n^{3/2}) \Rightarrow O(n^{1.5}) \Rightarrow O(n\sqrt{n})$$

Q1.) $f(n) = \sum_{i=1}^{n} i^3 = x$, choose for x.

$$\sum_{i=1}^{n} i^3 = \left[\frac{n(n+1)}{2}\right]^2 = \frac{n^2(n+1)^2}{4} = \frac{n^2[n^2+2n+1]}{4}$$

$f(n) = \frac{n^4 + 2n^3 + n^2}{4}$

- $O(n^4)$ ✓
- $\Omega(n^4)$ ✓
- $\Theta(n^4)$ ✓

From options:

$\Rightarrow \Theta(n^4), O(n^5), \Omega(n^4)$

Small / Little Notations:

* The bounds provided by Big notation (O, Ω) may or may not be tight.

$f(n)$
- $O(n)$ — tight bound
- $O(n^2)$ — loose bound
- $\Omega(1)$ — loose bound
- $\Omega(n)$ — tight bound

* The bounds provided by little / small notation is always not asymptotically tight (loose bound)

④. **Small-o (o):** proper upper bound

$f(n)$ is $o(g(n))$ if for all $c > 0$

$f(n) < c \cdot g(n)$, whenever $(n > n_0)$ $n_0 > 0$

→ Every small-o is big-o but not vice versa

eg: $f(n) = n$
- $O(n^2)$ ✓
- $O(n)$ ✓
- $o(n^2)$ ✓
- $o(n)$ ×
- $o(n \cdot \log n)$ ✓

⑤ **Small-omega (ω):** Proper lower bound.

$f(n)$ is $\omega(g(n))$, if for all $c > 0$ $f(n) > c \cdot g(n)$

- whenever $n > n_0$ $(n_0 > 0)$.

eg: $f(n) = n$
- $\Omega(n)$ ✓
- $\Omega(1)$ ✓
- $\omega(n)$ ×
- $\omega(\log n)$ ✓

→ Every small omega is big omega but not vice versa.

Analogy between Real numbers & A.S.N

Let a, b are real numbers and f's are positive functions.

1. If $f(n)$ is $O(g(n)) \longleftrightarrow a <= b$
2. If $f(n)$ is $\Omega(g(n)) \longleftrightarrow a >= b$
3. If $f(n)$ is $\Theta(g(n)) \longleftrightarrow a = b$
4. If $f(n)$ is $o(g(n)) \longleftrightarrow a < b$
5. If $f(n)$ is $\omega(g(n)) \longleftrightarrow a > b$

General Properties of Big-O notation:

Let $d(n)$, $e(n)$, $f(n)$, and $g(n)$ be functions mapping non-negative integers to non-negative reals then:

1. If $d(n)$ is $O(f(n))$ then $a.d(n)$ is $O(f(n))$ for any const $a > 0$
2. If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then $d(n) + e(n)$ is $O(f(n) + g(n))$.
3. If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then $d(n)e(n)$ is $O(f(n)g(n))$.
4. If $d(n)$ is $O(f(n))$ and $f(n)$ is $O(g(n))$, then $d(n)$ is $O(g(n))$.
5. If $f(n)$ is a polynomial of degree d (that is $f(n) = a_0 + a_1 n + \ldots + a_d n^d$) then $f(n)$ is $O(n^d)$.
6. n^x is $O(a^n)$ for any fixed $x > 0$ and $a > 1$
7. $\log n^x$ is $O(\log n)$ for any fixed $x > 0$.
8. $\log^x n$ is $O(n^y)$ for any fixed constants $x > 0$ and $y > 0$

Discrete properties of ASN

Properties \ ASN	O	Ω	Θ	o	ω
Reflexive	✓	✓	✓	×	×
Symmetric	×	×	✓	×	×
Transitive	✓	✓	✓	✓	✓
Transpose Symmetry	if $f(n)$ is $O(g(n))$ then $g(n)$ is $\Omega(f(n))$			if $f(n)$ is $o(g(n))$ then $g(n)$ is $\omega(f(n))$	

n is $O(n^2)$
n^2 is $O(n^3)$
$\therefore n$ is $O(n^3)$

Tricotomy Property: [For any two real no's (a,b)]

If 'f' and 'g' are positive functions

I. $f < g \Rightarrow O, o$
II. $f > g \Rightarrow \Omega, \omega$
III. $f = g \Rightarrow \Theta$

Does ASN/ASN functions obey tricotomy property always?

$\Rightarrow$ NO

$\rightarrow$ ASN/function (As) does not satisfy tricotomy property.

Examples

1. $f(n) = n$, $g(n) = n^2$ $(f \leq c.g)$
   $\hookrightarrow O, o.$
2. $f(n) = \log n$, $g(n) = \frac{1}{n}$ $(f > c.g)$
   $\hookrightarrow \Omega, \omega.$
3. $f(n) = n^2 + 10$; $g(n) = 10n^2 + 5$
   $o(n^2) = O(n^2) \Rightarrow \Theta(n^2)$
4. $f(n) = n$ $g(n) = n^{1+\sin n}$
   $= n^{1-1} = n^0$ $(f < g)$ —(i)
   $n = n^2$ $(f < g)$ —(ii)

$\rightarrow$ These two functions never converge.

Based on this we say
Min $(\sin x) = -1$
Max $(\sin x) = +1$.

State True / False

1. $100n \log n = O(n \log n)$ True

2. $2^{n+1} = O(2^n) \Rightarrow 2 \cdot 2^n = O(2^n)$ True

3. $2^{2n} = O(2^n)$ $(2^2)^n = 4^n$: False

4. $0 < x < y$ then $n^x = O(n^y)$ $2 < 3$ True
   ($n^2 < n^3$

5. $(n+k)^m \neq O(n^m)$ $(k, m) > 0$. False.

6. $\sqrt{\log n} = O(\log \log n)$ False

   $\log(\sqrt{\log n}) > \log(\log_2 \log_2 n)$

   $\frac{1}{2} \log \log n > \log \log \log n$

7. $\log(n)$ is $\Omega(1/n)$ True.

8. 2^{n^2} is $O(n!)$ False.

   $\log(2^{n^2})$ vs $\log(n!)$
   (n^n)

   $n^2 \log_2 2$ $n \log_2 n$

   $n^2 > n \log n$

9. n^2 is $O(2^{2 \log n})$ True
   ($2^{\log_2 n^2} = n^2$

10. $a^n \neq O(n^x)$, $a > 1$, $x > 0$ True

11. $2^{\log_2 n^2}$ is $O(n^2)$ True

27/7/2023

Homework:

(1.) Let $f(n)$ and $g(n)$ be positive functions if $f(n)$ is $O(g(n))$ then $f(n)$ always $O((f(n))^2)$? $\Rightarrow$ False

(2.) Is $2^{f(n)} = O(2^{g(n)})$? $\Rightarrow$ False

ex: $f(n) = 2n$, $g(n) = n$

$2^{2n} = O(2^n)$

$4^n \neq O(2^n)$

ex: $f(n) = n$

$g(n) = n^2$

$2^n = O(2^{n^2})$

ex:

1> $f(n) = n$, $(f(n))^2 = n^2$

2> $f(n) = \frac{1}{n}$ $(f(n))^2 = \frac{1}{n^2}$

$\frac{1}{n} > \frac{1}{n^2}$

$f(n) \neq O(f(n)^2)$

Q. Which one of the following is true for all positive functions $f(n)$?

$\Rightarrow f(n^2) = O(f(n^2))$, when $f(n)$ is polynomial

$f(n) = n^3$

$= f(n^2) = (n^3)^3 = n^6$

$f(n)^2 = (n^3)^2 = n^6$ $\Big\}$ $\therefore f(n^3) = \theta(f(n)^2)$ when $f(n)$ is polynomial.

The sum of two functions is governed by the dominant one.

$O(f(n)) + O(g(n)) \longrightarrow O(\max(f(n), g(n)))$

$\Omega(f(n)) + \Omega(g(n)) \longrightarrow \Omega(\max(f(n), g(n)))$

$\Theta(f(n)) + \Theta(g(n)) \longrightarrow \Theta(\max(f(n), g(n)))$

$O(f(n)) * O(g(n)) \longrightarrow O[f(n) * g(n)]$

↳ Similarly for Ω and Θ.

Asymptotic Comparisions

Q1. $f(n) = n$, $g(n) = \log n$ → $g(n)$ is $O(f(n))$

Q2. $f(n) = n^2 \log n$, $g(n) = n \cdot \log^{10} n$.

Soln ②

$f(n) = n^2 \log n$, $g(n) = n \cdot \log^{10} n$

$= (n \cdot \log) n$ $(n \log n) \log^9 n$

$= n$ $(\log n)^9$

$= \log n$ $> \log (\log n)^9$

$= 9 \cdot \log \log n$

$\Rightarrow g(n)$ is $O(f(n))$.

Q3. $f(n) = n^3$, $0 < n \leq 10,000$

$= n$, $n > 10,000$

$g(n) = n$, $0 < n \leq 100$

$= n^3$, $n \geq 100$

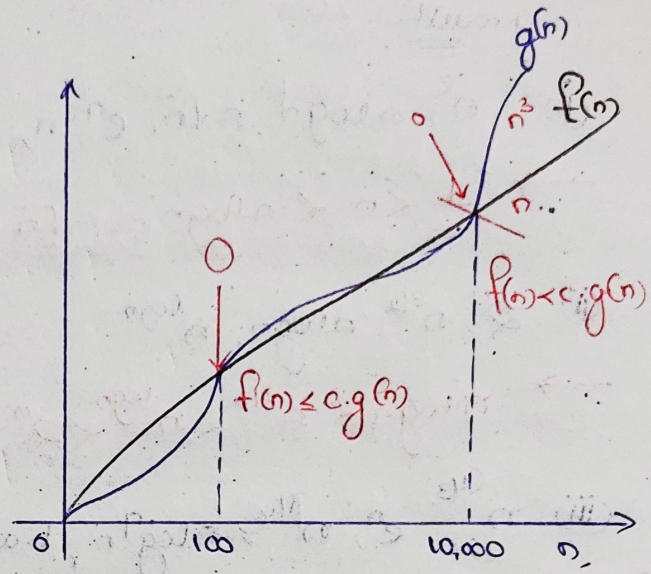
Q4 Two packages are available for processing a database having 10^x records. Package A takes time of $10 \cdot n \log n$ while package B takes a time of $0.0001 n^2$ for processing 'n' records. Determine the smallest integer x for which Package 'A' outperforms Package 'B'.

Sol^n: $f(n) = n^3$, $0 < n \leq 10,000$

③ $= n$, $n > 10,000$

$g(n) = n$, $0 < n \leq 100$

$= n^3$ $n > 100$



$f(n) = n^3$, $n > 100, \leq 10,000$

n, $n > 10,000$

$g(n) = n^3$, $n > 100, \leq 10,000$

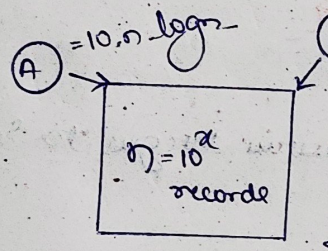
$= n^2$, $n > 10,000$

$\Rightarrow (f < g)$

→ $f(n)$ is $O(g(n))$, $n > 100$

→ $f(n)$ is $\neq o(g(n))$, $n > 10,000$

Sol^n ④



if $x = 2$

$n = 10^2$

A: $10 \cdot 10^2 \cdot \log_{10} 10^2$

$= 2000$ units

B: $10^{-4} \times (10^2)^2$

$= 10^{-4} \times 10^4 = 1$ unit

$\begin{cases} n \log n < n^2 \\ A < B \end{cases}$

if $x = 3$

$n = 10^3$

A: $10 \cdot 10^3 \cdot \log_{10} 10^3$

$= 30,000$

B: $10^{-4} \times (10^3)^2$

$= 100$

A → $10 \cdot n \cdot \log_{10} n$, B → $0.0001 n^2$

$10 \cdot n \log_{10} n < 0.0001 n^2$

$10 \cdot 10^x \cdot \log_{10} 10^x < 10^{-4} \times 10^{2x}$

$x < \frac{10^x}{10^5}$

$x = 6$

$6 < \frac{10^6}{10^5}$ ∴ $x = 6$

Q4. Arrange the functions in increasing order of rate of growth.

(i) $n^2, n\log n, n\sqrt{n}, e^n, n, 2^n, (1/n)$

→ $\frac{1}{n} < n < n\log n < n\sqrt{n} < n^2 < 2^n < e^n$

(ii) $2^n, n^{3/2}, n\log n, n^{\log n}$.

→ $n\log n < n^{3/2} < n^{\log n} < 2^n$.

(iii) $n^{(1/3)}, e^n, n^{7/4}, n\log^9 n, 1.001^n$.

→ $n^{1/3} < n.\log^9 n < n^{7/4} < 1.001^n < e^n$.

$n^{7/4}$

$= n^{1.75}$ $\quad n\log^9 n$

$= (n^{0.5}.n^{0.25})$ $\quad \log^9 n$

$(\sqrt{n}).n^{0.25} > (\log n)^9$.

Q5. Consider the following functions from positive integer to real number: $10, \sqrt{n}, n, \log_2 n, \frac{100}{n}$.

The correct arrangement of the above functions in increasing order of asymptotic complexity is:

→ $\frac{100}{n} < 10 < \log_2 n < \sqrt{n} < n$

Q6. Which of the following is True?

1. $f(n)$ is $O(g(n))$
2. $g(n)$ is NOT $O(f(n))$ } $f^n < g^{n^2}$ ← $f^n < (g^{n^2} = h^{n^2})$
3. $g(n)$ is $O(h(n))$
4. $h(n)$ is $O(g(n))$ } $g = h^{n^2}$.

Options C. → $h(n) \neq O(f(n))$
D. → $f(n) + h(n)$ is $O(g(n) + h(n))$. } then all true

Q7 $f(n) = 2^n$, $g(n) = n^n$.

$\log 2^n \quad \log n^n$

$n \cdot \log_2 2 \quad n \log_2 n$

$\Rightarrow f(n) = O(g(n))$

$n < n \log_2 n$

Q8. $f(n) = n \cdot 2^n$, $g(n) = 4^n$.

$\log_2(n \cdot 2^n) < \log_2 4^n$

$\Rightarrow f(n) = O(g(n))$

$\log_2 n + \log_2 2^n \quad n \log_2 4$

$n + \log n < n + n$

Homework

1. $f(n) = n^2 \log n$, $g(n) = n^{100}$
2. $f(n) = n^{\log n}$ $g(n) = 2^n$
3. $f(n) = \log_2 n$, $g(n) = \log_{10} n$.
4. $f(n) = 2^n$, $g(n) = n^{\sqrt{n}}$
5. $f(n) = n^{\log 2^n}$, $g(n) = n^{\log n^{10}}$
6. $\log n$, $\log^{10} n$, $\log \log n$, $(\log \log n)^{10}$, $(\log n)^{10}$.

Arrange in increasing order.

7. 2^{2^n}, $n!$ 4^n, 2^n.
8. $2^{\log n}$, $(\log n)^2$, $\sqrt{\log n}$, $\log \log n$

Q9. Let $w(n)$ and $A(n)$ repr respectively, the worst case and average case running time of algorithm exists $\forall$ size of n which is always true?

$\Rightarrow A(n) = O(w(n))$ Always true ✓

$\boxed{B(n) \leq A(n) \leq w(n)}$

$A(n) = w(w(n))$ always false and is remaining cases Sometimes true

Take home lessons

* $3n^2 - 100n + 6 = O(n^3)$ because I choose $C = 3$ and $3n^3 > 3n^2 - 100n + 6$
* $3n^2 - 100n + 6 = O(n^3)$ because I choose $C = 1$ and $n^3 > 3n^2 - 100n + 6$ where $n > 3$
* $3n^2 - 100n + 6 \neq O(n)$ because for any c I choose $cxn < 3n^2$ when $n > c$

- $3n^2-100n+6 = \Omega(n^2)$ because I choose $C=2$ and $2n^2 < 3n^2-100n+6$ where $n > 100$.
- $3n^2-100n+6 \neq \Omega(n^3)$, because I choose $C=3$ and $3n^2-100n+6 < n^3$ where $n > 3$.
- $3n^2-100n+6 = \Omega(n)$ because for any C I choose $Cn < 3n^2-100n+6$ when $n > 100$.
- $3n^2-100n+6 = \Theta(n^2)$ because both O and Ω apply.
- $3n^2-100n+6 \neq \Theta(n^3)$ because only O applies.
- $3n^2-100n+6 \neq \Theta(n)$ because only Ω applies.

Analysing Non-Recursive Algorithm

01. An element in an Array is called leader, if it is greater than all the elements to the right of it. The time complexity of the most efficient algorithm to print all leaders of the given Array of size 'n' is ____

	1	2	3	4	5	6	7	8	9
A	30	19	20	9	15	12	7	8	5
	L		L		L	L		L	

(i)

```
Algo BF_leader (A,n)
{ for i ← 1 to (n-1)
  { for j ← (i+1) to n
    { if (A[i] < A[j])
        break; }
    if (j = n+1) print (A[i])
  }
}
```

(i) Best case : (increasing order)

A | 5 | 10 | 15 | 20 |

Time = (n+1) : O(n)

(ii) Worst Case : (decreasing order)

A | 90 | 80 | 70 | 60 |

$= \frac{n(n+1)}{2} = O(n^2)$

(ii) Linear Search from Right-left:

	1	2	3	4	5	6	7
A	30	6	18	15	9	10	8
	L		L	L		L	

```
Algo EFF_Leader (A, n)
{ 1. L ← A[n]
  2. for i ← (n-1) down to 1
     { if (A[i] > L)
       { Print (A[i]);
         L ← A[i]; }
     } }
```

O(1)

Time : $1 + (n-1) \cdot O(1)$

$= n - 1 + 1$

$= O(n)$

$= \Omega(n)$

$\Theta(n)$

27/7/2023

Time Complexity Framework for Recursive Algorithms

① The time complexity of recursive algorithm is derived/represented as a recurrence relation.

② Solve Recurrence to get the value of a recurrence as a mathematical function (polynomial, logarithmic, exponential, constant).

③ Apply O, Ω, Θ.

1. Algorithm What(n)
{ if (n=1) return;
else
{ What(n-1);
B(n); }
} (function).

(1) Assume $B(n) = O(1)$.

Let $T(n)$ represent time complexity of What(n).

$T(n) = c, \quad n = 1 \quad (c > 0)$

$T(n) = a + T(n-1) + b, \quad (n > 1), \quad [a > 0, b > 0]$

↳ Recurrence

$T(n) = T(n-1) + d, \quad n > 1 \quad -(1) \quad [d = a+b]$

↳ constant.

Solving Recurrences

1.) Back Substitution (universal method) (also called Repeated Substitution).

2.) master method (Shortcut method)

3.) Recursion Tree

$T(n) = T(n-1) + d$ — ①

$T(n-1) = T(n-2) + d$ — ②

$T(n) = T(n-2) + d + d$ — ③

$= T(n-2) + 2d$ — ③

$= T(n-3) + 3d$ — ④

Generalization

$T(n-k) + kd$ — ⑤

$= T(1) + k.d$

$= c + kd$

$= c + d(n-1)$

$T(n) = c, \quad n=1$

$n - k = 1$

$k = n-1$

$T(n) = c + dn - d \quad n>1$

$O(n)$ } $\Theta(n)$ ✓

$\Omega(n)$ }

(ii) Assume $B(n) = O(n)$

$T(n) = c, \quad n=1$

$T(n) = a + T(n-1) + n$ — ①

$T(n-1) = T(n-2) + a + (n-1)$ — ②

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

$= T(n-2) + (n-1) + n + 2a$ — ③

$= T(n-3) + (n-2) + (n-1) + n + 3a$ — ④

$= T(n-k) + (n-(k-1)) + \ldots (n-1) + n + k.a$ — ⑤

$= T(1) + (n-(n-1-1)) + \dots + n + k \cdot a$

$= c + (2+3+\dots+n) + a(n-1)$

Let $c=1$.

$T(n) = (1+2+3+\dots+n) + an - a$

$= \sum_{i=1}^{n} i + an - a$

$$T(n) = \frac{n(n+1)}{2} + an - a$$

$\rightarrow O(n^2)$

$\rightarrow \Omega(n^2)$

$\Rightarrow \Theta(n^2)$

(iii) Assume $B(n) = O(1/n)$

$T(n) = c, \quad n=1$

$= a + T(n-1) + \frac{1}{n}$

$T(n) = T(n-1) + \frac{1}{n} + a \quad \text{—} (1)$

$T(n-1) = T(n-2) + \frac{1}{(n-1)} + a \quad \text{—} (2)$

$T(n) = T(n-2) + \frac{1}{(n-1)} + \frac{1}{n} + 2a \quad \text{—} (3)$

$= T(n-k) + \frac{1}{(n-(k-1))} + \dots + \frac{1}{n} + k \cdot a \quad \text{—} (4)$

$= T(1) + \left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} \dots + \frac{1}{n}\right) + a(n-1) \quad \text{—} (5)$

$= c + \text{“} + an - a$

(Let $c=1$)

$\rightarrow T(n) = \sum_{x=1}^{n} 1x + an - a$

$T(n) = \log n + an - a$

$O(n) \quad \Omega(n)$

$\Rightarrow \Theta(n)$

ex: $T(n) = T(n-1) + \frac{1}{n}$, $n>1$
$= c$, $n=1$
$\Big\} O(\log n)$

Q2. Algorithm Do_it(n)

```
{ if (n=1) return;
  else
  return (Do_it(n-1) + Do_it(n-1));
}
```

$T(n-1)$ $T(n-1)$

u_1 u_2

$T(n) = c, \quad n=1$

$= a + 2 \cdot T(n-1) + b$

$T(n) = 2 \cdot T(n-1) + d \quad (d = a+b)$

$T(n) = 2T(n-1) + d$ — ①

$T(n-1) = 2T(n-2) + d$ — ②

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

$= 4T(n-2) + 3d$ — ③

$= 2^2 \cdot T(n-2) + (2^2 - 1) \cdot d$

$= 2^3 + (n-3) + (2^3 - 1) \cdot d$

$\vdots \quad \vdots$

$= 2^k \cdot T(n-k) + (2^k - 1)d$ — ④

$= 2^k \cdot T(1) + 2^k \cdot d - d$

$= 2^{n-1} \cdot c + 2^{n-1} \cdot d - d$

$\boxed{T(n) = e \cdot 2^n - d}$ $< O(2^n), \Omega(2^n)$
$\Theta(2^n)$

Q3. Algorithm A(n)

$A(\sqrt{n})$

```
{ if (n==2) return;
  else return (A(√n)); }
```

$T(n) = c, \quad n = 2$

$= a + T(\sqrt{n}), \quad n > 2$

$T(n) = T(n^{1/2}) + 1.a \quad — 1.$

$T(n^{1/2}) = T(n^{1/4}) + a \quad — 2$

$T(n) = T(n^{1/4}) + 2a \quad — 3.$

$= T(n^{1/2^2}) + 2a$

$= T(n^{1/2^3}) + 3a$

$= T(n^{1/2^k}) + ka = T(2) + a.k$

$= c + a \log\log n$

$\boxed{T(n) \text{ is } O(\log\log n)}$

$n^{\frac{1}{2^k}} = 2$

$\frac{1}{2^k} \cdot \log n = \log_2 2$

$2^k = \log n$

$\boxed{k = \log\log n}$

Q4 Algorithm A(n)

```
{ if (n==2) return;
  else return(A(√n) + A(√n));
}
```

$T(n) = c, \quad n = 2$

$= a + 2T(\sqrt{n}) + b, \quad n > 2$

$T(n) = 2T(\sqrt{n}) + d, \quad n > 2 \quad [d = a + b > 0]$

$T(n) = 2.T(n^{1/2}) + d \quad — 1.$

$T(n^{1/2}) = 2.T(n^{1/4}) + d \quad — 2$

$T(n) = 2[2T(n^{1/4}) + d] + d$

$= 4.T(n^{1/4}) + 3d \quad — 3$

$= 2^2.T(n^{1/2^2}) + (2^2 - 1)d$

$= 2^3.T(n^{1/2^3}) + (2^3 - 1)d$

$= 2^k.T(n^{1/2^k}) + (2^k - 1)d$

$n^{1/2^k} = 2$

$\Rightarrow 2^k = \log n$

$T(n) = 2^k.T(n^{1/2^k}) + (2^k - 1).d$

$= \log n.c + \log n.d - d$

$\boxed{O(\log n)}$

$2^{\log_2 \log_2 n} = \log n^{\log_2 2}$

Q5 $T(n) = 2, \quad n = 2$

$= \sqrt{n} \cdot T(\sqrt{n}) + n, \quad n > 2$

$T(n) = n^{1/2} \cdot T(n^{1/2}) + n$ — ①

$T(n^{1/2}) = n^{1/4} \cdot T(n^{1/4}) + n^{1/2}$ — ②

$T(n) = n^{1/2}\left[n^{1/4} \cdot T(n^{1/4}) + n^{1/2}\right] + n$

$= n^{3/4} \cdot T(n^{1/4}) + 2n$ — ③

$\left(3/4 = 1 - \frac{1}{2^2}\right)$

$T(n) = n^{1-\frac{1}{2^2}} \cdot T\left(n^{1/2^2}\right) + 2n$ — ④

$= n^{1-\frac{1}{2^3}} \cdot T\left(n^{1/2^3}\right) + 3n$ — ⑤

$= n^{1-\frac{1}{2^k}} \cdot T\left(n^{1/2^k}\right) + k \cdot n$ — ⑥

$= \frac{n}{n^{1/2^k}} \cdot T(2) + n \cdot k$

$= \frac{n}{2} \cdot 2 + n \cdot \log\log n$

$= n + n \log\log n$

$n^{1/2^k} = 2$

$k = \log\log n$

$\boxed{T(n) = O(n \cdot \log\log n)}$

Q6

```
Algorithm Recur(n)
{ if(n==1) return;
  else
  { Recur(n/2);
    Recur(n/2);
    B(n); }
}
```

$T(n) = c, \quad n = 1$

$= a + 2 \cdot T(n/2) + b * n, \quad n > 1$

$T(n) = 2T(n/2) + d$ — ①

$T(n/2) = 2T(n/4) + d$ — ②

$T(n) = 2[2T(n/4) + d] + d$

$= 4T(n/4) + 3d$ — ③

$= 2^2 T(n/2^2) + (2^2 - 1)d$

$= 2^3 T(n/2^3) + (2^3 - 1)d$

$T(n) = 2^k \cdot T(n/2^k) + (2^k - 1) \cdot d$

$= n \cdot T(1) + (n-1)d$

$= c \cdot n + dn - d$

$\frac{n}{2^k} = 1$

$n = 2^k$

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

(i) Assume $B(n) = O(1)$

(ii) Assume $B(n) = O(n)$

$T(n) = 2 \cdot T(n/2) + n + a$ — ①

$T(n/2) = 2T(n/4) + n/2 + a$ — ②

$T(n) = 2[2T(n/4) + n/2 + a] + n + a$

$= 4 \cdot T(n/4) + 2n + 3a$ — (3)

$= 2^2 T(n/2^2) + 2n + (2^2 - 1) \cdot a$ — (4)

$= 2^3 T(n/2^3) + 3n + (2^3 - 1) a$ — (5)

$\vdots$

$= 2^k T(n/2^k) + k \cdot n + (2^k - 1) a$ — (6)

$= n \cdot c + n \cdot \log n + an - a$

$\downarrow$

$O(n \log n)$

$\Omega(n \log n)$

$\frac{n}{2^k} = 1$

$n = 2^k$

$k = \log n$

$\therefore T(n) = \Theta(n \log n)$

$\downarrow$

Q7

```
Algo Do-it (n)
{ if (n==1) return;
  else
  { Do-it (n/2);
    B(n); }
}
```

(i) $B(n) \rightarrow O(1)$

$T(n) = c, \quad n = 1$

$= a + T(n/2) + b, \quad n > 1$

$T(n) = T(n/2) + d$ — (1)

$T(n/2) = T(n/4) + d$ — (2)

$T(n) = T(n/4) + 2d$ — (3)

$= T(n/2^2) + 2d$ — (4)

$\vdots$

$= T(n/2^k) + k \cdot d$ — (5)

$= T(1) + d \cdot \log n$

$= c + d \cdot \log n$

$T(n) = \Theta(\log n)$

$\frac{n}{2^k} = 1$

$k = \log n$

(ii) $B(n) = O(n)$

$T(n) = c, \quad (n = 1)$

$= a + T(n/2) + n \cdot (n > 1)$

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

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

$T(n) = T(n/2) + n$ — (1)

$T(n/2) = T(n/4) + n/2$ — (2)

$T(n) = T(n/4) + \frac{n}{2^1} + \frac{n}{2^0}$ — (3)

$T(n) = T(n/2^2) + \frac{n}{2^1} + \frac{n}{2^0}$ — (3)

$= T(n/2^3) + \frac{n}{2^2} + \frac{n}{2^1} + \frac{n}{2^0}$ — (4)

$T(n) = T(n/2^k) + \frac{n}{2^{k-1}} + \frac{n}{2^{k-2}} \cdots + \frac{n}{2^0}$

$= T(1) + \frac{n}{2^0} + \frac{n}{2^1} \cdots + \frac{n}{2^{k-1}}$

$= c + n \left[\sum_{i=0}^{k-1} 1/2^i \right]$

$= c + n \left[\frac{1(1 - 1/2^k)}{1 - 1/2} \right]$

$= c + 2n \left[1 - 1/2^k \right]$

$= c + 2n - \frac{2n}{2^k}$

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

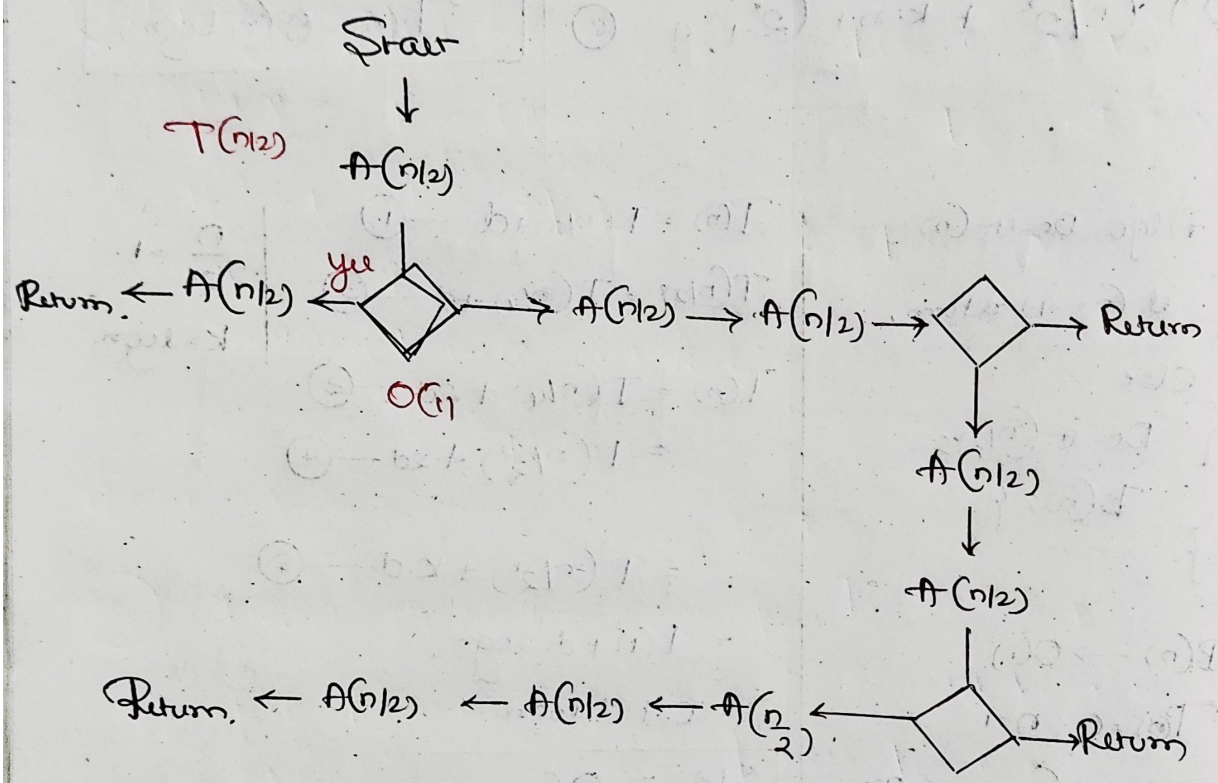
$\hookrightarrow T(n) = O(n)$

$\frac{n}{2^k} = 1$

$= n = 2^k$

Homework

→ The given diagram represents the flowchart of recursive algorithm A(n). Assume that all statements except for recursive call have order (1) time Complexity. Then the best case and worst case time complexity of the algorithm is.



Best Case : $T(n) = 2.T(n/2) + c \Rightarrow O(n)$

Worst Case : $T(n) = 8T(n/2) + c \Rightarrow O(?)$

(ii) $T(n) = 2T(n/2) + n\log n, \quad n>1$

$= c, \quad n=1$

$$T_n = \begin{cases} 8T(n/2)+c, & n>1 \\ a, & n=1 \end{cases}$$

$T(n) = 8T(n/2) + c$ — (1)

$T(n/2) = 8T(n/4) + c$ — (2)

$T(n) = 8(8T(n/4)+c) + c$

$= 64T(n/4) + 9c$ — 3

$= 8^3 T(n/2^3) + (2^3+1)c$ — 4

$\vdots$

$8^k T(n/2^k) + (2^{k+1}+1)c$

$T_n = 8^{\log n}.T(1) + \cdots$

$= n^{\log_2 8}.c + \cdots$

$O(n^3)$

(ii) $T(n) = 2T(n/2) + n\log n$ — ①

$T(n/2) = 2.T(n/4) + n/2.\log n/2$ — ②

$T(n) = 2[2.T(n/4) + n/2.\log n/2] + n\log n$ — ③

$= 4T(n/4) + n\log_2 n/2 + n.\log n/2^0$

$= 2^2 T(n/2^2) + n.\log_2 n/2 + n.\log n/2^0$

$= 2^k.T(n/2^k) + n.\log n/2^{k-1} + \ldots + n.\log n/2^0$ — ④

$= 2^k.T(n/2^k) + n.[\log n/2^{k-1} + \log n/2^{k-2} \ldots + \log n/2^0]$

$= 2^k.T(1) + n.\left[\log\left(\frac{n^k}{2^{0+1+2\ldots+k-1}}\right)\right]$

$= 2^k.T(1) + n\left[\log n^k - \log_2 2^{\frac{k(k-1)}{2}}\right]$

$\frac{n}{2^k} = 1$

$n = 2^k$

$k = \log n$

$= 2^k.c + n\left[k.\log n - \frac{k(k-1)}{2}\right]$

$= 2^k.c + n\left[2k.\log n - k^2 + k\right]$

$= n.c + \frac{n}{2}\left[2\log^2 n - \log^2 n + \log n\right]$

$= c.n + \frac{n}{2}\left[\log^2 n + \log n\right]$

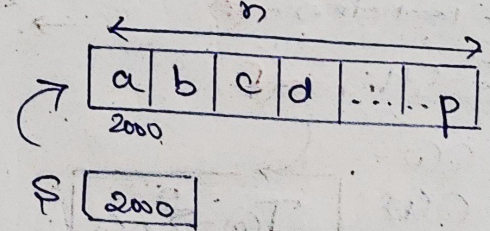
$= cn + \frac{1}{2}n.\log^2 n + \frac{n.\log n}{2}$

$\therefore T(n) = O(n.\log^2 n)$

Q8

```
void abc (char *s)
{ if (*s != '\0')
  { printf ("%c", *s);
T(n-1) abc (s+1);
T(n-1) abc (s+1); }
}
```

Let 's' be a pointer to a string of length 'n' characters. What is the time complexity?



$T(n) = c, \quad n=1$

$= a+b+2T(n-1), \quad n>1$

$T(n) = 2T(n-1)+d, \quad n>1 \; d>0$

$\boxed{T(n) = O(2^n)}$

ex: integer A[n] :

```
Algorithm RSum(A,n)
{ if(n=1) return (A[1]);
  else
    return (RSum(A, n-1)+A[n]);
}
```

(c) $T(n-1)$

$T(n) = c, \quad n=1$

$= T(n-1)+d, \quad n>1$

$T(n) = O(n)$

ex:

```
Algorithm Do_it(n)
{ if (n=1) return
  else
    return (Do_it(n-1)+n);
}
```

$T(n) = c, \quad n=1$

$= a+T(n-1)+b, \quad n>1$

$= T(n-1)+d$

$T(n) = O(n)$

Loop Complexities

for i ← 1 to n ⟺ for(i=1 ; i<=n ; ++i)

{ S1; S2; } ⟹ Total times the loop repeats = n.

eg: c=0

1. for i ← 1 to n —(n)

   c = c+1; —(1)

i=1 O(1)
i=2 O(1)
⋮
i=n O(1)

$T(n) = \sum_{i=1}^{n} O(1)$

n.

→ Time complexity depends on no of times the loop repeats (n) and complexity of statements in body of loop.

```
for i ← 0 to n { (n+1)
    c = c+1    { O(n)
```

n+c ⇒ O(n)

```
c = 0
for i ← 1 to n
    c = c+i,   c = (1+2+3...+n) = n(n+1)/2 = O(n²)
```

Time = O(n)
Value (c) = O(n²)

```
for i ← 1 to n/2 :
    c = c+1
```

Time = O(n)
Value = O(n)

```
for i ← 1 to n { O(1)
    break;
    read(x);
```

```
for i ← 2 to n
Body { if (x%i == 0)
         break;
```

Worst case : O(n)
Best case : O(1)

```
for i ← 1 to n
    B(n);
```

Time : O(n * O(B(n)));

(i) if B(n) = O(1) ⇒ O(n)
(ii) if B(n) = O(log n) ⇒ O(n log n)
(iii) if B(n) = O(n) ⇒ O(n²)

```
c = 0
for (i=1; i<n; ++i); i = n+1;
    c = c+i;
```

Time : O(n)
Value of c = n+1

```
c = 0
while(1)
{ c = c+1; }
```

Time is ∞

while loop

```
i = 1;
while (i <= n)
{ i = i+1; }
```

n

Time Complexity is O(n)

Nested loops:

```
for (i=1; i<n; i++)
{ for (j=1; j<=n; j++)
  { c=c+1; }
}
```

n

$i=1$, $j=1$ to n
$i=2$, $j=1$ to n
$i=n$, $j=1$ to n

$n \times n = n^2$

$O(n^2)$

Mutually Exclusive loops

```
{ 1. for i ← 1 to n : n
     c=c+1;
  2. for j ← 1 to n  n
     k=k+2;
}
```

Time: $O(n+m)$

: $O(\max(n,m))$

```
for i ← 1 to n : n
{ a) B(n); : log n.
  b) for j ← 1 to n/2
       c=c+1;        } O(n)
     k=0;
  c) while (k<=n)
       k++;          } O(n)
}
```

if $B(n) = O(\log n)$

Time $= O(n^2)$

Per iteration of 'i':

$(\log n + O(n) + O(n)) = O(n)$

Total_time $= n * [\log n + n + n]$

$= n \cdot \log n + n^2 + n^2 = O(n^2)$

Questions on loops : Determine Time Complexity.

①
```
for i ← 1 to n
    c=c+1;
```
$\Rightarrow O(n)$

②
```
for i ← 1 to n
  for j ← 1 to n/2
    c=c+1;
```
$\Rightarrow n \times n/2 = O(n^2)$

③
```
for i ← 1 to n
  for j ← 1 to n/4
    for k ← 1 to n
      break;
```
Time $= n \times \frac{n}{4} \times 1 \Rightarrow O(n^2)$

29/7/2023

1. Let $f(n)=n$ and $g(n)=n^{(1+\sin n)}$ where n is a positive integer. Which of following statements are correct?

I: $f(n)=O(g(n))$ ×

II: $f(n)=\Omega(g(n))$ ×

⇒ Neither of I nor II are correct.

2. Let f and g be functions of natural numbers given by $f(n)=n$ and $g(n)=n^2$. Which of the following statements is/are true?

⇒ $f \in O(g)$

$f \in o(g)$

Running time of Program segments with loops

④
```
for(i=1; i<=n; ++i)      — n
    for(j=1; j<=n; ++j)  — n
        for(k=n/2; k<=n; k+=n/2) — 2
            c=c+1;
```

$k=n/2+n/2=n+n/2$

$=3/2\ 3n/2 > n$

$n \times n \times 2 = 2n^2$

$=O(n^2)$

⑤
```
i=1
while(i<=n)
{ i=i*2; }
```

$i=1\times2^0, 1\times2^1, 1\times2\times2(2^2) \ldots 1\times2^k$

$i=2^k=n$

$k=\log_2 n$

Time: $O(k)$

$=O(\log_2 n)$

⑥
```
i=n
while(i>0)
{ i=i/2; }
```

assuming $n=16$

$i=16, 8, 4, 2, 1, 0$

$2^4, 2^3, \ldots 2^0$

$\therefore$ Time $= O(\log_2 n)$

⑦
```
for(int i=1; i<=n; i+=1)   → O(n)
    c=c+1;
```

⑧ for (i=1; i<=n; i=i+2) — i=1, 1+2, 1+2+2, ...
C=C+1; = 1+(2×0), 1+(2×1), ...
= 1+2×k = n k = (n-1)/2 = O(n)

⑨ for (i=1; i<=n; i=i+5) = 1, 1+5, 1+5+5
C=C+1; = 1+5×k ≠ n
k = (n-1)/5 ≈ n/5 = O(n)

Generalised form

for (i=1; i<=n; i=i+a)
↳ n/a

⑩ for (i=1; i<=n; i=i*2); = $\log_2 n$

⑪ for (i=1; i<=n; i=i*3) = $\log_3 n$ i = 1, 3, 9, 27
$i = 1 \times 3^k = n$
$k = \log_3 n$

Generalised form

for (i=1; i<=n; i=i*a);
= $O(\log_a n)$

⑫ for (i=1; i<=n; ++i) — n
for (j=1; j<=n; j=2*j) — $\log_2 n$ } $O(n \cdot \log n)$
C=C+1;

⑬ $n = 2^n$
for (i=1; i<=n; ++i) — n
for (j=1; j<=m; j=2*j) — $\log_2 n$ } $n \times \log_2 n = n \times \log_2 2^n$
$= n \times n = n^2$
$= O(n^2)$
C=C+1;

(14)
```
c=0
for i ← 1 to n
    for j ← i to n
        c = c+1;
```

Value of c:
i = 1, 2, 3, ..., n
c = n, (n+n-1), n+(n-1)+(n-2), ..., n+(n-1)+...+1

$$\text{Value of } c = \frac{n(n+1)}{2}$$

Time = $O(n^2)$

(15) $f(n) = \sum_{i=1}^{n} i \cdot O(n)$

$= O(n) * \sum_{i=1}^{n} 1$

$= O(n) * O(n) \Rightarrow O(n^2)$

$f(n) = \sum_{i=1}^{n} i = O(n)$ } Mathematical Summation

$\Updownarrow$

```
for i ← 1 to n
    c = c+1
```
} Programming loop

(16)

$f(n) = \sum_{i=1}^{k} O(n) \quad (k>0)$

$= O(n) \cdot \sum_{i=1}^{k} 1 \Rightarrow O(n) * k \Rightarrow O(n)$

(17)
```
i = n;
while (i>0) ---- log n times
{ j = 1;
  while (j<=n) } log n
  { j = 2*j; } for every iteration
  i = i/2; }
```

(log n × log n)

∴ Time = $O(\log_2^2 n)$

(18)
```
int fun (int n)
{ int i, j, p, q = 0;
  for (i=1; i<=n; ++i) ---- n
  { p = 0;
    for (j=n; j>1; j=j/2) } log n
        ++p;
    for (k=1; k<p; k=k*2) } log p
        ++q; }
return (q); }
```

The value of q returned by the function
→ $O(n \log \log n)$

$q = n * \log p$
$= n \times \log_2(\log_2 n)$ $\quad p = \log n$

Time Complexity:
$= O(n \cdot \log_2 n)$

→ If we put 'q=0' in second loop.

(i) then value of $q = O(\log\log n)$

(ii) time complexity $= O(n \cdot \log_2 n)$

(19)
```
for (i=1; i<=n; ++i)        ---- n
{   j=1                     ---- n
    while (j<=n)            ---- n × log n
      j = 2*j;
    for (k=1; k<=n; ++k)    ---- n → n^2
      c = c+1;
}
```

Time $= n^2 + n\log n + n$

$= O(n^2)$

(20) $n = 2^{2^k}$

```
for (i=1; i<=n; ++i)
{   j=2;
    while (j<=n)
    {   j = j*j;
        pf(++);  }
}
```

'++': $n(k+1)$

: $n \times \log\log + n$

$k=2$

$n = 2^4 = 16$

i=1	i=2	i=n
j=2,4,16	j=2,4,16	j=2,4,16
(+++)	(+++)	(+++)

$k=3$

$n = 2^8 = 256$

i=1

j=2,4,16,256

(++++)

Iteration : $O(k)$

Total iter $= O(n \times k)$

$= O(n \log\log n)$

$n = 2^{2^k}$

$k = \log_2 \log n$

(21) $n = 2^{2^k}$

```
for (i=2; i<=n; i = i*i)
    c = c+1;
```

Time $= O(\log\log n)$

assuming $n=16$

$i = 2, 4, 16$ $(2^{2^0}, 2^{2^1}, 2^{2^2})$

$i = 2^{2^k} = n$ $\quad k = \log\log n$

(22) $n = 2^{2^k}$

```
for (i=1; i*i<=n; i++)
    c = c+1;
```

Time $= i^2 \le n$

$i^2 = n \Rightarrow i = \sqrt{n}$

Time $O(1) = O(\sqrt{n})$

(23) for (i=n; i>=2; i=sqrt(i)) $n = 2^{2^k}$

c=c+i;

n = 256

$i = 256 : (256)^{1/2}, [(256)^{1/2}]^{1/2}, [((256)^{1/2})^{1/2}]^{1/2} = (n)^{1/2^k} \cdot 2$

$k = \log\log n$

16 → 2^{2^4} ; 4 → 2^{2^2} ; 2 → 2^{2^1} ; 2^{2^0}

$2^{2^k} < n \Rightarrow 2^{2^k} = n$

$k = \log\log n$

(24) A: Array [1...n] of binary (0|1)

```
f(n) = O(n);
Count : integer;
Count = 1
for i = 1 to n
{ if (A[i] == 1) Count++;
  else
  { f(Count);
    Count = 1; }
}
```

If this statement is removed.

Case 1 : O(n) } Best Case

Case 2 : O(n)

Case 3 : O(n)

Case 4 : $\left(\frac{n}{2} + \frac{n}{2} \times \frac{n}{2}\right) = O(n^2)$ } Worst Case

Time Complexity : Case

1) A[i] = 1, i = 1, n

Time = O(n)

2) A[i] = 0, i = 1, n

Time = O(n)

3) A[i] = 1 i = 1, n-1

= 0 i = n

1	1	1	1	1	1	1	0

Time = (n-1) + (n-1) + 1

= 2n - 1 = O(n)

4) A[i] = 1, i = 1, n/2

= 0, i = (n/2)+1, n

n/2 n/2-1

A:

1	1	1	1	0	0	0	0

Time: $\frac{n}{2} + \left(\frac{n}{2} + 1\right) O(1)$

= O(n)

Homework.

1. Consider the following function:

```
int unknown (int n)
{ int i, j, k = 0;
  for (i = n/2; i <= n; i++)
    for (j = 2; j <= n; j = j*2)
      k = k + n/2;
  return k; }
```

The return value of function is _____

2.
```
int fun (int n)
{ int i, j;
  for (i = 1; i <= n; i++){
    for (j = 1; j < n; j += i)
      { Printf ("%d %d", i, j); }
  }}
```

Time complexity of fun in terms of Θ notation is _____

Q1. Consider the following functions Function_1 and Function_2 expressed in pseudocode as follows:

Function_1

```
while n > 1 do
    for i = 1 to n do
        x = x + 1;
    end for
    n = ⌊n/2⌋;
end while
```

if n = 16: 1, 2, 3 ... $\left(\frac{n}{2^0}, \frac{n}{2^1}, \frac{n}{2^2}\right)$

$= O(n)$

Function_2

```
for i = 1 to 100*n
do
    x = x + 1;
end for
```

Time = 100 × n = O(n)

T(n) = 100 × n.

Let $f_1(n)$ and $f_2(n)$ denote the no. of times the statement "x = x + 1" is executed in "Function_1" and "Function_2" which of following statement is true?

Total time $= n + \frac{n}{2} + \frac{n}{4} \ldots \frac{n}{2^k}$

$= n\left[\sum_{i=1}^{k} \frac{1}{2^i}\right] = n\left(1 - \frac{1}{2^k}\right)$

$= n-1$

$= O(n)$

$\therefore f_1(n) \in \Theta(f_2(n))$ ✓

$f_1(n) \in O(n)$ ✓

(25) `K=1, i=1;`

`while (K<=n)`

`{ i++;`

`K = K+i; }`

i = 1, 2, 3, 4 ..... t

K = 1, (1+2), (1+2+3), (1+2+3+4) ... (1+2+3..t)

$(1+2+3 \ldots t) = n$

$\frac{t(t+1)}{2} = n$

$t^2 + t \approx 2n$

$t^2 \approx n \quad \therefore t = \sqrt{n}$

Time $= O(t)$

$= O(\sqrt{n})$

Space - Complexity

(memory)

Program / Algorithm

Instructions

Data (Input)

Space Complexity

Additional / Auxiliary Space

```
void swap(int a, int b)
{
    int temp;
1.  temp = a;
2.  a = b;
3.  b = temp;
}
```

Input

2B = Constant

O(1)

```
Algorithm Sum (A,n)
{ integer n, A[n], i, sum;        ⌈2B
    Sum=0;                        (2B
  for i ← 1 to n
    Sum = Sum + A[n];
  }
```

Time = O(n)
Space = O(1)

Input n, A[n]
Auxiliary space = 1 Bytes
O(1)

Space Complexity = We define the space used by an algorithm to be the number of memory cells (or words) needed to carry out the computational steps required to solve an instance of the problem excluding the space allocated to hold the input. In other words, it is only work space required by the algorithm

All definations of order of growth and asymptotic bounds pertaining to time complexity carry-over to space complexity. It is clear that the work space cannot exceed the running time of the algorithm, as writing into each memory cell requires atleast a constant amount of time.

Thus if we let T(n) and S(n) denote respectively, the time and space complexities of the algorithm, then

$$S(n) = O(T(n))$$

Example: In algorithm linear search, only one memory cell is used to hold the result of the search. If we add local variables eg for looping, we conclude that amount of space needed is O(1). This is also the case in algorithms Binary Search, Selection Sort and Insertion Sort.