Algorithms
Chapter 1: Algorithim Analysis

GATE CS Lectures
by Monalisa

MonalisaCS

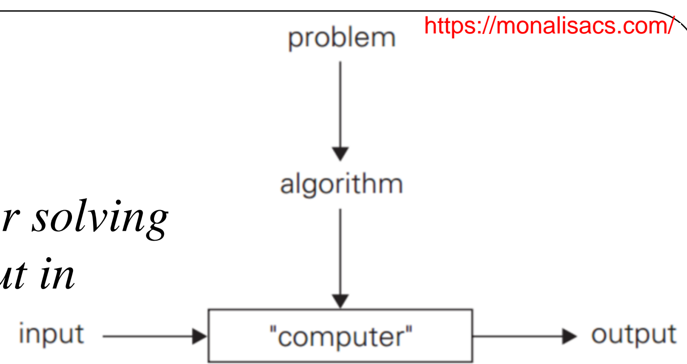
Section 5: Algorithms

- Searching, sorting, hashing. Asymptotic worst case time and space complexity. Algorithm design techniques : greedy, dynamic programming and divide-and-conquer . Graph traversals, minimum spanning trees, shortest paths
- Chapter 1:Algorithim Analysis:-Algorithm intro , Order of growth ,Asymptotic notation, Time complexity, space complexity, Analysis of Recursive & non recursive program, Master theorem]
- Chapter 2:Brute Force:-Sequential search, Selection Sort and Bubble Sort , Radix sort, Depth first Search and Breadth First Search.
- Chapter 3: Decrease and Conquer :- Insertion Sort, Topological sort,Binary Search .
- Chapter 4: Divide and conquer:-Min max problem , matrix multiplication ,Merge sort ,Quick Sort , Binary Tree Traversals and Related Properties .
- Chapter 5: Transform and conquer:- Heaps and Heap sort, Balanced Search Trees.
- Chapter 6: Greedy Method:-knapsack problem , Job Assignment problem, Optimal merge, Hoffman Coding, minimum spanning trees, Dijkstra's Algorithm.
- Chapter 7: Dynamic Programming:-The Bellman-Ford algorithm ,Warshall's and Floyd's Algorithm ,Rod cutting, Matrix-chain multiplication ,Longest common subsequence ,Optimal binary search trees
- Chapter 8: Hashing.
- Reference : Introduction to Algorithms by Thomas H. Cormen
- Introduction to the Design and Analysis of Algorithms, by Anany Levitin
- My Note

MonalisaCS

Introduction

https://monalisacs.com/



- The study of algorithms, sometimes called algorithmics.

What Is an Algorithm?

- *An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any input in a finite amount of time.*

1. The nonambiguity requirement for each step of an algorithm cannot be compromised.
2. The range of inputs for which an algorithm works has to be specified carefully.
3. The same algorithm can be represented in several different ways.
4. There may exist several algorithms for solving the same problem.

- An **algorithm** is any well-defined computational procedure that takes some value, or set of values, as **input** and produces some value, or set of values, as **output**.
- An algorithm is thus a sequence of computational steps that transform the i/p into the o/p.
- An ***instance of a problem*** consists of the input.
- An algorithm is said to be ***correct*** if, for every input instance, it halts with the correct output.

Characteristics of Algorithm

- Input ,Output ,Definiteness ,Finiteness ,Effectiveness

MonalisaCS

https://www.youtube.com/@MonalisaCS

Methodology of Analysis

- **A Priori analysis** means, analysis is performed prior to running it on a specific system.
- We determine the time and space complexity of an algorithm by just looking at the algorithm rather than running it on a particular system with a different memory, processor, and compiler.
- Independent of platform (Language ,Hardware ,Software..)
- Priori analysis is an absolute analysis
- In priori, we use asymptotic notations to determine time and space complexity as they are asymptotically same from computer to computer.
- **A Posteriori analysis** means, analysis of an algorithm only after running it on a system.
- It directly depends on the system and changes from system to system.
- Dependent on platform(Language ,hardware ,Software..)
- Posteriori analysis is a relative analysis.
- There are two kinds of efficiency:
- ***Time efficiency***, also called ***time complexity***,indicates how fast an algorithm runs.
- ***Space efficiency***, also called ***space complexity***, refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output.

MonalisaCS

Standard notations of common functions

Monotonicity

- A function $f(n)$ is ***monotonically increasing*** if $m \leq n$ implies $f(m) \leq f(n)$.
- It is ***monotonically decreasing*** if $m \leq n$ implies $f(m) \geq f(n)$.
- A function $f(n)$ is ***strictly increasing*** if $m < n$ implies $f(m) < f(n)$ and ***strictly decreasing*** if $m < n$ implies $f(m) > f(n)$.

Floors and ceilings

- For any real number x, we denote the greatest integer less than or equal to x by $\lfloor x \rfloor$ (read "the floor of x") and the least integer greater than or equal to x by $\lceil x \rceil$ (read "the ceiling of x").

Modular arithmetic

- For any integer a and any positive integer n, the value a mod n is the ***remainder*** (or ***residue***) of the quotient a=n .

Polynomials

- Given a nonnegative integer d, a ***polynomial in n of degree*** d is a function $p(n)$ of the form
- $p(n)=\sum_{i=0}^{d} a_i n^i$
- Where , constants $a_0, a_1, \ldots a_d$ are the ***coefficients*** of the polynomial and $a_d \neq 0$.
- A function $f(n)$ is ***polynomially bounded*** if $f(n)=O(n^k)$ for some constant k.

MonalisaCS

Exponentials

- For all n and a ≥ 1, the function a^n is monotonically increasing in n.
- For all real a >0, m, and n, we have the following identities:
- $a^0=1$, $a^1=a$, $a^{-1}=1/a$,
- $(a^m)^n= a^{mn}$
- $(a^m)^n= (a^n)^m$
- $a^m a^n= a^{m+n}$
- $a^m / a^n= a^{m-n}$

Logarithms

- We shall use the following notations:
- $\lg n = \log_2 n$ (binary logarithm),
- $\ln n = \log_e n$ (natural logarithm),
- $\lg^k n = (\lg n)^k$ (exponentiation),
- $\lg \lg n = \lg(\lg n)$ (Composition),
- For all real $a > 0$, $b > 0$, $c > 0$, and $n > 0$, logarithm bases are not 1
- $a^b=n$ then $\log_a n=b$, $a = b^{\log_b a}$,
- $\log_c (ab)=\log_c a+\log_c b$, $\log_c (a/b)=\log_c a-\log_c b$
- $\log_b a^n=n \log_b a$,

MonalisaCS

https://monalisacs.com/

- $\log_b a = \frac{\log_c a}{\log_c b}$,
- $\log_b (1/a) = -\log_b a$,
- $\log_b a = \frac{1}{\log_a b}$,
- $a^{\log_b c} = c^{\log_b a}$,

❖ Factorials

- The notation *n!* (read "n factorial") is defined for integers n≥ 0 as
- $n! = 1$ if $n = 0$
- $n! = n.(n-1)!$ if $n > 0$
- Thus, n!= 1. 2. 3….. n.
- A weak upper bound $n! \leq n^n$, since each of the n terms in the factorial product is at most n.

❖ Fibonacci numbers

- We define the ***Fibonacci numbers*** by the following recurrence:
- $F_0 = 0$,
- $F_1 = 1$,
- $F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$.
- Each Fibonacci number is the sum of two previous ones ,
- yielding the sequence 0,1,1,2,3,5,8,13,21,….

MonalisaCS

https://www.youtube.com/@MonalisaCS

Order of growth

- Constant — $1, 10000000, 2^{10000000000000}, \ldots$
- Double logarithmic — Log log n
- Logarithmic — log n
- Polylogarithmic — $(\log n)^c$
- Fractional power or Root — $(n)^{1/c}, \sqrt[c]{n}$
- Linear — n,2n,..
- Linearithmic — n log n
- Polynomial — $n^2, n^3, n^a, \ldots$
- Exponential — $2^n, 3^n, a^n, \ldots$
- Factorial — n!

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

MonalisaCS

Using Limits for Comparing Orders of Growth:

- $\lim_{n\to\infty} \frac{t(n)}{g(n)} = 0$ ⇒ t(n) has smaller order of growth than g(n)
- $\lim_{n\to\infty} \frac{t(n)}{g(n)} = \text{constant}$ ⇒ t(n) has same order of growth as g(n)
- $\lim_{n\to\infty} \frac{t(n)}{g(n)} = \infty$ ⇒ t(n) has larger order of growth than g(n)

https://monalisacs.com/

- Ex : Compare order of growth of $\log_2 n$ and $\sqrt{n}$
- $\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n\to\infty} \frac{(\log_2 e)1/n}{1/2\sqrt{n}}$
- $=2 \log_2 e \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0$
- $\log_2 n < \sqrt{n}$

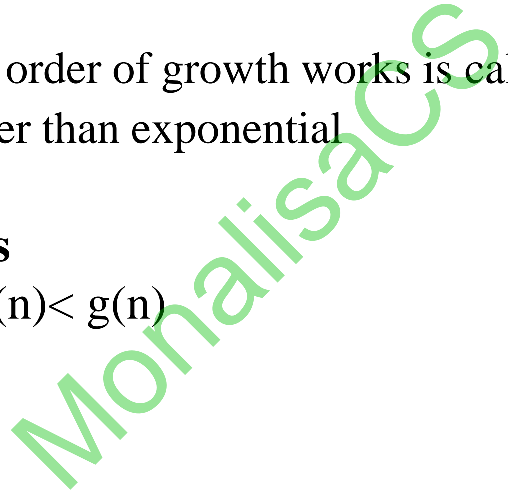
❖ **Threshold** :The value after which order of growth works is called threshold.

- Ex: Polynomial algorithm run faster than exponential
- After threshold point(4) $n^2 < 2^n$

n	n^2 <	2^n
1	1	2
2	4	4
3	9	8
4	16	16
5	25	32
6	36	64

❖ **Comparison between functions**

- Ex 1: $f(n) = 2^{100}$, $g(n) = \log n \Rightarrow f(n) < g(n)$
- Ex 2: (GATE CS 2001,Q:1.16)
- $f(n) = n^2 \log n$, $g(n) = n(\log n)^{10}$
- $n^2 \log n$? $n(\log n)^{10}$
- $n.(n \log n)$? $n \log n .(\log n)^9$
- n ? $(\log n)^9$
- n > $(\log n)^9$
- $f(n) > g(n)$



https://www.youtube.com/@MonalisaCS

- Ex 3: (GATE CS 1994 , Q: 1.23)
- $f(n) = n^3$, $0 \le n \le 10{,}000$
- $= n^2$, $n > 10{,}000$
- $g(n) = n$, $0 \le n \le 100$
- $= n^3$, $n > 100$
- $f(n) < g(n)$
- Ex 4: n , $\log n$, $\sqrt{n}$, $n \log n$, $\log^2 n$, n^n , $n^2 \log n$, 2^{2n}
- $\log n$, $\log^2 n$, $\sqrt{n}$, n , $n \log n$, $n^2 \log n$, 2^{2n} , n^n
- Ex 5: (GATE CS 2000, Q: 2.17) $3n^{\sqrt{n}}$, $2^{\sqrt{n} \log n}$, $n!$
- $2^{\sqrt{n} \log n} = n^{\sqrt{n} \log 2} = n^{\sqrt{n}}$
- $2^{\sqrt{n} \log n}$, $3n^{\sqrt{n}}$, $n!$
- Ex 6: $n^{3/2}$, $\sqrt{n}$, $\log n$, $1/n$, e^n , e^{2n} [e=2.7]
- $1/n$, $\log n$, $\sqrt{n}$, $n^{3/2}$, e^n , e^{2n}
- Ex 7: (GATE IT 2008,Q10) $n^{1/3}$, e^n , $n^{7/4}$, $n \log^9 n$, 1.0000001^n
- $n^{1/3}$, $n \log^9 n$, $n^{7/4}$, 1.0000001^n , e^n
- Ex 8: (GATE CS 2008,Q39) $f(n)=2^n$, $g(n)=n!$, $h(n)=n^{\log n}$
- $n^{\log n} < 2^n < n!$

MonalisaCS

Worst-Case, Best-Case, and Average-Case Efficiencies

- ALGORITHM *SequentialSearch* (*A[0…,n-1],K*)
- //Input : An array *A[0..n-1]* and a search key *K*
- //Output : The index of the first element in *A* that matches *K or -1* if no matching founds.
- *i = 0*
- ***while*** *i<n and A[i]≠ K* ***do***
- *i = i+1*
- ***if*** *i<n* ***return*** *i*
- ***else return*** *-1*
- In the **worst case**, when there are no matching elements or the first matching element happens to be the last one on the list, the algorithm makes the largest number of key comparisons among all possible inputs of size *n*: $C_{worst(n)} = n$.
- The ***worst-case running time***, that is, the longest running time for *any* input of size n.
- It guarantees that for any instance of size *n*, the running time will not exceed *Cworst(n)*.
- The input class for which the algorithm does maximum work hence taking maximum time are worst case input & worst case running time.
- Ex: Quick sort behave worst case for sorted input ,running time $O(n^2)$.

MonalisaCS

https://monalisacs.com/

- The ***best-case efficiency*** of an algorithm is its efficiency for the best case input of size *n*, for which the algorithm runs the fastest among all possible inputs of that size.
- For example, the best-case inputs for sequential search are lists of size *n* with their first element equal to a search key; $C_{best(n)} = 1$ for this algorithm.
- The input class for which the algorithm does minimum work hence taking minimum time are best case input & best case running time.
- Ex: Quick sort behave best case for random order ,running time O(n log n).
- ***Average-case running time*** is in between best case & worst case .
- Let's consider again sequential search. The standard assumptions are that
- (a) the probability of a successful search is equal to p $(0 \leq p \leq 1)$
- (b) the probability of the first match occurring in the *i*th position of the list is the same for every *i*.
- In the case of a successful search, the probability of the first match occurring in the *i*th position of the list is *p/n* for every *i*,and the number of comparisons made by the algorithm is *i*.
- In the case of an unsuccessful search, the number of comparisons will be *n* with the probability of such a search being $(1 - p)$. Therefore,
- $C_{avg(n)} = [1 \times \frac{P}{n} + 2 \times \frac{P}{n} + \ldots i \times \frac{P}{n} + \ldots\ldots n \times \frac{P}{n}] + n \times (1\text{-}p)$
- $= \frac{P}{n} [1 + 2 + \ldots\ldots i + \ldots\ldots + n] + n(1\text{-}p)$
- $= \frac{P}{n} \times \frac{n(n+1)}{2} + n(1\text{-}p) = \frac{p(n+1)}{2} + n(1\text{-}p)$
- If p = 1,the average number of key comparisons made by
- sequential search is (n + 1)/2

MonalisaCS

https://www.youtube.com/@MonalisaCS

- If worst case W(n) ,Best case B(n) ,Average case A(n)
- $B(n) \leq A(n) \leq W(n)$
- Merge sort / Heap sort B=A=W
- Quick Sort (B=A)<W
- Linear search /Binary search B<(A=W)

Asymptotic notation

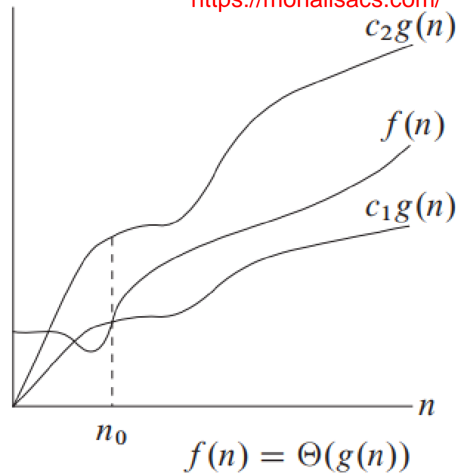
- Asymptotic notation primarily use to describe the running times of algorithms.
- There are two types of asymptotic notation.
- Big
  - Big Oh(O), Asymptotic upper bound, ≤
  - Big Omega(Ω), Asymptotic lower bound,≥
  - Theta(Θ), Asymptotically tight bound ,=
- Small
  - Little Oh (o),Proper Upper bound ,<
  - Little Omega(ω),Propper Lower bound ,>

❖ **Θ-notation**

- For a given function g(n),we denote by $\Theta(g(n))$ the set of functions
- $\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$.

MonalisaCS

- The value of $f(n)$ sandwiched between $c_1 g(n)$ and $c_2 g(n)$ inclusive
- "$f(n) \in \Theta(g(n))$" indicate that $f(n)$ is a member of $\Theta(g(n))$.
- "$f(n) = \Theta(g(n))$" express the same notion.
- for all $n \geq n_0$, the function $f(n)$ is equal to $g(n)$ to within a constant factor.
- We say that $g(n)$ is an ***asymptotically tight bound*** for $f(n)$.
- Ex : $\frac{1}{2}n^2 - 3n = \Theta(n^2)$ lets determine c_1, c_2, and n_0
- $c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$
- For all $n \geq n_0$, dividing by n^2
- $c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$
- Let $n_0 = 7$
- $c_1 \leq \frac{1}{2} - \frac{3}{7} = \frac{7-6}{14} = \frac{1}{14} \leq c_2$
- Let $c_1 = \frac{1}{14}$ and $c_2 = \frac{1}{2}$ and $n_0 = 7$



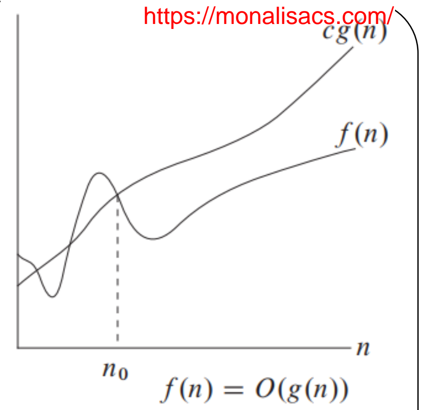
MonalisaCS

O-notation

- O-notation provides an ***asymptotic upper bound***.
- For a given function g(n),we denote by $O(g(n))$ ("big-oh of g of n" or just "oh of g of n") the set of functions

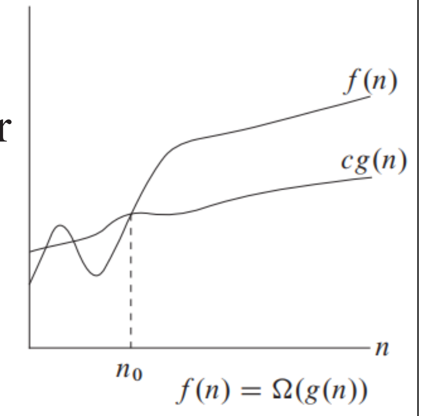
https://monalisacs.com/

- $O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$.
- $f(n) = O(g(n))$ indicate that a $f(n)$ is a member of the set $O(g(n))$
- $f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$,
- Since Θ notation is a stronger notion than O-notation.
- Written set-theoretically , we have $\Theta(g(n)) \subseteq O(g(n))$.
- Ex : an^2+bn+c where a>0 is in $\Theta(n^2)$ also in $O(n^2)$ also in $O(n^3)$.



❖ **Ω-notation**

- Ω-notation provides an ***asymptotic lower bound***.
- For a given function g(n),we denote by Ω (g(n)) ("big-omega of g of n" or just "omega of g of n") the set of functions
- $\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$.
- Ex : an^2+bn+c where a>0 is in $\Omega(n^2)$ also in $\Omega(n)$ also in $\Omega(1)$.



❖ o-**notation**

- The asymptotic upper bound provided by *O*-notation may or may not be asymptotically tight.
- The bound $2n^2 = O(n^2)$ is asymptotically tight, but $2n = O(n^2)$ is not.

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- We use o-notation to denote an upper bound that is not asymptotically tight.
- We formally define $o(g(n))$ ("little-oh of g of n") as the set
- $o(g(n))=\{f(n)$: for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$.
- Ex : $2n=o(n^2)$ but $2n^2 \neq o(n^2)$.
- The main difference is that in $f(n)= O(g(n))$, the bound $0 \leq f(n) \leq cg(n)$ holds for *some* constant $c > 0$, but in $f(n)= o(g(n))$, the bound $0 \leq f(n) < cg(n)$ holds for *all* constants $c > 0$.
- The relation $f(n)= o(g(n))$ implies that
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ $\Rightarrow$ ***f(n)*** has smaller order of growth than ***g(n)***

❖ **ω-notation**

- We use ω -notation to denote a lower bound that is not asymptotically tight.
- $f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$
- We define $\omega(g(n))$ ("little-omega of g of n") as the set
- $\omega(g(n))=\{f(n)$: for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0\}$.
- Ex: $n^2/2=\omega(n)$, but $n^2/2 \neq \omega(n^2)$.
- The relation $f(n)= \omega(g(n))$ implies that
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ $\Rightarrow$ ***f(n)*** has larger order of growth than ***g(n)***

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

Examples of Asymptotic Notation :

- $f_1(n)=2^{100}$ — $O(1), \Omega(1), \Theta(1)$
- $f_2(n)=2^{100}+\log n$ — $O(\log n), \Omega(\log n), \Theta(\log n)$
- $f_3(n)=2n+1000$ — $O(n), \Omega(n), \Theta(n)$
- $f_4(n)=n^2+\log n$ — $O(n^2), \Omega(n^2), \Theta(n^2)$
- $f_5(n)=\sum_{i=1}^{n} 1$ — $O(n), \Omega(n), \Theta(n)$
- $f_6(n)=\sum_{i=1}^{n} i$
- $=1+2+\ldots n$
- $=n(n+1)/2$ — $O(n^2), \Omega(n^2), \Theta(n^2)$
- $f_7(n)=\sum_{i=1}^{n} i^2$
- $=1^2+2^2+3^2+\ldots\ldots n^2$
- $=n(n+1)(2n+1)/6$ — $O(n^3), \Omega(n^3), \Theta(n^3)$
- $f_8(n)=\sum_{i=1}^{n} \log i$
- $=\log 1+\log 2+\ldots\log n$
- $=\log 1\times2\times3\times4\ldots n=\log n!=O(\log n^n)$ — $O(n \log n)$
- $f_9(n)=\sum_{i=1}^{n} 1/2^i$
- $=1/2+1/2^2+1/2^3\cdots\cdots+1/2^n$
- $=1-1/2^n$ — $O(1)$

MonalisaCS

https://monalisacs.com/

- $f_{10}(n) = n! = 1 \times 2 \times 3 \ldots\ldots \times n$
- $1 \times 1 \times 1 \times \ldots \times 1 \leq 1 \times 2 \times 3 \ldots\ldots \times n \leq n \times n \times n \ldots\ldots \times n$
- $1 \leq n! \leq n^n$ $\Omega(1), O(n^n)$
- $f_{11}(n) = \log n! = O(\log n^n)$ $O(n \log n)$
- True/False

1. $100n\log n = O(n\log n)$ True
2. $\sqrt{\log n} = O(\log \log n)$ False
3. $1/n = O(\log n)$ True
4. $2^{n+1} = O(2^n)$ True
5. $2^{2n} = O(2^n)$ False
6. $0 < x < y, n^x = O(n^y)$ True
7. $(n+k)^m = O(n^m)$,m,k>0 True
8. $2^{n^2} = O(n)$ False
9. $\sum_{i=1}^{n} n/i = O(n \log n)$ True

- $f(n) = n,\ g(n) = n^2$ $\Rightarrow f(n) = O(g(n))$ or $g(n) = \Omega(f(n))$
- $f(n) = 2^{10000},\ g(n) = \log n$ $\Rightarrow f(n) = O(g(n))$ or $g(n) = \Omega(f(n))$
- $f(n) = 2^n,\ g(n) = n^2$ $\Rightarrow g(n) = O(f(n))$ or $f(n) = \Omega(g(n))$

MonalisaCS

https://www.youtube.com/@MonalisaCS

Properties of Asymptotic Notations

- **Analogy Between Asymptotic Notation & Real number :** Let f, g functions and a,b real numbers
- $f(n) = O(g(n)) \Leftrightarrow a \leq b$, $f(n) \leq cg(n)$
- $f(n) = \Omega(g(n)) \Leftrightarrow a \geq b$, $f(n) \geq cg(n)$
- $f(n) = \Theta(g(n)) \Leftrightarrow a = b$, $c_1 g(n) \leq f(n) \leq c_2 g(n)$
- $f(n) = o(g(n)) \Leftrightarrow a < b$, $f(n) < cg(n)$
- $f(n) = \omega(g(n)) \Leftrightarrow a > b$, $f(n) > cg(n)$

ASN	Reflexivity	Symmetry	Transitivity	Transpose symmetry
O	Yes	No	Yes	Yes
Ω	Yes	No	Yes	Yes
Θ	Yes	Yes	Yes	No
o	No	No	Yes	Yes
ω	No	No	Yes	Yes

- **Relational properties**

Reflexivity :

- $f(n) = \Theta(f(n))$, $f(n) = O(f(n))$, $f(n) = \Omega(f(n))$

Symmetry:

- $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$

Transitivity:

- $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
- $f(n) = O(g(n))$ and $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
- $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$
- $f(n) = o(g(n))$ and $g(n) = o(h(n)) \Rightarrow f(n) = o(h(n))$
- $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$

Transpose symmetry:

- $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$
- $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$

MonalisaCS

https://monalisacs.com/

- $f(n)$ is ***asymptotically smaller*** than $g(n)$ if $f(n) = o(g(n))$ and $f(n)$ is ***asymptotically larger*** than $g(n)$ if $f(n) = \omega(g(n))$
- **Trichotomy:** For any two real numbers a and b, exactly one of the following must hold: $a < b$, $a = b$, or $a > b$.
- Although any two real numbers can be compared, not all functions are asymptotically comparable.
- For example, we cannot compare the functions n and $n^{1+\sin n}$ using asymptotic notation,
- Since the value of the $1+\sin n$ oscillates between 0 and 2, taking on all values in between.
- **General Properties:**
- If $f(n) = O(g(n))$ then $a \times f(n) = O(g(n))$,a>0
- If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$
- then $f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\})$ and $f_1(n) \times f_2(n) = O(g_1(n) \times g_2(n))$
- Ex: $2^{100} = O(1)$, $\log n = O(\log n)$
- $2^{100} + \log n = O(\max\{1, \log n\}) = O(\log n)$
- Ex : $3n^2 = O(n^2)$, $2\log n + 100 = O(\log n)$
- $3n^2 \times (2\log n + 100) = O(n^2 \log n)$
- Ex: $f(n) = O(g(n))$, $g(n) \neq O(f(n))$, $g(n) = O(h(n))$, $h(n) = O(g(n))$
- a) $f(n) = O(h(n))$ True b) $f(n) + g(n) = O(g(n) + h(n))$ True
- c) $h(n) \neq O(f(n))$ True d) $f(n) \times g(n) \neq O(g(n) \times h(n))$ False

MonalisaCS

https://www.youtube.com/@MonalisaCS

Mathematical Analysis of Nonrecursive Algorithms

- **Analyzing the Time Efficiency of Nonrecursive Algorithms**

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation. (As a rule, it is located in the innermost loop.)
3. Check whether the number of times the basic operation is executed depends only on the size of an input. It also depends on some property, worst-case, average-case, and, best-case efficiencies .
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.
5. Using standard formulas and rules of sum , establish its order of growth.

- $\sum_{i=l}^{u} 1 = u - l + 1$ where $l \le u$ are lower and upper integer limits
- $\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i$, $\sum_{i=l}^{u}(a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$
- $\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1+2+3\ldots n = \frac{n(n+1)}{2}$, $\sum_{i=1}^{n} i^2 = 1^2+2^2+3^2\ldots n^2 = \frac{n(n+1)(2n+1)}{6}$
- **Ex 1:** Finding the value of the largest element in a list of *n* numbers.
- Algorithm: *MaxElement (A[0…n-1])*
- *maxval← A[0]*
- *for i ←1 to n-1 do*
- *if A[i]>maxval*
- *maxval ← A[i]*
- *return maxval*

MonalisaCS

https://monalisacs.com/

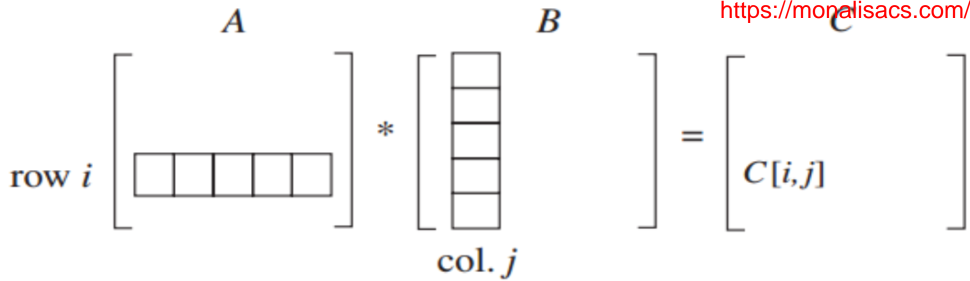
- There are two operations in the loop's body:
- Since the comparison is executed on each repetition of the loop and the assignment is not, we should consider the comparison to be the algorithm's basic operation.
- The number of comparisons will be the same for all arrays of size *n*.
- Let us denote *C(n)* the number of times this comparison is executed.
- $C(n)= \sum_{i=1}^{n-1} 1$=n-1-1+1=n-1
- O(n), Ω(n), Θ(n)
- **Ex 2:** Element uniqueness problem.
- Algorithm: *UniqueElement (A[0…n-1])*
- *for i← 0 to n-2 do*
- *for j ←i+1 to n-1 do*
- *if A[i]=A[j] return false*
- *return true*
- The number of element comparisons depends not only on n but also on whether there are equal elements in the array and, if there are , which array positions they occupy.
- $C_{worst}(n) = \sum_{i=0}^{n-2} \cdot \sum_{j=i+1}^{n-1} 1$
- $= \sum_{i=0}^{n-2}$((n-1)-(i+1)+1) == $\sum_{i=0}^{n-2}$(n-1-i)

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- $= \sum_{i=0}^{n-2}(n-1-i) = \sum_{i=0}^{n-2}(n-1) - \sum_{i=0}^{n-2} i$
- $=(n-1)\sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i$
- $=(n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} = \Theta(n^2)$



- **Ex 3:Matrix Multiplication**
- Given two n×n matrices A and B,C=A.B Scalar (dot) products of the rows of matrix A and the column of matrix B
- Where C[i,j]=A[i,0]B[0,j]+….A[i,k]B[k,j]+….A[i,n-1]B[n-1 ,j] for every pair 0≤i,j ≤n-1.
- Algorithm: *MatrixMultiplication (A[0…n-1,0…n-1],B[0…n-1,0…n-1])*
- *for i← 0 to n-1 do*
- *for j ←0 to n-1 do*
- *C[i,j] ←0.0*
- *for k ←0 to n-1 do*
- *C[i,j] ←C [i,j]+A [i,k]*B [k,j]*
- *return C*
- $C(n) = \sum_{i=0}^{n-1} \cdot \sum_{j=0}^{n-1} \cdot \sum_{k=0}^{n-1} 1$
- $= \sum_{i=0}^{n-1} \cdot \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n \sum_{j=0}^{n-1} 1$
- $= \sum_{i=0}^{n-1} n^2 = n^2 \sum_{i=0}^{n-1} 1 = n^3 = \Theta(n^3)$

MonalisaCS

https://www.youtube.com/@MonalisaCS

Frequency Count Method

https://monalisacs.com/

- **For Loop**
- Ex 1:
- *for (i=1;i≤ n;i++)*
- *{ stmt; }*

i=1(1)	i≤ n (n+1)	stmt (n)	i++ (n)
1	1,2,……n,n+1	1,2,……n	2,……n,n+1

Θ(n)

- Ex 2:
- *for (i=n;i≥ 1;i--)*
- *{ stmt; }*

Iteration	1	2	…	n
i	n	n-1	…	1

Θ(n)

- Ex 3:
- *for (i=1;i≤ n;i+=2)*
- *{ stmt; }*

Iteration	1	2	3	…k	k+1(fail)
i	1	1+2	1+2*2	…1+2*(k-1)	1+2*k

- $1+2*k>n$
- $2k>n-1$
- $k>(n-1)/2$
- $k \cong n$

Θ(n)

- Ex 4:
- *for (i=1;i≤ n;i=i*2)*
- *{ stmt; }*

Iteration	1	2	3	…k	k+1(fail)
i	1	2	2^2	…$2^{(k-1)}$	$2^{(k)}$

- $2^{(k)}>n$
- $\log 2^k > \log n$
- $k>\log n$
- $k \cong \log n$

Θ(logn)

- Ex 5:
- *for (i=n;i ≥ 1;i=i/2)*
- *{ stmt; }*

Iteration	1	2	3	…k	k+1(fail)
i	n	n/2	$n/2^2$	…$n/2^{(k-1)}$	$n/2^k$

- $n/2^k<1 \Rightarrow n<2^k \Rightarrow \log n<k$
- $k \cong \log n$ **Θ(logn)**

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- Ex 6:
- *for (i=1,p=0;p ≤ n;i++)*
- *{ p=p+i; }*

i	1	2	3	…k	k+1(fail)
p	0+1	1+2	1+2+3	…1+2+…k	1+2+..k+1

- 1+2+3…k+1>n
- (k+1)(k+2)/2>n
- $k^2 \cong n \Rightarrow k \cong \sqrt{n}$

$\Theta(\sqrt{n})$

- Ex 7:
- *for (i=1;i*i ≤ n;i++)*
- *{ stmt; }*

i	1	2	…k	k+1(fail)
i^2	1	2^2	$…k^2$	$(k+1)^2$

- $(k+1)^2>n$
- $k^2 \cong n$
- $k \cong \sqrt{n}$

$\Theta(\sqrt{n})$

- Ex 8:
- *for (i=2;i ≤ n;i=i²)*
- *{ stmt; }*

Iteration	1	2	3	…k	k+1(fail)
i	2^{2^0}	2^{2^1}	2^{2^2}	$…2^{2^{k-1}}$	2^{2^k}

- $2^{2^k}>n$
- $\log 2^{2^k}>\log n$
- $2^k>\log n$
- $k>\log \log n$
- $k \cong \log \log n$

$\Theta(\log \log n)$

- Ex 9:
- *for (i=n;i ≥2;i=√i)*
- *{ stmt; }*

Iteration	1	2	3	…k	k+1(fail)
i	n	$\sqrt{n}=n^{1/2}$	$n^{1/2^2}$	$…n^{1/2^{k-1}}$	$n^{1/2^k}$

$\Theta(\log \log n)$

- $n^{1/2^k}<2 \Rightarrow \log n^{1/2^k}<\log 2 \Rightarrow 1/2^k \log n<1$
- $\log n< 2^k \Rightarrow k \cong \log \log n$

- Ex 10:
- *for (i=0;i <n;i++)*
- *{ stmt; }*
- *for (j=0;j <n;j++)*
- *{ stmt; }*

- n+n=2n
- $\Theta(n)$

MonalisaCS
https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- Ex 11:
- *p=0*
- *for (i=1;i <n;i=i*2)*
- *{ p++; }*
- *for (j=1;j <p;j=j*2)*
- *{ stmt; }*

Iteration	**1**	**2**	**3**	**…k**	**k+1(fail)**
i	1	2	2^2	$…2^{(k-1)}$	2^k
p	0	1	2	…k-1	k

Iteration	**1**	**2**	**3**	**…m**	**m+1(fail)**
j	1	2	2^2	$…2^{(m-1)}$	2^m

- $2^k \geq n$
- $k \geq \log n$ **Θ(logn)**
- $k \cong \log n$
- $2^m \geq p$
- $m \geq \log p$
- $m \cong \log p$
- **Θ(log p)=Θ(log logn)**

- Ex 12:
- *for (i=1;i ≤ n;i++)*
- *{ for (j=1;j ≤ n;j++)*
- *{ stmt; }*
- *}*

i	**1**	**2**	**…**	**n**
j	1,2..n (n time)	n time	…	n time

- for each i ,j runs n time
- $n*n=n^2$ **Θ(n²)**
- **n+n+n…n(n time)**
- **$=n*n=n^2$**

- Ex 13:
- *for (i=1;i ≤ n;i++)*
- *{ for (j=n/2;j ≤ n;j+=n/2)*
- *{ stmt; }*
- *}*

i	**1**	**2**	**…**	**n**
j	n/2,n (2 time)	2 time	…	2 time

- for each i ,j runs 2 time n*2=2n **Θ(n)**
- **2+2+2…2(n time)=2*n=2n**

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- Ex 14:
- *for (i=1;i ≤ n;i++)*
- *{ for (j=1;j*j ≤ n;j++)*
- *{ stmt; }*
- *}*

i	1	2	…	n
j2	$1^2,2^2,3^2\ldots(\sqrt{n})^2(\sqrt{n}$ time)	$\sqrt{n}$ time	…	$\sqrt{n}$ time

- for each i ,j runs $\sqrt{n}$ time n*$\sqrt{n}$=n$\sqrt{n}$ **$\Theta(n\sqrt{n})$**
- **$\sqrt{n}+\sqrt{n}\ldots\ldots\sqrt{n}$(n time)=n$\sqrt{n}$**

- Ex 15:
- *for (i=1;i ≤ n;i++)*
- *{ for (j=1;j ≤i;j++)*
- *{ stmt; }*
- *}*

i	1	2	…	n
j	1(1time)	1,2(2 time)	…	1,2..n(n time)

- for each i ,j runs i time
- **1+2+3…n=n(n+1)/2** **$\Theta(n^2)$**

- Ex 16:
- *for (i=1;i ≤ n;i++)*
- *{ for (j=1;j ≤ n;j=j*2)*
- *{ stmt; }*
- *}*

i	1	2	…	n
j	$1,2,2^2$..(log n)	log n time	…	logn time

- for each i ,j runs logn time
- n*log n **$\Theta(n\log n)$**
- **logn+logn +…….logn(n time) =nlogn**

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- Ex 17:
- *for (i=1;i ≤ n;i++)*
- *{ for (j=1;j ≤ n;j=j+i)*
- *{ stmt; } }*

i	1	2	3	… n
j	1,2,3,..n(n time)	1,3,5.. (n/2 time)	1,4,7..(n/3 time)..	1(1 time)

- for each i ,j runs n/i time
- **n+n/2+n/3 +….n/n =n(1+1/2+1/3+…1/n)=nlogn** **Θ(nlogn)**

- Ex 18:
- *for (i=1;i ≤ n;i++)*
- *{ for (j=1;j ≤ n;j++)*
- *{for (k=1;k ≤ n;k++)*
- *{ stmt; } } }*

i	1			2	… n
j	1	2	….n	n time	… n time
k	1,2,3,..n(n time)	n time	n time	n^2 time	… n^2 time

- for each i ,j runs n time , k runs n^2 time.
- **n^2+ n^2 + n^2 +…(n time)= n^2 *n=n^3** , $\sum_{i=1}^{n} \cdot \sum_{j=1}^{n} \cdot \sum_{k=1}^{n} 1$ **Θ(n^3)**

- Ex 19:
- *for (i=1;i ≤ n;i++)*
- *{ for (j=1;j ≤ i;j++)*
- *{for (k=1;k ≤ j;k++)*
- *{ stmt; } } }*

i	1	2		… n
j	1	1	2	… n time
k	1(1 time)	1	1,2(1+2 time)	… (1+2+..n) time

- for each i ,j runs i time ,
- k runs 1+2..j time.
- **1+(1+2)+…..(1+2..n)** $=\sum_{n=1}^{n} \frac{n(n+1)}{2}$ **Θ(n^3)**
- $\sum_{i=1}^{n} \cdot \sum_{j=1}^{i} \cdot \sum_{k=1}^{j} 1 = \sum_{i=1}^{n} \cdot \sum_{j=1}^{i} j = \sum_{i=1}^{n} \frac{i(i+1)}{2}$ = **Θ(n^3)**

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- Ex 20:
- *for (i=1;i ≤ n;i++)*
- *{ for (j=1;j ≤ i²;j++)*
- *{for (k=1;k ≤ n/2;k++)*
- *{ stmt; } } }*

i	1	2	… n
j	1(1 time)	1,2,3,4(4 time)	… n^2 time
k	1,2..n/2(n/2 time)	1,2..n/2(n/2 time)=n/2*4 time	… n/2 *n^2 time

- for each i ,j runs i^2 time , k runs n/2*i^2 time.
- **$n/2+ n/2*2^2 + n/2*3^2 +\ldots n/2*n^2$**
- **$= n/2(1+2^2+3^2+\ldots\ldots n^2)=n/2 *\{n(n+1)(2n+1)/6\}$** **Θ($n^4$)**
- $\sum_{i=1}^{n} \cdot \sum_{j=1}^{i^2} \cdot \sum_{k=1}^{n/2} 1 = \sum_{i=1}^{n} \cdot \sum_{j=1}^{i^2} n/2 = \sum_{i=1}^{n} \frac{n}{2} * i^2$
- =n/2*n(n+1)(2n+1)/6

- **While Loop**
- Ex 1:
- *i=1*
- *while (i ≤ n)*
- *{ stmt;*
- *i++; }*

Iteration	**1**	**2**	**…**	**n**
i	1	2	…	n

- **Θ(n)**

- Ex 2:
- *i=1*
- *while (i ≤ n)*
- *{ stmt;*
- *i=i*2; }*

Iteration	**1**	**2**	**3**	**…k**	**k+1(fail)**
i	1	2	2^2	…$2^{(k-1)}$	$2^{(k)}$

- $2^{(k)}>n \Rightarrow \log 2^k > \log n$
- $k>\log n \Rightarrow k \cong \log n$ **Θ(logn)**

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- Ex 3:
- *i=1;k=1;*
- *while (k ≤ n)*
- *{ stmt;*
- *k=k+i; i++ }*

Iteration	1	2	3	…m	m+1(fail)
k	1	1+1	1+1+2	…1+1+2+..m-1	
i	1	2	3	…m	

- 1+2+3+4+..m>n
- m(m+1)/ 2 > n
- $m^2>n \Rightarrow m \cong \sqrt{n}$
- $\Theta(\sqrt{n})$

- Ex 4:
- *i=1;*
- *while (i ≤ n)*
- *{ k=n;*
- *while (k>0)*
- *{ k=k/2;}*
- *i=i*2; }*

i	1	2	2^2	…2^m
k	n,n/2…1(logn time)	logn time	logn time	…logn time

- $2^m>n$
- m ≅ logn time i runs

- For each i , k runs logn time =logn*logn $\Theta(\log^2 n)$
- **logn+logn+...logn(logn time)=logn*logn**

i	n	n/2	$n/2^2$	…$n/2^k$
j	0	0+n/2	$n/2+n/2^2$	…$n/2+n/2^2+…n/2^k$

- $2^k=n$
- $n/2+n/2^2+…n/2^k$
- $n(1/2+1/2^2+…..1/2^k)$

- GATE CS 2006,Q 15: Consider the following C-program fragment in which i, j and n are integer variables. *for(i = n, j = 0; i > 0; i /= 2, j +=i);*
- Let val(j) denote the value stored in the variable j after termination of the for loop.
- Which one of the following is true?
- **Ans : (C) val(j) = Θ(n)**
- (A) val(j) = $\Theta(\log n)$ (B) val(j) = $\Theta(\sqrt{n})$
- (C) val(j) = $\Theta(n)$ (D) val(j) = $\Theta(n\log n)$

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- GATE CS 2007,Q 51:Consider the following C program segment:

```
int IsPrime (n)
 {
    int i, n;
    for (i=2; i<=sqrt(n);i++)
       if(n%i == 0)
        {printf("Not Prime \n"); return 0;}
      return 1;
}
```

- Let T(n) denote number of times the for loop is executed by the program on input n. Which of the following is TRUE?
- (A) $T(n)=O(\sqrt{n})$ and $T(n)=\Omega(\sqrt{n})$ (B) $T(n)=O(\sqrt{n})$ and $T(n)=\Omega(1)$
- (C) $T(n)=O(n)$ and $T(n)=\Omega(\sqrt{n})$ (D) None of the above
- Worst Case : $T(n)=O(\sqrt{n})$
- Best Case : When n is an even number body of for loop is executed only once(due to "return 0" inside if) $T(n)=\Omega(1)$
- Ans: (B) $T(n)=O(\sqrt{n})$ and $T(n)=\Omega(1)$

MonalisaCS

https://www.youtube.com/@MonalisaCS

Mathematical Analysis of Recursive Algorithms

General Plan for Analyzing the Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.
4. Set up a recurrence relation, with an appropriate initial condition , for the number of times the basic operation is executed.
5. Solve the recurrence or, at least, ascertain the order of growth of its solution.

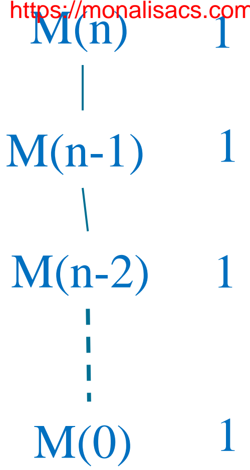
- **Three methods for solving recurrences**
- In the *substitution method*, we guess a bound and then use mathematical induction to prove our guess correct.
- The *recursion-tree method* converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence.
- The *master method* provides bounds for recurrences of particular form

MonalisaCS

- **Ex 1:Factorial function** F(n)=n!, n!=1*2*…..(n-1)*n for n≥1 and 0!=1

https://monalisacs.com/

- Algorithm: *F(n)*
-   *if (n=0) return 1*
-   *else return F(n-1)*n*
- The basic operation of the algorithm is multiplication .
- Number of executions we denote *M(n)*.
- F (n)=F(n-1)*n , for n>0
- M(n)=M(n-1)+1 , for n>0 [M(n-1) to compute F(n-1),1 for multiplication]
- Such equations are called ***recurrence relations*** or, ***recurrences*.**
- To determine a solution uniquely, we need an ***initial condition*** that tells us the value with which the sequence starts .
- if n=0 return 1. M(0)=0 [no multiplication when n=0]
- The recurrence relation and initial condition :
- M(n)=M(n-1)+1 for every n>0, M(0)=0
- ***Method of backward substitutions*.**
- M(n)=M(n-1)+1 [substitute M(n-1)=M(n-2)+1]
- =M(n-2)+1+1=M(n-2)+2 [substitute M(n-2)=M(n-3)+1]
- =M(n-3)+1+2=M(n-3)+3
- ………….
- General formula for the pattern: M(n-i)+i ,
- The initial condition M(0)=0, n-i=0⇒ n=i
- M(0)+n=0+n=n **O(n)**

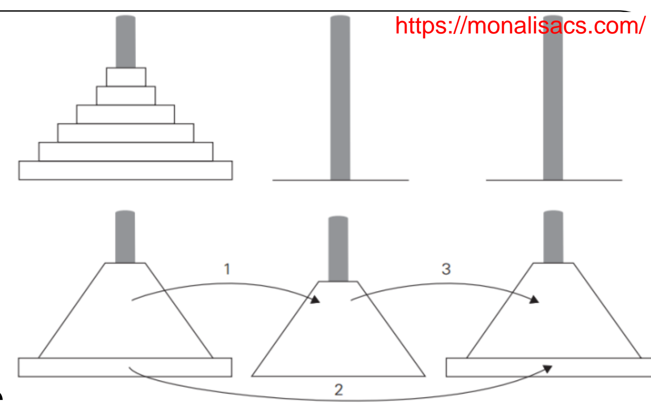


- n-k=0 ⇒k=n
- Height n
- 1+1+..1(n times)
- =n

MonalisaCS
https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

❖ **Ex 2:** the ***Tower of Hanoi*** **puzzle.**



- In this puzzle, we have n disks of different sizes that can slide onto any of three pegs.
- Initially, all the disks are on the first peg in order of size,largest on the bottom & smallest on top.
- The goal is to move all the disks to the third peg, using the second one as an auxiliary, if necessary.
- We can move only one disk at a time, it is forbidden to place a larger disk on top of a smaller one.
- To move $n > 1$ disks from peg 1 to peg 3 (with peg 2 as auxiliary),
- We first move recursively $n - 1$ disks from peg 1 to peg 2 (with peg 3 as auxiliary),
- Then move the largest disk directly from peg 1 to peg 3,
- Finally, move recursively $n - 1$ disks from peg 2 to peg 3 (using peg 1 as auxiliary).
- If $n = 1$, we simply move the single disk directly from the source peg to the destination peg.
- Clearly, the number of moves $M(n)$ depends on n only, Recurrence equation for it:
- $M(n)=M(n-1)+1+M(n-1)$
- With the initial condition $M(1) = 1$,
- Recurrence relation for the number of moves M(n):
- $M(n)=2M(n-1)+1$ for $n>1$
- $M(1) = 1$

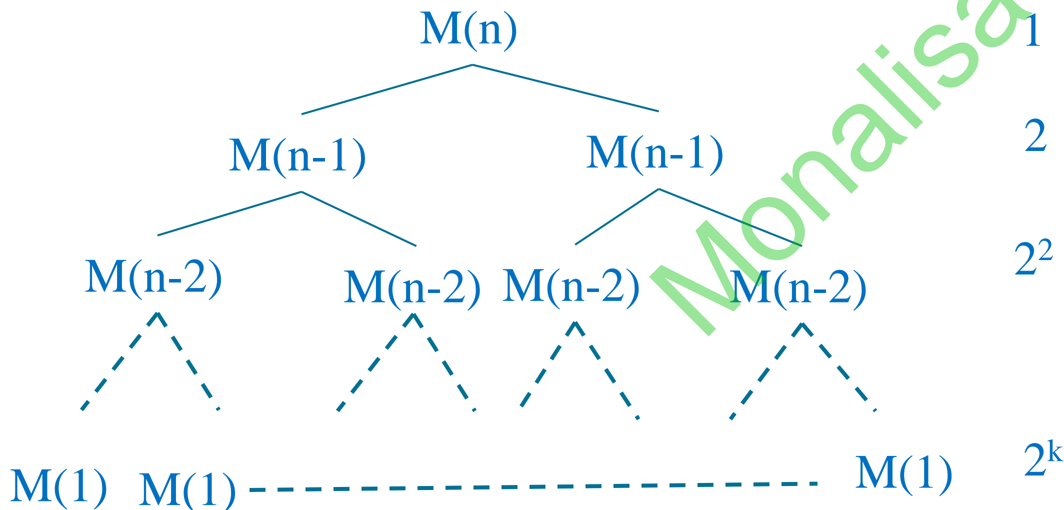
MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

Method of backward substitutions.

- $M(n)=2M(n-1)+1$ [substitute $M(n-1)=2M(n-2)+1$]
- $2\{2M(n-2)+1\}+1=2^2M(n-2)+2+1$ [substitute $M(n-2)=2M(n-3)+1$]
- $2^2\{2M(n-3)+1\}+2+1=2^3M(n-3)+2^2+2+1$
- …..
- $M(n)=2^iM(n-i)+2^{i-1}+\ldots2^2+2+1=2^iM(n-i)+2^i-1$ $[\sum_{i=0}^{i-1}2^i = 2^i - 1]$
- Initial condition $M(1) = 1$, $n-i=1 \Rightarrow i=n-1$
- $M(n)=2^{n-1}M(1)+2^{n-1}-1= 2^{n-1}+2^{n-1}-1= 2^n-1$ **$O(2^n)$**



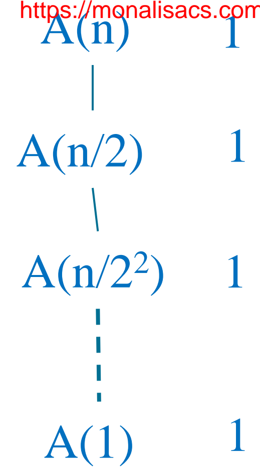
- $1+2+2^2+\ldots\ldots.2^k=2^{k+1}-1$
- Assume $n-k=1 \Rightarrow k=n-1$
- $2^{n-1+1}-1=2^n-1$
- **$O(2^n)$**

MonalisaCS

https://www.youtube.com/@MonalisaCS

Ex 3: The number of binary digits in n's binary representation.

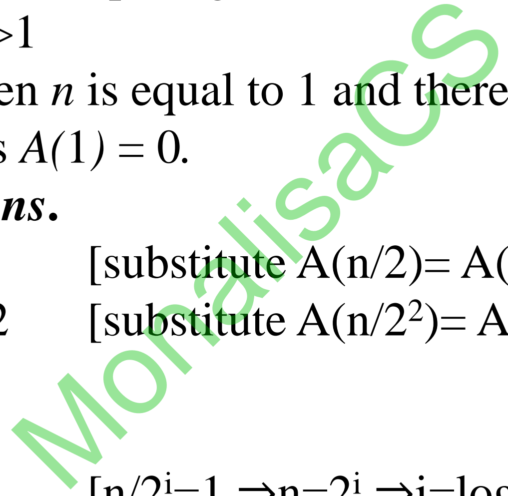
- Algorithm: *BinRec(n)*
- *if (n=1) return 1*
- *else return BinRec(⌊n/2⌋)+1*
- The number of additions made in computing *BinRec*(⌊n/2⌋) is A(⌊n/2⌋)
- $A(n)=A(\lfloor n/2 \rfloor)+1$ for $n>1$
- Since the recursive calls end when *n* is equal to 1 and there are no additions , the initial condition is $A(1) = 0$.



Method of backward substitutions.

- $A(n)=A(\lfloor n/2 \rfloor)+1$ [substitute $A(n/2)= A(n/2^2)+1$]
- $=\{A(\lfloor n/2^2 \rfloor)+1\}+1= A(\lfloor n/2^2 \rfloor)+2$ [substitute $A(n/2^2)= A(n/2^3)+1$]
- $= A(n/2^3)+3$
- ....
- $A(n/2^i)+i$ [$n/2^i=1 \Rightarrow n=2^i \Rightarrow i=\log n$]
- $A(1)+\log n=0+\log n=\log n$
- $A(n)=$ **$O(\log_2 n)$**

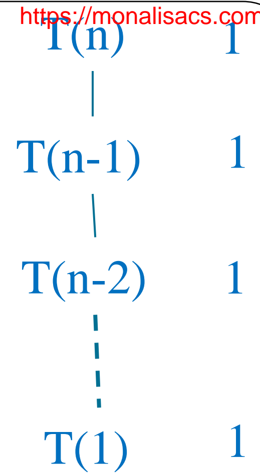
- $n/2^k=1 \Rightarrow k=\log_2 n$
- Height $\log_2 n$
- $1+1+..1(\log_2 n$ times$)$
- $=\log_2 n$

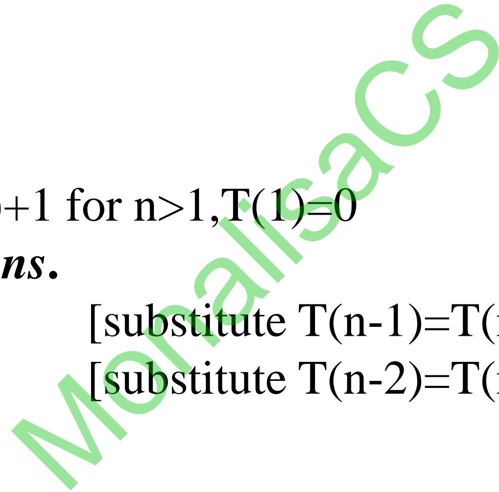

MonalisaCS

Ex 4a : (Decrease-and-Conquer Method)

https://monalisacs.com/

- Algorithm: What(*n*)
- *{ if (n=1) return 1*
- *else*
- *{What(n-1)*
- *call B( );*
- *} }*
- Let B( ) =O(1)
- Recurrence relation T(n)=T(n-1)+1 for n>1,T(1)=0
- ***Method of backward substitutions.***
- T(n)=T(n-1)+1 [substitute T(n-1)=T(n-2)+1]
- ={T(n-2)+1}+1=T(n-2)+2 [substitute T(n-2)=T(n-3)+1]
- ={T(n-3)+1}+2=T(n-3)+3
- ………….
- General formula for the pattern: T(n-i)+i ,
- The initial condition T(1)=0, n-i=1⇒ i=n-1
- T(1)+(n-1)=0+n-1=n-1
- **O(n), Θ(n), Ω(n)**





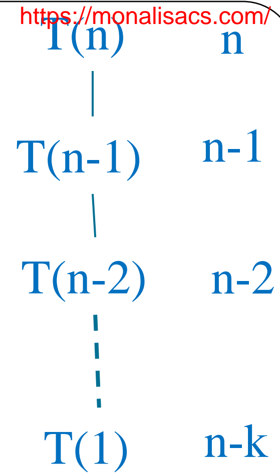
- n-k=1 ⇒k=n-1
- Height n-1
- 1+1+…..1(n-1 times)
- (n-1) ≅n

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

Ex 4b:

- Let B( ) =O(n)
- Recurrence relation T(n)=T(n-1)+n for n>1,T(1)=0
- ***Method of backward substitutions.***
- T(n)=T(n-1)+n [substitute T(n-1)=T(n-2)+(n-1)]
- =T(n-2)+(n-1)+n [substitute T(n-2)=T(n-3)+(n-2)]
- =T(n-3)+(n-2)+(n-1)+n
- ………….
- General formula for the pattern: T(n-i)+(n-i+1)+…..(n-1)+n
- The initial condition T(1)=0, n-i=1⇒ i=n-1
- T(1)+{2+3+….(n-1)+n} ≅ 1+2+3+…n
- $=\frac{n(n+1)}{2}$ ≅n^2
- **O(n^2), Θ(n^2), Ω(n^2)**



- n-k=1 ⇒k=n-1
- n+n-1+…..+2+1
- $\frac{n(n+1)}{2}$ ≅n^2

Ex 4c:

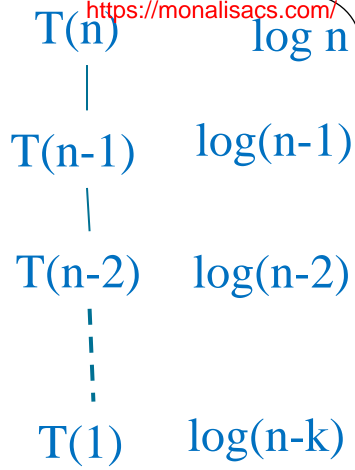
- Let B( ) =O(log n)
- Recurrence relation T(n)=T(n-1)+log n for n>1,T(1)=0
- ***Method of backward substitutions.***
- T(n)=T(n-1)+log n [substitute T(n-1)=T(n-2)+log (n-1)]
- =T(n-2)+log (n-1)+log n [substitute T(n-2)=T(n-3)+log (n-2)]

MonalisaCS

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- $=T(n-3)+\log(n-2)+\log(n-1)+\log n$
- ………….
- General formula for the pattern: $T(n-i)+\log(n-i+1)+\ldots\log(n-1)+\log n$
- The initial condition $T(1)=0$, $n-i=1 \Rightarrow i=n-1$
- $T(1)+\{\log 2+\log 3+\ldots\log(n-1)+\log n\}$
- $\cong \log 1+\log 2+\log 3+\ldots\log(n-1)+\log n=\log(1*2*3\ldots*n)=\log n!$
- $\log n!$ $O(\log n^n)=$ **$O(n \log n)$**
- $\log n+\log(n-1)+\ldots\log(n-k)$ $[n-k=1 \Rightarrow k=n-1]$
- $\log n+\log(n-1)+\ldots\log 1$
- $\log(n*(n-1)*\ldots3*2*1)=\log n!$
- $\log n!=O(\log n^n)=$ **$O(n \log n)$**



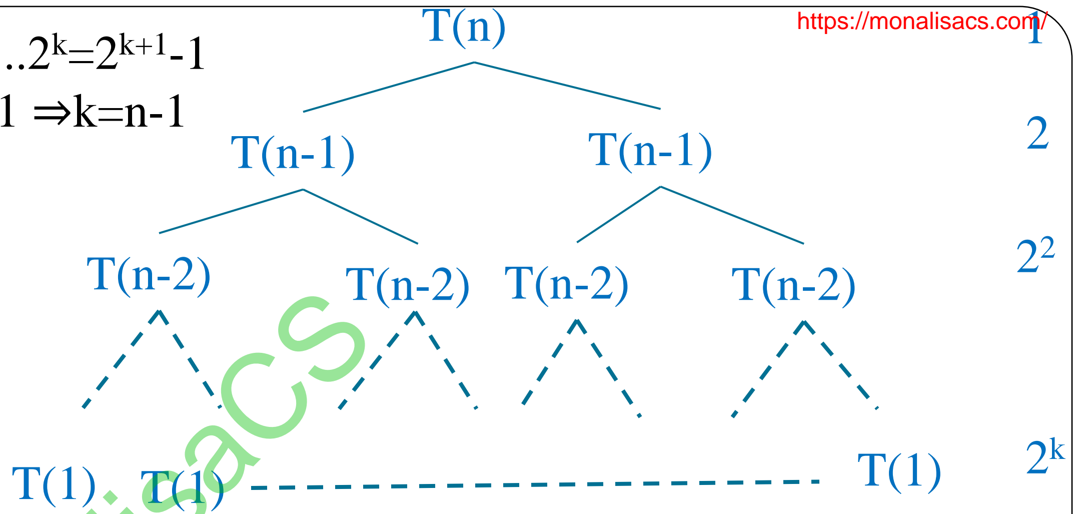
❖ Ex 4d:

- Recurrence relation $T(n)=n*T(n-1)$ for $n>1, T(1)=1$
- ***Method of backward substitutions.***
- $T(n)=n*T(n-1)$ [substitute $T(n-1)=(n-1)*T(n-2)$]
- $=n*(n-1)*T(n-2)$ [substitute $T(n-2)=(n-2)*T(n-3)$]
- $= n*(n-1)*(n-2)*T(n-3)$
- ………………
- General formula for the pattern: $n*(n-1)*(n-2)*\ldots(n-k+1)T(n-k)$ $[n-k=1]$
- $n*n-1*\ldots\ldots2*1=n!$ $O(n^n)$

MonalisaCS

https://www.youtube.com/@MonalisaCS

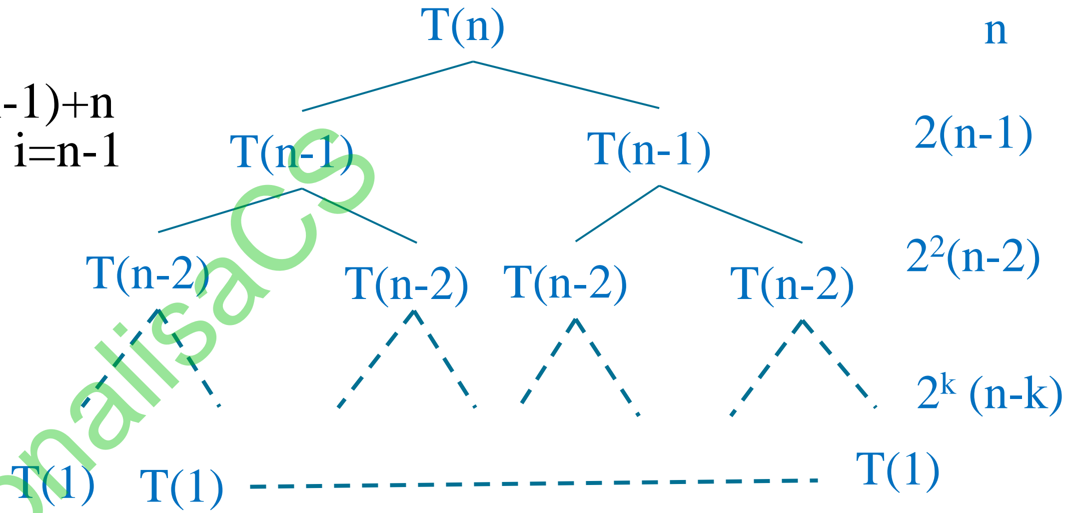
- **Ex 5a:**
- Algorithm: Do(n)
- { *if (n=1) return 1*
- *else*
- *{Do(n-1)*
- *Do(n-1)*
- *call B( );*
- *} }*
- $1+2+2^2+\ldots\ldots..2^k=2^{k+1}-1$
- Assume n-k=1 ⇒k=n-1
- $2^{n-1+1}-1=2^n-1$
- $\mathbf{O(2^n)}$



- Recurrence relation
- T(n)=2T(n-1)+1 for n>1,T(1)=1
- ***Method of backward substitutions.***
- T(n)=2T(n-1)+1 [substitute T(n-1)=2T(n-2)+1]
- $=2^*\{2T(n-2)+1\}+1=2^2*T(n-2)+2+1$ [substitute T(n-2)=2T(n-3)+1]
- $=2^2*\{2T(n-3)+1\}+2+1=2^3T(n-3)+2^2+2+1$
- …………..
- General formula for the pattern: $2^iT(n-i)+2^{i-1}+\ldots.2^2+2+1$
- $2^iT(n-i)+2^i-1$
- The initial condition T(1)=1, n-i=1⇒ i=n-1
- $2^{n-1}T(1)+2^{n-1}-1=2^n-1$ $\mathbf{O(2^n)}$
- **Ex 5b:** T(n)=2T(n-1)+n for n>1,T(1)=1
- ***Method of backward substitutions.***

https://monalisacs.com/
MonalisaCS
https://www.youtube.com/@MonalisaCS

- $T(n)=2T(n-1)+n$ [substitute $T(n-1)=2T(n-2)+(n-1)$]
- $=2*\{2T(n-2)+(n-1)\}+n=2^2*T(n-2)+2(n-1)+n$ [substitute $T(n-2)=2T(n-3)+(n-2)$]
- $=2^2*\{2T(n-3)+(n-2)\}+2(n-1)+n=2^3T(n-3)+2^2(n-2)+2(n-1)+n$
- ………….
- General formula for the pattern:
- $2^iT(n-i)+2^{i-1}(n-i+1)+\dots 2^2(n-2)+2(n-1)+n$
- The initial condition $T(1)=1$, $n-i=1 \Rightarrow i=n-1$
- $2^{n-1}T(1)+2^{n-2}*2+\dots..2(n-1)+n$
- $2^{n-1}*1+2^{n-2}*2+\dots..2(n-1)+n$
- $2^{n-1}*(1+2^{-1}*2+2^{-2}*3+\dots.+2^{-n+1}*(n))$
- $\cong n*2^{n-1}$ **$O(n2^n)$**
- $n+2(n-1)+2^2(n-2)+\dots.2^k(n-k)$
- Assume $n-k=1 \Rightarrow k=n-1$
- $n+2(n-1)+2^2(n-2)+\dots.2^{n-1}(1)$
- **$O(n2^n)$**



❖ **Ex 6a:**

- Algorithm: $A(n)$
- *{ if (n=2) return 1*
- *else*
- *return* $A(\sqrt{n})$
- *B( ) }*

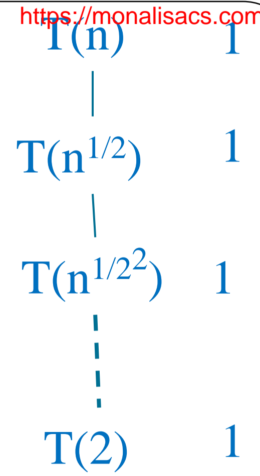
- Recurrence relation
- $T(n)=T(\sqrt{n})+1$ for $n>2$, $T(2)=1$

https://monalisacs.com/
MonalisaCS
https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- ***Method of backward substitutions.***
- $T(n)=T(\sqrt{n})+1$ [substitute $T(n^{1/2})=T(n^{1/2^2})+1$]
- $= T(n^{1/2^2})+1+1= T(n^{1/2^2})+2$ [substitute $T(n^{1/2^2})= T(n^{1/2^3})+1$]
- $= T(n^{1/2^3})+3$
- …………
- General formula for the pattern: $T(n^{1/2^i})+i$
- The initial condition $T(2)=1$, $n^{1/2^i}=2$
- $\Rightarrow 1/2^i \log n=\log 2=1$
- $2^i=\log n \Rightarrow i=\log\log n$
- $T(2)+\log\log n=1+\log\log n$
- **O(log log n)**

- $n^{1/2^k}=2$
- $1/2^k \log n=\log 2 =1$
- $2^k=\log n$
- $k=\log\log n$
- Hieght $=\log\log n$
- $1+1+\ldots.1$(log logn times)
- log log n



❖ **Ex 6b:** $T(n)=2T(\sqrt{n})+1$

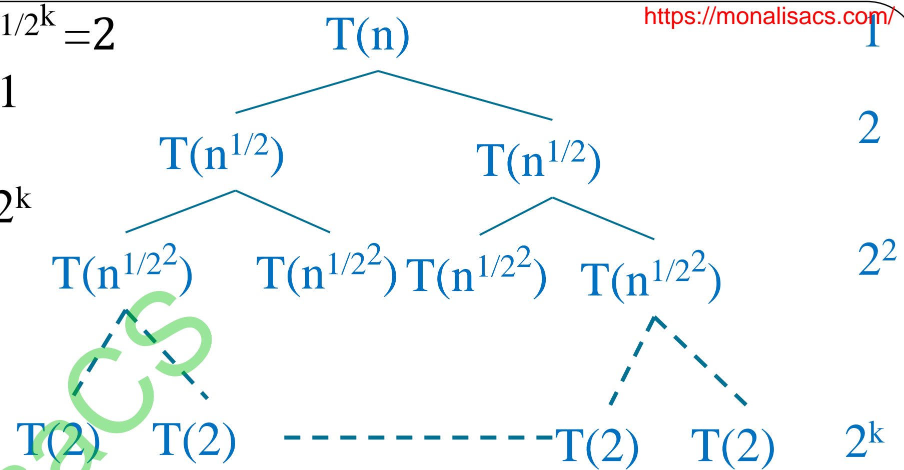
- ***Method of backward substitutions.***
- $T(n)=2T(\sqrt{n})+1$ [substitute $T(n^{1/2})=2T(n^{1/2^2})+1$]
- $= 2*\{2T(n^{1/2^2})+1\}+1=2^2T(n^{1/2^2})+2+1$ [substitute $T(n^{1/2^2})= 2T(n^{1/2^3})+1$]
- $= 2^2\{2T(n^{1/2^3})+1\}+2+1= 2^3T(n^{1/2^3})+2^2+2+1$
- …………
- General formula for the pattern: $2^iT(n^{1/2^i})+2^{i-1}+\ldots.2^2+2+1= 2^iT(n^{1/2^i})+2^i-1$
- The initial condition $T(2)=1$, $n^{1/2^i}=2$

Monalisacs

https://www.youtube.com/@MonalisaCS

- $1/2^i \log n = \log 2 = 1$
- $2^i = \log n$
- $2^i T(n^{1/2^i}) + 2^i - 1$
- $\log n * T(2) + \log n - 1$
- $\log n * 1 + \log n - 1$
- $= 2\log n - 1$
- **O(log n)**

- $T(n^{1/2^k}) = T(2) \Rightarrow n^{1/2^k} = 2$
- $1/2^k \log n = \log 2 = 1$
- $2^k = \log n$
- $1+2+2^2+2^3+\ldots\ldots 2^k$
- $= 2^{k+1} - 1$
- $2 \log n - 1$
- **O(log n)**



❖ **Ex 6c:** $T(n) = \sqrt{n}\, T(\sqrt{n}) + n$

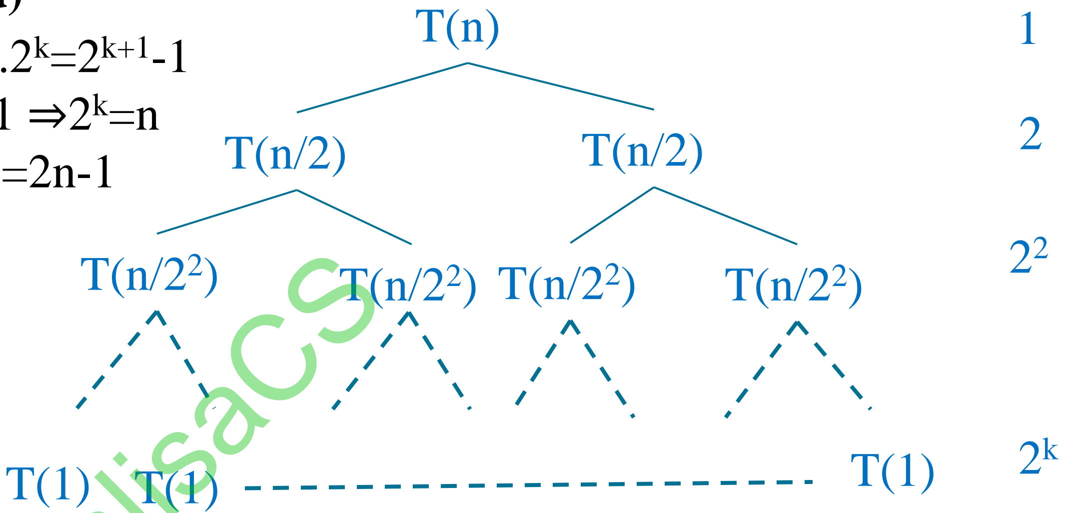
- ***Method of backward substitutions.***
- $T(n) = \sqrt{n}\, T(\sqrt{n}) + n$ [substitute $T(n^{1/2}) = n^{1/2^2} T(n^{1/2^2}) + n^{1/2}$]
- $= n^{1/2} * \{n^{1/2^2} T(n^{1/2^2}) + n^{1/2}\} + n = n^{3/4} T(n^{1/2^2}) + 2n$
- $= n^{1-1/2^2} T(n^{1/2^2}) + 2n$
- ………….
- General formula for the pattern: $n^{1-1/2^i} T(n^{1/2^i}) + i*n = n/n^{1/2^i} T(n^{1/2^i}) + i*n$
- The initial condition $T(2)=1$, $n^{1/2^i} = 2$
- $2^i = \log n \Rightarrow i = \log \log n$
- $n/2*T(2) + \log \log n * n = n/2 + n \log \log n$
- **O(n log log n)**

https://monalisacs.com/
MonalisaCS
https://www.youtube.com/@MonalisaCS

Ex 7a: (Divide-and-Conquer Method)

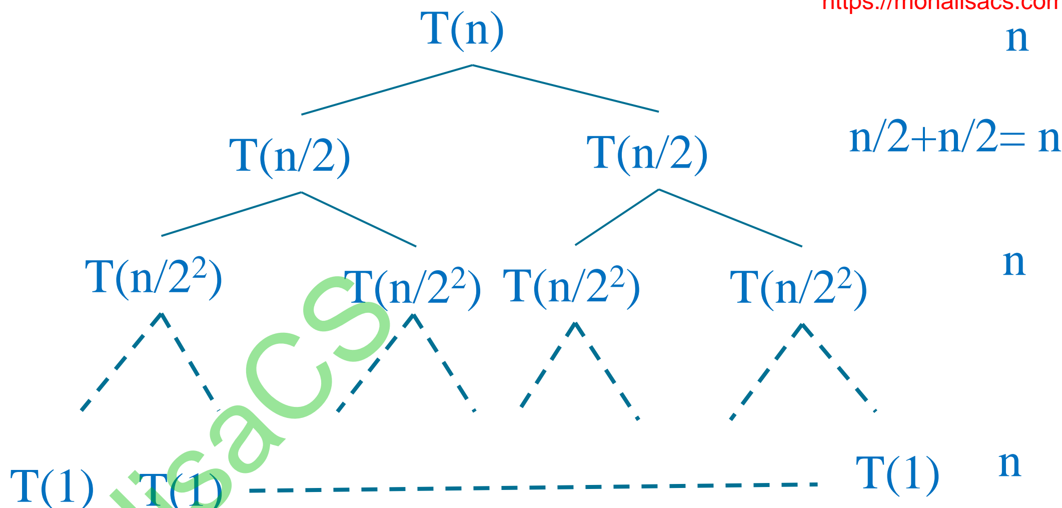
- Algorithm: Foo(n)
- *{ if (n=1) return 1*
- *else*
- *Foo(n/2)*
- *Foo(n/2)*
- *B( ) }*
- a :B=O(1) ,b :B=O(n)
- Recurrence relation
- T(n)=T(n/2)+T(n/2)+1 for n>1,T(1)=1
- T(n)=2T(n/2)+1
- ***Method of backward substitutions.***
- T(n)=2T(n/2)+1 [substitute $T(n/2)= 2T(n/2^2)+ 1$]
- $= 2\{2T(n/2^2)+ 1\}+1= 2^2T(n/2^2)+ 2+1$ [substitute $T(n/2^2)= 2T(n/2^3)+ 1$]
- $= 2^3T(n/2^3)+ 2^2+2+1$
- ………….
- General formula for the pattern: $2^iT(n/2^i)+ 2^{i-1}+\ldots.2^2+2+1 = 2^iT(n/2^i)+ 2^i-1$
- The initial condition T(1)=1, $n/2^i =1 \Rightarrow 2^i=n$
- nT(1)+ n-1 =2n-1 **O(n)**

- $1+2+2^2+\ldots\ldots..2^k=2^{k+1}-1$
- Assume $n/2^k=1 \Rightarrow 2^k=n$
- $2^{k+1}-1 =2*2^k-1=2n-1$
- **O(n)**



MonalisaCS

- **Ex 7b:**
- Recurrence relation
- $T(n)=2T(n/2)+n$ for $n>1, T(1)=1$
- ***Method of backward substitutions.***
- $T(n)=2T(n/2)+n$
- [substitute $T(n/2)= 2T(n/2^2)+ n/2$]
- $= 2\{2T(n/2^2)+ n/2\}+n= 2^2T(n/2^2)+ 2n$
- [substitute $T(n/2^2)= 2T(n/2^3)+ n/2^2$]
- $= 2^3T(n/2^3)+ 3n$
- ………….
- General formula for the pattern: $2^iT(n/2^i)+i*n$
- The initial condition $T(1)=1$, $n/2^i=1 \Rightarrow 2^i=n \Rightarrow i=\log n$
- $nT(1)+ n*\log n = n+n \log n$
- **O(n log n)**
- Assume $n/2^k=1 \Rightarrow 2^k=n \Rightarrow k=\log n$
- Height $=\log n$
- n+n+n+…..n(log n times) =n log n
- **O(n log n)**



https://monalisacs.com/
MonalisaCS
https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

The master method for solving recurrences

- There are different types of master method according to recurrence form
- **Master Theorem for Decrease & conquer Recurrence**
- $T(n)=aT(n-b)+f(n)$ [$a>0, b>0, T(d)=c$ Initial condition, $n>d$]
- *Case 1:if* $a<1$, $T(n)$ *is* $O(f(n))$
- *Case 2:if* $a=1$, $T(n)$ *is* $O(n*f(n))$
- *Case 3:if* $a>1$, $T(n)$ *is* $O(a^{n/b}*f(n))$
- **Ex 4a:** $T(n)=T(n-1)+1$
- $a=1, b=1, f(n)=1$, Case 2, $T(n)=O(n*f(n))=O(n*1)=O(n)$
- **Ex 4b:** $T(n)=T(n-1)+n$
- $a=1, b=1, f(n)=n$, Case 2, $T(n)=O(n*f(n))=O(n*n)=O(n^2)$
- **Ex 4c:** $T(n)=T(n-1)+\log n$
- $a=1, b=1, f(n)=\log n$, Case 2, $T(n)=O(n*f(n))=O(n*\log n)=O(n \log n)$
- **Ex 4d:** $T(n)=n*T(n-1)$
- $a=n, b=1$, Case 3, $T(n)=O(a^{n/b}*f(n))=O(n^n)$
- **Ex 5a:** $T(n)=2T(n-1)+1$
- $a=2, b=1, f(n)=1$, Case 3, $T(n)=O(a^{n/b}*f(n))=O(2^{n/1}*1)=O(2^n)$
- **Ex 5b:** $T(n)=2T(n-1)+n$
- $a=2, b=1, f(n)=n$, Case 3, $T(n)=O(a^{n/b}*f(n))=O(2^{n/1}*n)=O(n2^n)$
- Ex 1: $T(n)=1/2T(n-1)+\log n$
- $a=1/2, b=1, f(n)=\log n$, Case 1, $T(n)=O(f(n))=O(\log n)$

MonalisaCS

https://www.youtube.com/@MonalisaCS

Master Theorem for Divide & conquer Recurrence

- $T(n) = aT(n/b) + f(n)$ [a>0 ,b>1 ,f (n) is a +ve function]
- *Case 1: if $f(n)=O(n^{\log_b a-\epsilon})$, for some $\epsilon > 0$ then $T(n)$ is $\Theta(n^{\log_b a})$*
- *Case 2: if $f(n)=\Theta(n^{\log_b a} * \log^k n)$, for some k*

a) *$k \geq 0$ then $T(n)$ is $\Theta(n^{\log_b a} * \log^{k+1} n)$*

b) *$k=-1$ then $T(n)$ is $\Theta(n^{\log_b a} * \log \log n)$*

c) *$k<-1$ then $T(n)$ is $\Theta(n^{\log_b a})$*

1. If $f(n) = O(n^{\log_b a-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

- *Case 3: if $f(n)=\Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.*

- Ex 7a: $T(n)=2T(n/2)+1$
- a=2 ,b=2 ,f(n)=1
- $n^{\log_b a} = n^{\log_2 2} = n$
- $1<n$ or $f(n)=O(n^{\log_b a})$ case 1
- $\Theta(n^{\log_b a}) = \mathbf{\Theta(n)}$
- Ex 7b: $T(n)=2T(n/2)+n$
- a=2 ,b=2 ,f(n)=n
- $n^{\log_b a} = n^{\log_2 2} = n$
- $n=n$ or $f(n)= \Theta(n^{\log_b a} \log^0 n)$ case 2
- $\Theta(n^{\log_b a} * \log^{k+1} n) = \mathbf{\Theta(n \log n)}$

- Ex 1: $T(n)=2T(n/2)+n\log n$
- a=2 ,b=2 ,f(n)=nlogn
- $n^{\log_b a} = n^{\log_2 2} = n$
- nlogn vs n ,$f(n)=\Theta(n^{\log_b a} * \log^1 n)$ case 2
- $k=1 \geq 0$ then $\Theta(n^{\log_b a} * \log^{k+1} n) = \mathbf{\Theta(n \log^2 n)}$
- -------------------
- $n\log n > n$
- or $f(n)= \Omega(n^{\log_b a+\epsilon})$ case 3
- $2f(n/2)=2 * n/2 * \log(n/2)$
- $=n \log(n/2) \leq n\log n$
- $\Theta(f(n)) = \mathbf{\Theta(n \log n)}$

MonalisaCS

- Ex 2: $T(n)=4T(n/2)+n$
- $a=4$, $b=2$, $f(n)=n$
- $n^{\log_b a} = n^{\log_2 4} = n^2$
- $n<n^2$ or $f(n)=O(n^{\log_b a})$ case 1
- $\Theta(n^{\log_b a})$ $=\mathbf{\Theta(n^2)}$
- Ex 3: $T(n)=8T(n/2)+n^2$
- $a=8$, $b=2$, $f(n)=n^2$
- $n^{\log_b a} = n^{\log_2 8} = n^3$
- $n^2<n^3$ or $f(n)=O(n^{\log_b a})$ case 1
- $\Theta(n^{\log_b a})$ $=\mathbf{\Theta(n^3)}$
- Ex 4: $T(n)=T(2n/3)+1$
- $a=1$, $b=3/2$, $f(n)=1$
- $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$
- $1=1$ or $f(n)=\Theta(n^{\log_b a} \log^0 n)$ case 2
- $\Theta(n^{\log_b a} * \log^{k+1} n) = \mathbf{\Theta(\log n)}$
- Ex 5: $T(n)=2T(n/2)+n/\log n$
- $a=2$, $b=2$, $f(n)=n\log^{-1} n$
- $n^{\log_b a} = n^{\log_2 2} = n$
- $n\log^{-1} n$ vs n, $f(n)=\Theta(n^{\log_b a} * \log^{-1} n)$ case 2
- $k=-1$ then $\Theta(n^{\log_b a} * \log\log n) = \mathbf{\Theta(n \log\log n)}$
- Ex 6: $T(n)=T(n/3)+n$
- $a=1$, $b=3$, $f(n)=n$
- $n^{\log_b a} = n^{\log_3 1} = n^0 = 1$
- $n>n^0$ or $f(n)=\Omega(n^{\log_b a+\epsilon})$ case 3
- $af(n/b) \le cf(n)$
- $1f(n/3)=n/3$
- $=n/3 \le n$
- $\Theta(f(n)) = \mathbf{\Theta(n)}$
- Ex 7: $T(n)=9T(n/3)+n^{2.5}$
- $a=9$, $b=3$, $f(n)=n^{2.5}$
- $n^{\log_b a} = n^{\log_3 9} = n^2$
- $n^{2.5}>n^2$ or $f(n)=\Omega(n^{\log_b a+\epsilon})$ case 3
- $af(n/b) \le cf(n)$
- $9f(n/3)=9(n/3)^{2.5}=9*\frac{n^{2.5}}{3^2\sqrt{3}}=\frac{n^{2.5}}{\sqrt{3}} \le n^{2.5}$
- $\Theta(f(n)) = \mathbf{\Theta(n^{2.5})}$

https://monalisacs.com/
MonalisaCS
https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- **Variation of Master Theorem for Divide & conquer Recurrence**
- $T(n) = T(\alpha n) + T((1-\alpha)n) + cn$ $[0<\alpha<1, c>0]$ then $T(n)=\Theta(n \log n)$
- Ex 1: $T(n)=T(n/2)+T(n/2)+n = 2T(n/2)+n$
- $\alpha=1/2$, $c=1$
- $\Theta(n \log n)$
- Ex 2: $T(n)=2T(n/2)+5n$
- $\alpha=1/2$, $c=5$
- $\Theta(n \log n)$
- Ex 3: $T(n)=T(n/3)+T(2n/3)+2n$
- $\alpha=1/3$, $c=2$
- $\Theta(n \log n)$
- Ex 4: $T(n)=T(2n/5)+T(3n/5)+n$
- $\alpha=2/5$, $c=1$
- $\Theta(n \log n)$

MonalisaCS

Space complexity

- Space complexity is a function describing the amount of memory (space) an algorithm takes to execute the algorithm.
- Non recursive algorithm :Extra space required to execute algorithm called space complexity .
- **Ex 1:** Algorithm: *MaxElement (A[0…n-1])*
- *maxval← A[0]*
- *for i ←1 to n-1 do*
- *if A[i]>maxval*
- *maxval ← A[i]*
- *return maxval*
- Input :A[0…n-1] .
- Extra space require to run algorithm is ‘i’ size . its a constant
- Space complexity O(1) ,Time complexity O(n).
- **Ex 2:** Algorithm*(A[0…n-1])*
- *int i, B[n];*
- *for i ←1 to n-1 do*
- *B[i] ← A[i]*
- *return B[i];*
- Input :A[0…n-1]
- Extra space require to run algorithm are ‘i & B[n]’ size . 1+n
- Space complexity O(n) ,Time complexity O(n).

MonalisaCS

https://monalisacs.com/

- Recursive algorithm :number of stack space required in control stack/run time stack for push/pop of activation record called space complexity.
- **Ex 1:**Algorithm: *F(n)*
- *if (n=0) return 1*
- *else return F(n-1)*n*
- Stack space required =n.
- Space complexity O(n) ,
- T(n)=T(n-1)+1 for n>0,T(0)=0 ,Time complexity O(n).
- **Ex 2:**Algorithm: Do*(n)*
- *{ if (n=1) return 1*
- *else {Do(n-1)*
- *Do(n-1)*
- *call B( );*
- *} }*
- Stack space required =n.
- Space complexity O(n) ,
- T(n)=2T(n-1)+1 for n>1,T(1)=1 ,Time complexity $O(2^n)$.

https://www.youtube.com/@MonalisaCS

https://monalisacs.com/

- Let T(n)=Time complexity ,S(n)=Space complexity ,T(n)< = > S(n) depend on algorithm .
- An in-place algorithm is an algorithm that does not need an extra space and produces an output in the same memory that contains the data by transforming the input 'in-place'. However, a small constant extra space used for variables is allowed.
- But may require a small non-constant extra space for its operation. Usually, this space is O(log n).
- If $S(n) \leq \log n$ in-place algorithm .
- Bubble sort, Selection Sort, Insertion Sort, Heapsort are in-place algorithm .
- If $S(n) > \log n$, not in-place algorithm .
- Merge Sort. merge sort requires O(n) extra space.So it is not in-place algorithm .
- GATE CS 2009,Q 35:The running time of an algorithm is represented by the following recurrence relation: $T(n)=n \quad n \leq 3$
- $T(n)=T(n/3)+cn \quad$ otherwise
- Which one of the following represents the time complexity of the algorithm?
- (A)$\Theta(n)$ (B)$\Theta(n\log n)$ (C)$\Theta(n^2)$ (D)$\Theta(n^2\log n)$
- $a=1, b=3, f(n)=cn$
- $n^{\log_b a} = n^{\log_3 1} = n^0 = 1$
- $cn>1$ or $f(n)= \Omega(n^{\log_b a+\epsilon})$ case 3 ,$af(n/b) \leq cf(n)$
- $1f(n/3)=n/3 \Rightarrow n/3 \leq n$
- $\Theta(f(n))=\boldsymbol{\Theta(n)}$

MonalisaCS

https://www.youtube.com/@MonalisaCS