Objective

(i) what is an algorithm

(ii) how to analyse an algorithm

(iii) study of different design techniques

Pre requisite

(i) C

(ii) Data structure (how it stored, accessed & manipulated data)

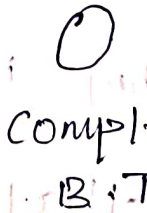
linear → stack, queue, array, linked list

Non-linear - Tree, graph.

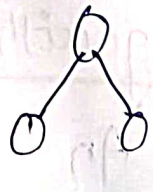
* In linear data structure traversal occure is only in one unique way i.e top to bottom, front to rear etc.

* In Non-linear data structure traversal occurs in multiple ways i.e in, pre, post, BFS, DFS etc.

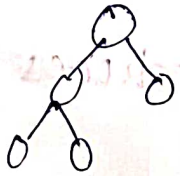
Tree → Binary Tree
Tree → N-ary Tree



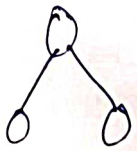
complete B.T



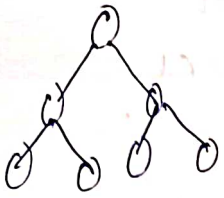
C. B. T



Almost



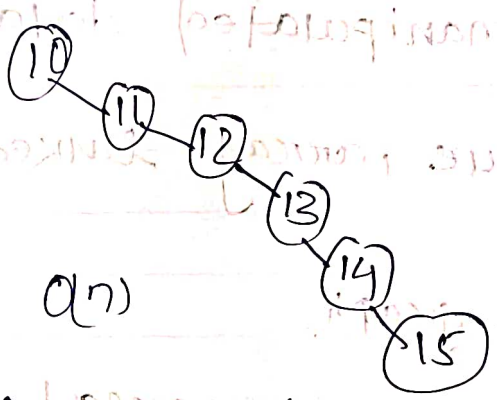
height $\log(3)$



$\log(7)$ { must be a complete binary tree } ~~long~~ $\log(n)$ height of tree)

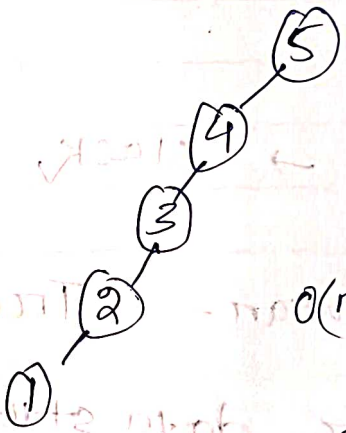
BST

10, 11, 12, 13, 14, 15 , 5, 4, 3, 2, 1



O(n)

(Right skewed)



O(n)

left skewed

Objective

(I) Algorithm

(II) Analysis

(III) Design

Algorithm

→ An algorithm is any well defined computational procedure that takes an input / set of inputs & produces an output or set of output.

→ It is a computational procedure which transforms the input to its corresponding output.

Representation of an algorithm :

(I) flow chart

(II) pseudocode → uses the programming element

(I) variable
(II) constant
(III) operators
(IV) Branching constructs (IF, IF else)
(V) loop constructs.

Q write an algorithm to add two integers.

A ADD (x, y)

(I) input x, y
(II) compute result = x + y
(III) print result
(IV) Exit

Properties of an algorithm

(I) Finiteness [It must stop with finite number of steps]

(II) Each step must be well defined (It must not be ambigious)

(III) correctness : An algorithm is said to be correct if it stops/ Halts with correct output for every instance of the problem.

$n = 10$
$y = 20$
one instance of input.

$n = 15$
$y = 30$
another instance of input.

$[-2^{31}$ to 0 to $2^{31}-1]$ range of 'int' datatype

* Every algorithm followed correctness proof.

Design methods

(I) Divide & conquer

(II) Dynamic progreamming approach

(III) Greedy approach

(IV) Recurrences

Analysis

Sorting (Ascending)

Input: a sequence of 'n' numbers

$\langle a_1, a_2, a_3 \ldots \ldots a_n \rangle$

output: A permutation of the given sequence

$\langle a_1', a_2' \ldots \ldots a_n' \rangle$

such that $a_1' \leq a_2' \leq \ldots \ldots \leq a_n'$

Instance

$\langle 15, 10, 25, 3, 4, 2 \rangle$ input

$\langle 2, 3, 4, 10, 15, 25 \rangle$

a_1' a_n'

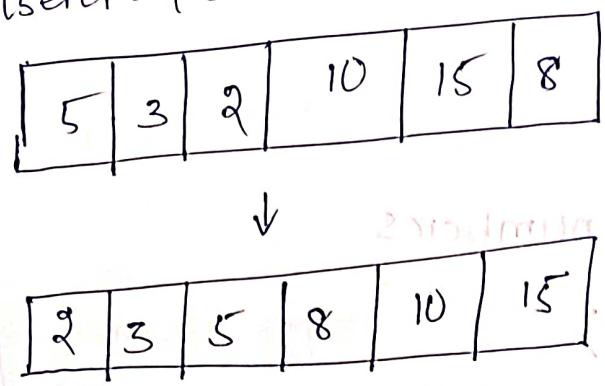
Insertion sort

→ Insertion sort is a good algorithm for a smaller input sequence

→ Each element to be sorted is taken as the key element

→ It works like most people arrange the playing cards in their left hand.

Insertion sort



Assumption

(I) Let length(A) is no of elements present in A.

(II) Array index starts from 1

Algorithm for insertion sort :

Insertion sort (A)

(I) for j = 2 to length(A)

(II) key = A[j]

(III) i = j-1

(IV) while (i >= 1 && A[i] > key)

(V) A[i+1] = A[i]

(VI) i = i-1

(VII) A[i+1] = key.

(VIII) Exit

Analysis of Algorithm

→ The analysis of algorithms are done by considering the resources the algorithm uses.

* Resources algorithm uses:

(i) computer memory
(ii) Hardware
(iii) computational time (most often used)

major factors

computation time / Running time

(i) Time complexity
(ii) space complexity (how much memory it takes)

Running time of insertion sort

→ The time taken by insertion sort for 4 inputs varies from 1000 inputs.

→ Even if the input size / length of the sequence is same still the insertion sort running time varies depending how nearly the elements are sorted.

→ In general the time of insertion sort depends on the size of input.

19-01-2024

Analysis

Running time

→ It is the sum of execution time of each statement of the algorithm.

cost: computation time of that statement

let c_i is the time taken by ith statement of algorithm for each execution.

eg: ADD(x,y)

	cost	time	Total
(I) input x,y	c_1	1	$c_1+c_2+c_3+c_4$
(II) compute result = x+y	c_2	1	
(III) print result	c_3	1	
(IV) exit	c_4	1	

Print(n)	cost	time	$T(n)$ =
(I) input n	c_1	1	$c_1+c_2(n+1)+c_3n+c_4$
(II) for i = 1 to n	c_2	n+1	$= c_1+c_2+c_4+(c_2+c_3)n$
(III) print(i)	c_3	n	$= c_{11}+c_{12}n$
(IV) exit	c_4	1	$= O(n)$

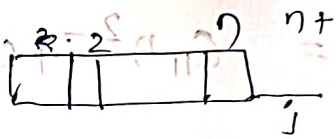
$[c_{11} = c_1+c_2+c_4,\ c_{12} = c_2+c_3]$

'n'- size of input

→ The time complexity of an algorithm depends on the input size.

computation time for insertion sort

	cost	time
(I) for j = 2 to n	c_1	n
(II)	c_2	n-1
(III)	c_3	n-1
(IV)	c_4	$\sum_{j=2}^{n} t_j$
(V)	c_5	$\sum_{j=2}^{n} (t_j - 1)$
(VI)	c_6	11
(VII)	c_7	n-1
(VIII)	c_8	1



n+1 -1 = n

3	10	15	16	17	2

j	i/tj
2	2
3	3
4	4
n	n

let t_j is no of times line 4 executes for each value of j.

$$\sum_{j=2}^{n} t_j = 2+3+4+\cdots+n$$

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

$$\sum_{j=2}^{n} (t_j - 1)$$

$$= 1+2+3+\cdots+n-1$$

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

$$= \frac{n(n-1)}{2}$$

$T(n) = c_1 n + c_2 n + c_2 + c_3 n$

$T(n) = c_1 n + c_2 n - c_2 + c_3 n - c_3 + c_4 \frac{n^2}{2} + c_4 \frac{n}{2} - c_4$

$+ c_5 \frac{n^2}{2} - c_5 \frac{n}{2} + c_6 \frac{n^2}{2} - c_6 \frac{n}{2} + c_7 n - c_7 + c_8$

$= c_{11} n^2 + c_{12} n + c_{13}$

$= O(n^2)$

$\frac{n(n-1)}{2} c_5 -$

$= \frac{n^2}{2} c_5 + \frac{n}{2} c_5$

$c_{11} = \frac{c_4 + c_5 + c_6}{2}$

$c_{12} = c_1 + c_2 + c_3 + c_7 + c_4 - c_5 + \frac{c_6}{2}$

$c_{13} = -c_2 - c_3 - c_4 - c_7 + c_8$

→ When we analyse an algorithm different cases of the algorithm are considered.

(I) worst case

(II) average case

(III) Best case.

→ In insertion sort worst case means the array is in reverse order.

6	5	4	3	1

(II) Average case : means it is nearly sorted (mixed case)

15	91	3	4	100

(III) Best case: The array is already sorted

1	2	3	4	5

→ In insertion sort the ~~worst case~~ ~~to~~ avg time complexity of worst case & average case is $O(n^2)$

24-01-2024

Best case Time complexity of insertion sort

Algo	cost	Time
(I) for $j = 2$ to n.	c_1	n
(II) key = A[j]	c_2	$n-1$
(III) $i = j-1$	c_3	$n-1$
(IV) while ($i >= 1$ && A[i] > key)	c_4	$n-1$
(V) A[i+1] = A[i]	c_5	0

(VI) $i = i-1$ c_6 0

(VII) $A[i+1] = key$ c_7 $n-1$

(VIII) Exit c_8 1

Total time

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5 \times 0 + c_6 \times 0 + c_7 \times (n-1) + c_8$$

$$= (c_1 + c_2 + c_3 + c_4 + c_7) n + (-c_2 - c_3 - c_4 - c_7 + c_8)$$

$$= c_{11} n + c_{12}$$

highest powere is '1' so the orderers is

$O(n)$

Note:

* Insertion sort is the only sorting algorithm whose best case time is $O(n)$ or linear in number of inputs.

Problem

Let there be an array of size 1 million. There are two computers C_1 who executes 1 billion instruction per second. C_2 who executes 10 million instruction per second. C_1 runs an algorithm whose running time is $O(n^2)$ & C_2 runs an algorithm whose running time is $O(n \log n)$. compute how much time each machine will take to sort the given array.

Ans

Array size(n) $= 10^6$

C_1 speed $= 10^9$ instruction/sec

C_2 speed $= 10 \times 10^6$ instruction/sec

$= 10^7$ instruction/sec

C_1

The algorithm takes $= O((10^6)^2)$

$= 10^{12}$

Time to sort the array $= \frac{10^{12}}{10^9}$

$= 10^3$ sec

C_2

The algo C_2 runs takes $= O(n \log_2 n)$

$= O(10^6 \log_2 10^6)$

$= \log_2 10^6 = \log_2 2^{20} = 20$

$= 10^6 \times 20$

$2^{10} = 1024 \approx 10^3$

$10^6 \approx 2^{20}$

$\log_2 2^3 = 3 \log_2 2$
$= 3$

$= 10^6 \times 10^1 \times 2$

Time taken by $c_2 = \frac{10^6 \times 20}{10^7}$

$= 2$ sec

Note

Though the speed of the computer is very high, still its execution time is more because it runs an algorithm whose running time is more than the algorithm which runs in c_2.

→ so it is essential to develop efficient algorithm to solve a problem than the speed of the computer.

Bubble Sort

29-01-2024

Array | 91 | 16 | 5 | 31 | 41 | 13 |

* 'n' no of elements
no of comparision & swapping is $(n-1)$

| 16 | 5 | 31 | 41 | 13 | 91 |

$n-(n-1)$

→ 1st iteration.

| 16 | 5 | 31 | 13 | 41 | 91 |

→ 2nd iteration.

5	16	13	31	41	91

→ 3rd iteration
(n-3)

5	13	16	31	41	91

→ 4th iteration
(n-4)

5	13	16	31	41	91

→ 5th iteration
(n-5)

$n-(n-1)$
$= n-n+1$
$= 1$

$[(n-n-1)$ comparision$]$ *

*$(n-1)^{th}$ iteration the array is sorted.

Algorithm

Bubble sort(A)

(I) for i = 1 to n-1

(II) for j = 1 to n-i

(III) If (A[j] > A[j+1])

(IV) swap (A[j], A[j+1])

(V) Exit

1. for i = 1 to n-1
2. for j = 1 to n-i
3. if (A[j] > A[j+1])
4. { swap (A[j], A[j+1])
5. End.

Running time $= (n-1)+(n-2)+(n-3)+\cdots+1$

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

$= c_1 n^2 + c_2 n$

Biggest power is '2'

So, time complexity = $O(n^2)$ for all the 3 cases (worst, avg, best)

Linear Search

* Search

→ It is the process of finding a given element from a given input sequence.

→ Input: A sequence of elements & an element which is to be searched.

→ output: position where the elements present message whether the element is present or not

Two searching Algorithm

(I) linear search

(II) Binary search

10	21	3	4	8	91

searching element

91

Algorithm for linear search

linear search (A, ele)

(I) flag = 0

(II) for i = 1 to n

(iii) if (ele == A[i])

(iv) flag = 1

(v) break

(vi) if (flag == 1)

(vii) print ("element is present")

(viii) else

(ix) print ("element is not present")

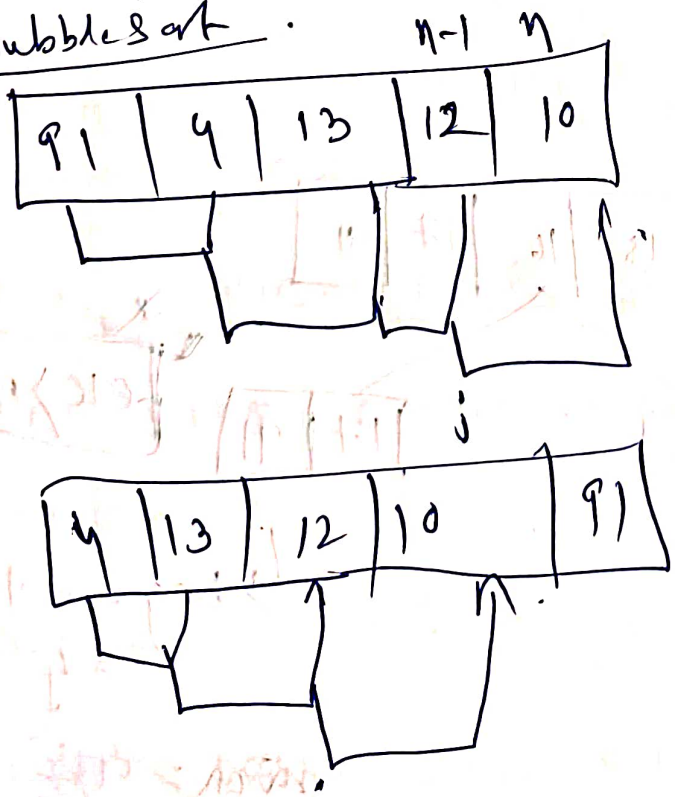
(x) Exit

Running time =

Ele at first position ← Best case: Element is present at loc 1 O(1)

Avg case:
worst case: { O(n) → linear time search algorithm

Bubble sort.



j	i
j = n-1	1
n-2	2
1. n-2	2. 2

= (n-1) + (n-2) + ... 1 ∴ i = n-1

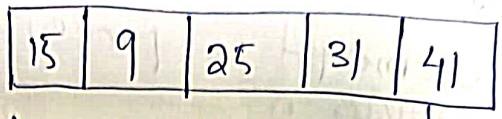
Binary search

31-01-2024

Pre condition:-

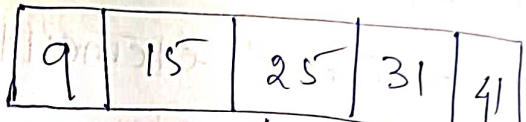
→ Array must be sorted

Input: Any array in any order



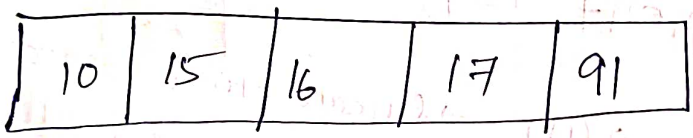
↓ sort

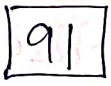
* must sort the array



↓ Binary search

Output: element present / Absent





element to be searched

begin = 1

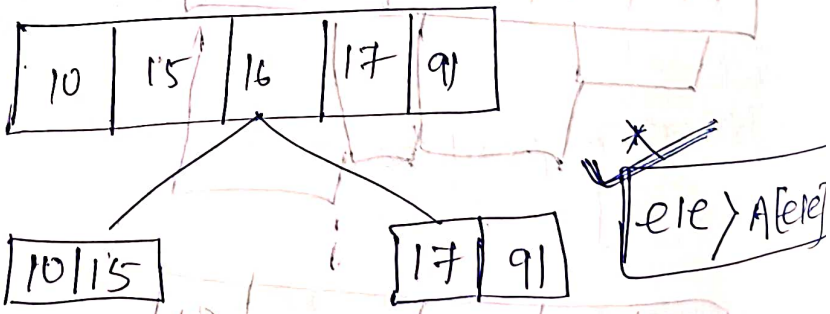
end = 5

$mid = \frac{beg + end}{2}$

= 5

ele = A[mid]

91 ≠ 16



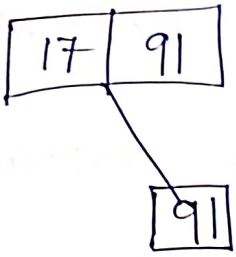
↳ beg = mid + 1

= 4

mid = 4+5/2

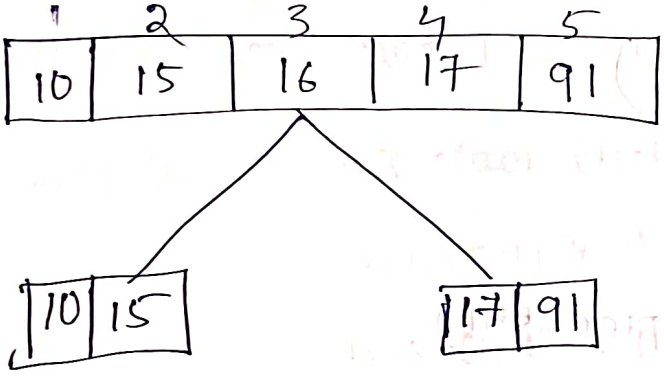
mid = 4

ele ≠ A [mid]



begin = mid + 1 = 5
end = 5
mid = 5
ele = A [mid]

eg:



10
ele

beg-1
end = mid - 1
mid = $\frac{1+2}{2}$
= 1
ele = A[mid]
search element

begin = 1
end = 5
mid = 3
ele ≠ A [mid]
ele > or < A [mid]
ele < A [mid] so search will be continues on the left array.

Time complexity of B.S

→ Best case of the element to be search is the middle element of the input array = O(1)

* height of B.S.T having 'n' no of elements $O(\log_2 n)$

→ worest case time of BS = $O(\log_2 n)$

Algorithm for B.S

Binary Search (A, beg, end, ele)

(1) flag = 0

(2) while (beg <= end)

(3) { mid = (beg + end)/2

(4) if (ele == A[mid])

(5) flag = 1

(6) print (element is present)

(7) break

(8) else if (ele > A[mid])

(9) beg = mid + 1

(10) else

(11) end = mid - 1

(12) } // end of while

(13) if (flag == 0)

(14) print ("element is absent")

(15) exit

02-02-24

Recursive ~~function~~ Binary search :-

* ~~function~~ call itself, to solve the same problem till the ~~base condition~~

* A function call itself to solve the same problem with smaller size ouput, till the base condition arise

* Base condition means a point where the problem can not be divided farther.

eg: fact(n) = n x fact(n-1)

= n*(n-1) x fact(n-2)

= n*(n-1) x (n-2) * fact(n-3)

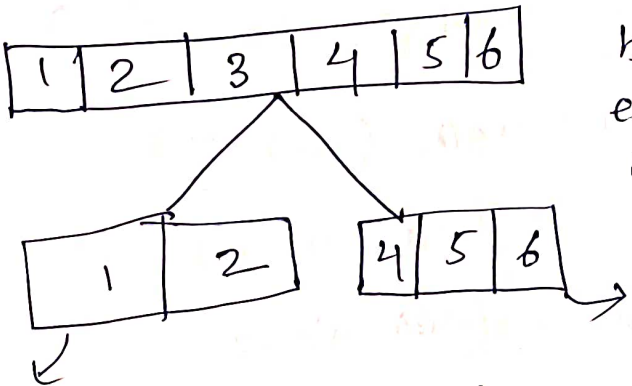
⋮

= n*(n-1)*.... * fact(1) Base condition.

eg: sum(n)

= n + sum(n-1)

= n + (n-1) + ... + sum(1) Base condition



beg = 1
end = 6
mid = 3

1
search element
A[mid] ≠ ele

BS(A, mid+1, end, ele)

BS(A, beg, mid-1, ele)

Recursive BS (A, beg, end, ele)

(I) Flag = 0

(II) If (beg <= end)

(III) mid = (beg + end) / 2

(IV) If (ele == A[mid])

(V) flag = 1

(VI) print (element is present)

(VII) return

(VIII) else If (ele < A[mid])

(IX) return recursive BS(A, beg, mid-1, ele)

(X) else

(XI) return Recursive BS (A, mid+1, end, elem).

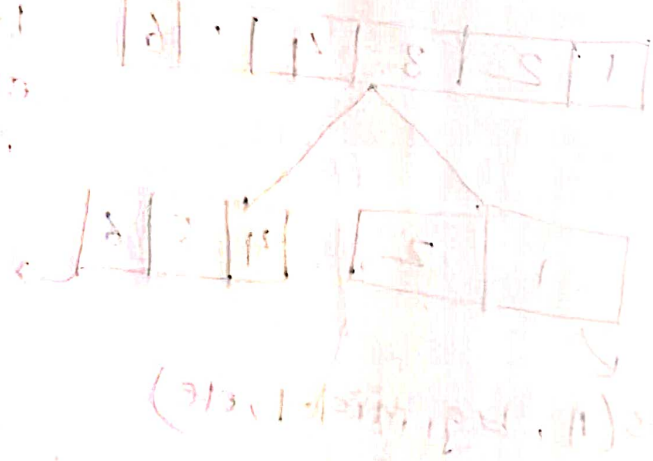
(XII) }

(XIII) return flag

(XIV) Exit

Time complexity

O(log n)



Growth of function

$T(n) = O(n^2)$

↓

input size

$(n \geq 1)$

→ The orders of growth of running time of an algorithm gives a characterisation of efficiency of an algorithm when n is large. $(n \to \infty)$

→ The asymptotic efficiency $(n \to \infty)$ of an algorithm is with the large input size the orders of growth of running time

→ The asymptotic efficiency of an algorithm is given by asymptotic notation.

Asymptotic Notations

→ There are 5 standard asymptotic notation used to represent the running time of an algorithm.

(I) Θ (Theta) - notation

(II) O (Big-Oh) notation

(III) Ω (omega) notation

(IV) o (little-oh) notation

(V) ω (little omega) notation

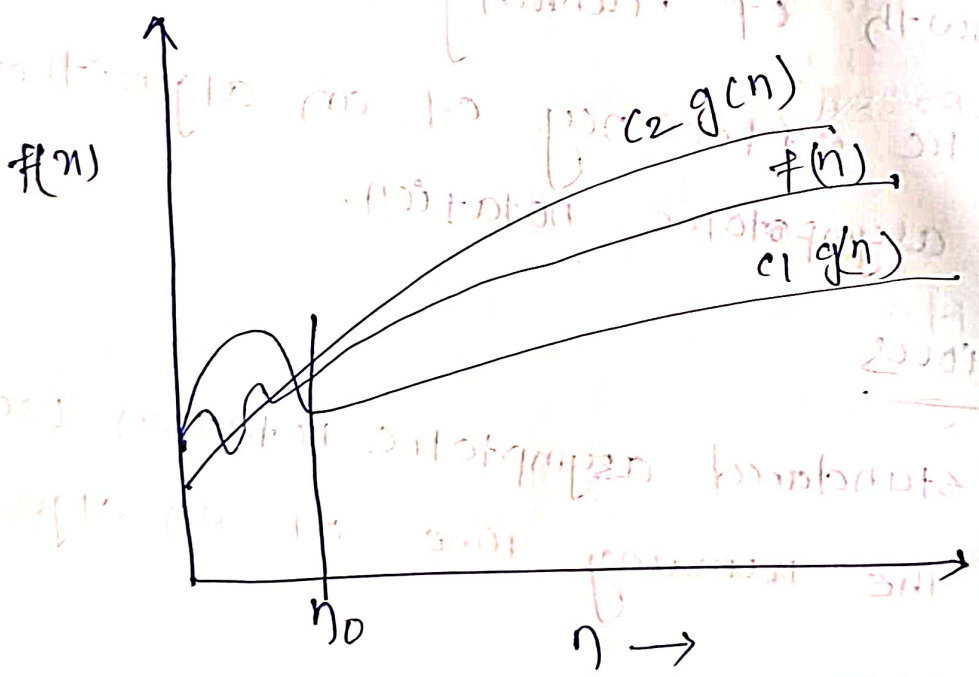
Θ - Notation

For a given function $g(n)$, we denote a set of functions $\Theta(g(n)) = \{ f(n)$: There exist positive constants c_1, c_2, n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for $\forall n \geq n_0 \}$

$f(n)$ & $g(n)$ are non-negative function.

c_1, c_2, n_0 are strictly +ve

(n = input size)



* $g(n)$ is asymptotic tight bound of $f(n)$.

eg: $3n+4 = \Theta(n)$ $\quad c_1 n \leq 3n+4 \leq c_2 n$

02-02-2024

Q Prove that

$$n^2+5n+6 = \Theta(n^2)$$

Ans

$$f(n) = n^2+5n+6$$
$$g(n) = n^2$$

Prove that $c_1 g(n) \leq n^2+5n+6 \leq c_2 g(n) : \forall n \geq n_0$ —— (1)

Step 1: Find c_1, c_2 & n_0 such that eq^n (1) satisfies for $\forall n \geq n_0$

eq^n (1) can be written as $c_1 n^2 \leq n^2+5n+6 \leq c_2 n^2$

$$\Rightarrow c_1 \leq 1+\frac{5}{n}+\frac{6}{n^2} \leq c_2 \quad \text{—— (2)}$$

Left inequality

$c_1 \leq x$

Right inequality

$x \leq c_2$

n	x
1	12
2	5
3	3.3
⋮	⋮
∞	1

$c_1 = 1$

$c_2 = 12$

$1 \leq 1+\frac{5}{n}+\frac{6}{n^2} \leq 12$

$n_0 = 1$

for $c_1 = 1$, $c_2 = 12$ & $n_0 = 1$, The eq^n (1) holds for all $n \geq n_0 = 1$. Hence $n^2 + 5n + 6 = O(n^2)$ proved.

Q Prove that $\frac{n^2}{2} - 3n = O(n^2)$

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

$g(n) = n^2$

To prove :- $c_1 n^2 \leq \frac{n^2}{2} - 3n \leq c_2 n^2$ —— (1)

find c_1, c_2 and n_0 such that eq^n (1) holds for all $n \geq n_0$

eq^n (1) can be written as $c_1 n^2 \leq \frac{n^2}{2} - 3n \leq c_2 n^2$

$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$ —— (2)

$\frac{1}{2} - \frac{3}{n}$ = X

n	X
1	-2.5
2	-1
3	-1/2
4	-1/4
5	-1/10
6	0
7	1/14

n	X
8	1/8
⋮	⋮
∞	1/2

7 → n_0

for $c_1 = \frac{1}{14}$ & $c_2 = -1/2$ and $n_0 = 7$, eq (1) holds

Hence $\frac{n^2}{2} - 3n = \Theta(n^2)$

Q Prove that $an^2+bn+c = \Theta(n^2)$

$f(n) = an^2+bn+c$

$g(n) = n^2$

To prove $c_1 n^2 \leq an^2+bn+c \leq c_2 n^2$ — (1)

find c_1, c_2 and n_0 such that eq(1) holds for all $n \geq n_0$

eq(1) can be written as $c_1 \leq a + \frac{b}{n} + \frac{c}{n^2} \leq c_2$ — (2)

x

n	x
1	a+b+c
2	$(a+\frac{b}{2}+\frac{c}{4})$
3	$(a+\frac{b}{3}+\frac{c}{9})$
⋮	⋮
∞	1

$n_0 = 1$

$c_1 = a$

$c_2 = a+b+c$

for $c_1 = a$ & $c_2 = a+b+c$ & $n_0 = 1$, eq (1) holds

Hence $an^2+bn+c = \Theta(n^2)$

Ans

05-02-2024

Q Prove $3n+4 = \Theta(n)$

$f(n) = 3n+4$

$g(n) = n$

Proof c_1, c_2, and n_0 such that

$$c_1 n \leq 3n+4 \leq c_2 n$$

$$\Rightarrow c_1 \leq 3 + \underbrace{\frac{4}{n}}_{x} \leq c_2$$

n	x
1	7
2	5
3	.
.	.
.	.
∞	3

$c_1 = 3$

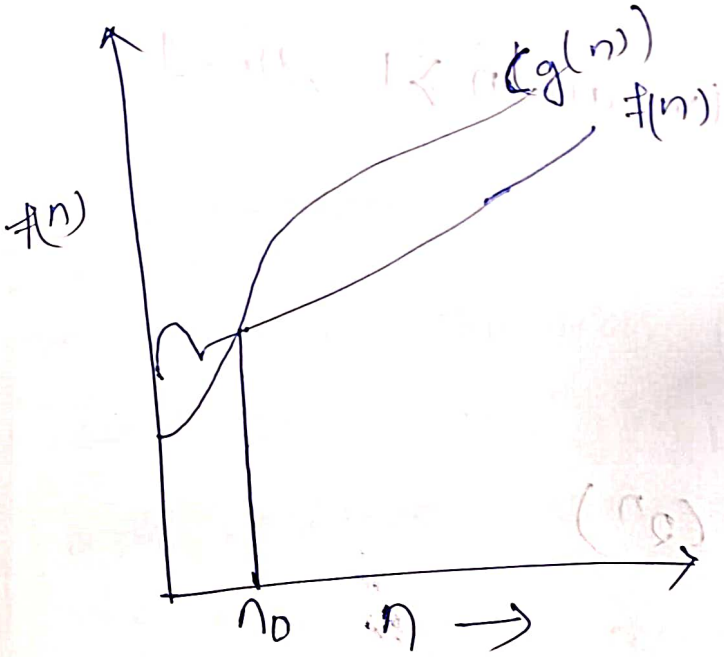
$c_2 = 7$

$n_0 = 1$ (an $n \geq 1$)

For $c_1 = 3$, $c_2 = 7$ & $n_0 = 1$ $3n+4 = \Theta(n)$ (proved)

Big-oh (O) Notation

→ For a given function $g(n)$, we denote a set of functions $O(g(n)) = \{ f(n) :$ There exist +ve constants c & n_0 such that $0 \leq f(n) \leq c g(n)$ $\forall$ $n \geq n_0 \}$.



Q prove that $3n^2+4n+5 = O(n^2)$

$f(n) = 3n^2+4n+5$

$g(n) = O(n^2)$ n^2

we have to proof,

By definition O-notation

$0 \leq f(n) \leq cg(n)$

$0 \leq 3n^2+4n+5 \leq cn^2$ ——— (1)

find c & n_0 such that eq^n (1) holds for all

$n \geq n_0$

$3n^2+4n+5 \leq cn^2$

$3+\frac{4}{n}+\frac{5}{n^2} \leq c$

let $x = 3+\frac{4}{n}+\frac{5}{n^2}$

x	n
1	12
2	6.2
3	4.5
∞	3

$C = 12 \quad n_0 = 1$

$3 + \frac{4}{n} + \frac{5}{n^2} \leq C$ for all $n \geq 1$, $n_0 = 1$

for $C = 12$, $n_0 = 1$

$8n^2 + 4n + 5 = O(n^2)$

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

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

$g(n) = 2^n$

we have to proof

$0 \leq f(n) \leq c\,g(n)$

$0 \leq 2^{n+1} \leq C \cdot 2^n$ ——— ①

$2 \leq C$

~~find~~ for ~~$C=2, n_0=1$ such that eq^n ① holds for all~~

for $C = 2$ & $n_0 = 1$, $2^{n+1} = O(2^n)$ proved)

Q Prove that $n^2 + 5n + 6 = O(n^2)$

$f(n) = n^2 + 5n + 6$

$g(n) = n^2$

$0 \leq f(n) \leq c\,g(n)$

$0 \leq n^2 + 5n + 6 \leq cn^2$

$0 \leq 1 + \frac{5}{n} + \frac{6}{n^2} \leq C$

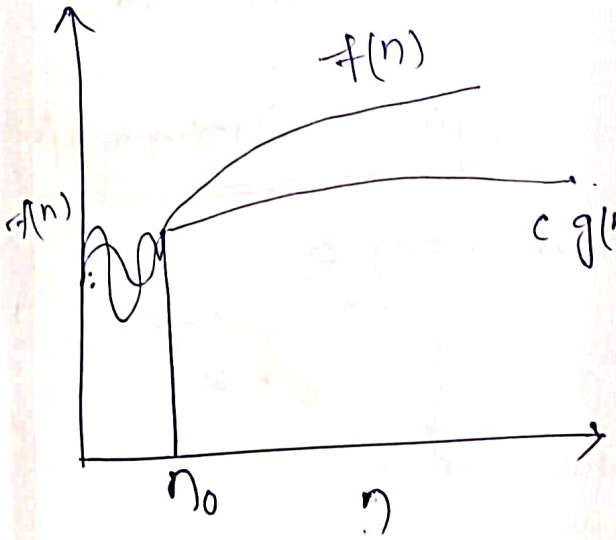
n	x
1	12
2	5
3	3.3
⋮	
∞	1

$c = 12$

$n_0 = 1$, thus $n^2 + 5n + 6 = O(n^2)$ (proved)

Ω - Notation

for a given function $g(n)$, we denote a 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 $n \geq n_0 \}$



→ Θ - defines tight bound of $f(n)$

→ O - defines upper bound of $f(n)$

→ Ω - defines lower bound of $f(n)$

Prove than $n^2 + 5n + 6 = \Omega(n^2)$

Ans $f(n) : n^2 + 5n + 6$

$g(n) = n^2$

$0 \leq cn^2 \leq n^2 + 5n + 6$

$0 \leq c \leq 1 + \frac{5}{n} + \frac{6}{n^2}$ — (1)

n	X
1	12
2	5
3	3.3
⋮	⋮
∞	1

$\Rightarrow c \leq 1 + \frac{5}{n} + \frac{6}{n^2}$

$c=1 \leq 1 + \frac{5}{n} + \frac{6}{n^2}, \forall n \geq 1, \quad n_0 = 1$

$c=1, n_0=1, n^2+5n+6 = \Omega(n^2)$ (proved)

Theorem

→ For given functions $f(n)$ & $g(n)$,

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

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

Q Prove that $2^{2n} = O(2^n)$

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

$g(n) = 2^n$

$0 \leq f(n) \leq c\,g(n)$

$0 \leq 2^{2n} \leq c\,2^n$

$0 \leq 2^n \leq c$ which is possible only for $n=1$

↳ not possible for $n \geq 1$

So $2^{2n} = O(2^n)$ ✗ not possible

Asymptotic Notation in Eq?

07-02-2024

$5n^2+6n+4 = 5n^2+f(n)$

$= 5n^2+\Theta(n)$ where $f(n) = 6n+4$

$= \Theta(n)$

$5n^2+6n+4 = \Theta(n^2)$

$\Rightarrow 5n^2 + f(n) = \Theta(n^2)$

$\Rightarrow 5n^2 + \Theta(n) = \Theta(n^2)$

→ Asymptotic notations can present both in the left or right sight of a equation.

Asymptotic Tight

$4n^3 = \Theta(n^3)$, meaning is $4n^3 \leq cn^3$ for some +ve c & n_0

Asymptotic tight

$4n^2 = O(n^3)$ Asymptotically not tight

$4n^2 \leq cn^3$

$f(n)$ $g(n)$

$4n^2 = \Theta(n^2)$ - Asymptotically tight (at some point $f(n) = g(n)$)

$4n = O(n^2)$ - not asymptotic tight (can never become equal)

o (little oh)

For a given function $g(n)$, we denote a set of function. $o(g(n)) = \{ f(n) :$ for any constant $c > 0$, $\exists$ +ve constant n_0 such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0 \}$

* If $f(n) = o(g(n))$

then $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$

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

ω (little omega)

→ For a given function $g(n)$, we denote a set of functions $\omega(g(n)) = \{ f(n)$ such that for any constant $c > 0$, $\exists$ +ve n_0, such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0 \}$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

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

is $2^{2n} = O(2^n)$? not possible

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

$g(n) = 2^n$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{2^{2n}}{2^n} = \lim_{n\to\infty} 2^n = \infty$$

by definition of little omega

$2^{2n} = \omega(2^n)$ (big-oh is upper bound)

Q Prove that $4n = O(n^2)$

$f(n) = 4n$

$g(n) = n^2$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

$$= \lim_{n\to\infty} \frac{4n}{n^2}$$

$$\Rightarrow \lim_{n\to\infty} \frac{4}{n} = 0$$

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

$\Rightarrow 4n = o(n^2)$ proved.

Θ-notation

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

$\exists c_1, c_2, n_0$ (+ve, constants) such that $\forall\, c_1 g(n) \leq f(n) \leq c_2 g(n)$

O-notation

$\exists$ c, n_0 (+ve) such that $0 \leq f(n) \leq c \cdot g(n)$

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

Ω-notation

$\exists$ c & n_0 (+ve) s.t $0 \leq c\, g(n) \leq f(n)$

$f(n) = \Omega(g(n))$ Asymptotically tight

o-notation

$f(n) = o(g(n))$, for $\forall c > 0$, $\exists n_0$ such that

$0 \leq f(n) < c\, g(n)$

↳ Asymptotically not tight

w-notation

$f(n) = w(g(n))$, for $\forall c > 0$, $\nexists n_0$ such that

$0 \leq c\, g(n) < f(n)$ Asymptotically not tight

Note

* $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$ where $c > 0$

then $f(n) = \Theta(g(n)) \rightarrow f(n) = O(g(n))$ & $f(n) = \Omega(g(n))$

* $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$

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

* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

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

09-02-2024

Exercise

(1) for any real a & b, $b>0$, prove that $(n+a)^b = \Theta(n^b)$

(2) prove that $n! = O(n^n)$

(3) Prove that $n! = \omega(2^n)$

(4) prove that $\log(n!) = \Theta(n \log n)$

Ans

$\lim_{n\to\infty}$

Ans

(1) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{(n+a)^b}{n^b}$

$= \lim_{n\to\infty} \left(\frac{n+a}{n}\right)^b$

$= \lim_{n\to\infty} \left(1+\frac{a}{n}\right)^b$

$= \left(1+\frac{a}{\infty}\right)^b$

$= (1+0)^b$

$= 1$

so $\lim_{n\to\infty} \frac{(a+n)^b}{n^b} = 1 \Rightarrow (a+n)^b = \Theta(n^b)$

(Proved)

(2) $n! = O(n^n)$

An approximate value of $n!$ is given as:

(i) $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1+\Theta\left(\frac{1}{n}\right)\right)$ where $\Theta\left(\frac{1}{n}\right) = \frac{c}{n}$

(Sterlings approximation)

(ii) $n! = \left(\frac{n}{e}\right)^n$

To prove

$$\lim_{n\to\infty} \frac{n!}{n^n} - ?$$

~~compute~~

compute

$$\frac{n!}{n^n} = \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)}{n^n}$$

$$= \frac{\sqrt{2\pi n}\left(1+\Theta\left(\frac{1}{n}\right)\right)}{e^n}$$

$$= \frac{\sqrt{2\pi n}\left(1+\frac{c}{n}\right)}{e^n}$$

derivation

$$\frac{d}{dn}\left(\sqrt{2\pi n}\left(1+\frac{c}{n}\right)\right)$$

$$= \frac{1}{2\left(\sqrt{2\pi n}\right)} 2\pi\left(1+\frac{c}{n}\right) + \sqrt{2\pi n}\left(+\frac{c}{n^2}\right)$$

$$= \frac{1}{\sqrt{2\pi n}}\left(1+\frac{c}{n}\right) - \sqrt{2\pi n}\cdot\frac{c}{n^2}$$

$$= \frac{1\left(1+\frac{c}{n}\right) - \sqrt{2\pi n}\cdot\frac{c}{n^2}}{\sqrt{2\pi n}\, e^n}$$

$$= \frac{\pi + \frac{c\pi}{n} - \frac{2c\pi}{n}}{\sqrt{2\pi n}\, e^n}$$

$$= \frac{\pi - \frac{c\pi}{n}}{e^n \sqrt{2\pi n}}$$

$$= \frac{n!}{n^n}$$

So $\lim_{n\to\infty} \frac{n!}{n^n} = \lim_{n\to\infty} \frac{\pi - c\pi/n}{e^n \sqrt{2\pi n}}$

$$= \frac{\pi - c\pi/\infty}{\infty}$$

$$= \frac{\pi - 0}{\infty}$$

$$= 0$$

$\lim_{n\to\infty} \frac{n!}{n^n} = 0 \Rightarrow n! = o(n^n)$ (proved)

(3) $n! = w(2^n)$

$$n! = \left(\frac{n}{e}\right)^n$$

Proof

$$\lim_{n\to\infty} \frac{n!}{2^n} = \lim_{n\to\infty} \frac{(n/e)^n}{2^n}$$

$$= \lim_{n\to\infty} \frac{\log\left(\frac{n}{e}\right)^n}{\log 2^n}$$

$$= \lim_{n\to\infty} \frac{n(\log n - \log e)}{n \log 2}$$

$$= \lim_{n\to\infty} \left(\frac{\log n}{\log 2} - \frac{\log e}{\log 2}\right)$$

$$= \lim_{n\to\infty} (\log n - \log e)$$

$= \infty - c$

$= \infty$

So $\lim_{n \to \infty} \frac{n!}{2^n} = \infty \Rightarrow n! = \omega(2^n)$ (proved)

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

Proof

$$\lim_{n \to \infty} \frac{\log(n!)}{n \log n} = \lim_{n \to \infty} \frac{\log\left(\left(\frac{n}{e}\right)^n\right)}{n \log n}$$

$$= \lim_{n \to \infty} \frac{n \log n - n \log e}{n \log n}$$

$$= \lim_{n \to \infty} \left(1 - \frac{\log e}{\log n}\right)$$

$$= 1 - \frac{c}{\infty}$$

$$= 1 - 0$$

$$= 1$$

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

& compare $n!$, $n \log n$, 2^n, $\log(n!)$, $\log(n!)$, n^2, n

Ans

$$n < n \log n \leq \log n! < 2^n < n! < n^2$$

12-02-2024

Prove that $\sum_{i=1}^{n} \log(i) = \Theta(n \log n)$

Sol?

$$\sum_{i=1}^{n} \log(i) = \log(1) + \log(2) + \cdots + \log n)$$

$$= \log(1 \times 2 \times 3 \times \cdots \times n) \quad (\because \log(ab) = \log a + \log b)$$

$$= \log(n!)$$

$$= \Theta(n \log n) \quad \text{(proved)}$$

formula

(I) $b^{\log_b a} = a$ eg: $2^{\log_2 a} = a$

(II) $a^{\log_b c} = c^{\log_b a}$

(III) $\log a^K = (\log a)^K$

(IV) $\log_b a = \frac{1}{\log_a b}$

(V) $\ln a = \log_e a$

Polynomial

→ A polynomial of degree d, where d is non-negative is defined as P(n)

$$P(n) = \sum_{i=0}^{d} a_i n^i \qquad a_i = \text{coefficient of } n^i$$

eg: $p(n) = 5n^3 + 2n^2 + 5n + 6$

Note

A function $f(n)$ is called polynomially bounded. if $f(n) = O(n^K)$ for some constant $K > 0$.

Exponent

for any real $a > 0$, and m, n (integer) an exponent takes the form a^m or a^n

(I) $O(\log n)$ - logarithmic function (It takes less time

(II) $O(n^2)$ - polynomial function (It takes normal time)

(III) $O(2^n)$ Exponential function (It takes much more time)

Prove that $n^K = O(2^n)$ Rule: $n^K < 2^n < n! < n^n$

Compare the following fun

$n \log n, n^2, n^{100}, n!, 2^n$

logarithmic : $n \log n$

polynomial : n, n^2, n^{100}

Exponential : 2^n

$n!$

$n < n\log n < n^2 < n^{100} < 2^n < n!$

Sort the following function in asymptotic order of growth.

$\log(n!)$, 2^n, $2^{\log n}$, n^{200}, $n!$, n^2, $2^{\log n} = 2^{\log 2^n} = n$

Ans $2^{\log n} < \log(n!) < n^2 < n^{200} < 2^n < n!$

$\log(n!)$, $\frac{2^{\log n}}{n}$, n^{1000}, 2^n, $\sqrt{\log n}$, $\frac{4^{\log n}}{n^2}$, 2^{2^n}.

Ans $\sqrt{\log n} < 2^{\log n} < \log(n!) < 4^{\log n} < n^{1000} < 2^n < 2^{2^n}$

19-02-2024.

Divide & conquer

→ Design paradigm

→ It is a recursive method of solving the problem

→ At each step of recursion, it follows 3 essential steps.

(i) Divide

(ii) conquer

(iii) combine

(i) Divide

→ In divide step the problem is divided into sub problems that are smaller instances of the same problem.

(ii) conquer

→ In conquer step the subproblems are solved by recursively calling the same method or algorithm.

(iii) combine

→ It combines the results of the subproblems to get the solution of original problem.

Merge sort

→ merge sort follows divide & conquer approach.

→ Given a sequence of n numbers

→ It sorts the sequence using following steps.

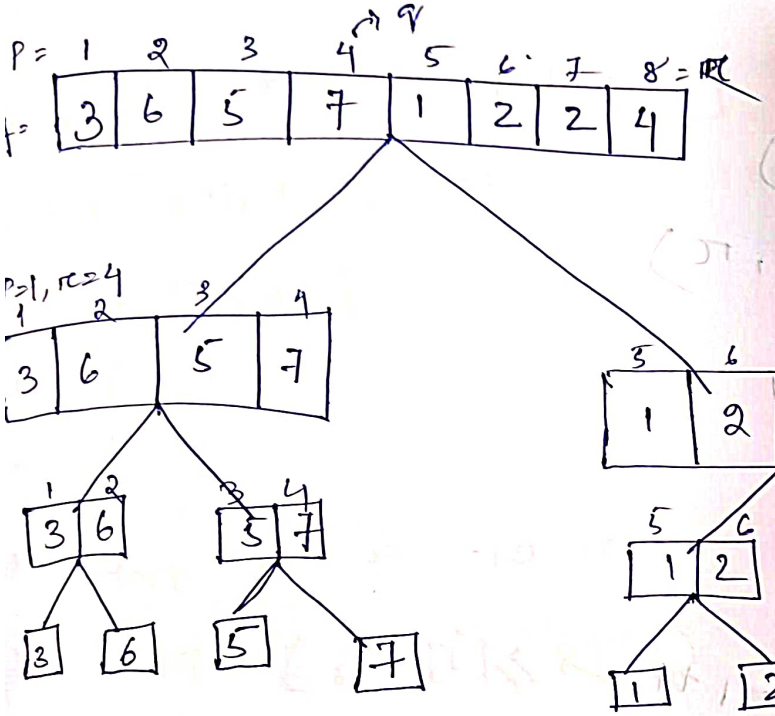
(i) Divide :- It divide the sequence of n numbers to two subsequences each of size $n/2$.

(ii) conquer :- The subsequences are sorted by calling merge sort.

(iii) combine:- combine the sorted subsequences to get the whole sequence sorted.

Eg:

input: $\langle 3, 6, 5, 7, 1, 2, 2, 4 \rangle$



p (beginning index) = 1

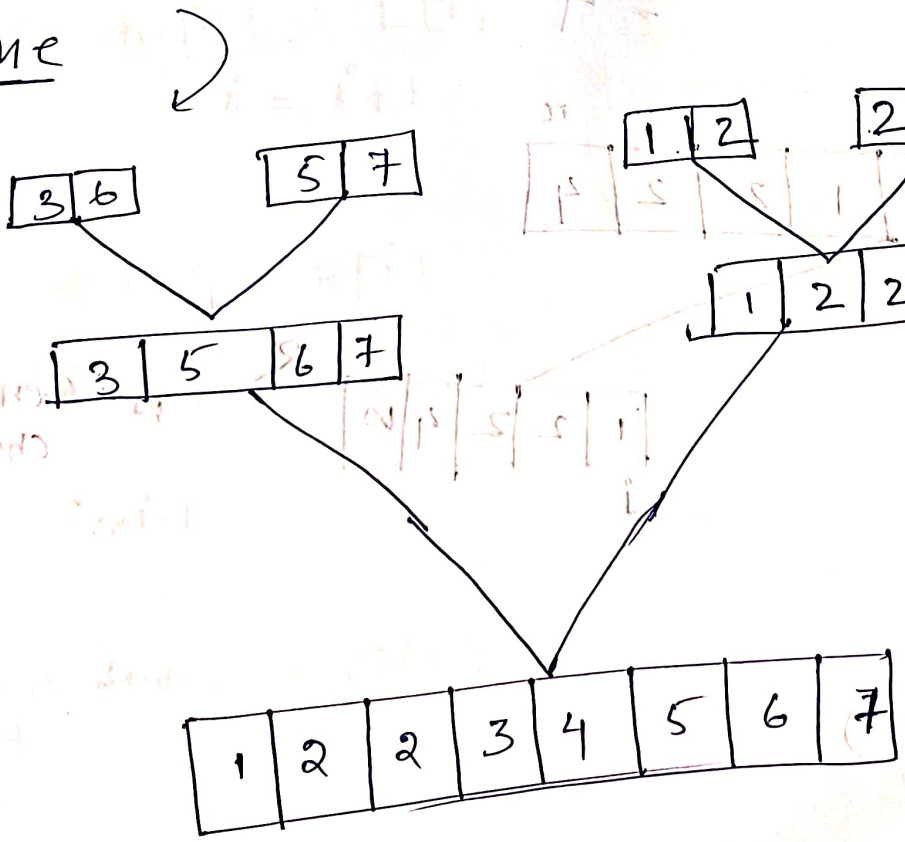
r (end index) = 8

q (mid index) = $\frac{p+r}{2}$

$5+8/2 = 6$

$p < r$
$p == r$
divide and merge sort

combine



Algorithm for Merge Sort :

merge sort (A, P, r)

p = starting index

r = end index

(I) If (p<r)

{

(II) q = p+r/2

(III) merge sort (A, p, q)

(IV) merge sort (A, q+1, r)

(V) merge (A, p, q, r)

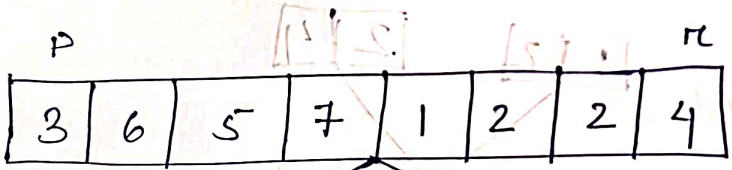
}

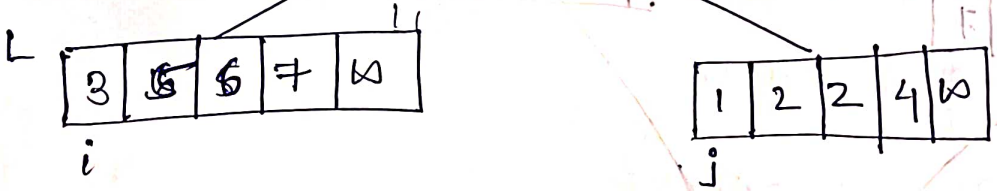
(VI) Exit

Merge

$n_1 = q - p + 1$

4-1+1 = 4





∞ - Sentinal Character

Algorithm

Merge (A, p, q, r)

(I) $n_1 = q - p + 1$

(II) $n_2 = r - q$

(III) Assume two subarray L of length $n_1 + 1$ and

R of length n_2+1

(iv) for $i = 1$ to n_1

v) $L[i] = A[p+i-1]$

vi) for $j = 1$ to n_2

vii) $R[j] = A[q+j]$

viii) $L[n_1+1] = \infty$

(ix) $R[n_2+1] = \infty$

(x) $i = 1$

(xi) $j = 1$

(xii) for $k = p$ to r

(xiii) if ($L[i] \leq R[j]$)

(xiv) $A[K] = L[i]$

(xv) $i = i+1$

(xvi) else

(xvii) $A[K] = R[j]$

(xviii) $j = j+1$, $K = K+1$

(xix) Exit

Merge time $= \Theta(n)$

21-02-2024

Time taken by merge sort

Step-1 Divide

Let $D(n)$ is the division time to divide an array of n elements

$D(n) = \Theta(1)$ in merge sort

Step-2 conquer

It divides the array of size n into two sub arrays of size $n/2$ each.

Step-3 combine time

$\Theta(n)$

If $T(n)$ is the total time taken by merge sort.

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + \Theta(1) + \Theta(n) & \text{if } n > 1 \\ \text{if } n = 1 \text{ then } T(n) = \Theta(1) \end{cases}$$

This is called recurrence eqn of merge sort.

Recurrence

→ A recurrence is an eqn or an inequality which describes the value of a function in terms of its value in smaller input size.

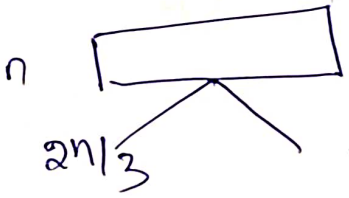
$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(1) + \Theta(n), \quad n > 1$$

→ The time complexity of recursive algorithmare often represented as recurrence relation.

eg: factorial

fact(n) = n * fact(n-1)

$$T(n) = \begin{cases} 0=1 & \text{if } n=1 \\ T(n-1)+1 & \text{if } n>1 \end{cases}$$



* $T(n) = T(2n/3) + T(n/3) + D(n) + C(n)$

Methods to solve recurrence:

→ Master Method

* General form of recurrence for divide & conquer approach.

$$T(n) = \begin{cases} O(1) & \text{if } n=1 \\ T(\frac{n}{b}) + C(n) + D(n) & \text{if } n>1 \end{cases}$$

→ Recurrence Tree method

→ Substitution method

→ Iterative method

→ Change of variable method

23-02-2024

Recurrence of Fibonacci series

$fib(n) = fib(n-1) + fib(n-2)$

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

→ A recurrence relation need to be solved to compute the asymptotic bound of the recurrence

eg: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$T(n) = \{$

lets take the recurrence of merge sort

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n > 1 \end{cases}$$

$$= \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases} \quad \text{—— (1)}$$

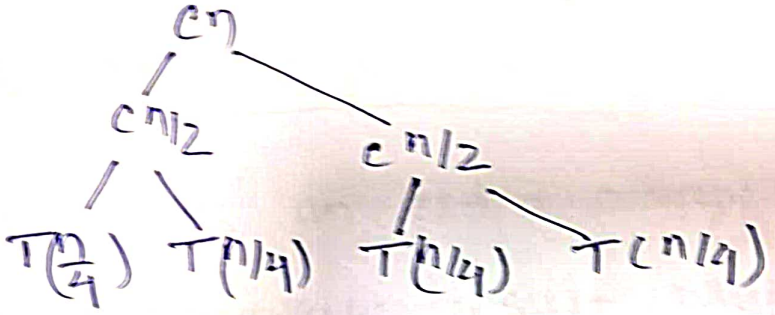
eq^n (1) can be written as $T(n) = 2T\left(\frac{n}{2}\right) + cn$ —— (2)

Represent eq^n (2) in a tree

(a)

$T\left(\frac{n}{2}\right) = 2\left(T\left(\frac{n}{4}\right)\right) + c\frac{n}{2}$ — (3)

fig (a) can be expanded by using eq^n (3)

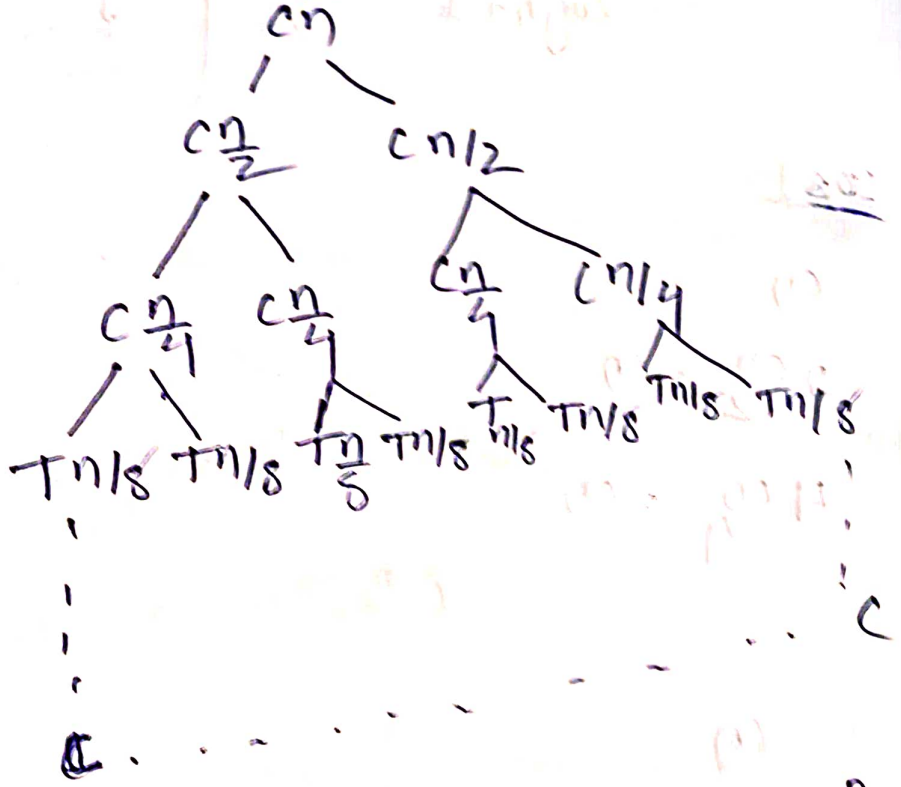


(b)

$T\left(\frac{n}{4}\right) = 2\left(T\left(\frac{n}{8}\right)\right) + c\frac{n}{4}$ — (4)

fig b can be expanded by using eq^n (4)

Recursion Tree method



cost of a node at level 0 = $cn = c\frac{n}{2^0}$

at level 1 cost of a single node = $c\frac{n}{2} = c\frac{n}{2^1}$

cost of a single node at level $i = c\frac{n}{2^i}$

Let the base condition arises at level K i.e

$$c\frac{n}{2^K} = c$$

$$\Rightarrow \frac{n}{2^K} = 1$$

$$\Rightarrow n = 2^K$$

$$\Rightarrow \boxed{K = \log n}$$

→ How many total no of levels present in the given tree $= K+1$

$= \log n + 1$ ∵ [$\log n$ = height of complete Binary tree]

Level	cost
0	cn
1	$2\frac{cn}{2} = cn$
2	$4\frac{cn}{4} = cn$
⋮	
K	cn

The total cost $T(n) = cn + cn + cn + \cdots cn$

$= (K+1)cn$

$= (\log n + 1)\, cn$

$= cn\log n + cn$

$= \Theta(n \log n)$

$T(n) = \Theta(n \log n)$ [Time complexity of merge sort]

Limitations of Merge sort :-

→ merge sort uses an extra array of size 'n' store the intermediate result while merging.

→ its space complexity is $(2n)$ where other algorithm is n

Master Method

$T(n) = a\,T(\frac{n}{b}) + f(n)$, $a \geq 1$, $b > 1$ and $f(n)$ is a given function.

the asymptotic bound of $T(n)$ takes the following form.

case-1

If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$,

then $T(n) = \Theta(n^{\log_b a})$

case-2

If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \log n)$

Case-3

If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$ and $af(\frac{n}{b}) \leq cf(n)$ for constant $c < 1$, then

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

26-02-2024

masters method

Q Solve the given recurrence using

$$T(n) = 9T(\frac{n}{3}) + n$$

A $a = 9,\ b = 3,\ f(n) = n$

$$n^{\log_b a} = n^{\log_3 9}$$

$$= n^{\log_3 3^2}$$

$$= n^2$$

assuming $\varepsilon = 1$, we can write $f(n) = n = O(n)$ or

$$f(n) = n \leq cn, \quad c > 1$$

$$= O(n)$$

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

$$= O(n^{\log_b a - \varepsilon}) \quad \text{where } \varepsilon = 1$$

So soln is $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$

Q $T(n) = T\left(\frac{2n}{3}\right) + 1$. solve the given recurrence using master method.

$a = 1, \quad b = 3/2 \quad f(n) = 1$

$n \log_b^a = n \log_{3/2}^1$

$= n^0$

$= 1$

$f(n) = \Theta(n \log_b^a)$

soln is $T(n) = \Theta(n \log_b^a \log n)$

$= \Theta(1 \cdot \log n)$

$= \Theta(\log n)$

Q solve the given recurrence $T(n) = 3T\left(\frac{n}{4}\right) + n \log n$ using master method.

$T(n) = 3T\left(\frac{n}{4}\right) + n \log n$

$a = 3 \quad b = 4, \quad f(n) = n \log n$

$n \log_b^a = n \log_4^3$

$= n^{0.793}$

[Case-3 $0.793 < f(n)$]

$n \log_b^a = n \log_4^3 = n^{0.793}$

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

$$= \Omega(n)$$

$$= \Omega(n \log_b{}^a + 0.207)$$

$$= \Omega(n^{\log_b a} + \epsilon) \quad \text{where } \epsilon = 0.207$$

↳ condition 1

condition-2

$$af\left(\frac{n}{b}\right) = a\left(\frac{n}{b}\right)\log\left(\frac{n}{b}\right)$$

$$= \frac{3}{4}n(\log n - \log 4)$$

$$= \frac{3}{4}n(\log n - \log 4)$$

$$= \frac{3}{4}n\log n - \frac{3}{4}n \times 2 \quad [\because \log 4 = 2]$$

$$= \frac{3}{4}n\log n - \frac{3}{2}n$$

$$\leq \frac{3}{4}n\log n$$

$$\leq c\,f(n) \quad \text{where } c = \frac{3}{4} < 1$$

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

$$f\left(\frac{n}{b}\right) = \frac{n}{b}\log\left(\frac{n}{b}\right)$$

As both the condition of case 3 of master theorem satisfies so the soln is $T(n)$ is

$$\boxed{T(n) = \Theta(f(n)) = \Theta(n\log n)}$$

Q using master method solve $T(n) = 2T(\frac{n}{2}) + n$

$T(n) = 2T(\frac{n}{2}) + n$

$a = 2, \; b = 2, \; f(n) = n$

$n \log_b^a = n \log_2^2$

$= n^1$

$= n$

$f(n) = \Theta(n \log_b^a)$

soln is $T(n) = \Theta(n^{\log_b^a} \log n)$

$= \Theta(n \log n)$

Substitution Method

→ substitution method works in 2 steps

(i) Guess the soln

(ii) Prove the soln by method of induction

→ There are different types of substitution method

(i) forward substitution method

(ii) Backward substitution method

Forward substitution

In this method the variable $n = 1, 2, 3, \ldots$ so on till it can guess a soln.

→ Then the soln is proved by method of induction.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) + 1 & \text{if } n \geq 2 \end{cases}$$

Soln

$T(1) = 1$

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

$T(3) = T(3-1) + 1 = 2 + 1 = 3$

let $T(K) = K$, then prove that $T(K+1) = K+1$

$T(K+1) = T(K+1-1) + 1$

$= T(K) + 1$

$= K + 1$ proved

then

$T(n) = n \leq cn \quad c \geq 1$

$= O(n)$

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

6. $T(n) = \begin{cases} 1 & \text{if } n=1 \\ 2T(n-1)+1 & \text{if } n \geq 2 \end{cases}$ solve this recurrence by forcwarrd substitution.

Ans $T(1) = 1$

~~$T(2) = 2T(1-1)+1 = 2$~~

~~$T(3) =$~~

Ans $T(1) = 1 = 2^{1}-1$

$T(2) = 2 \cdot T(1) + 1 = 3 = 2^{2}-1$

$T(3) = 2 \cdot T(2) + 1 = 2 \cdot 3 + 1 = 7 = 2^{3}-1$

$T(4) = 2 \cdot T(3) + 1 = (2 \cdot 7 + 1 = 15 = 2^{4}-1$

let $T(K) = 2^{K}-1$, then prove that $T(K+1) = 2^{K+1}-1$

$T(K+1) = 2 \cdot T(K+1-1) + 1$

$= 2T(K) + 1$

$= 2(2^{K}-1) + 1$

$= 2 \cdot 2^{K} - 2 + 1 = 2^{K+1} - 1$ proved

Hence we can write

$T(n) = 2^{n}-1 \leq C 2^{n}$ where $C \geq 1$

$= O(2^{n})$

Backward Substitution

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n-1) + 1 & \text{if } n > 1 \end{cases}$$

Sol?

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

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

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

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

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

$= 2^3 T(n-3) + 4 + 3$

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

Step-K

$= 2^K T(n-K) + 2^K - 1$

$\vdots$

let $K = n-1$

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

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

$= 2^{n-1} + 2^{n-1} - 1$

$= 2 \cdot 2^{n-1} - 1$

$= 2^n - 1$

So $T(n) = 2^n - 1 \leq c 2^n$ for $c > 1$

$= O(2^n)$

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

$T(n-2) = 2T(n-3) + 1$

Q. Solve the following recurrence using backward substitution

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(\frac{n}{2}) + n & \text{if } n \geq 2 \end{cases}$$

Sol?

$$T(n) = 2T(\tfrac{n}{2}) + n$$

$$= 2 \cdot (2T(\tfrac{n}{4}) + \tfrac{n}{2}) + n$$

$$= 2^2 T(\tfrac{n}{4}) + 2n$$

$$= 2^2 (2T(\tfrac{n}{8}) + \tfrac{n}{4}) + 2n$$

$$= 2^3 T(\tfrac{n}{8}) + 2^2 \cdot \tfrac{n}{4} + 2n$$

$$= 2^3 T(\tfrac{n}{8}) + 3n$$

$$\vdots$$

$T(\frac{n}{2}) = 2T(\frac{n}{4}) + \frac{n}{2}$

$T(\frac{n}{4}) = 2T(\frac{n}{8}) + \frac{n}{4}$

Let the basecase arises at step K.

$$= 2^K T(\tfrac{n}{2^K}) + Kn \quad \text{(at step K)} \longrightarrow (1)$$

$T(\frac{n}{2^K}) = 1 \Rightarrow \frac{n}{2^K} = 1$

$\Rightarrow n = 2^K$

$\Rightarrow K = \log n$

At step K the base case arises:

So $T(\frac{n}{2^K}) = 1$ (as it is base case)

$\rightarrow \frac{n}{2^K} = 1$

$\Rightarrow n = 2^K \Rightarrow K = \log n$

Put the value of K in eq^n (1)

$$T(n) = 2^K T\left(\frac{n}{2^K}\right) + Kn \qquad \left(2^K = 2^{\log n} = n\right)$$

$$= n \cdot 1 + \log n \cdot n$$

$$= n \log n + n$$

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

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