Algorithms
Chapter 4: Divide and conquer

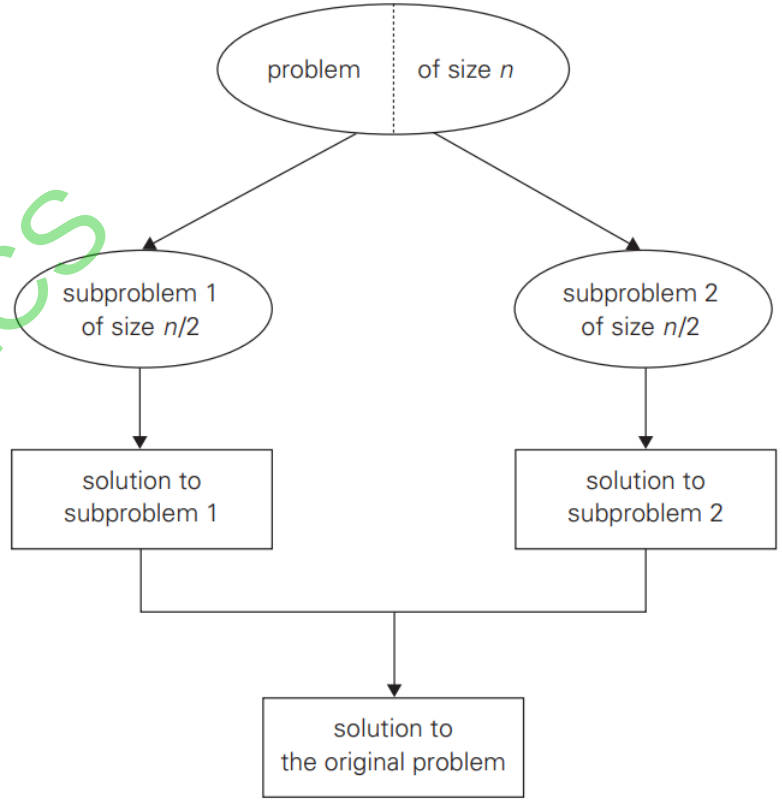
GATE CS Lectures
by Monalisa

- 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

- Chapter 4: Divide and conquer
- Min max problem ,
- Matrix multiplication ,
- Merge sort ,
- Quick Sort ,
- Binary Tree Traversals and Related Properties .

Divide and conquer

- Divide-and-conquer algorithms work according to the following general plan:
- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
- **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
- **Combine** the solutions to the subproblems into the solution for the original problem.
- ***Divide-and-conquer recurrence*** $T(n) = aT(n/b) + f(n$
- When the subproblems are large enough to solve recursively, we call that the ***recursive case***.
- Once the subproblems become small enough that we no longer recurse,we say that the recursion "bottoms out" and that we have gotten down to the ***base case***.



Master Theorem for Divide & conquer Recurrence

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

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 $T(n) = \Theta(f(n))$. ■

- 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>-1 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 $\Theta(n^{\log_b a})$*
- *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 :
- T(n)=2T(n/2)+1 T(n)=2T(n/2)+nlogn
- T(n)=T(2n/3)+1 T(n)=T(n/3)+n

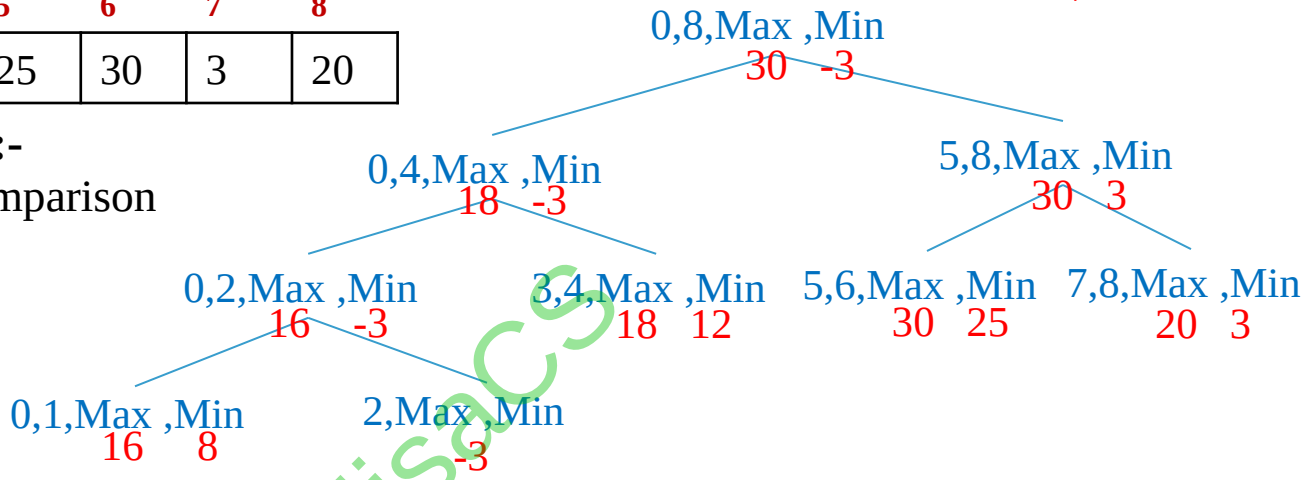
Min Max problem:

- *Algorithm MAXMIN(A , n , Max , Min)*
- *Max=Min=A[0]*
- *for i ← 1 to n-1*
- *{if(A[i] > Max)*
- *Max=A[i];*
- *else if(A[i] < Min)*
- *min=A[i]; }*
- Best case comparison =n-1 [Ascending order]
- Worst case comparison =2(n-1) [Descending order]
- Average case comparison=$\frac{3n}{2}$-1=1.5n-1 [Random order]
- Example :

0	1	2	3	4	5	6	7	8
8	16	-3	12	18	25	30	3	20

- Max=Min=A[0]=8
- i=1,16>8, Max=16
- i=2,-3>16,-3<8, Min=-3
- i=3,12>16,12<-3, no change
- i=4,18>16,Max=18
- i=5,25>16,Max=25
- i=6,30>25, Max=30
- i=7,3>30,3<-3, no change
- i=8,20>30,20<-3, no change
- Max=30 ,Min=-3

0	1	2	3	4	5	6	7	8
8	16	-3	12	18	25	30	3	20



- **Algorithm DC MaxMin :-**
- Let T(n) total number of comparison
- T(n)=0 n=1
- =1 n=2
- $=2T(\frac{n}{2})+2$ n>2
- $T(n)=2T(\frac{n}{2})+2$
- $=2(2T(\frac{n}{2^2})+2)+2$
- $=2^2T(\frac{n}{2^2})+2^2+2$
- $T(n)=2^kT(\frac{n}{2^k})+\sum_{i=1}^{k} 2^i$
- $=2^kT(\frac{n}{2^k})+\frac{2(2^k-1)}{2-1}$
- $[\frac{n}{2^k}=2 \Rightarrow n=2^{k+1} \Rightarrow k+1=\log n]$
- $T(n)=\frac{n}{2}T(2)+n-2$

- $= \frac{n}{2} + n - 2 = \frac{3n}{2} - 2$
- In all case $T(n) = \frac{3n}{2} - 2 = 1.5n - 2$
- Time complexity for both algorithm =O(n)
- **GATE CSE 2007 | Question: 50**
- An array of n numbers is given, where n is an even number. The maximum as well as the minimum of these n numbers needs to be determined. Which of the following is TRUE about the number of comparisons needed?
- **(A)** At least 2n – c comparisons, for some constant c, are needed.
- **(B)** At most 1.5n – 2 comparisons are needed.
- **(C)** At least $nLog_2n$ comparisons are needed.
- **(D)** None of the above.
- Ans: **(B)** At most 1.5n – 2 comparisons are needed.

Strassen's Algorithm for matrix multiplication

- If A and B are square n×n matrices, then in the product C=A.B, we define the entry C_{ij}, for i, j = 1, 2 …., n, by $C_{ij} = \sum_{k=1}^{n} A_{ik} . B_{kj}$ [Eq 1]
- We must compute n^2 matrix entries, and each is the sum of n values.
- *SQUARE-MATRIX-MULTIPLY(A, B)*

```
1.  n= A:rows
2.  let C be a new n × n  matrix
3.  for i = 1 to n
4.     for j = 1 to n
5.          C_ij = 0
6.          for k =1 to n
7.  C_ij = C_ij + A_ik . B_kj
8.  return C
```

- Each of the triply-nested **for** loops runs exactly n iterations, and each execution of line 7 takes constant time, SQUARE-MATRIX-MULTIPLY takes $\Theta(n^3)$ time.
- **A simple divide-and-conquer algorithm**
- C= A . B, we assume that n is an exact power of 2 in each of the n × n matrices.
- Suppose that we partition each of A, B, and C into four n/2 × n/2 matrices.
- $A=\begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix}$, $B=\begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix}$, $C=\begin{vmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{vmatrix}$

- So we rewrite the equation C=A.B as $\begin{vmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{vmatrix} = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \cdot \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix}$
- $C_{11}=A_{11}.B_{11}+A_{12}.B_{21}$ $C_{12}=A_{11}.B_{12}+A_{12}.B_{22}$ $C_{21}=A_{21}.B_{11}+A_{22}.B_{21}$ $C_{22}=A_{21}.B_{12}+A_{22}.B_{22}$
- Hence, recurrence for **divide-and-conquer** algorithm:
- $T(n)=\begin{cases} \Theta(1) & \text{if } n \leq 2 \\ 8T(n/2)+\Theta(n^2) & \text{if } n > 2 \end{cases}$
- By the master method the solution T (n)= $\Theta(n^3)$.
- **Strassen's method**
- Instead of performing eight recursive multiplications of n/2 × n/2 matrices ,it performs only seven.
- The cost of eliminating one matrix multiplication will be several new additions of n/2 × n/2 matrices, but still only a constant number of additions.
- Strassen's method has four steps:
- 1. Divide the input matrices A and B and output matrix C into n/2 × n/2 submatrices.
- This step takes Θ (1) time by index calculation.
- 2. Create 10 matrices $S_1,S_2,...S_{10}$, each of which is n/2×n/2 and is the sum or difference of two matrices created in step 1.
- We can create all 10 matrices in $\Theta(n^2)$ time.
- 3. Using the submatrices created in step 1 and in step 2,recursively compute seven matrix products P_1, P_2 ,…..P_7. Each matrix P_i is n/2 × n/2.

- 4. Compute the desired submatrices C_{11}, C_{12}, C_{21}, C_{22} of the result matrix C by adding and subtracting various combinations of the P_i matrices. We can compute all four submatrices in $\Theta(n^2)$ time.
- Let us assume that once the matrix size n gets down to 2, we perform a simple scalar multiplication, just as in line 4 of SQUARE-MATRIX-MULTIPLY-RECURSIVE.
- When $n > 2$, steps 1, 2, and 4 take a total of $\Theta(n^2)$ time, and step 3 requires us to perform seven multiplications of $n/2 \times n/2$ matrices.
- Hence, recurrence for Strassen's algorithm:
- $$T(n)=\begin{cases}\Theta(1) & \text{if } n \le 2 \\ 7T(n/2)+\Theta(n^2) & \text{if } n > 2\end{cases}$$
- By the master method the solution $T(n)= \Theta(n^{\lg 7})= \Theta(n^{2.807})$.

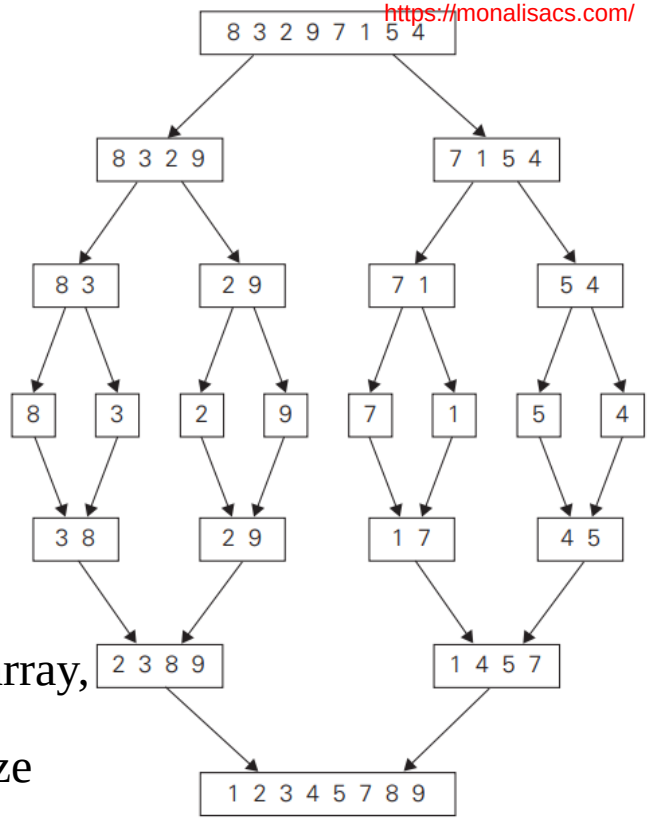
$$C_{11}=A_{11}.B_{11}+A_{12}.B_{21}$$
$$C_{12}=A_{11}.B_{12}+A_{12}.B_{22}$$
$$C_{21}=A_{21}.B_{11}+A_{22}.B_{21}$$
$$C_{22}=A_{21}.B_{12}+A_{22}.B_{22}$$

- **Step 2:**
- $S_1 = B_{12}-B_{22}$, $S_2 = A_{11}+A_{12}$, $S_3 = A_{21}+A_{22}$, $S_4 = B_{21}-B_{11}$, $S_5 = A_{11}+A_{22}$,
- $S_6 = B_{11}+B_{22}$, $S_7 = A_{12}-A_{22}$, $S_8 = B_{21}+B_{22}$, $S_9 = A_{11}-A_{21}$, $S_{10} = B_{11}-B_{12}$,
- We must add or subtract matrices 10 times, this step take $\Theta(n^2)$ time.
- **Step 3:** $P_1=A_{11}.S_1$ $P_2=S_2.B_{22}$ $P_3=S_3.B_{11}$
- $P_4=A_{22}.S_4$ $P_5=S_5.S_6$ $P_6=S_7.S_8$ $P_7=S_9.S_{10}$
- **Step 4**
- $C_{11}= P_5 + P_4 - P_2 + P_6$ $C_{12}= P_1 + P_2$
- $C_{21}= P_3 + P_4$ $C_{22}= P_5 + P_1 - P_3 - P_7$

Mergesort

- The ***merge sort*** algorithm follows the divide-and-conquer paradigm. It operates as follows.
- **Divide:** Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.
- **Conquer:** Sort the two subsequences recursively using merge sort.
- **Combine:** Merge the two sorted subsequences to produce the sorted answer.
- **ALGORITHM** *Mergesort(A[0..n - 1])*
- /Sorts array *A*[0..*n* - 1] by recursive mergesort
- //Input: An array *A*[0..*n* - 1] of orderable elements
- //Output: Array *A*[0..*n* - 1] sorted in nondecreasing order
- **if** *n* > 1
- copy *A*[0..*n*/2 - 1] to *B*[0..*n*/2 - 1]
- copy *A*[*n*/2..*n* - 1] to *C*[0..*n*/2 - 1]
- *Mergesort(B[0..n/2 - 1])*
- *Mergesort(C[0..n/2 - 1])*
- *Merge(B, C, A)*
- The key operation of the merge sort algorithm is the merging of two sorted sequences in the “combine” step.We merge by calling procedure MERGE.
- It ***merges*** them to form a single sorted array A.
- MERGE procedure takes time Θ(n).

- **ALGORITHM** *Merge(B[0..p - 1], C[0..q - 1], A[0..p + q - 1])*
- //Merges two sorted arrays into one sorted array
- //Input: Arrays *B*[0..*p* - 1] and *C*[0..*q* - 1] both sorted
- //Output:Sorted array *A*[0..*p* + *q* - 1] of the elements of *B* and *C*
- $i \leftarrow 0; j \leftarrow 0; k \leftarrow 0$
- **while** $i < p$ **and** $j < q$ **do**
- **if** $B[i] \leq C[j]$
- $A[k] \leftarrow B[i]; i \leftarrow i + 1$
- **else** $A[k] \leftarrow C[j]; j \leftarrow j + 1$
- $k \leftarrow k + 1$
- **if** $i = p$
- copy $C[j..q - 1]$ to $A[k..p + q - 1]$
- **else** copy $B[i..p - 1]$ to $A[k..p + q - 1]$



- **Divide:** The divide step just computes the middle of the subarray, which takes constant time. Thus, D(n)= Θ (1).
- **Conquer:** We recursively solve two subproblems, each of size n/2, which contributes 2T(n/2) to the running time.
- **Combine:** We have already noted that the MERGE procedure on an n-element subarray takes time Θ(n), and so C(n)=Θ(n).
- The recurrence relation for the number of key comparisons $C(n)$ is $C(n) = 2C(n/2) + C_{merge}(n)$ for $n > 1$, $C(1) = 0$.

- $C_{merge}(n)$, the number of key comparisons performed during the merging stage.
- At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needing to be processed is reduced by 1.
- In the worst case, neither of the two arrays becomes empty before the other one contains just one element .
- Number of comparison in Best case=n_1 ,Worst case=$n_1+n_2-1=n-1$ [let $|B|=n_1,|C|=n_2$]
- Therefore, for the worst case, $C_{merge}(n) = n - 1$, and we have the recurrence
- $C_{worst}(n) = 2C_{worst}(n/2)+n-1$ for n>1 ,$C_{worst}(1)=0$
- T(n)=$\begin{cases} c & if\ n = 1, \\ 2T(n/2) + cn & if\ n > 1, \end{cases}$
- By master theorem running time of merge sort is Θ(nlogn) [Best ,Worst ,Average case]
- Space complexity =c + n + log n(Stack)=O(n)
- Merge sort is not in place as it need extra memory but stable.
- **2-way merge sort/Bottom-up Merging**
- $n=2^k$
- Let A = 31 , 28 , 17 , 65 , 35 , 42 , 86 , 25
- 1st pass: 28,31, 17,65, 35,42, 25,86
- 2nd pass:17,28,31,65, 25,35,42,86
- 3rd pass:17,25,28,31,35,42,65,86
- Number of passes require=$\log_2 n$,Time complexity = $\Theta(n\log_2 n)$

- **Pros and cons of Merge Sort:**
- Pros:
- 1.Large size problem
- 2.Linked list (work on both array and linked list)
- 3.External sorting (sort in external memory)
- 4.Stable (order doesn't change for repeated element.)
- Ex:3,8,2,3,7,4,after sort:2,3,3,4,7,8
- Cons:
- 1.Extra space(not in place sort)
- 2.No small problem (insertion sort work better than merge sort for small problem)
- 3.recursive
- GATE CS 1995 | Question: 1.5,ISRO 2011
- Merge sort uses:
- (A)Divide and conquer strategy (B)Backtracking approach
- (C) Heuristic search (D) Greedy approach
- Ans : a)Divide and conquer strategy
- GATE CS 1995 | Question: 1.16
- For merging two sorted lists of sizes *m* and *n* into a sorted list of size *m*+*n*, we require comparisons of
- (a) O(m) (b)O(n) (c)O(m+n) (d)O(logm+logn)
- Ans : (c)O(m+n)

- GATE CS 1999 | Question: 1.14, ISRO2015-42
- If one uses straight two-way merge sort algorithm to sort the following elements in ascending order: 20, 47, 15, 8, 9, 4, 40, 30, 12, 17
- then the order of these elements after second pass of the algorithm is:
- (a)8, 9, 15, 20, 47, 4, 12, 17, 30, 40 (b)8, 15, 20, 47, 4, 9, 30, 40, 12, 17
- (c)15, 20, 47, 4, 8, 9, 12, 30, 40, 17 (d)4, 8, 9, 15, 20, 47, 12, 17, 30, 40
- Sol:
- 1st pass:20,47, 8,15, 4,9, 30,40, 12,17
- 2nd pass:8,15,20,47, 4,9,30,40, 12,17
- Ans : (b)8, 15, 20, 47, 4, 9, 30, 40, 12, 17
- ISRO2018,2020
- Of the following sort algorithms, which has execution time that is least dependent on initial ordering of the input? (A)Insertion sort
- (B)Quick sort (C)Merge sort (D)Selection sort
- Ans : (C)Merge sort
- ISRO2018-35
- Given two sorted list of size m and n respectively. The number of comparisons needed the worst case by the merge sort algorithm will be:
- (A)m×n (B)maximum of m and n
- (C)minimum of m and n (D)m+n−1
- Ans: (D)m+n−1

Quicksort

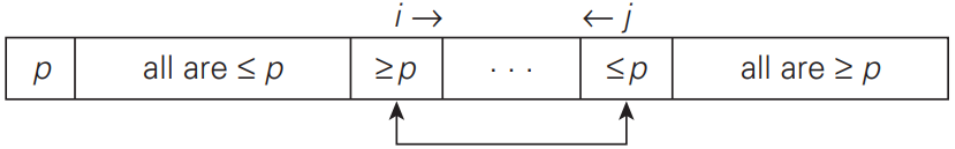
- **Description of quicksort**
- Quicksort, applies the divide-and-conquer paradigm .
- The three-step divide-and-conquer process for sorting array $A[l....r]$
- **Divide:** Partition (rearrange) the array $A[l....r]$ into two subarrays $A[l..s - 1]$ and $A[s + 1..r]$ such that each element of $A[l..s - 1]$ is less than or equal to $A[s]$, which is, in turn, less than or equal to each element of $A[s + 1..r]$. Compute the index s as part of this partitioning procedure.
- **Conquer:** Sort the two subarrays $A[l..s - 1]$ and $A[s + 1..r]$ by recursive calls to quicksort.
- **Combine:** Because the subarrays are already sorted, no work is needed to combine them: the entire array $A[l....r]$ is now sorted.

$$\underbrace{A[0] \ldots A[s-1]}_{\text{all are} \le A[s]} \quad A[s] \quad \underbrace{A[s+1] \ldots A[n-1]}_{\text{all are} \ge A[s]}$$

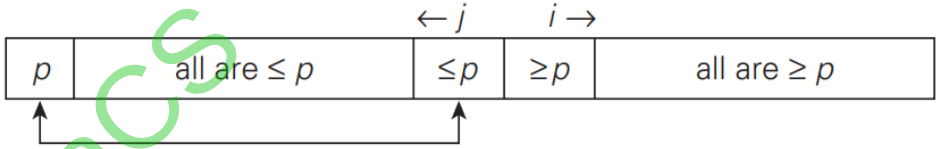
- Obviously, after a partition is achieved, $A[s]$ will be in its final position in the sorted array, and we can continue sorting the two subarrays to the left and right of $A[s]$ independently.
- Note the difference with merge sort :
- There, entire work happens in combining their solutions;
- Here, the entire work happens in the division stage, with no work required to combine the solutions to the subproblems.

- **ALGORITHM** *Quicksort(A[l..r])*
- //Input: Subarray of array *A*[0..*n* - 1], defined by its left and right indices *l* and *r*
  //Output: Subarray *A*[*l*..*r*] sorted in nondecreasing order
- **if** *l* < *r*
- *s* ←*Partition(A[l..r]) //s* is a split position
- *Quicksort(A[l..s - 1])*
- *Quicksort(A[s + 1..r])*
- C.A.R. Hoare, the prominent British computer scientist who invented quicksort .
- We will now scan the subarray from both ends,comparing the subarray's elements to the pivot.
- The **left-to-right scan**, denoted below by index pointer *i*, starts with the second element.
- Since we want elements smaller than the pivot to be in the left part of the subarray, this scan skips over elements that are smaller than the pivot and stops upon encountering the first element greater than or equal to the pivot.
- The **right-to-left scan**, denoted below by index pointer j, starts with the last element of the subarray.
- Since we want elements larger than the pivot to be in the right part of the subarray, this scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot.

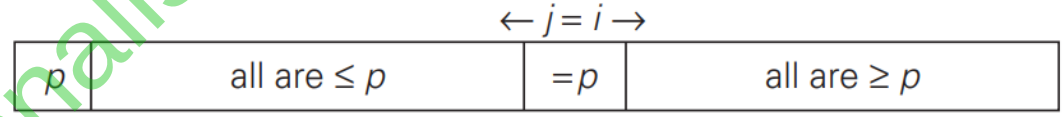
- After both scans stop, three situations may arise.
- If scanning indices *i* and *j* have not crossed, i.e.,*i* < *j*, we simply exchange *A*[*i*] and *A*[*j*] and resume the scans by incrementing *i* and decrementing *j*, respectively:



- If the scanning indices have crossed over, i.e., *i* > *j*, we will have partitioned the subarray after exchanging the pivot with *A*[*j*]:



- Finally, if the scanning indices stop while pointing to the same element, i.e., *i* = *j*,the value they are pointing to must be equal to *p* .
- Thus, we have the subarray partitioned, with the split position *s* = *i* = *j*:



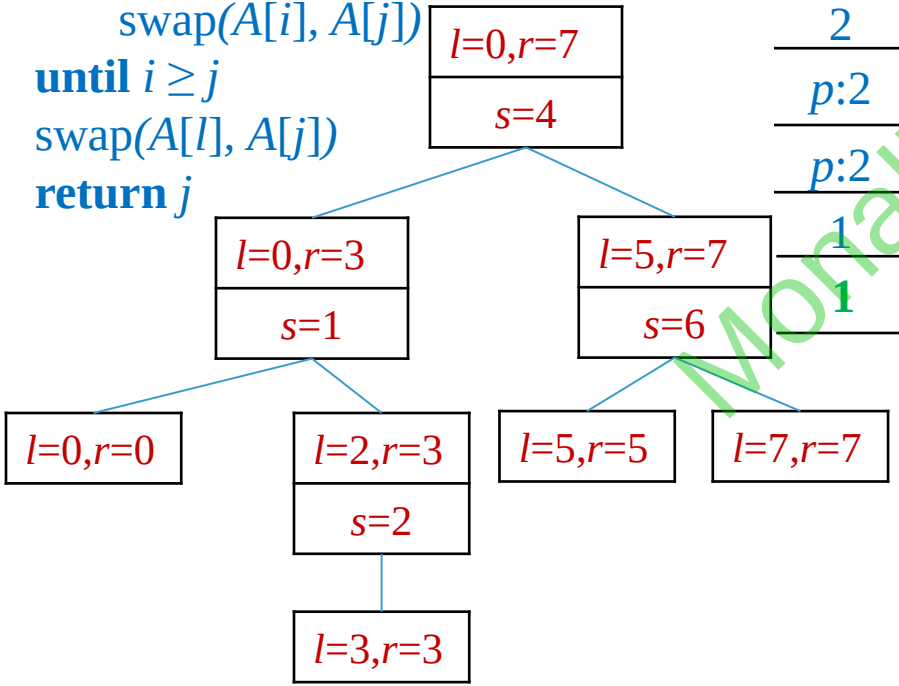
- We can combine this with the case of crossed-over indices *(i > j)* by exchanging the pivot with *A*[*j*] whenever *i* ≥ *j*.
- **ALGORITHM** *HoarePartition(A[l..r])*

  //Partitions a subarray by Hoare's algorithm, using the first element as a pivot

  //Input: Subarray of array *A*[0..*n* - 1], defined by its left(*l*) and right(*r*)*(l < r)*

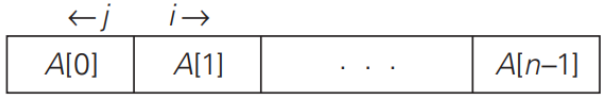
  //Output: Partition of *A*[*l*..*r*], with the split position returned by this function.

- $p \leftarrow A[l]$
- $i \leftarrow l; j \leftarrow r + 1$
- **Repeat**
- **repeat** $i \leftarrow i + 1$ **until** $A[i] \geq p$
- **repeat** $j \leftarrow j - 1$ **until** $A[j] \leq p$
- *if*$(i<j)$
- swap$(A[i], A[j])$
- **until** $i \geq j$
- swap$(A[l], A[j])$
- **return** j



0	1	2	3	4	5	6	7
p:5	3	1	9	8	2	4	7
p:5	3	1	i:9	8	2	j:4	7
p:5	3	1	4	i:8	j:2	9	7
p:5	3	1	4	j:2	i:8	9	7
2	3	1	4	**5**	8	9	7
p:2	i:3	j:1	4		p:8	i:9	j:7
p:2	j:1	i:3	4		p:8	j:7	i:9
1	**2**	3	4		7	**8**	9
1		p,j:3	i:4		**7**		**9**
		3	4				
			4				

- **Performance of quicksort**
- The running time of quicksort depends on whether the partitioning is balanced or unbalanced, which in turn depends on which elements are used for partitioning.
- If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort.
- If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort.
- **Best-case partitioning**
- In the most even possible split, PARTITION produces two subproblems, each of size no more than n/2, since one is of size $\lfloor n/2 \rfloor$ and one of size $\lceil n/2 \rceil$-1.
- In this case, quicksort runs much faster. The recurrence for the running time is then
  $T(n) = 2T(n/2) + \Theta(n)$ for $n > 1$, $T(1) = 0$.
- By case 2 Master Theorem, $T(n) \in \Theta(n \log_2 n)$;
- **Worst-case partitioning**
- In the worst case, all the splits will be skewed to the extreme: one of the two subarrays will be empty, and the size of the other will be just 1 less than the size of the array being partitioned.
- This situation will happen, for increasing arrays, if $A[0..n - 1]$ is a strictly increasing array and we use $A[0]$ as the pivot, the left-to-right scan will stop on $A[1]$ while the right-to-left scan will go all the way to reach $A[0]$, indicating the split at position 0:



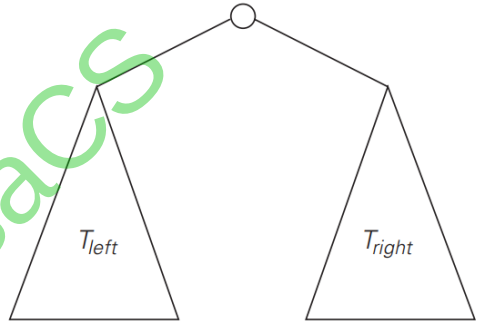
- So, after making $n + 1$ comparisons to get to this partition and exchanging the pivot $A[0]$ with itself, the algorithm will be left with the strictly increasing array $A[1..n - 1]$to sort.
- This sorting of strictly increasing arrays of diminishing sizes will continue until the last one $A[n - 2..n - 1]$ has been processed.
- The total number of key comparisons made will be equal to
- $C_{worst}(n) = (n + 1) + n + \ldots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2)$
- The Quicksort recurrence for already sorted array
- $T(n) = T(n\text{-}1) + T(0)+\Theta(n)=T(n\text{-}1)+\Theta(n)$ for $n > 1$, $T(0) = \Theta(1)$.
- The partitioning costs $\Theta(n)$ time . By substitution method $T(n) \in \Theta(n^2)$;
- Therefore, the worst-case running time of quicksort is no better than that of insertion sort.
- Moreover, the $\Theta(n^2)$ running time occurs when the input array is already completely sorted—a common situation in which insertion sort runs in O(n) time.
- **Average-case partitioning**
- Assuming that the partition split can happen in each position s with the same probability $1/n$, we get the following recurrence relation:

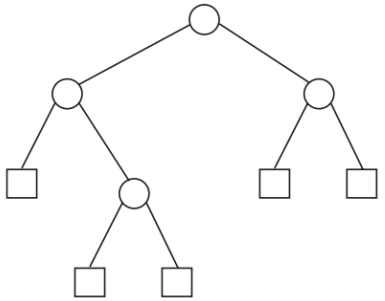
- $C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1,$
- $C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$
- Its solution ≅1.39n $\log_2$n
- Thus, on the average, quicksort makes only 39% more comparisons than in the best case.
- Better pivot selection methods such as ***randomized quicksort*** that uses a random element or the ***median-of-three*** method that uses the median of the leftmost, rightmost, and the middle element of the array .
- Like any sorting algorithm, quicksort has weaknesses. It is not **stable**. It requires a stack to store parameters of subarrays that are yet to be sorted.
- The size of this stack can be made to be in O(log n) by always sorting first the smaller of two subarrays obtained by partitioning, it is worse than the O(1) space efficiency of heapsort.
- ISRO-2013-12
- Which of the following sorting algorithms has the minimum running time complexity in the best and average case?
- (A) Insertion sort, Quick sort (B)Quick sort, Quick sort
- (C) Quick sort, Insertion sort (D)Insertion sort, Insertion sort
- Ans : (A) Insertion sort, Quick sort
- GATE CS 1994 | Question: 1.19, ISRO2016-31

- Algorithm design technique used in quicksort algorithm is?
- (A) Dynamic programming (B)Backtracking (C) Divide and conquer (D)Greedy method
- Ans: (C) Divide and conquer
- GATE CS 2006 | Question: 52
- The median of n elements can be found in O(n) time. Which one of the following is correct about the complexity of quick sort, in which median is selected as pivot?
- (A)$\Theta(n)$ (B)$\Theta(n\log n)$ (C)$\Theta(n^2)$ (D)$\Theta(n^3)$
- As we choose the pivot a median element .So, good splits,best case $\Theta(n\log n)$.
- Ans : (B)$\Theta(n\log n)$
- ISRO2015-12
- A machine needs a minimum of 100 sec to sort 1000 names by quick sort. The minimum time needed to sort 100 names will be approximately
- (A)50.2 sec (B)6.7 sec (C)72.7 sec (D)11.2 sec
- Running time of quick sort = $\Theta(n \lg n) = c\, n \lg n$
- For n = 1000, $100 = c * 1000 * \lg 1000 = c*1000*10$(approx $\lg 1000=9.966$)
- $\Rightarrow c=\frac{100}{10000}= 0.01$
- For n = 100, Running time = $0.01 * 100 * \lg 100 = 1* 6.64 \cong 6.7$
- Ans : (B)6.7 sec

Binary Tree Traversals and Related Properties

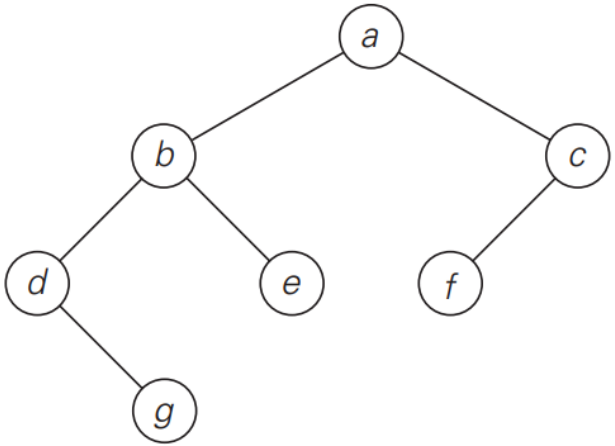
- In this section, we see how the divide-and-conquer technique can be applied to binary trees.
- A ***binary tree*** T is defined as a finite set of nodes that is either empty or consists of a root and two disjoint binary trees T_L and T_R called, respectively, the left and right subtree of the root.
- **ALGORITHM** *Height(T)*
- //Computes recursively the height of a binary tree
- //Input: A binary tree T
- //Output: The height of T
- **if** $T = \emptyset$ **return** -1
- **else return** $\max\{Height(T_{left}), Height(T_{right})\} + 1$



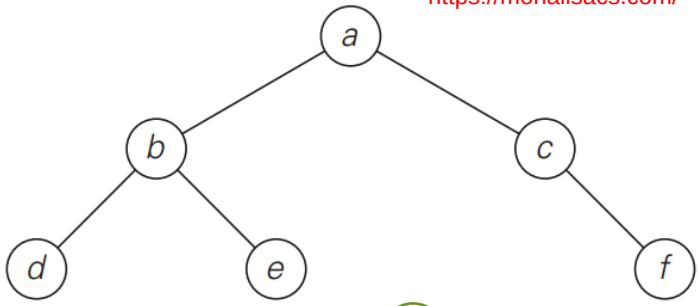


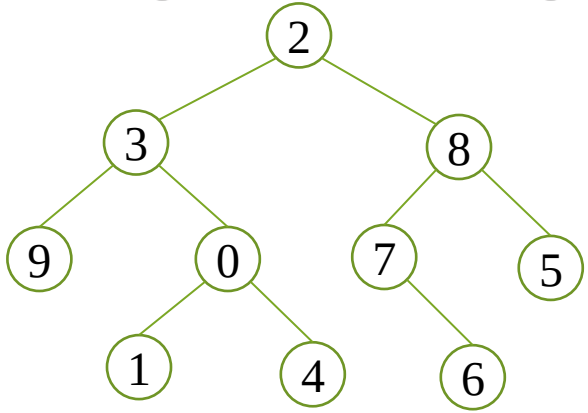
- We measure the problem's instance size by the number of nodes $n(T)$ in a given binary tree T.
- $A(n(T)) = A(n(T_{left})) + A(n(T_{right})) + 1$ for $n(T) > 0$,
- $A(0) = 0$.
- The extra nodes(squares) are called ***external***; the original nodes(circles) are called ***internal***.
- The number of external nodes x is always 1 more than the number of internal nodes n: $x = n + 1$.

- Consider the total number of nodes, both internal and external. Since every node, except the root, is one of the two children of an internal node, we have the equation $2n + 1 = x + n$
- Equality also applies to any nonempty ***full binary tree***, in which, by definition, every node has either zero or two children .
- The most important divide-and-conquer algorithms for binary trees are the three classic traversals: preorder, inorder, and postorder.
- All three traversals visit nodes of a binary tree recursively, i.e., by visiting the tree's root and its left and right subtrees. They differ only by the timing of the root's visit:
- In the ***preorder traversal***, the root is visited before the left and right subtrees are visited.
- In the ***inorder traversal***, the root is visited after visiting its left subtree but before visiting the right subtree.
- In the ***postorder traversal***, the root is visited after visiting the left and right subtrees.
- Finally, not all questions about binary trees require traversals of both left and right subtrees.
- For example, the search and insert operations for a binary search tree require processing only one of the two subtrees.
- Accordingly, we considered them not as applications of divide-and conquer but rather as examples of the decrease and conquer technique.



preorder: *a, b, d, g, e, c, f*
inorder: *d, g, b, e, a, f, c*
postorder: *g, d, e, b, f, c, a*





- Traverse the following binary tree
- **a.** in preorder. **b.** in inorder. **c.** in postorder.
- Preorder: a,b,d,e,c,f
- Inorder:d,b,e,a,c,f
- Postorder:d,e,b,f,c,a
- Draw a binary tree with 10 nodes in such a way that the inorder and postorder traversals of the tree yield the following lists:9,3,1,0,4,2,7,6,8,5 (inorder) and 9,1,4,0,3,6,7,5,8,2 (postorder).