Module - 2

Brute Force Approaches (contd):

Exhaustive Search:

* Exhaustive Search is simply a brute-force approach to combinatorial problems.

* It suggests generating each and every element of the problem domain, selecting those of them that satisfy all the constraints, and then finding a desired element.

Let's illustrate exhaustive search by applying it to 2 important problems i.e,

1) Travelling Salesman Problem:

* The travelling salesman problem asks to find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started.

* The problem can be conveniently modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances.

* Then the problem can be stated as the problem of finding the shortest Hamiltonian circuit* of the graph.

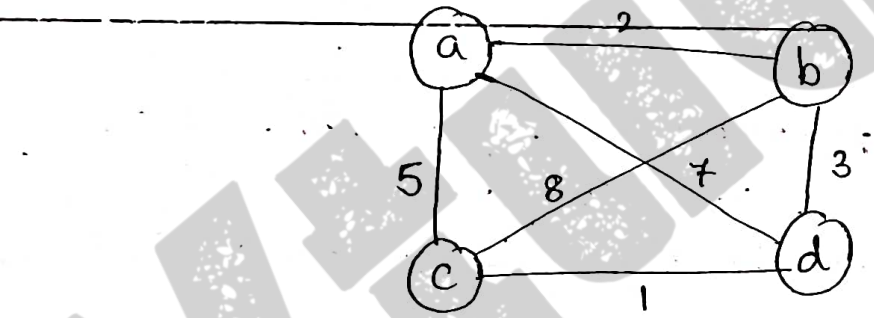
VTUCode

* Hamiltonian circuit: is defined as a cycle that passes thro all the vertices of the graph exactly once. It is named after the Irish mathematician Sir William Rowan Hamilton.

* Hamiltonian circuit can also be defined as a sequence of n+1 adjacent vertices $v_{i_0}, v_{i_1}, \ldots v_{i_{n-1}}, v_{i_0}$, where the first vertex of the sequence is the same as the last one and all other n-1 vertices are distinct.

* Thus, we can get all the tours by generating all the permutations of n-1 intermediate cities, compute the tour lengths, and find the shortest among them.

Ex:



The figure presents a small instance of the problem

* An inspection of figure reveals 3 pairs of tours that differ only by their direction.

* So, we could cut the number of vertex permutations by half.

Ex: choose any 2 intermediate vertices, say, b and c, and then consider only permutations in which b precedes c.

i.e,

Tour	Length
1) $a \to b \to c \to d \to a$	$l = 2+8+1+7 = 18$
2) $a \to b \to d \to c \to a$	$l = 2+3+1+5 = 11$ optimal
3) $a \to c \to b \to d \to a$	$l = 5+8+3+7 = 23$
4) $a \to c \to d \to b \to a$	$l = 5+1+3+2 = 11$ optimal.
5) $a \to d \to b \to c \to a$	$l = 7+3+8+5 = 23$
6) $a \to d \to c \to b \to a$	$l = 7+1+8+2 = 18$

~~* The total no of permutations needed~~ is still $\frac{1}{2}(n-1)!$; which makes the exhaustive-search approach impractical for all but very small values of n.

Ex:

Given n jobs $\langle j1, j2, j3, j4 \rangle$ and n persons $\langle p1, p2, p3, p4 \rangle$, it is required to assign all n jobs to all n persons with the constraint that one job has to be assigned to one person and the cost involved in completing all the jobs should be minimum.

	J_1	J_2	J_3	J_4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

Tour	Length
1) A → B → C → D → A :	9 + 3 + 8 + 7 = 20: optimal
2) A → B → D → C → A :	2 + 7 + 9 + 5 = 23
3) A → C → B → D → A :	7 + 8 + 7 + 7 = 29
4) A → C → D → B → A :	7 + 8 + 6 + 6 = 27
5) A → D → B → C → A :	8 + 6 + 3 + 5 = 22
6) A → D → C → B → A :	8 + 9 + 8 + 6 = 31

∴ The shortest route is A B C D A with a total distance of 70.

VtuCode

2) Knapsack Problem:

* Given n items of known weights $w_1, w_2 \dots w_n$ and values $v_1, v_2 \dots v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

* The exhaustive-search approach to this problem leads to generating all the subsets of the set of n items given, computing the total weight of each subset in order to identify feasible subsets, and finding a subset of the largest value among them.

* The number of subsets of an n-element set is 2^n; the exhaustive search leads to a $\Omega(2^n)$ algorithm, no matter how efficiently individual subsets are generated.

Ex:

1) Solve the following instance of knapsack problem where W = 10,

Item	1	2	3	4
Weight	7	3	4	5
value	42	12	40	25

Soln:

We need to list all the possible subsets and their calculate their respective weight and value.
Whichever, subset has the highest value is called optimal.

VTUCode

Subset	Total Weight	Total value
{1}	7	42
{2}	3	12
{3}	4	40
{4}	5	25
{1, 2}	10	54
{1, 3}	11	Not feasible (NA)*
{1, 4}	12	NA
{2, 3}	7	52
{2, 4}	8	37
{3, 4}	9	65
{1, 2, 3}	14	NA
{1, 2, 4}	15	NA
{1, 3, 4}	16	NA
{2, 3, 4}	12	NA
{1, 2, 3, 4}	19	NA

∴ The subset {3, 4} has the optimal solution.

6) Assignment Problem:

* Here, there are n people who need to be assigned to execute n jobs, one person per job.

i.e, each person is assigned to exactly one job and each job is assigned to exactly one person.

* The cost that would accure if the ith person is assigned to the jth job is known as quantity C[i,j] for each pairs i, j = 1, 2, ... n.

* The problem is to find an assignment with the minimum total cost.

Ex:

A small instance of this problem follows, with the table entries representing the assignment costs C[i, j]:

	Job1	Job 2	Job 3
Person 1	9	2	7
Person 2	6	4	3
Person 3	5	8	1

* The problem is to select one element in each row of the matrix so that all selected elements are in different columns and the total sum of the selected elements is the smallest possible.

Vtucode

*

Cost Matrix	Cost:
1) ⟨1, 2, 3⟩	9 + 4 + 1 = 15
2) ⟨1, 3, 2⟩	9 + 3 + 8 = 20
3) ⟨2, 1, 3⟩	2 + 6 + 1 = 9
4) ⟨2, 3, 1⟩	2 + 3 + 5 = 10
5) ⟨3, 2, 1⟩	7 + 4 + 5 = 16
6) ⟨3, 1, 2⟩	7 + 6 + 8 = 21

∴ The optimal cost is 9.

Where, person 1 → Job2, person 2 → Job 1, person 3 → Job3.

* The requirements of the assignment problem imply that there is a one-to-one correspondence b/w feasible assignments and permutations of the first n integers.

* Therefore, the exhaustive-search approach to the assignment problem would require generating all the permutations of integers 1, 2 . . n, computing the total cost of each assignment by summing up the corresponding elements of the cost matrix, and finally selecting the one with the smallest sum.

Decrease - And - Conquer:

* The decrease - and - conquer technique is based on exploiting the relationship b/w a solution to a given instance of a problem and a solution to its smaller instance.

* once a relationship is established, it can be exploited either top down or bottom up.

* The bottom-up variation is usually implemented iteratively, starting with a solution to the smallest instance of the problem; it is called sometimes the incremental approach.

There are 3 major variations of decrease - and - conquer.

1) Decrease by a constant
2) Decrease by a constant factor
3) Variable size decrease.

1) Decrease by a constant:

* In the decrease by a constant variation, the size of an instance is reduced by the same constant on each iteration of the algorithm.

* Typically, this constant is equal to one, although other constant size reductions do happen occasionally.

* Consider, as an example, the exponentiation problem of computing a^n where $a \neq 0$ and n is a non-negative integer.

* The relationship b/w a solution to an instance of size n and an instance of size $n-1$ is obtained by the obvious formula $a^n = a^{n-1} \cdot a$.

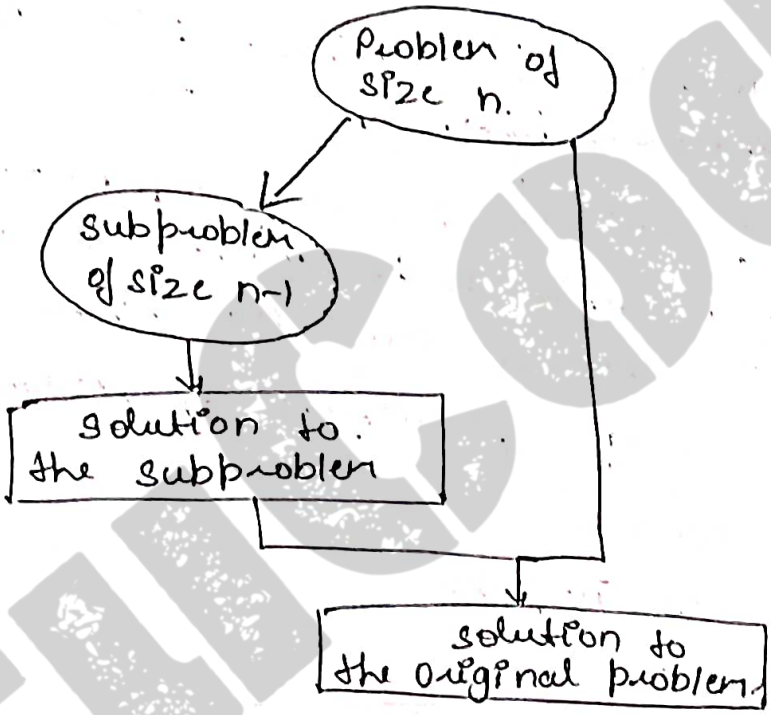


Fig: Decrease-(by one)-and-conquer technique.

* So the function $f(n) = a^n$ can be computed either "top down" by using its recursive definition or "bottom up" by multiplying 1 by a n times.

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 0, \\ 1 & \text{if } n = 0. \end{cases}$$

2) Decrease-by-a constant factor:

* This technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm
* In most applications, this constant factor is equal to two.

Ex:

* If the instance of size n is to compute a^n, the instance of half its size is to compute $a^{n/2}$, with the obvious relationship b/w the two: $a^n = (a^{n/2})^2$.

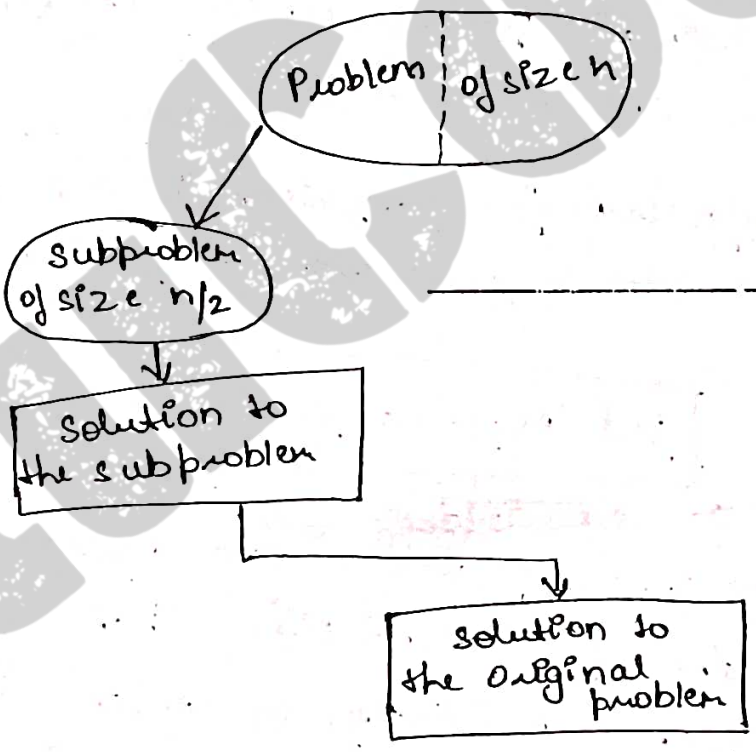


Fig: Decrease-(by-half)-and-conquer technique.

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive,} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd} \\ 1 & \text{if } n = 0. \end{cases}$$

* If we compute a^n recursively according to the formula and measure the algorithm's efficiency by the number of multiplications, we should expect the algorithm to be in $\Theta(\log n)$ because, on each iteration, the size is reduced by about a half at the expense of one or two multiplications on each iteration.

Eg: Is Binary search.

3) Variable - Size - Decrease:

* The size-reduction pattern varies from one iteration of an algorithm to another.

* Euclid's algorithm for computing the greatest common ~~divisor provides~~ a good example of such a situation.

Formula: $\boxed{gcd(m, n) = gcd(n, m \bmod n)}$

* Though the value of the second argument is always smaller on the right-hand side than on the left-hand side, it decreases neither by a constant nor by a constant factor.

Insertion Sort:

* We consider an application of the decrease-by-one technique to sort an array $A[0 \ldots n-1]$.
* We assume that the smaller problem of sorting the array $A[0 \ldots n-2]$ has already been solved to give us a sorted array of size $n-1$: $A[0] \leq \ldots \leq A[n-2]$.
* We need is to find an appropriate position for $A[n-1]$ among the sorted elements and insert it there.
* This is usually done by scanning the sorted subarray from right to left until the first element smaller than or equal to $A[n-1]$ is encountered to insert $A[n-1]$ right after the element.
* The resulting algorithm is called straight insertion sort or Insertion sort.

Algorithm: InsertionSort $(A[0 \ldots n-1])$

```
// sorts a given array by insertion sort
// Input: An array A[0..n-1] of n orderable elements
// output: Array A[0..n-1] sorted in nondecreasing order.
for i ← 1 to n-1 do
    v ← A[i]
    j ← i-1
    while j ≥ 0 and A[j] > v do
        A[j+1] ← A[j]
        j ← j-1
    A[j+1] ← v
```

Example:

Given an array: [25, 2, 4, 32, 1, 3].

Let n be the no of elements i.e n = 6.

for i = 1 to n-1 do:
This loop iterates through each element of the array starting from index 1.

1) i = 1. i.e A[1], which is 2 in our example.

Key (v) = A[i]

∴ v = 2.

j = i - 1 → j is set to the index before the current element.

1) j = 0. Since i is 1, 'j' becomes 0.

while j >= 0 and A[j] > v do:

→ Since j is 0 and A[j] (which is 25) is greater than v (which is 2), the loop is entered.

A[j+1] = A[j]

→ shift the element at index 'j' to the right.

j = j - 1

→ Decrement j, ∴ becomes -1.

A[j+1] = v.

→ Insert the v (which is 2) into its current position.
Now the array is [2, 25, 4, 32, 1, 3].

This process continues for each element of the array until the array is fully sorted.

Vtucode

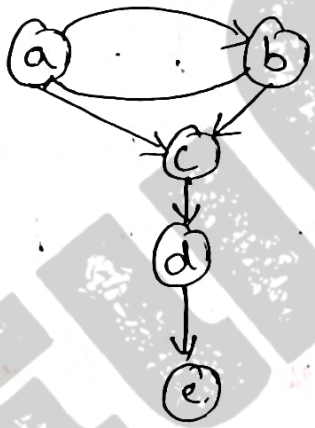
Topological sorting:

* Before we move to the topological sorting, we must know about the graphs.

* A Directed graph or digraph, is a graph with directed ~~graphs themselves~~ specified for all its edges.

* A directed cycle in a digraph is a sequence of three or more of its vertices that starts and ends with the same vertex and in which every vertex is connected to its immediate predecessor by an edge directed from the predecessor to the successor.

Ex: 1)



a, b, a is a directed cycle in the above graph.

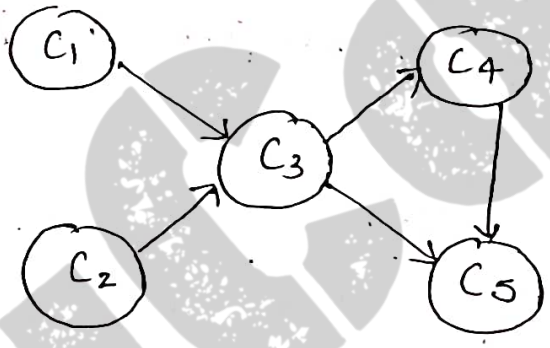
* Conversely, if a DFS forest of a digraph has no back edges, the diagraph is a dag, an acronym for directed acyclic graph.

Ex: 2)

consider a set of five required courses {C1, C2, C3, C4, C5} a part-time student has to take in some degree program.

* The courses can be taken in any order as long as the following courses prerequisites are met:

C1 and C2 has no prerequisites, C3 requires C1 and C2, C4 requires C3 and C5 requires C3 and C4.



* The student can take only one course per term.
* The question is whether we can list its vertices in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends.

This problem is called topological sorting.

* For topological sorting, to be possible, a diagraph has must be a dag.
* There are two efficient algorithms that both verify whether a diagraph is a dag and, if it is, produce an ordering of vertices that solves the topological sorting problem.

First Algorithm: Depth-First Search (DFS):

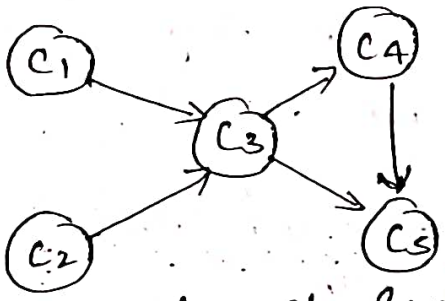
* DFS is an algorithm used for traversing or searching tree or graph data.
* It starts at a designated node and explores as far as possible along each branch before backtracking.
* Along the path, or order, if in which vertices become dead-ends, they are popped out of the traversal stack.
* Reversing this order yields a solution to the topological sorting problem, provided, of course, no back edge has been encountered during the traversal.

Ex:

* When a vertex v is popped off a DFS stack, no vertex u with an edge from u to v can be among the vertices ~~popped off before v~~.

* Hence, any such vertex u will be listed after v in the popped-off order list, and before v in the reversed list.

In Ex2:



Let us consider the edge as $C_1 \rightarrow u$

So, the traversal starts from C_1

i) $C_1 \longrightarrow C_3 \longrightarrow C_4 \longrightarrow C_5$

stack.

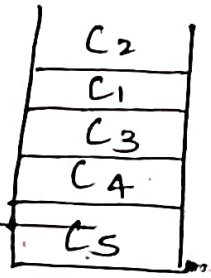
ii) Since C_5 does not have any vertex further connected it is moved into the stack and ~~starts~~ DFS again starts with C_4.

iii) Again C_4 does not have any path except C_5, C_4 is also popped into the stack.

So on, for C_3 and C_1 also.

iv) Now, we start DFS from C_2, as C_2 does have traversal from $C_2 \rightarrow C_3$, But C_3 is also already in the stack. We stop at C_2.

* When we, reverse the order of the stack, ~~we get the topological sorting~~ of the given graph.



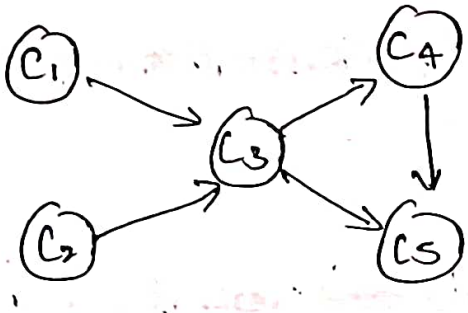
i.e C_2, C_1, C_3, C_4, C_5.

Second Algorithm: Decrease-(by-one)-and-conquer

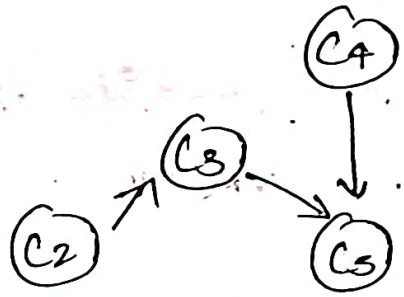
* Repeatedly, Identify in a remaining digraph a source, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it.

* The order in which the vertices are deleted yields a solution to the topological sorting problem.

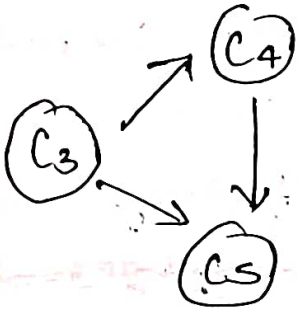
Ex:



delete C_1 →

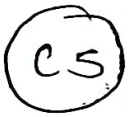


delete C_2 →



delete C_3 →

delete C_4 →



delete C_5 →

The solution obtained is C_1, C_2, C_3, C_4, C_5.

---

* Note:

The solution obtained by the source-removal algorithm is different from one obtained by the DFS-algorithm.

* Both are correct, of the topological sorting problem.

Divide - and - Conquer:

* Divide - and - conquer is probably the best - known general algorithm design technique.

* The algorithm works according to the following general plan:

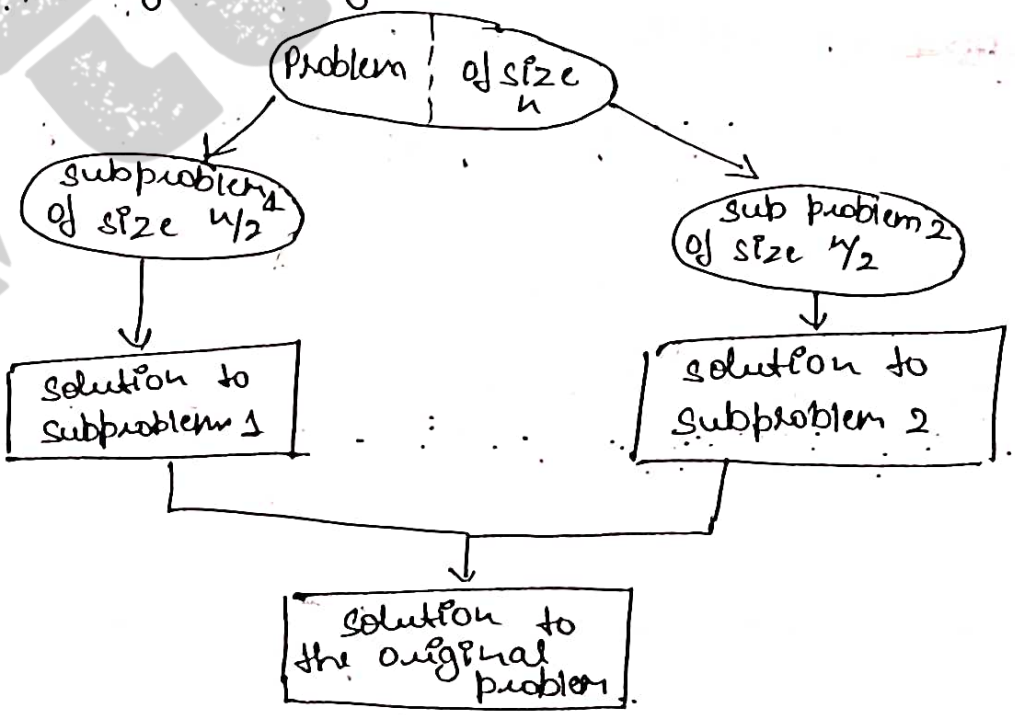
1) A problem is divided into several subproblems of the same type, ideally of about equal size.

2) The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough)

3) If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

---

* The divide - and - conquer technique depicts the case of dividing a problem into 2 smaller subproblems, by far the most widely occuring case.



Ex:

Let us consider the problem of computing the sum of n numbers $a_0 \cdots a_{n-1}$.

If $n > 1$, we can divide the problem into 2 instances of the same problem: to compute the sum of the first $\lfloor n/2 \rfloor$ nos and to compute the remaining $\lceil n/2 \rceil$ nos

i.e,

$$a_0 + \cdots + a_{n-1} = (a_0 + \cdots a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \cdots + a_{\lceil n-1 \rceil}).$$

* In the most typical case of divide-and-conquer a problem's instance of size n is divided into 2 instances of size $n/2$.

---

* More generally, an instance of size n can be divided into b instances of size n/b, with a of them needing to be sloved.

* Assuming that size n is a power of b to simplify our analysis, we get the following recurrence for the running time $T(n)$:

$$\boxed{T(n) = a \cdot T(n/b) + f(n)}$$

where, $f(n) \rightarrow$ is a function that accounts for the time spent on dividing an instance of size n into instances of size n/b and combining their solutions. This is called the general divide-and-conquer recurrence.

* The order of growth of its solution $T(n)$ depends on the values of the constants a and b and the order of growth of the function $f(n)$.

* The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the following theorem

Master Theorem:

If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence, then,

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Analogous results hold for the O and Ω notations too.

Ex:

The recurrence for the number of additions $A(n)$ made by the divide-and-conquer sum-computation algorithm on inputs of size $n = 2^k$ is.

By the formula $\rightarrow T(n) = aT(n/b) + f(n)$

we get, $A(n) = 2 \cdot A(n/2) + 1$

$\therefore a = 2$, $b = 2$, and $d = 0$; since $a > b^d$

$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \underline{\Theta(n)}$.

Merge Sort:

* Merge Sort is a perfect example of a successful application of the divide-and-conquer technique.

* It sorts a given array $A[0..n-1]$ by dividing it into two halves $A[0..\lfloor n/2 \rfloor - 1]$ and $A[\lfloor n/2 \rfloor..n-1]$, sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

Algorithm 1:

```
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 non-decreasing 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 merging of 2 sorted arrays can be done as follows:

* Two pointers (array indices) are initialized to point to the first elements of the array being merged.

* The elements pointed to are compared; and the smaller of them is added to a new array being constructed.

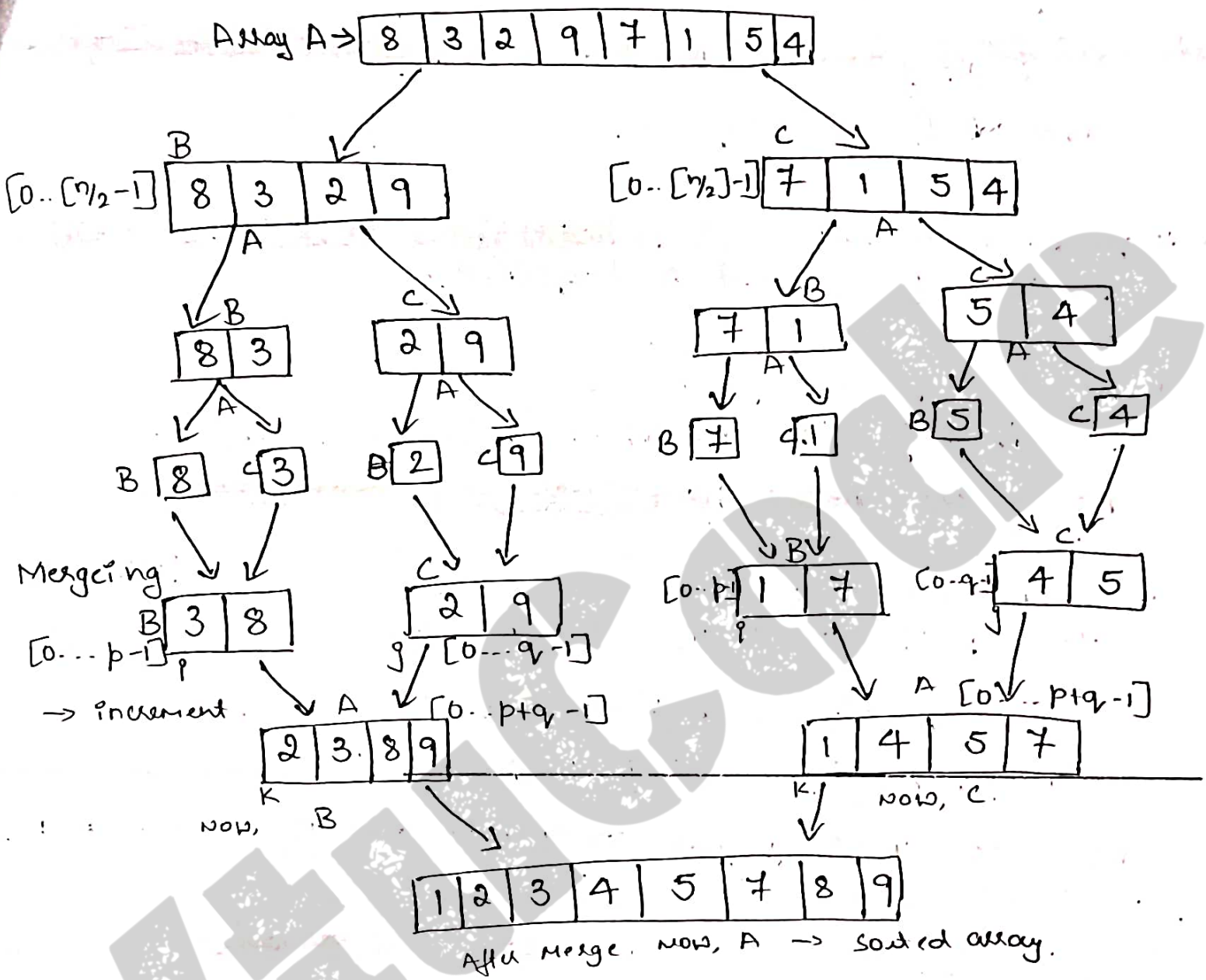
* The index of the smaller element is incremented to point to its immediate successor in the array it was copied from.

* This operation is repeated until one of the 2 given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

Algorithm 2:

```
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 & C.
  i ← 0; j ← 0; k ← 0.
While i < p and j < q do
    if B[i] ≤ C[j]
        A[k] ← B[i]; i ← i+1
    else A[k] ← C[j]; j ← j+1
    k ← 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]
```

Example:



* It begins from Algorithm, where the whole array is called A and be is being divided into 2 halves. as B and C. i.e B [0 -- ⌈n/2⌉ - 1] and C [0 -- ⌈n/2⌉ - 1].
* The same is repeated till all the elements are divided seperately.
* once all are divided, the function merge is called. i.e Merge (B, C, A).

* Merge (B [0 -- p-1], C [0 --- q-1], A [0 -- p+q-1])
  i.e, B array values are from 0 to p-1; C -> 0 to q-1
  and A [0 to p+q-1].

* We have initalized i, j, k = 0 for iteration of array ~~A, B~~, B, C, and A respectively.

* While i < p and j < q do.
  i.e, check for the condition, of array B and C.
  Above ex: i = 0 & p = 1 and j = 0 & q = 1.

* if B[i] ≤ C[j]
  i.e, B[0] ≤ C[0] ⇒ 3 ≤ 2.

  A[k] ← B[i]; i ← i+1
  or A[k] ← C[j]; j ← j+1

As, 2 is small, j gets incremented and is saved in A[k]
k ← 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].

As, i ≠ P,
C gets copied to A

The same repeated for all the conditions and gets merged to A array. Hence gets the sorted list.
The time complexity of merge sort is O(n log n).

Analysis:

Assuming for simplicity that n is a power of 2, the recurrence relation for the no of key comparisons C(n) is

$$C(n) = 2C(n/2) + C_{merge}(n) \text{ for } n>1,$$

$$C(1) = 0.$$

Let us analyse $C_{merge}(n)$, the no of key comparisons performed during the merge stage.

At each step, exactly one comparision is made, after which the total no of elements in the 2 arrays still needing to be processed is reduced by 1.

In the worst case, neither of the 2 arrays becomes empty before the other one contains just one element (eg: smaller elements may come from the alternating arrays).

∴ for the worst case, $C_{merge} = n-1$, and we have recurrence

$$C_{worst}(n) = 2 \cdot C_{worst}(n/2) + n-1 \quad \text{for } n>1,$$

$$C_{worst}(1) = 0.$$

Hence, Acc to theorem (Master theorem),

$$\boxed{C_{worst}(n) = n \log_2 n - n + 1.}$$

Vtucode

Analysis of Merge sort using Master theorem,

If the time for merging operation is proportional to n, then

$$T(n) = \begin{cases} a & n=1, \ a \text{ is constant} \\ 2T(n/2)+c \cdot n & n>1, \ c \text{ is constant.} \end{cases}$$

Assume $n = 2^k$, then:

$$T(n) = 2 \cdot T(n/2) + c \cdot n$$

$$= 2[2 \cdot T(n/4) + c \cdot n/2] + c \cdot n$$

$$= 4T(n/4) + 2cn$$

$$= 4[2T(n/8) + (n/4)] + 2cn$$

$$= 8T(n/8) + cn + 2cn$$

$$= 2^3 T(n/2^3) + 3cn$$

$$= 2^k T(n/2^k) + kcn$$

$$= nT(1) + kcn \qquad // n = 2^k.$$

$$T(n) = na + cn \log n.$$

If $2^k \leq n \leq 2^{k+1}$, then $T(n) \leq T(2^{k+1})$

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

Vtucode

Quick Sort:

* Quick sort is the other important sorting algorithm that is based on the divide-and-conquer approach.

* Quick sort divides its input elements according to their value.

* It is divided into partition. i.e A partition is an arrangement of the array's elements so that all the problem to the left of some element A[s] are less than or equal to A[s]; and all the elements to the right of A[s] are greater than or equal to it.

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

* After the partition is achieved, A[s] will be in its final position in the sorted array and we can continue ~~sorting~~ sorting the 2 subarrays to the left and to the right of A[s] independently.

Pseudocode of quicksort: call Quicksort (A[0--n-1])

P.T.O.

Algorithm 1:

```
Quicksort (A [l ... r])
// sorts a subarray by quicksort
// 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 ascending order.
if l < r
    s ← partition (A[l ... r]    // s is a split position
    quicksort (A [l ... s-1])
    quick sort (A [s+1 ... r])
```

Partitioning:

* We start by selecting a pivot - an element with respect to whose value we are going to divide the subarray.
* There are several different strategies for selecting a pivot where in here we use the simplest strategy of selecting the subarray's first element: p = A[l].
* Make first element as the pivot element P.
* Set i (low index to point to 0) and j to high which is n-1.

* Increment i until $A[i] \geq p$. If condition is met stop incrementing i.

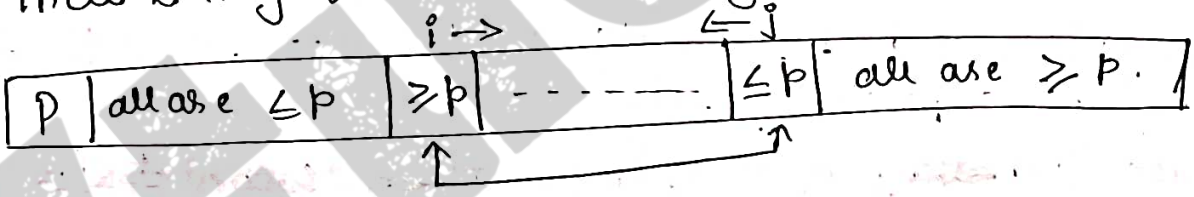
* Decrement j until $A[j] \leq p$. If condition is met stop decrementing j.

* compare the positions of i and j (three case may arise $<, >, =$)

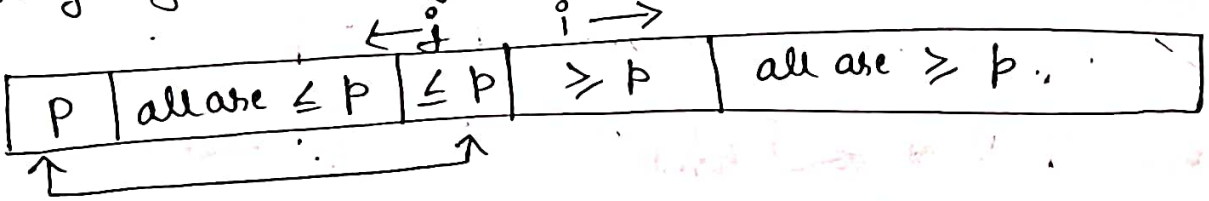
i) If $i < j$ perform swap $A[i]$, $A[j]$ and continue incrementing i and from the same position.

* ~~If scanning index has crossed~~

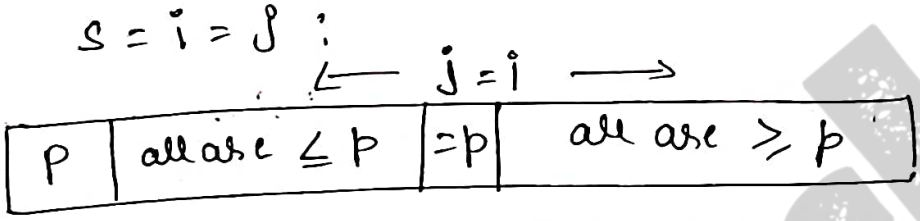
* If scanning index i and j have not crossed, i.e $i < j$, we simply exchange $A[i]$ and $A[j]$ and resume scanning by incrementing i and decrementing j, respectively.



* If $i > j$ then swap pivot element with $a[j]$; return j, as the split position. 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 combine the last case with the case of crossed-over indices ($i > j$) by exchanging the pivot with $A[j]$ whenever $i \geq j$. j is the split position.

Algorithm:

Hoare Partition (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 & right indices l and r ($l < r$)
// Output: Partition of A[l..r], with the split position returned as this function's value.

```
p ← A[l]
i ← l ; j ← r+1
repeat
    repeat i ← i+1 until A[i] ≥ p
    repeat j ← j-1 until A[j] ≤ p
    swap(A[i], A[j])
```

until $i \geq j$

swap (A[i], A[j]) // undo last swap when $i \geq j$

swap (A[l], A[j])

return j.

Note:

That index i can go out of the subarray's bounds in this pseudocode. Rather than checking for this possibility every time index i is incremented, we can append to array A[0..n-1] a "sentinel" that would prevent index i from advancing beyond position n.

Ex:

0	1	2	3	4	5	6	7
Elements to sort							
5 pivot no	3 (i)	1	9	8	2	4	7 (j)
5	3	1	9 (i)	8	2	4 (j)	7
compares i & j value to p. no							
5	3	1	4 (i)	8	2	9 (j)	7
swaps i & j value							
5	3	1	4	8 (i)	2 (j)	9	7
5	3	1	4	2 (i)	8 (j)	9	7
5	3	1	4	2 (j)	8 (i)	9	7
swap of pivot no As j crosses i							
2	3	1	4	5	8	9	7
2	3 (i)	1	4 (j)		8	9 (i)	7 (j)

2	3	1	4		8	7	9	
	(i)	(j)				(i)	(j)	
2	1	3	4		8	7	9	
	(j)	(i)				(j)	(i)	
1	2	3	4		7	8	9	
1					7			
		3	4				9	
			(i j)					
			4					
1	2	3	4	5	7	8	9	
Sorted Array:								

Fig: Array's transformations with pivots.

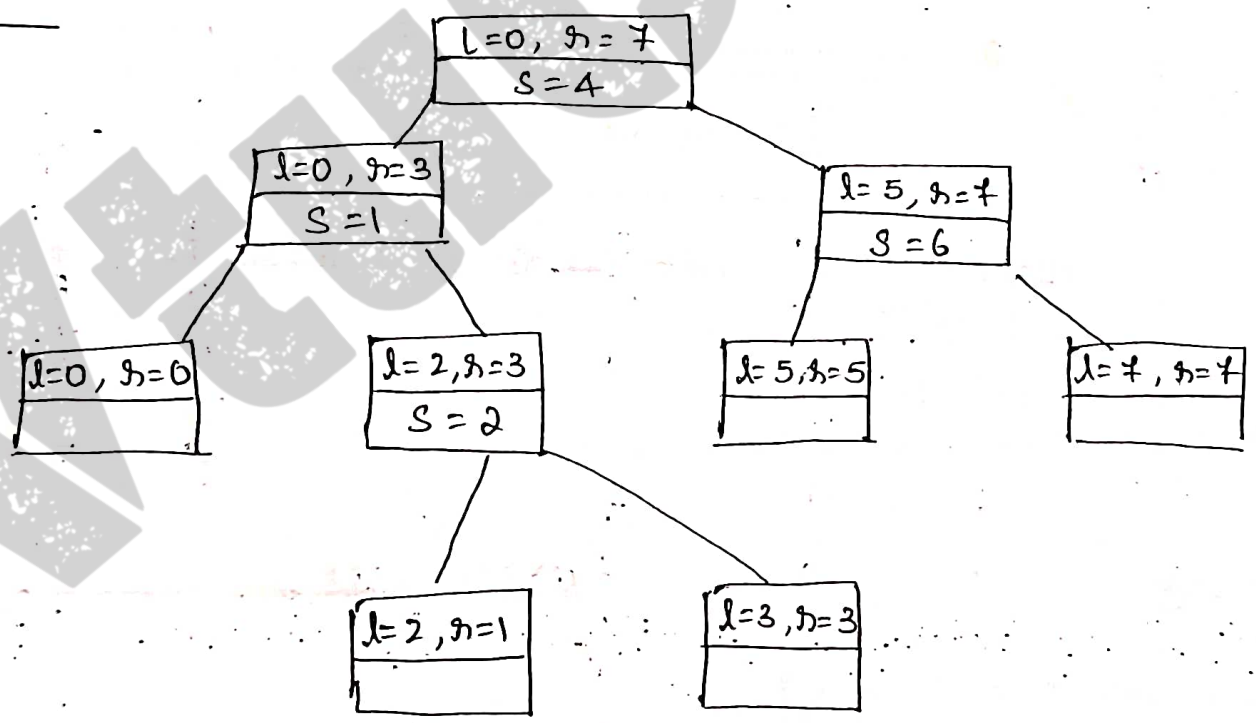


Fig: Tree of recursive calls to quicksort with input values l and r of subarray bounds & split position s of a partition obtained.

Analysis:

* We start our discussion of quick sort's efficiency by noting that the no of key comparisons made before a partition is achieved is n+1, if the scanning indices cross over and n if they coincide.

Best case:

If all the splits happen in the middle of corresponding subarrays, we will have the best case. The no of comparisons in the best case satisfies the recurrence:

$$C_{best}(n) = 2 \cdot C_{best}(n/2) + n \text{ for } n > 1;$$
$$C_{best}(1) = 0.$$

According to Master theorem, $C_{best}(n) \in (n \log^2 n)$;

Solving it exactly for $n = 2^k$ yields, we get

$$\boxed{C_{best}(n) = n \log 2 \cdot n}$$

Worst case:

All the splits, will be skewed to the extreme:
one of the 2 subarrays will be empty and the size of the other will be just 1 less than the size of the subarray being partitioned.

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.

The total no of key comparisons made will be equal

$$C_{worst}(n) = (n+1) + n + \cdots + 3$$

$$= (n+1)(n+2)\,2 - 3 \in \Theta(n^2)$$

Average case:

Let $C_{avg}(n)$ be the avg no of key comparisons made by the quicksort on a randomly ordered array of size n

A partition can happen in any position s ($0 \le s \le n-1$) after $n+1$ comparisons are made to achieve the partition

After the partition, the left and right subarrays will have s and $n-1-s$ elements, respectively.

Assuming that the partition split can happen in each position s with the same probability $1/n$, we get the following recurrence relation:

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)]$$

for $n > 1$.

$C_{avg}(0) = 0$, $C_{avg}(1) = 0$.

Its solution, which is much trickier than the worst-case analyses, turns out to be

$$C_{avg}(n) \approx 2n \ln n \approx 1.39\, n \log_2 n.$$

Analysis according to master theorem:

Best-case:

The best case of quick sort occurs, when the problem's instance is divided into 2 equal parts on each recursive call of the of algorithm. So the recurrence relation will be.

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ T(n/2)+T(n/2)+n & \text{otherwise // it is written as } 2T(n/2)+n \end{cases}$$

$\therefore a=2, b=2, f(n)=n=n^1=n^d$, so $d=1$.

So, according to master theorem $T(n)$ is given by

$$\therefore T(n) = \begin{cases} O(n^d) & \text{if } a<b^d \\ O(n^d \log bn) & \text{if } a=b^d \\ O(n^{\log_b a}) & \text{if } a>b^d \end{cases}$$

For the quick sort algorithm, $a=b^d$ holds good.

$\therefore T(n) = O(n^d \log n) = O(n^1 \log_2 n) \in O(n \log n)$.

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

Worst case:

The worst case of the quick sort occurs, when at each invocation of the quick sort algorithm, the current array is partitioned into 2 subarrays with one of them being empty.

This situation occurs when the input list is arranged in either ascending or descending order.

Ex: 10 11 12 13.

10 11 12 13

10 11 12 13

10 11 12 13

The recurrence relation for the above situation is:

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

$T(0)$ → To sort left subarray; $T(n-1)$ → To sort right subarray.

$$\therefore T(n) = T(0) + T(n-1) + C_n$$

$$T(n) = T(n-1) + C_n \quad // T(0) = 0$$

By the method of backward substitution,

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

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

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

$$= T(n-4) + C(n-3) + C(n-2) + C(n-1) + C_n$$

$$\vdots$$

$$= T(n-i) + C(n-i-1) + (n-i-2) \cdots + (n-3) + (n-2) + (n-1)$$

$$= T(n-n) + C[(n-(n-1)) + (n(n-2)) + \cdots + (n-3) + (n-2) + (n-1) + n]$$

$$= T(0) + C[1 + 2 + 3 + 4 \cdots + (n-3) + (n-2) + (n-1) + n]$$

$$= C[1 + 2 + 3 + 4 + \cdots (n-3) + (n-2) + (n-1) + n] \quad // T(0) = 1$$

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

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

Average Case Analysis:

The average case of the quick sort will appear for typical or random or real time input.

* Where the given array may not be exactly positioned into 2 equal parts as in the best case or the sub-arrays are not skewed as in the worst case.

Here, the pivot element is placed at an arbitrary position from 1 to n.

Let k be the position of the pivot element, as shown,

Then, $T(n) = T(k-1) + T(n-k) + (n+1)$

$T(n) = (n+1) \; 2 \log(n+1) = n \log n$

$$T(n) = \in O(n \log n)$$ //

Binary Tree and Traversals:

* 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, left and right subtree of the root.

* Since the definition itself divides a binary tree into 2 smaller structures of the same type, the left subtree and the right subtree, many problems about binary trees can be solved by applying the divide-and-conquer technique.

Ex:

Let us consider a recursive algorithm for computing the height of a binary tree.

* The height is defined as the length of the longest path from the root of a leaf, it can be computed ~~as the~~ maximum of the heights of the root's left and right subtrees plus 1.

Algorithm:

```
Height (T)
// computes recursively the height of a binary tree
// Input: A binary tree T
// output: The height of T.
if T = ∅ return -1
else return max {Height (T_left), Height (T_right)}.
```

* Measure the problem's instance size by the no. of nodes $n(T)$ in a binary tree T.

* The No. of comparisons made to compute the maximum of 2 nos and the number of additions $A(n(T))$ made by the algorithm are the same. ∴ The recurrence relation for $A(n(T))$:

$$A(n(T)) = A(n(T_{left})) + A(n(T_{right})) + 1$$

for $n(T) > 0$

$$A(0) = 0.$$

Note:

* For a empty tree, the comparison $T = \phi$ is executed once but there are no additions, and for a single node tree, the comparison and addition numbers are 3 & 1, respectively.

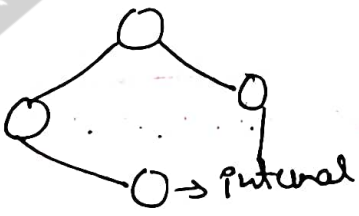
---

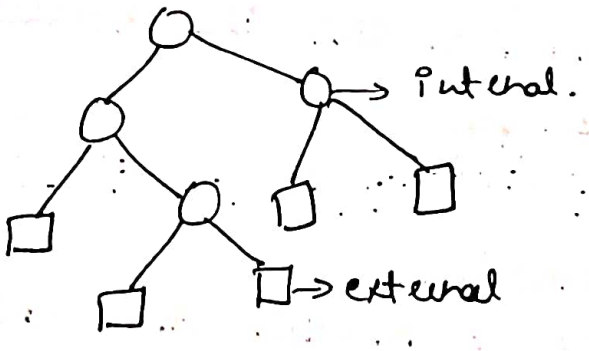
* It helps in the analysis of tree algorithms to draw the tree's extension by replacing the empty subtrees by special nodes.

The extra nodes are called external.

The original nodes are called internal.

* The extension of the empty binary tree is a single external node.





* It is easy to see that the Height algorithm makes exactly one addition for every internal node of the extended tree, and it makes one comparison to check whether the tree is empty for every internal node & external node.

* From, the above diagram and many other egs, it is easy to hypothesize that the no of external nodes x is always 1 more than the no of internal nodes n.

$$x = n+1.$$

* To prove this equality, consider the total no of nodes, both internal and external.

* Since every node, except the root, is one of the 2 children of an internal node, we have eq:

$$2n+1 = x+n.$$

* The no of comparisons to check whether the tree is empty is

$$C(n) = n + x = 2n+1.$$

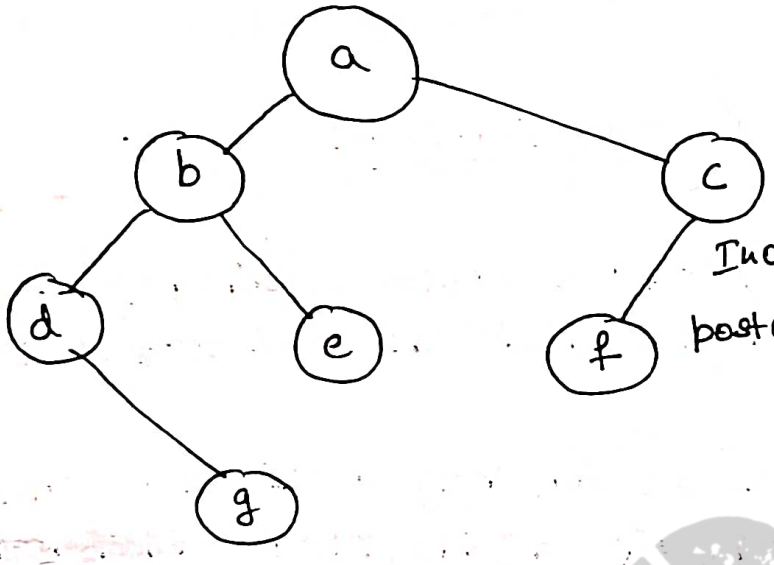
and the no of addition is $A(n) = n$.

The most important divide-and-conquer algorithms for binary trees are the 3 classic traversals:

1) Preorder
2) Inorder
3) Postorder.

All the 3 traversals visit nodes of a binary tree recursively, i.e by visiting the tree's root and its left and right subtree. They differ only by the timing of the root's visit.

In pre-order traversal, the root is visited before the left and right subtree are visited.



Preorder: a, b, d, g, e, c, f

Inorder: d, g, b, e, a, f, c

postorder: g, d, e, b, f, c, a.

* In the inorder traversal, the root is visited after visiting its left subtree but before visiting the rightsubtree.
* In the postorder traversal, the root is visited after visiting the left and right subtrees (in that order).
* As to their efficiency analysis, it is ~~identical to the above~~ analysis of the height algorithm because a recursive call is made for each node of an extended binary tree.

Vtucode

Multiplication of Large Integers and Strassen's Matrix Multiplication.

→ Multiplication of Large Integers:

* Some applications, notably modern cryptography, require manipulation of integers that are over 1000 decimal digits long.
* Since such integers are too long to fit in a single word of a modern computer, they require special treatment.
* If we use the conventional pen and pencil algorithm for multiplicating 2 ~~non~~ n-digit integers, each of the n digit of the first no is multiplied by each of the n digits of the second no for the total of n^2 digit multiplication
* To demonstrate the basic idea of the algorithm, let us start with a case of 2-digit integers, say 23 & 14.

$\therefore 23 = 2 \cdot 10^1 + 3 \cdot 10^0$ & $14 = 1 \cdot 10^1 + 4 \cdot 10^0$

Now, let us multiply,

$$23 * 14 = (2 \cdot 10^1 + 3 \cdot 10^0) \times (1 \cdot 10^1 + 4 \cdot 10^0)$$

$$= (2 \times 1) 10^2 + (2 * 4 + 3 \times 1) 10^1 + (3 \times 4) 10^0.$$

The last formula yields the correct answer of 322, but it uses the same four digit multiplications as the pen-and-pencil algorithm.

* Fortunately, we can compute the middle term with just one digit multiplication by taking advantage of the products 2×1 and 3×4 that need to be computed:

$$2 \times 4 + 3 \times 1 = (2 + 3) * (1 + 4) - 2 \times 1 - 3 \times 4.$$

* For any pair of 2-digit nos $a = a_1 a_0$ and $b = b_1 b_0$, their product c can be computed by the formula,

$$C = a * b = C_2 10^2 + C_1 10^1 + C_0,$$

where,

$C_2 = a_1 * b_1$ is the product of first digits.

$C_0 = a_0 * b_0$ is the product of their second digit.

$C_1 = (a_1 + a_0) * (b_1 + b_0) - (C_2 + C_0)$ is the product of the sum of the a's digit & the sum of the b's digit minus the sum of C_2 & C_0.

* When multiplying 2 n-digit integers a and b where n is positive even no.

* Let's divide both no in the middle — divide & conquer rule.

* So, a's digit → first half — a_1 & second half is a_0 same for b also.

* $a = a_1 a_0 \Rightarrow a = a_1 10^{n/2} + a_0$ &

$b = b_1 b_0 \Rightarrow b = b_1 10^{n/2} + b_0$.

$$\therefore C = a * b = (a_1 \cdot 10^{n/2} + a_0) * (b_1 \cdot 10^{n/2} + b_0)$$

$$= (a_1 * b_1) 10^n + (a_1 * b_0 + a_0 * b_1) 10^{n/2} + (a_0 * b_0)$$

$$C = C_2 10^n + C_1 10^{n/2} + C_0$$

where,

$C_2 \rightarrow a_1 * b_1 \rightarrow$ is the product of first halves

$C_0 = a_0 * b_0 \rightarrow$ is the product of second halves.

$C_1 = (a_1 + a_0) * (b_1 + b_0) - (C_2 + C_0)$ is the product of the sum of the a's halves and the sum of the b's halves minus the sum of C_2 and C_0.

* If $n/2$ is even, we can apply the same method for computing the products C_2, C_0 and C_1.

* Thus if n is a power of 2, we have a recursive algorithm for computing the product of 2-digit integers. In its pure form, the recursion is stopped when n ~~no of that~~ becomes 1.

* Since multiplication of n-digit nos requires three multiplications of $n/2$-digit nos, the recurrence for the no of multiplication $M(n)$ is:

$$M(n) = 3M(n/2) \quad \text{for } n > 1, \quad M(1) = 1.$$

Solving it by backward substitution for $n = 2^k$ yields

$$M(2^k) = 3M(2^{k-1})$$
$$= 3[3M(2^{k-2})]$$
$$= 3^2 M(2^{k-2})$$
$$\cdots$$
$$= 3^i M(2^{k-i})$$
$$= 3^k M(2^{k-k}) = \underline{\underline{3^k}}$$

Since $k = \log_2 n$,

$$M(n) = 3^{\log_2 n}$$
$$= n^{\log_2 3} \approx \underline{\underline{n^{1.585}}}$$

For addition & subtraction:

Let $A(n)$ be the no of digit additions & subtractions executed by the above algorithm in multiplying 2 n-digit decimal integers.

* Besides $3A(n/2)$ of these operations needed to compute the three products of $n/2$-digit no. The formula requires 5 additions & one subtraction. Hence, the recurrence is

$$A(n) = 3A(n/2) + cn \quad \text{for } n > 1, \quad A(1) = 1.$$

By Master Theorem, we obtain

$$\boxed{A(n) \in \Theta(n^{\log_2 3})}$$

Strassen's Matrix Multiplication

Ex: (By Divide-and-conquer Algo).

Given:

$$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2.$$

The idea is to decrease the no of multiplication from 4 to 3

$$(A_1 + A_2) * (B_1 + B_2) = A_1 B_1 + (A_1 * B_2 + A_2 * B_2) + A_2 B_2.$$

i.e, $(A_1 * B_2 + A_2 * B_1) = (A_1 + A_2) * (B_1 + B_2) - A_1 B_1 - A_2 B_2,$

which requires only 3 multiplications at the expense of (4-1) extra add/sub.

Recurrence for the no of multiplications $M(n)$:

$$M(n) = 3M(n/2), \quad M(1) = 1.$$

Soln: $M(n) = 3^{\log_2 n} = n^{\log_2 3} = n^{1.585}$.

Ex: $2135 * 4014$

It requires 9 digit multiplications as opposed to 16.

$$= (21 * 10^2 + 35) * (40 * 10^2 + 14)$$

$$= (21\ 40) * 10^4 + C_1 * 10^2 + 35\ 14$$

where $C_1 = (21 + 35) * (40 + 14) - 21\ 40 - 35\ 14$, &

$$(21 * 40) = (2 * 10 + 1) * (4 * 10 + 0)$$

$$= (2\ 4) * 10^2 + (2 * 10 + 1\ 0$$

where $L_2 = (2+1) * (4+0) - 2\ 4 - 1\ 0$, etc

Strassen's Matrix Multiplication:

By: Divide and conquer Method:

* Multiplication of 2x2 matrices: By using divide & conquer method or approach, we can reduce the no of multiplication such an algorithm was published by V. Strassen in 1969.

* The principal insight of the algorithm lies in the discovery that we can find the ~~opposed to the~~ product of C of 2 2x2 matrix A & B with just seven multiplication as opposed to the eight required by the brute-force algorithm.

* This is accomplished by using the following set formulas:

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{bmatrix}$$

where,

$M_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$,

$M_2 = (a_{10} + a_{11}) * b_{00}$,

$M_3 = a_{00} * (b_{01} - b_{11})$,

$M_4 = a_{11} * (b_{10} - b_{00})$,

$M_5 = (a_{00} + a_{01}) * b_{11}$,

$M_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$,

$M_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$.

* Thus, to multiply 2x2 matrices, Strassen's algorithm makes seven multiplications & 18 addition / subtractions, whereas, the brute-force algorithm requires 8 multiplications & four additions.

Multiplication of n x n matrices:

Let A & B be 2 n x n matrices where n is a power of 2. (If n is not a power of 2, matrices can be padded with rows & columns of zeros).

* We divide A, B & their product C into four $n/2 * n/2$ submatrices each as follows:

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

* It is not difficult to verify that one can treat these submatrices as no to get the correct product.

Ex:

C_{00} can be computed as either as $A_{00} * B_{00} + A_{01} * B_{10}$ or as $M_1 + M_4 - M_5 + M_7$, ~~If the seven products with the no replaced by the corresponding submatrices~~

+ If the 7 products of $n/2 * n/2$ matrices are computed recursively by the same method, we have Strassen's algorithm for matrix multiplication.

Analysis:

If $M(n)$ is the no of multiplications made by strassen's algorithm in multiplying 2 $n \times n$ matrices, we get the following recurrence relation for it,

$$M(n) = 7M(n/2) \quad \text{for } n>1, \; M(1) = 1.$$

Since $n = 2^k$,

$$M(2^k) = 7M(2^{k-1}) =$$
$$= 7[7M(2^{k-2})]$$
$$= 7^2 M(2^{k-2}) = \ldots$$
$$= 7^i M(2^k i) = \ldots$$
$$= 7^k M(2^{k-k})$$
$$= 7^k$$

Since $k = \log_2 n$,

$$M(n) = 7^{\log_2 n}$$
$$= n^{\log_2 7}$$
$$M(n) \approx n^{2.807}$$

This implies $M(n) \approx O(n^{2.807})$ which is smaller than n^3 required by the brute-force algorithm.

Vtucode