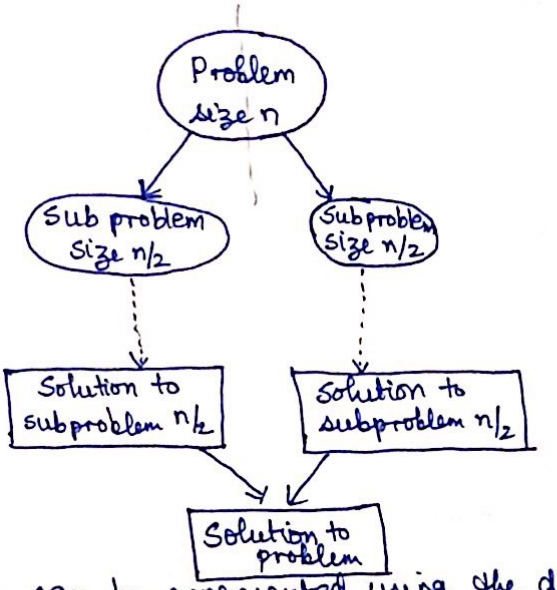
Divide & Conquer

Divide & conquer is the most common problem solving design technique.
Here, the given problem is divided into sub-problems of equal size which are recursively solved and the solutions of the sub-problems are merged (combined) to get the solution to the actual problem.



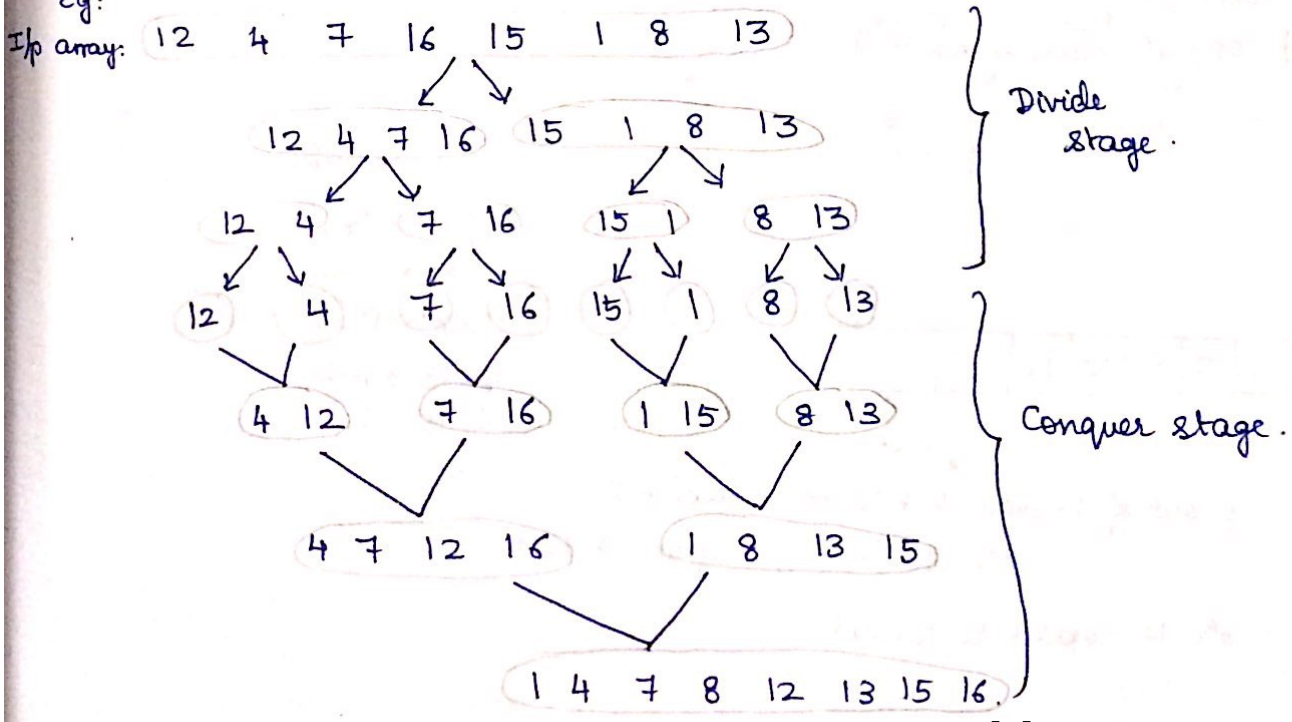
The divide & conquer can be represented using the diagram given.
used for: Merge sort, quick sort, binary trees, matrix multiplication

Merge Sort

Steps used in merge sort are:

Step 1: The given array is divided into two equal halves
Step 2: Recursively sort the left half
Step 3: Recursively sort the right half
Step 4: Merge the sorted left & right array into a single array.

Eg:



Algorithm to perform Merge sort

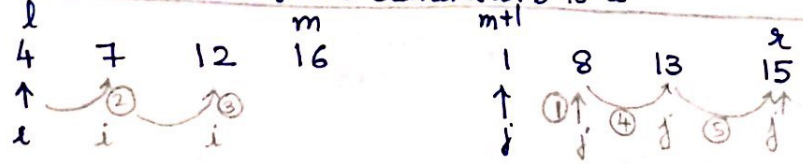
```
Algorithm MergeSort(A, l, r)        → index of 1st el
                                    → index of last el
//Recursively sorts the element using merge sort
//Input: An unsorted array A, with index of first element marked as 'l'
// index of last element marked as 'r'.
//Output: A sorted array A

if (l<r)
    m = (l+r)/2;
    MergeSort (A, l, m);
    MergeSort (A, m+1, r);
    Merge (A, l, m, r);
```

Algo. to merge two sorted arrays

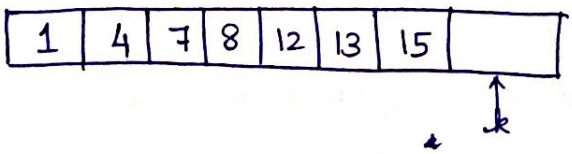
```
Algorithm Merge (A, l, m, r)
//Merges two sorted arrays together
//Input: 2 sorted arrays A[l...m] & A[m+1....r]
//Output: A merged sorted array
 i ← l
 j ← m+1
 k ← l
 while (i<=m and j<=r)
     if A[i] < A[j]                    // comparison
         b[k++] = a[i++]
     else
         b[k++] = a[j++]
 while i<=m
         b[k++] = a[i++]          } Assignment only.
 while j <= r
         b[k++] = a[j++]
 { copy the elements in b to a
```

if starting index is 0, replaces array when called recursively.

```
for(i=l; i<=r; i++)
 a[i] = b[i]
```



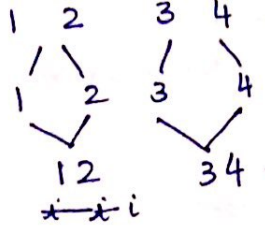
4<1 → False.
4<8 → True.
7<8 → True
12<8 → False
16<13 → False
16<15 → False



j out of bounds, but 'i' not completed.

∴ shift all 'i' from a to b.

∴ 16 copied to array b.



1 2

i out of bounds.

move 'j' from curr. loc^n to r.

∴ B : 1 2 3 4

A : 1 2 3 4.

Basic operation

Comparison (if(a[i]<a[j]))

Analysis of merge sort. 1 comparison for 2 i/p elements

The basic operation in merge sort is the comparison.

Let C(n) denotes the total number of comparisons done/carried to sort 'n' elements.

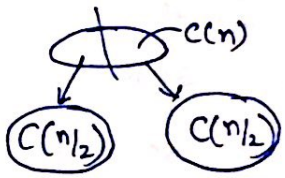
if (n=1) BASE CASE.

$C(n) = 0$

i.e $C(1) = 0$

if (n>1)

$C(n) = 2C(n/2) + C_{merge}^{(n)}$



↳ cost reqd. to merge 'n' elements
i.e., no. of merge opⁿ for all 'n' elements
total no. of comparisons

In worst case,

$C_{merge}^{(n)} = n-1$ comparisons

∴ $C(n) = 2C(n/2) + (n-1) = C_{worst}(n)$.

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

$f(n) = n^{\log_b a} \cdot (\log n)^k$ $k=0$

$n \cdot \log n^{k+1} = n+1$

$a = 2, b = 2$ $2C(n/2)$ has higher order of growth than $f(n)$.

$f(n) = n$.

$n^{a \log_b a} = n^{\log_2 2} = n^1 = n$. $\Theta(n^{\log_b a} \cdot \log n^{k+1})$

$C_{worst}^{(n)} = \Theta(n \log n)$.

$n^1 \cdot \log n^{0+1} = n \log n$.

$C_{worst}(n) = n \log n - n + 1$

$n \log n > n$ & > 1.

∴ $C_{worst}(n) = n \log n$.

If assignment is considered as the basic operation, then basic operation of merge sort in all three cases would be $n \log n$.

For the ascending & descending set of elements, if comparison is taken as the basic operation, then the count of ascending & descending is less than the random set.

If, the basic operation was considered to be assignment (increase count before 'if', & inside both 'while'), then, the count would be same for all 3 cases.

$C(n) = n \log n$.

$\Rightarrow$ if $n = 8$,

$C(n) = 24$.

(No. of comparisons in ascending/descending < No. of comparisons for random).

Eg Best case: 1 2 3 4 5 6 7 8

Worst case: 1 3 5 7 2 4 6 8

Q: Sort all the characters in the word quicksort, using merge sort

0 1 2 3 4 5 6 7 8

① Q U I C K S O R T m = 4.

∴ must do merge sort of:

② Q U I C K S O R T

③ Q U I C K S O R T.

④ Q U I C K S O R T.

⑤ Q U I C K S O R T.

At step 5, for all, l = r, ∴ fn only returns to calling fn.

from 5 to 4:

Q U I C K S O R T

4 to 3:

Q U I C K S O R T.

I Q U C K O S R T.

Array A: I Q U C K O S R T

if out of bound, copy rest

if out of bounds after placing O copy rest

Merge on ① I Q U C K & ② O S R T.

C I K Q U & O R S T

Merge: C I K Q U O R S T

C I K O Q R S T U

∴ Final array:

C I K O Q R S T U.

Quick Sort — Most efficient sorting algo.

It is based on divide & conquer, may not be two equal halves, uses concept of partition

We take an element (usually first) as a reference (pivot) & after partition we place the pivot in its actual position in the sorted array such that: ~~elements to~~

- elements to the left of pivot ≤ to pivot
- elements to the right of pivot > than ~~pivot~~ pivot.

Eg pivot

10 5 12 1 17 15 23.

↓ opn.

5 1 [10] 12 17 15 23 } unequal parts.

sub-parts may not be sorted.

In randomized quick sort, any pivot can be taken } Not in syllabus.

Usually, first or last element are considered as pivot.

Partition

Index of first = l, Index of last = r.

∴ I/p → A [l...r]

pivot = A[l]

i = l+1

j = r

if pivot ≥ a[i] —true→ i++.

else

if pivot < a[j] —true→ j--

else if i<j —true→ swap(a[i], a[j])

↓ false

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

return j.

1 5 [10] 12 17 15 23

[1] 5 [10] [12] 17 15 23

[1] 5 [10] [12] 15 [17] 23

l, i, j, r.

10 5 12 1 17 15 23

10 ≥ 5 ∴ i shifts ahead.

10 ≱ 12, ∴ j--

10 ~~≥~~ < 23 j--

10 < 15 j--

10 < 17 j--

10 < 1 NO

i<j yes.

swap 12 & 1.

10 5 1 12 17 15 23.

10 ≥ 1 ✓ ∴ i++.

10 < 12 ✓ ∴ j--.

10 5 ~~12~~ 1 12 17 15 23

10 ≱ 12, 10 ≮ 1.

i ≮ j

∴ swap 10 & a[j]

swap 10 & 1.

∴ 1 5 10 12 17 15 23.

In quick sort, which is based on divide & conquer, to divide the problem instance, we use partition.

In partition, we generally take either the first element or the last element as pivot and at the end of the partition, we are trying to place the pivot element in its actual position in the sorted array.

```
Algorithm quicksort(A, l, r)
// Sorts a set of elements recursively using quick sort
// Input: An array A, which holds the element to be sorted, indexed from l to r
// Output: A sorted array
    if l < r
      s ← partition(A, l, r)
      quicksort(A, l, s-1)
      quicksort(A, s+1, r)

Algorithm partition(A, l, r)
// Partitions a sub-array A indexed from l to r.
// Input: A sub-array A
// Ouput: It places the pivot element in its actual position in sorted array A
      pivot ← A[l]
       i ← l+1
       j ← r
       while True
            while pivot ≥ A[i] & i ≤ r  increment i
            while pivot < A[j]  decrement j
            if (i < j)
                swap(A[i], A[j])
            else
                swap(A[l], A[j])   → DO NOT swap with pivot
                return j.
```

No additional/temporary array reqd.

∴ Quicksort is known as "in-place sorting technique" as it does not use extra memory to sort.

Me

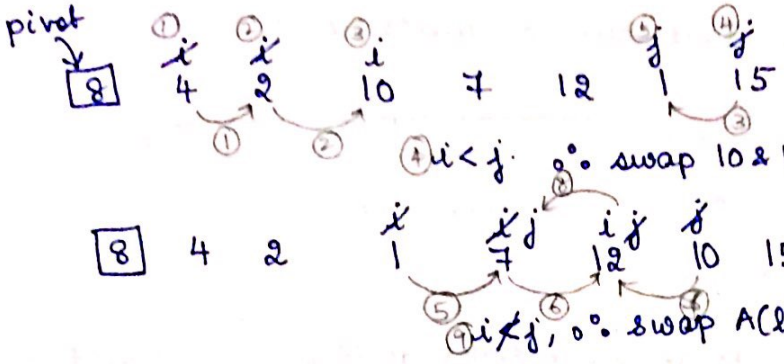
Basic operation: Comparison.

N

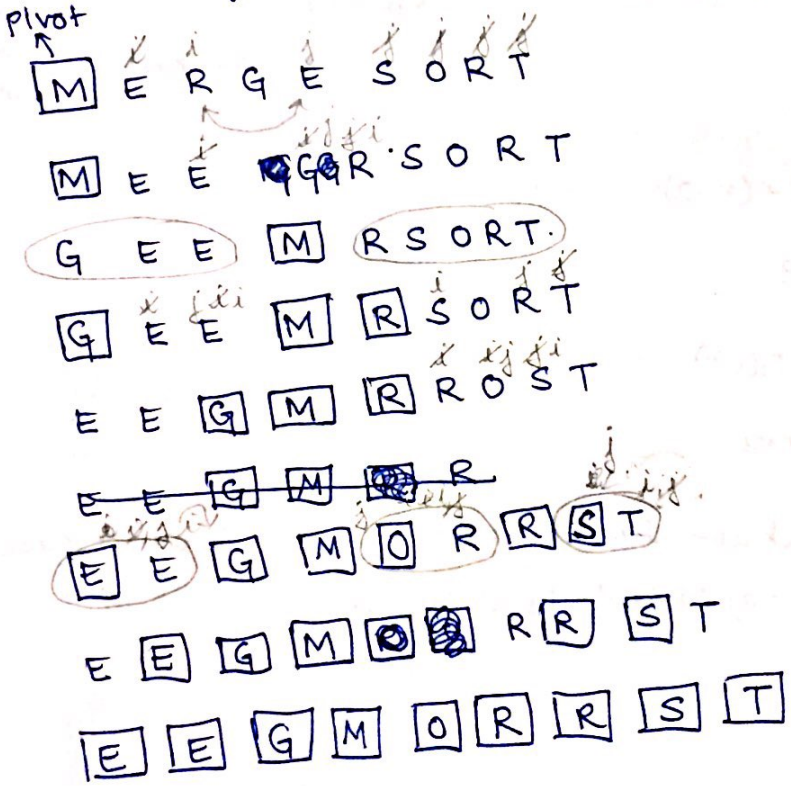
```
pivot ≥ A[i]
pivot < A[j]   } increment count.
```

Eg: Trace quicksort for the elements

8 4 2 10 7 12 1 15



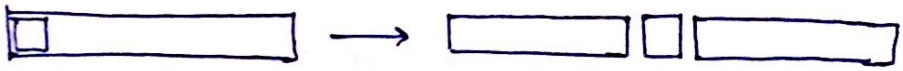
Sort the characters of the word mergesort using quick sort.



∴ Final array A: EEGMORRST.

Efficiency:

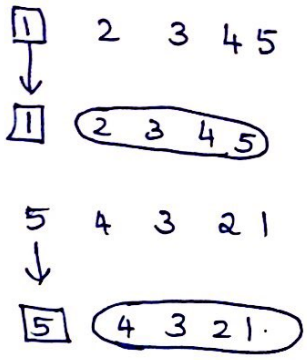
If the pivot element divides the array into two equal parts, then we come across the best case of quick-sort.



This has value same as efficiency of merge sort.

$\therefore T_{best}(n) = \Theta(n \log n)$.

If the input supplied to the quick sort algorithm is a sorted array or if the elements are in the reverse (descending) order, then, it exhibits the worst case.



The recurrance relation exhibiting the worst case of quicksort can be denoted as $T(n) = T(n-1) + \text{no. of comparisons}$

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

Eg. $n = 2$

3 comparisons.

1 2

1 with 2 → twice

1 with 1 → once

$\therefore 3$.

By substitution:

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

$= \frac{(n+1)(n+2)}{2} - 3$.

$\approx \Theta(n^2)$,i.e. $O(n^2)$.

if $n = 5$, 18 comparisons.

The average case of quicksort lies in between the best & worst case, which is actually equivalent to $\sim 1.38 n \log n$.

Multiplication of two large integers

In brute force, if 'n' denotes the size of a & b, the no. of multiplication operations is n^2. (traditional method of multiplying two 'n' digit numbers).

Eg. To multiply 2 two-digit numbers, we require 4 operations.

Similarly, to multiply 2 three-digit numbers, we require 9 operations.

By using the concept of divide & conquer, we can reduce the time complexity from n^2 to $n^{1.59}$.

Let a & b be two n-digit numbers

& the objective is to find c, as $c = a \times b$.

Let a be represented as

$$a = a_1 a_0$$

& b be represend as

$$b = b_1 b_0$$

where, a_0, a_1, b_0, b_1 have a size $n/2$. (divide each term into two parts).

The product 'c' is calculated with the following steps:

Step 1 $\rightarrow$ $c_0 = a_0 \times b_0$ — 1 multiplication opern.

Step 2 $\rightarrow$ $c_2 = a_1 \times b_1$ — 1 multiplication opn.

Step 3 $\rightarrow$ $c_1 = (a_0 + a_1) * (b_0 + b_1) - (c_2 + c_0)$. — 1 multiplication opn.

Step 4 $\rightarrow$ $c = c_2 10^n + c_1 10^{n/2} + c_0$ — Shift operation.

If M(n) is no. of multiplication operations.

$\therefore$ $M(n) = 3M\left(\frac{n}{2}\right)$ + time spent in addn, subn & shift.

Each operation is performed on a size of $n/2$

$f(n)$ is asymptotically slower than $n^{\log_b a}$.

$\therefore$ $M(n) = \Theta(n^{\log_2 3})$

$= \Theta(n^{1.59})$

Eg.

① $a = 23$ $\quad a_1 = 2,\ a_0 = 3$

$b = 46$ $\quad b_1 = 4,\ b_0 = 6$

$c_0 = a_0 b_0 = 3 \times 6 = 18$

$c_2 = a_1 b_1 = 2 \times 4 = 8$

$c_1 = (2+3) * (4+6) - (18+8)$

$= 5 \times 10 - 26$

$= 50 - 26$

$= 24.$

$c = 8 \times 10^2 + 24 \times 10 + 18$

$= 800 + 240 + 18$

$= 1058.$

② Compute 236×1212 using divide & conquer.

$a = 0236 \quad a_1 = 2,\ a_0 = 36$

$b = 1212 \quad b_1 = 12,\ b_0 = 12.$

$c_0 = a_0 b_0 = 36 \times 12 = 432$

$c_2 = a_1 b_1 = 2 \times 12 = 24.$

$c_1 = (2+36)(12+12) - (432+24)$

$= 38 \times 24 - 456$

$= 456.$

$c = c_2 \times 10^n + c_1 \times 10^{n/2} + c_0$

$c = 24 \times 10^4 + 456 \times 10^2 + 432$

$= 286032.$

Analysis:

The recurrance relation to multiply two n-digit numbers using divide & conquer can be represented as

$M(n) = 1$ if $n = 1$ (Base case)

else,

$M(n) = 3M\left(\frac{n}{2}\right) + cn$ → time to add / subtract & shift

$M(n) = \begin{cases} 1 & \text{if } n = 1 \\ 3M\left(\frac{n}{2}\right) + cn & \text{otherwise, } n > 1 \end{cases}$

$= 3M(n/2) = 3M(2^{k-1})$ if $n = 2^k$

$= 3^2 M(2^{k-2})$

$= 3^3 M(2^{k-3})$

$\vdots$

$= 3^k M(2^{k-k})$

$= 3^k M(2^0)$

$= 3^k$

$\therefore$ if $n = 2^k$.

$k = \log_2 n$.

$\therefore M(n) = 3^{\log_2 n}$

$= n^{\log_2 3}$

$= n^{1.59}$

Alternatively, use Master Theorem.

Strassen's Matrix Multiplication

The brute force method of multiplying two matrices of order $n \times n$ yields a time complexity of n^3.

Using divide & conquer, Strassen has reduced the time complexity from n^3 to $n^{2.81}$

$$A = \begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 2 \\ 1 & 2 \end{bmatrix}$$

$C = A \times B$

$$= \begin{bmatrix} 1\times1+1\times1 & 1\times2+1\times2 \\ 2\times1+2\times1 & 2\times2+2\times2 \end{bmatrix}.$$

No. of multiplications

$= 8$

$= n^3$

(Here, $n=2$).

Multiplication is the basic operational step.
(Takes more time than add^n).

Let A & B denotes two matrices of the same order, represented as

$$A = \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix}, \qquad B = \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix}$$

The product

$C = A \times B$

is represented as: $\begin{bmatrix} C_1 & C_2 \\ C_3 & C_4 \end{bmatrix}$

where,

$C_1 = E + I + J - G$

$C_2 = D + G$

$C_3 = E + F$

$C_4 = D + H + J - F$

$D = A_1(B_2 - B_4)$

$E = A_4(B_3 - B_1)$

$F = (A_3 + A_4)B_1$

$G = (A_1 + A_2)B_4$

$H = (A_3 - A_1)(B_1 + B_2)$

$I = (A_2 - A_4)(B_3 + B_4)$

$J = (A_1 + A_4)(B_1 + B_4)$.

Brute force.

$$M(n) = \begin{cases} 1, & n = 1 \\ 8\,M\left(\frac{n}{2}\right), & n > 1 \end{cases}$$

$M(n) = n^{\log_2 8}$

$= n^3$.

Strassen's

$$M(n) = \begin{cases} 1, & n = 1 \\ 7\,M\left(\frac{n}{2}\right) + \text{time to add/sub}, & n > 1 \end{cases}$$

Case 1, $f(n)$ asymptotically slower

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

$= n^{2.81}$.

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix}$$

$A_1 = 1,\ A_2 = 2,\ A_3 = 3,\ A_4 = 4$

$B_1 = 1,\ B_2 = 1,\ B_3 = 2,\ B_4 = 2.$

$D = A_1(B_2 - B_4)$

$= 1(-1) = -1.$

$E = A_4(B_3 - B_1) = 4(2-1)$

$= 4$

$F = (A_3 + A_4)B_1$

$= (3+4)1 = 7$

$G = (A_1 + A_2)B_4$

$= (1+2)2$

$= 6$

$H = (A_3 - A_1)(B_1 + B_2)$

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

$I = (A_2 - A_4)(B_3 + B_4)$

$= (2-4)(2+2) = -2 \times 4 = -8.$

$J = (A_1 + A_4)(B_1 + B_4)$

$= (1+4)(1+2) = 15.$

$C_1 = E + I + J - G$

$= 4 - 8 + 15 - 6$

$= 5$

$C_2 = D + G$

$= -1 + 6 = 5$

$C_3 = E + F = 4 + 7$

$= 11$

$C_4 = D + H + J - F$

$= -1 + 4 + 15 - 7$

$= 11.$

$\therefore C = \begin{bmatrix} 5 & 5 \\ 11 & 11 \end{bmatrix}$

The recurrance relation for multiplying two matrices using divide & c can be represented as:

$$M(n) = \begin{cases} 1 & , n = 1 \\ 7M(n/2) + Cn & , \text{otherwise} \end{cases}$$

Cn time for add^n/sub

Using Master theorem on this relation,

we get time complexity as $n^{\log_2 7}$

$= n^{2.81}$.

DECREASE & CONQUER

Decrease & conquer is almost similar to divide & conquer, but, rather than dividing the problem into two equal halves, we decrease the size of the problem instance by a constant factor.

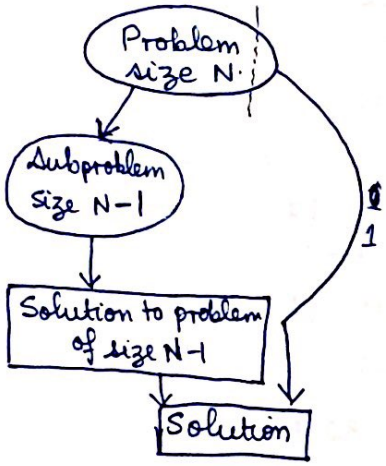
There are 3 variations of decrease & conquer

i) Decrease by a constant ("Decrease by 1") → Similar to traversal One node visited in each iteration.
   Eg. Insertion sort, DFS, BFS.

ii) Decrease by a constant factor → If factor = 2, similar to divide & conquer But, only one instance solved i.e, no merging done.
   Eg. Fake coin, Josephus problem. Eg. binary search.

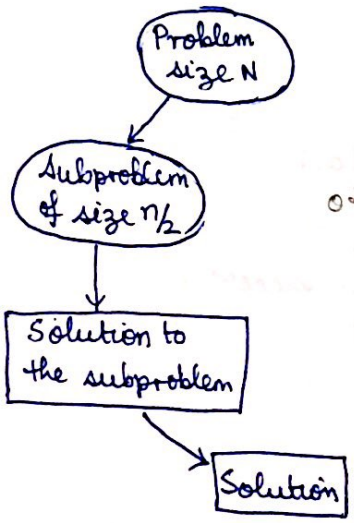
iii) Variable size decrease
   Eg. Find GCD by Euclid's algorithm,
   Find median by using partition.

The ~~algorithm~~ diagram to represent decrease by a constant.



Eg.
$a^n = a^{n-1} \cdot a$.

Diagram to represent decrease by a constant factor is.



Can be $n/2$, $n/3$, $n/4$ i.e, any constant factor.

other part of problem need not be solved

Eg.

$a^n = (a^{n/2})^2$; n is even.

$= (a^{\frac{n-1}{2}})^2 \cdot a$; n is odd.

Generally, in decrease & conquer, we exploit the relation between a solution to a given instance of the problem & a solution to the ~~smaller~~ instance of the problem

Insertion sort

Given A[0....n-1] to sort, assume A[0....n-2] is sorted.
∴ only posⁿ of A[n-1] has to be found.

Eg 12 2 7 54 32 17

Itern 1: size 1 → 12

1st iteration

2 → 12 2 2 12 } can be given as 1st iteration.
compare

2nd iteration

3 → 2 12 7

2 7 12 7>2, ∴ no exchange

3rd iteration

4 → 2 7 12 54 54>12,
no change & no more comparisons.

4th iteration

5 → 2 7 12 54 32

2 7 12 32 54 32>12.
no change & no more comparisons.

5th iterations

6 → 2 7 12 32 54 17

2 7 12 32 17 54

2 7 12 17 32 54

'n' elements need (n-1) iterations

Here, comparison done L to R.

Can be R to L.

Largest element pushed to last.

Binary search can be used

Get posⁿ by binary search & place accordingly.

Algorithm:

```
Algorithm InsertionSort (A[0....n-1])
// Sorts a given array using insertion sort
// Input: An array A, indexed from 0 to n-1
// Output: A sorted array A
for i ← 1 to n-1 do
    item ← A[i]
    j ← i-1
    while j ≥ 0 and A[j] > item
        A[j+1] ← A[j]
        j ← j-1
    A[j+1] ← item.
```

j	item					
12	2	7	54	32	17	j≥0 & 12>2 °° shift 12 to right & j → -1
2.	12	7	54	32	17	°° item at j+1 = at 0 j=1, 12>7, shift 12, decrease j.
						j=0, 2 not ≥ 7 °° out of while.
2	7	12	54	32	17	°° 7 at j+1, i.e 7 at 1.

etc.

Basic operation comparison (A[j] > item).

Trace the following i/p using insertion sort & find the efficiency (no. of times basic opn performed).

1	2	3	4	5	2 with 1, 1 ≯ 2 °° out of while.
1	2	3	4	5	3 with 2, 2 ≯ 3, places & steps no shifting.
1	2	3	4	5	4 with 3, 4 > 3 no shifting.
1	2	3	4	5	5 with 4, 5 > 4, no shifting.

°° (n-1) comparisons occur.

Analysis.

① Best case: The best case occurs when all the elements are in the sorted order. Here, the comparison A[j] > item is executed once in each iteration. This can be represented as

$$\sum_{i=1}^{n-1} 1 = n-1+1-1 = n-1 \text{ comparisons.}$$

$= \Theta(n)$ efficiency.

② Worst case

If all the elements in the i/p array are in the reverse order, then, the insertion sort exhibits the worst case.

Here, the comparison statement A[j] > item will be executed maximum number of times in each iteration,

which can be represented as:

$$= \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1$$

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

$$= \sum_{i=1}^{n-1} i$$

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

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

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

Average case:

$$(n-1) \leq avg \leq n^2.$$

$\Theta(n^2)$ is considered. $\because \frac{n^2}{4}, \frac{n^2}{8}$ also n^2.

Insertion sort is the only algorithm which has a linear time efficiency when the array is already sorted.

Trace the following i/p for Insertion Sort.
elements: 'c', 'o', 'm', 'p', 'u', 't', 'e', 'r'.

		No. of comparisons
	C O M P U T E R	
Iteration 1	C | O M P U T E R	1
Iteration 2	C O | M P U T E R	2
Iteration 3	C M O | P U T E R	1
Iteration 4	C M O P | U T E R	1
Iteration 5	C M O P U | T E R	2
Iteration 6	C M O P T U | E R	5
Iteration 7	C E M O P T U | R	3
	C E M O P R T U	

Graph Reversal

TREE TRAVERSAL

A root/ref. node is necessary in a tree traversal.

If tree is traversed in pre, in or post order, the traversal are unique.

Once a node is visited, we never visit again.

GRAPH TRAVERSAL

Root/ref node is not is not necessary in a tree traversal. (any node)

The traversal may not be unique.

Additional data structure to keep track of whether node is visited or not, because we may return to the same node.

A graph may be repd as:

- Array: "Adjacency matrix", 2D array.
- Linked list: "Adjacency list".

} possible forms of i/p.

'n' nodes in graph, n x n adjacency matrix. ∴ time efficiency of any algo (worst case) = n^2.

'n' nodes, adjacency list shows no. of vertices & no. of edges.

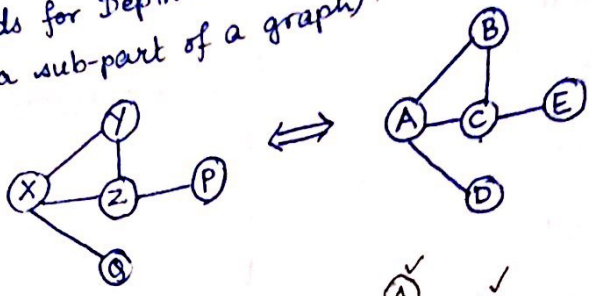
∴ time efficiency depends on |V| & |E|

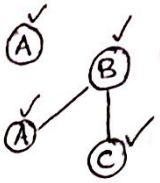
Basic/Most popular graph traversal

- DFS (Depth First Search)
- BFS (Breadth First Search)

DFS

This stands for Depth First Search.
(A tree is a sub-part of a graph)





Visit A, mark as visited. A_1

Visit not-visited edges. B_2

∴ visit B.

Visit neighbours of B C_3

(A already visited). ∴ visit C

Visit neighbours of C E_4.

∴ visit E

all neighbours have already been visited

E: 'dead end', no other nodes can be reached.

∴ $E_{4,1}$ ↳ first dead node.

Returns to C, dead end $C_{3,2}$.

Returns to B, dead end $B_{2,3}$

Returns to A,

has one non-visited node

∴ visits D. D_5.

Output: A_1 B_2 C_3 E_4 D_5

Types of

Tree edge : One visited node & one not-visited ~~edge~~ node.

Back edge: ↳ Connects two visited ~~edges~~ node.

~~Back edge~~ : Always connects node to its grandparent.

DFS uses a stack

* Node push order, popping order.

(BFS uses queue).

Algorithm DFS(G)

// Implements a depth-first search traversal on a graph G.

// Input: A graph G, represented as G = {V, E}

// Output: A graph G, with its vertices marked with consecutive integers in the order they have been first encountered by DFS algorithm.

Mark each vertex in V with 0 as a mark of being unvisited

count ← 0

one iteration per component ← for each vertex v in V

if v is marked with zero

dfs(v)

(Pop when node is dead end)

Algorithm dfs(v) → ∵ recursive, we say stack is used. Uses System Stack

// visits recursively all the unvisited vertices connected to vertex v & assigns them an integer number in the order they are encountered via variable count

count ← count + 1

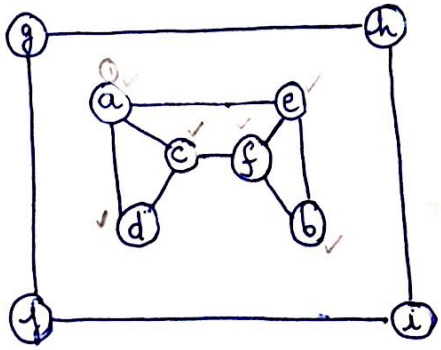
mark vertex v with count

for each vertex w in V adjacent to v

if w is marked with 0

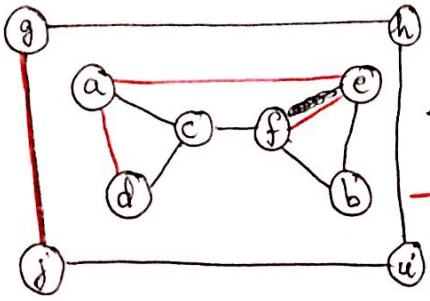
dfs(w)

Perform DFS Traversal on the given graph.



$a_{1,6}$
$c_{2,5}$
$d_{3,1}$
$f_{4,4}$
$b_{5,3}$
$e_{6,2}$

$g_{7,10}$
$h_{8,9}$
$i_{9,8}$
$j_{10,7}$



Tree edges.

Back edges.

Between nodes that have been executed denote by dashed lines.

Exam If asked, write push & pop order

Also, redraw graph with tree & back edges.

In case of a DFS traversal, we come across two types of edges.

1. Tree edge:

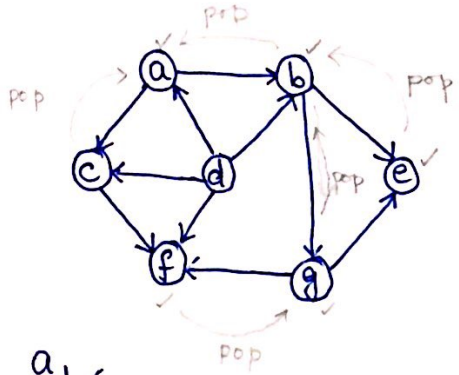
   The edge between a visited & an unvisited vertex are tree edges. (denoted with solid lines)

2. Back edges:

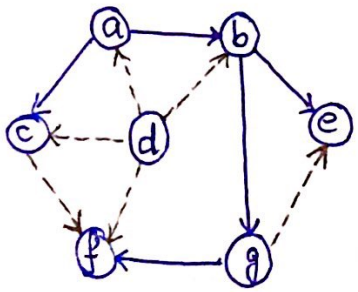
   Connects vertices to previously visited vertices other than the immediate predecessor in the traversal.
   (denoted with dashed lines)

A dead end are those nodes in the DFS Traversal from which we can't traverse to any more unvisited nodes.

Perform a DFS Traversal on the given graph.

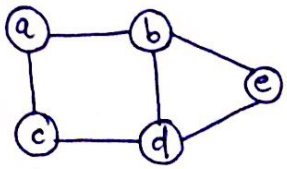


$a_{1,6}$
$b_{2,4}$
$c_{3,1}$
$g_{4,3}$
$f_{5,2}$
$c_{6,5}$
$d_{7,7}$ } for loop traversed again.

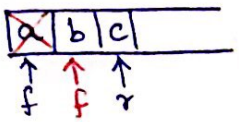


BFS

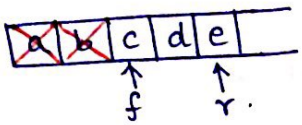
This stands for Breadth First Search
Data structure used is a queue.

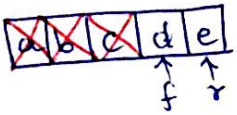


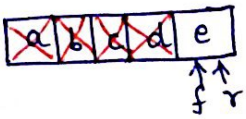
If starting vertex is 'a', place element in queue.
visit elements that have not been visited.

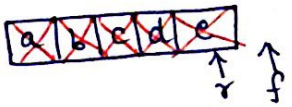


All unvisited nodes of 'a' are visited
then, 'a' is deleted.









∴ Queue empty, i.e, all nodes in tree are visited.

Appears similar to
level-order traversal

BFS is a popular graph traversal method, wherein we visit all the nodes in a given graph level-wise

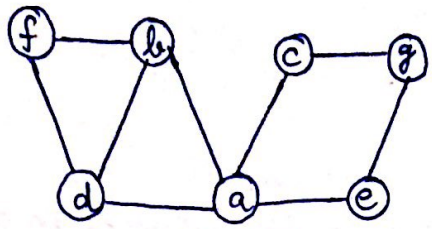
The general procedure that we follow in BFS are:

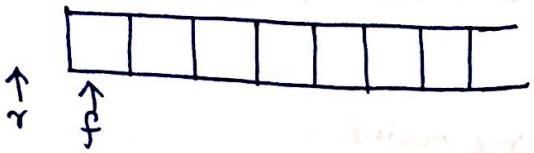
Step 1: Initialise a queue and insert the starting vertex onto the queue

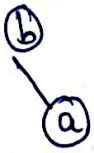
Step 2: Until queue becomes empty, delete a node and find the neighbours of the deleted node which are not visited, and insert them onto the queue.

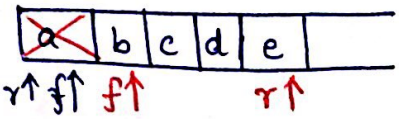
Step 3: Repeat step 2 until queue becomes empty.

Example

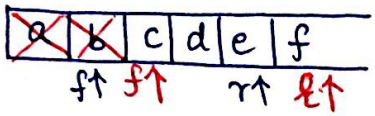




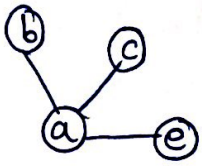


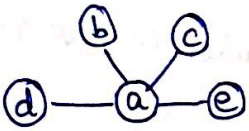


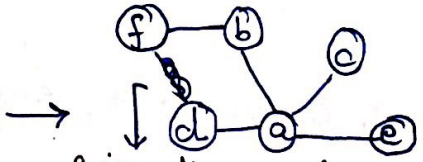
a, then its neighbours are placed in queue.



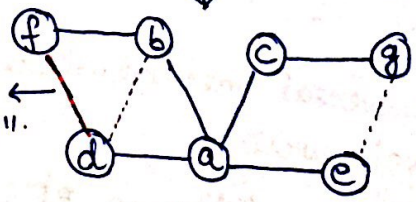
b & its neighbours are added.







f was discovered while exploring 'b'.



Dotted line ← "Cross edges".

Algorithm BFS(G)

//Implements a breadth first search traversal on a graph G

//Input: A graph $G = \{V, E\}$

//Output: A graph G, with its vertices marked with consecutive integers in the order of their visit.

Mark each vertex in V with zero, as a mark of being unvisited

count ← 0

for each vertex v in V do

if v is marked with zero

bfs(v)

Algorithm bfs(v)

//V visits all the unvisited vertices connected to vertex v, and assigns them the order in which they are visited via the global variable count

count ← count + 1

Mark v with count and initialise a queue with vertex v

while the queue is not empty do

for each vertex w adjacent to ● (w is in V) front vertex do

if w is marked with zero //w is unvisited.

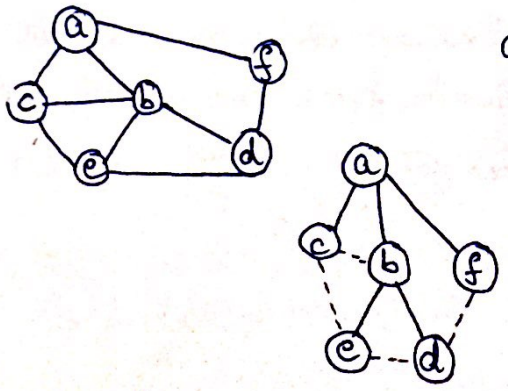
count ← count + 1

Add w to the queue

Remove a vertex ● from the front of the queue // remove front vertex from queue.

——— x ———

Perform a BFS traversal on the graph.



$a_1\ b_2\ c_3\ f_4\ d_5\ e_6$

	a	b	c	d	e	f
a	0	1	1	0	0	1
b	1	0	1	1	1	0
c						
d						
e						
f						

Any edge connecting edge of same level: cross edge.

~~Cross edges: The edges that connects two vertices which are in the same level~~

~~If in different levels, no edge drawn.~~

Cross edges: The edge that connects a visited vertex which is not the immediate predecessor of a given vertex is termed as the cross edge.

Common Q

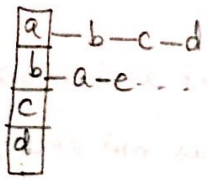
Difference between DFS & BFS.

DFS

1. Data structure used is stack
2. In DFS, we have tree & back edges
3. Two ordering: we maintain a note of order in which they are pused & order in which they are popped
4. Applications of DFS are:
   Graph connectivity, articulation point
5. Efficiency is: (checks whether value is 0 or 1 for each value of matrix)
   - Order of $|V|^2$ if input is in the form of adjacency matrix
   - Order of $|V|+|E|$ if input is in the form of adjacency list. (depends on no. of vertices & no. of edges with each)

BFS.

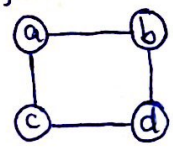
1. Data structure used is queue
2. In BFS, we have tree & cross edges
3. One-ordering:
   Order of insertion is same as order of deletion.
4. Applications of BFS are:
   Graph connectivity and minimum edge path.
   Used in spanning tree.
5. Efficiency is same as DFS.

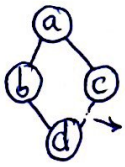


Quiz Q

Articulation point: A vertex of a connected graph is said to be its articulation point if its removal with all edges incident to it breaks the graph into disjoint pieces.

Eg. of BFS





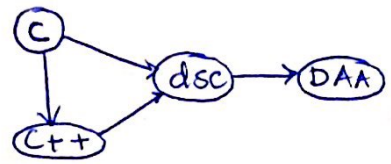
→ dashed line ∵ cross edge.

Topological ordering

It is not a sorting method

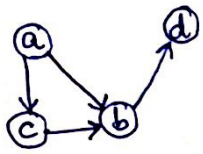
Topological ordering of a directed acyclic graph (DAG) is the linear ordering of all the vertices in the graph, such that if there is an edge from vertex 'u' to 'v', u is placed before 'v' in the ordering.

Eg.

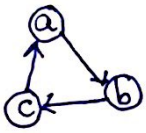


ordering : C C++ DSC DAA

Eg.



Order : a c b d.



Forms a cycle.
∴ topological sorting not possible.

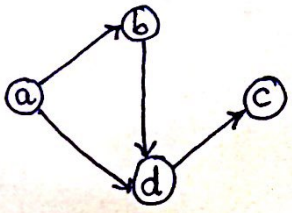
There are two methods to get the topological ordering :

i) DFS Method
ii) Vertex deletion method ("Source removal method").

① DFS Method (Lab program 4).

In this method, we traverse the given graph using DFS method, and, we should note down the order in which the nodes become the dead end. (the popping order of the nodes of the ~~ver~~ graph should be recorded.) By reversing the popping order, we get the topological ordering.

Eg.1.

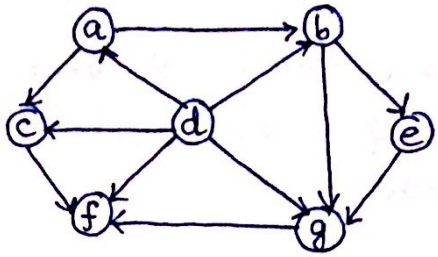


$a_{1,4}$
$b_{2,3}$
$d_{3,2}$
$c_{4,1}$

Popping order:
c d b a

Topological ordering:
a b d c

Eg.2.



$a_{1,6}$
$b_{2,4}$
$e_{3,1}$
$g_{4,3}$
$f_{5,2}$
$c_{6,5}$
$d_{7,7}$.

Popping order:

e f g b c a d

Topological ordering:

d a c b g f e

② Vertex deletion method

The general procedure for vertex deletion is:

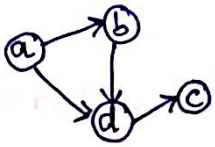
Step 1: Note down the in-degree of all the nodes

Step 2: Place the node which has in-degree of zero

Step 3: Decrement the in-degree of those nodes where there is an incoming edge from the node which was placed in step 2.

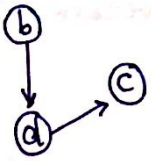
Step 4: Repeat step 2 & step 3 until all the nodes have been placed.

Eg.



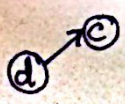
in(a) = 0
in(b) = 1
in(c) = 1
in(d) = 2

'a' placed.



in(b) = 0
in(c) = 1
in(d = 1

'b' placed.



in(c) = 1
in(d) = 0

'd' placed

c

in(c) = 0

'c' placed.

Array:

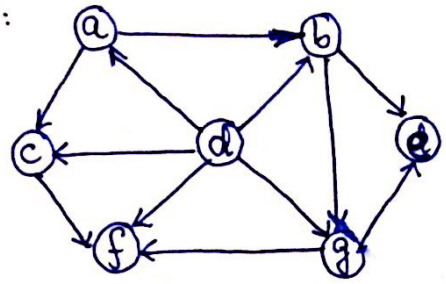
in(a) = 0
in(b) = 1 0
in(c) = 1 1 1 0
in(d) = 2 1 0

Topological ordering: a b d c

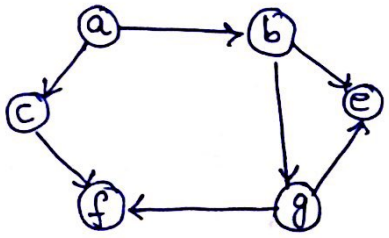
While taking i/p as adjacency matrix, increase in-degree by one if there is an incoming edge.

Push vertices with degree 0 onto stack

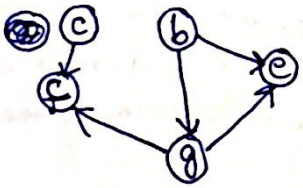
Eg 2:



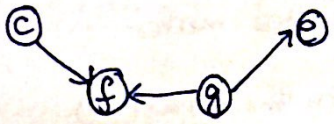
in(a) = 1
in(b) = 2
in(c) = 2
in(d) = 0 Place 'd'.
in(e) = 2
in(f) = 3
in(g) = 2



in(a) = 0
in(b) = 1
in(c) = 1
in(e) = 2
in(f) = 2
in(g) = 1 Place 'a'.



in(b) = 0
in(c) = 0 Place 'b'
in(e) = 2 Assuming 'b' is encountered
in(f) = 2 first alphabetically.
in(g) = 1



in(c) = 0 Place 'c'.
in(e) = 1
in(f) = 1
in(g) = 0

in(e) = 1
in(f) = 1 Place 'g'.
in(g) = 0

in(f) = 0
in(e) = 0 Place 'e', then 'f'. (Alphabetically).

∴ in(a) = 1 0
in(b) = 2 1 0
in(c) = 2 1 0 0
in(d) = 0 0
in(e) = 2 2 2 1 1 0
in(f) = 3 2 2 2 1 0
in(g) = 2 1 1 0 0

Topological order: d a b c g e f.

i/p array ↖ ↗ array with whether visited/not

```
void dfs(int a[][], int v[], int n, int s)
                              ↙ no. of vertices   ↘ source vertex
  {
   v[s] = 1;
   for(i=0; i<n; i++)
       if((a[s][i]==1)&&(v[i]==0))
         dfs(a, v, n, i);
  }
```

For using for topological sorting:

```
int result[];
int k=0;
 void dfs(int a[][], int v[], int n, int s)
 {
   v[s]=1;
   for(i=0; i<n; i++)
    if((a[s][i]==1)&& (v[i]==0))
       dfs(a,v,n,i);
       result[k++]=i;
 }
```

NOTE
This works for connected graphs.
for disconnected:

```
for (i=0; i<n; i++)
   if
    dfs(a,v,n,i)
  result[k++]=s (source)
```

print from (n-1) to 0 in calling funct

As it is recursive, when it becomes a dead end, it returns to the calling function.

∴ push them into array.

For loop in main will take care of unvisited nodes.

In main, print from the end of array(k) to the start(0).

This is because array result holds the elements in their popping order

The reverse of this array is the topological ordering.

Program to implement topological ordering using vertex deletion.

```
main()
{
int n, a[][], v[] ={0}, i, j, in[], st[], top=-1, d, result[], k=0;
• → Read n (no. of nodes of graph=n)
  → Read the adjacency matrix
    for (i=0; i<n; i++)
       {
        for (j=0; j<n; j++)
          {
           scanf ("%d", &a[i][j]);
if(a[i][j]==1)  in[j]++;
          }
       }

    for (i=0; i<n; i++)
    {
      if (in[i] ==0)
          st[++top]=i;
    }
    while (top! =-1)
      {
        d = st[top--];
        result[k++] = d;
        for (i=0; i<n; i++)
          {
            if (a[d][i]==1)
                  in[i]--; }
       for(i=0; i<n; i++)
          if (in[i]==0)
             st[++top]=i;
          }
      }

    for(i=k; i>=0; i--)
     printf ("%d", result[k]);
}
```

a b

Selection problem & computing median

(Variable size decrease).

Selection problem is a problem is which we find the K^{th} smallest or largest element in a given set of 'n' elements.

If 'k' has a value of $\lceil n/2 \rceil$, then, the k^{th} element is called the median.

To find the median, we shall use the concept of partition as used earlier in quick sort and let 's' denote the position of partition.

After partition, if $S = k$, then the S^{th} element is the median.

If $S < k$, we need to do partition on the right side.

If $S > k$, we need to do partition on the left side.

Eg. Compute the median for the given set of elements.

p↓ 4 1 10 9 7 12 8 2 15

$n = 9$

$k = \lceil n/2 \rceil = \lceil 4.5 \rceil = 5$.

∴ Median is the 5th smallest element.

4 > 1, i++

4 ⊁ 10, stop

4 < 15, j--

4 ⊀ 2, stop.

i < j, swap.

4 1 2 9 7 12 8 10 15.

swap pivot & j.

2 1 4 9 7 12 8 10 15.

Corresponding 's' value = 3.

S < k. ∴ right of it partition

9 7 8 12 10 15

p→ 8 7 9 12 10 15

Corresponding 's' value = 6.

S > k, ∴ perform partition on left

8 ≥ 7, but end, ∴ i not incremented

8 ⊀ 7.

i ⊀ j

∴ swap pivot

7 (8)

S = k, ∴ pivot element is the median.

∴ 8 is the median.

Quicksort: After partition, done again on left AND right half.

Finding median: After partition, either

Either stop ($S = k$)

Partition left

Partition right.

Size does not decrease by half or by any constant value.

it is decreased by variable size. ∴ "Variable size decrease".

Efficiency of finding median using decrease & conquer can be found by

$$C(n) = C(n/2) + n+1$$

$f(n) = n+1$.

$n^{\log_b a} = n^{\log_2 1} = n^0 = 1$.

∴ $f(n)$ is faster than $n^{\log_b a}$

∴ $C(n) = \Theta(n)$. // Average case.

$C(n) = \Theta(n^2)$. // Worst case: Order 1 2 3 4 5 6 7 8 9

↳ 1 compared $(n-1)$ times

2 compared $(n-2)$ times

3 compared $(n-3)$ times

∴ summation gives an answer of order n^2

5 times iteration to get median.

1 compared, 2 compared etc.

But positions do not change

Find the median for the given elements

p→	20	10	60	30	50	40	$n=6$, $k=3$.
	10	20	60	30	50	40	$s=2$, $s<k$. ∴ take right → Partition 1
			60	30	50	40	$s=6$, $s>k$, ∴ take left → Partition 2
			40	30	50	60	$s=4$, $s>k$, ∴ take left. → Partition 3.
			30	40	50		Partition 4

(30).

(Algorithm will have condn for 1 element in fn call.

∴ 30 is the median. ($s=k$).

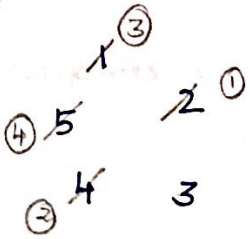
E/ Algorithm to find GCD using Euclid's algorithm is also an example for variable size decrease.

Josephus problem (Decrease by constant factor)

Problem statement:

Let n be the number of people, numbered from 1 to n, standing in a circle. Starting the count with person 1, we eliminate every second person until only one person (or survivor) is left.

Eg $n = 5$



$J(5) = 3$

(Survivor is 3 if $n = 5$).

(Take binary rep^n of n & shift to get answer

$J(101) = 011 = 3$. right shift

$J(110) = 101 = 5$

$J(111) = 111 = 7$

If killing every second person) circular shift.

If there are even number of people, then, the survivor can be calculated using equation:

$$J(2k) = 2J(k) - 1$$

Base case

$J(k) = 1$, for $k = 1$.

Eg. $k = 4$.

$J(8) = J(100) = 001 = 1$.

$J(2k) = 2J(4) - 1$.

$= 2J(2 \times 2) - 1$

$= 2(2J(2) - 1) - 1$

$= 4J(2) - 2 - 1$.

$= 4J(2 \times 1) - 3$

$= 4(2J(1) - 1) - 3$

$= 8J(1) - 4 - 3$

$= 8 \times 1 - 4 - 3$

$= 8 - 7$

$= 1$

For odd number of people

$$J(2k+1) = 2J(k)+1.$$

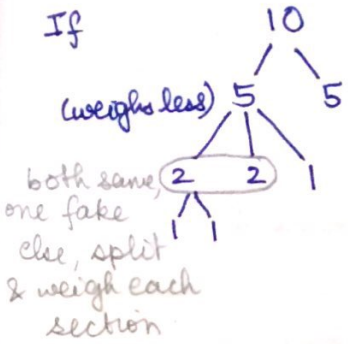
Fake coin problem (Decrease by constant factor)

Problem statement / Objective:

To identify a fake coin (Fake coin weighs less/more compared to other coins) amongst the given set of 'n' coins.

Procedure:

Divide the given coins into two sets, & weigh them on a weighing scale or pan. Further, divide that set which weighs less.



Prerequisite: must know if coin weighs less or more
Assume it weighs less.

Ref. textbook for theory.

Basic operation: weighing the coins.
& the total number of weighing operation done can be represented as

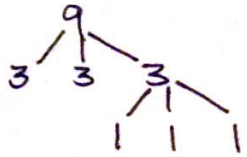
$$W(n) = W(\lceil n/2 \rceil) + 1 \quad \forall\, n > 1.$$

Base case, $W(1) = 0$

Similar to recurrance of binary search.

∴ Efficiency of identifying fake coin = $\log_2 n$

Instead of 2 parts, 3 parts (or any no. of parts can made)



If divided into 'k' subsets, efficiency of algorithm is $\log_k n$

Algorithm for generating combinatorial object (Decrease by constant)

① Generating subset

② Generating permutation.

Generating permutation.

Eg. 3, No. of permutation = 6.

1 2 3

Start with 1 : 1

Insert 2, to right & left of 1 : 12 21

Insert 3 from right to left (12) : 312 132 123 } 6 permutations.

Insert 3 from ~~right~~ to ~~right~~ left (21) : 321 231 213 }

$n! = n(n-1)!$

It is an example of decrease by constant.

To find the number of permutations for a value of 'n', we need the number of permutations for $(n-1)$

For $n = 3$, we can find no. of permutations by inserting 1 element at a time.

1. Start with 1	1
2. Insert 2 to L & R of 1	12 21
3. Insert 3, R to L of 12	312 132 231
Insert 3, R to L of 21	321 231 213

Bottom up approach

Johnson Trotter algorithm.

According to this algorithm, it is possible to get the same order of permutations of 'n' elements without explicitly generating the permutation of smaller value of 'n'.

The main idea in this algorithm is the association of direction with each digit in the permutation.

Eg: $\overrightarrow{1}\ \overrightarrow{2}\ \overleftarrow{3}$ } direcⁿ held in a data structure

Here, a digit (or component) is said to be a mobile digit, if its arrow points to a smaller digit. ∴ here, 3 is the mobile digit.

$\overrightarrow{1}\ \overleftarrow{2}\ \overrightarrow{3}\ \overrightarrow{4}\ \overleftarrow{5}$

2 & 5 are mobile digits

Algorithm

Algorithm Johnson-Trotter(n)

// Implements Johnson-Trotter algorithm for generating permutation

// Input: A positive integer 'n'.

// Output: A list of permutations of the set 1 to 'n'.

Initialise the first permutation with $\overleftarrow{1}, \overleftarrow{2}, \overleftarrow{3} \ldots\ldots \overleftarrow{n}$

While there exists a mobile integer k do

find the largest mobile integer k

swap 'k' & the adjacent integer its arrow points to

reverse the direction of all integers that are larger than 'k'.

Eg. n=3

1) $\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}$ — 2 & 3 are mobile integers. 3 is largest. ∴ swap 3 & 2.

2) $\overleftarrow{1}\ \overleftarrow{3}\ \overleftarrow{2}$ — k=3. ∴ no reversing direction. 3 mobile, 2 not. k=3. swap 3 & 1.

3) $\overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{2}$ — Now, 2 is a mobile integer

4) $\overleftarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}$ — 2 was mobile. ∴ rev. direcⁿ of values > 2, i.e, reverse 3 direction

$\overrightarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}$ — swap 2 & 3, ∴ 3 mobile.

5) $\overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{1}$ — 3 mobile, swap 3 & 1.

6) $\overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{3}$ — No mobile integer exists. ∴ algorithm stops.

List of possible permutations.

Trace Johnson-Trotter algorithm for $n = 4$.

$n = 4$.

1) $\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}\ \overleftarrow{4}$ — 2,3,4 mobile, 4 largest.

2) $\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{4}\ \overleftarrow{3}$

3) $\overleftarrow{1}\ \overleftarrow{4}\ \overleftarrow{2}\ \overleftarrow{3}$

4) $\overleftarrow{4}\ \overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}$ — 2, 3, mobile; 3 largest

5) $\overleftarrow{4}\ \overleftarrow{1}\ \overleftarrow{3}\ \overleftarrow{2}$

$\overrightarrow{4}\ \overleftarrow{1}\ \overleftarrow{3}\ \overleftarrow{2}$

6) $\overleftarrow{1}\ \overrightarrow{4}\ \overleftarrow{3}\ \overleftarrow{2}$

7) $\overleftarrow{1}\ \overleftarrow{3}\ \overrightarrow{4}\ \overleftarrow{2}$

8) $\overleftarrow{1}\ \overleftarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}$ — 3, [illegible] is mobile

9) $\overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{2}\ \overrightarrow{4}$ — Reverse 4.

$\overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{4}$

10) $\overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{4}\ \overleftarrow{2}$

11) $\overleftarrow{3}\ \overleftarrow{4}\ \overleftarrow{1}\ \overleftarrow{2}$

12) $\overleftarrow{4}\ \overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{2}$ — 2 is mobile.

13) $\overleftarrow{4}\ \overleftarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}$

$\overrightarrow{4}\ \overleftarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}$

14) $\overrightarrow{3}\ \overrightarrow{4}\ \overleftarrow{2}\ \overleftarrow{1}$

15) $\overrightarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}\ \overleftarrow{1}$

16) $\overrightarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{4}$

17) $\overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{1}\ \overrightarrow{4}$ — 3 is mobile

$\overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{1}\ \overleftarrow{4}$

18) $\overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{4}\ \overleftarrow{1}$

19) $\overleftarrow{2}\ \overleftarrow{4}\ \overleftarrow{3}\ \overleftarrow{1}$

20) $\overleftarrow{4}\ \overleftarrow{2}\ \overleftarrow{3}\ \overleftarrow{1}$

21) $\overleftarrow{4}\ \overleftarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}$ — 3 is mobile.

$\overrightarrow{4}\ \overleftarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}$

22) $\overleftarrow{3}\ \overrightarrow{4}\ \overleftarrow{2}\ \overleftarrow{1}$

23) $\overleftarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}\ \overleftarrow{1}$

24) $\overleftarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{4}$

All 24 permutations.

Freq. mistakes (If asked in exam, try, if you don't get in 5-10 moves, try a diff question