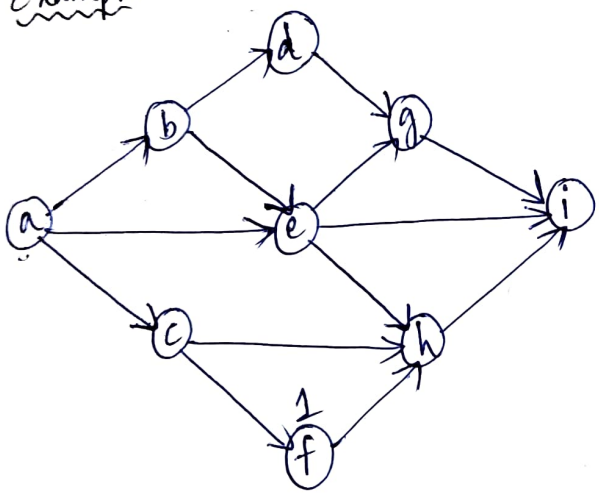
Topological Sorting

Ordering a Graph

Suppose we have a directed acyclic graph (DAG) of courses, and we want to find an order in which the courses can be taken.

* There are many problems involving a set of tasks in which some of the tasks must be done before others.
* Topological sort is a method of arranging the vertices in a directed acyclic graph (DAG), as a sequence, such that no vertex appear in the sequence before its predecessor.
* Topological sort is not unique.
* The following are all topological sort of the graph below:

Example:-



S1 = {a, b, c, d, e, f, g, h, i}

S2 = {a, c, b, f, e, d, h, g, i}

S3 = {a, b, d, c, e, g, f, h, i}

S4 = {a, c, f, b, e, h, d, g, i}, etc.

Topological Sort :- Given a digraph G = (V, E), a total ordering of G's vertices such that for every edge (v, w) in E, vertex v precedes w in the ordering.

Toposort runtime

function topologicalSort():
map: = {each vertex[] its in-degree} // O(V)
queue: = {all vertices with in-degree=0}
ordering: = {}
Repeat until queue is empty // O(V)
Dequeue the first vertex v from the queue // O(1)
ordering += v // O(1)
Decrease the in-degree of all v's // O(E) for all passes.
neighbors by 1 in the map
queue += {any neighbors whose in-degree is now 0}.

Overall: O(V+E); essentially O(V) time on a sparse graph.

Topological sort Algorithm

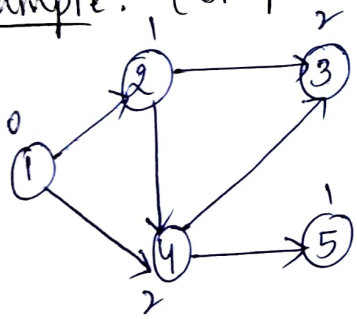
```
Algorithm topologicalSort(graph)
{
  stack: = [];
  visited: = [False, ..., False], //length N
  For each vertex v in graph do
  {
    if not visited[v] then
    {
      DFS(v, visited, stack);
    }
  }
  print(stack);
}
Algorithm DFS(v, visited, stack)
  {
    visited[v]: = true;
    for each neighbour u of v do
```

```
    if not visited [u] then
    {
      DFS (u, visited, stack);
    }
  }
  push(v);
}
```

Applications of Topological sorting

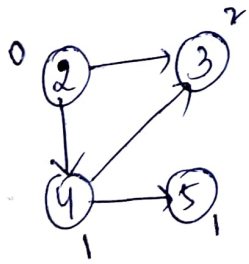
* Packet transfer
* Course selection
* Order compilation tasks
* Data serialization.

Example: (Simple method)



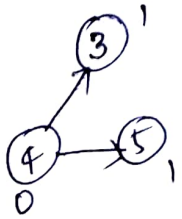
Step1: Find indegree of each degree.

Step2: Start with node having indegree of '0' and delete the edges.



Sol: 1,

Step3: Repeat with indegree of '0' and delete edges.



Sol: 1,2

③ 0

⑤ 0

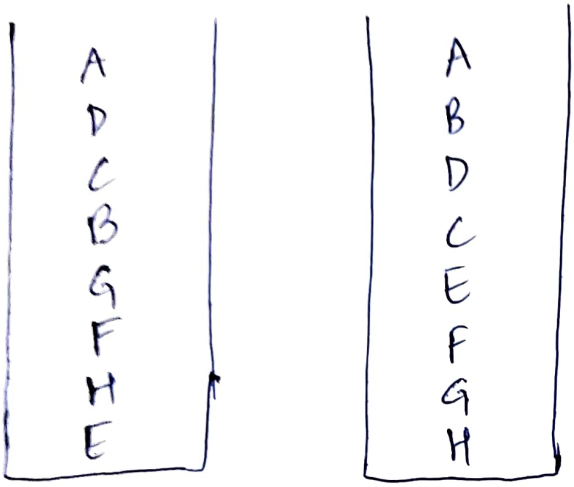
Sol: 1, 2, 4

Final solution: 1, 2, 4, 3, 5

or

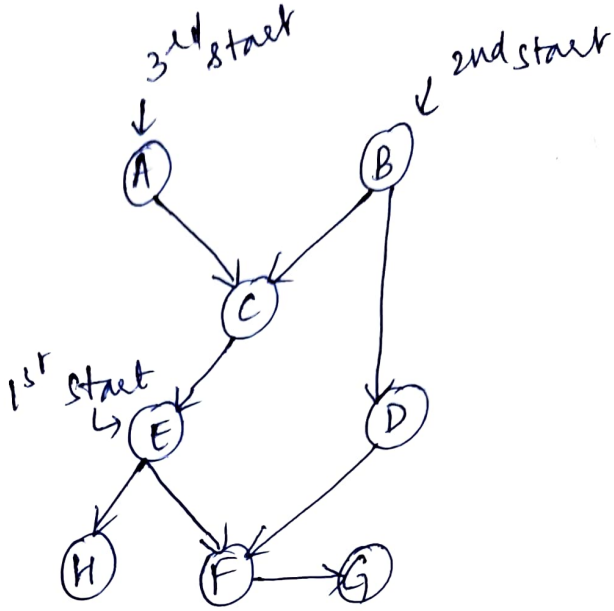
1, 2, 4, 5, 3

Another method



visited nodes

stack



Solution: A B D C E F G H

(1) Can start at any node. Lets take E send to visited nodes.

(2) visit all nodes children to "E": H, send to visited nodes.

(3) H has no children, so send to stack

(4) Go back to E and take other child

(5) Take child of F i.e G.

(6) G has no children, so send to stack.

(7) Go back to F, F has no children.

(8) H, E also don't have children, write them in the stack

(9) Take B as starting node, and take its children as C

(10) Follow the same procedure until all the nodes are placed in the stack.

Connected components and Spanning Trees

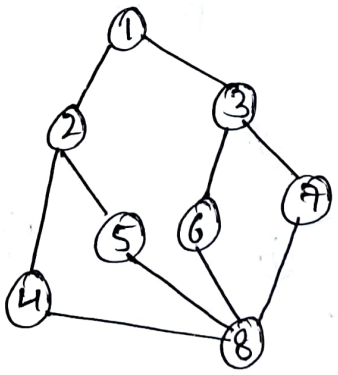
Let us consider a graph G, G is said to be connected undirected graph if all the vertices of G will get visited on the first call to BFS (Breadth-First Search), and G is said to be not connected, if all the vertices of G will get visited for atleast two calls to BFS.

Hence, BFS can be used to determine whether G is connected or not.

→ The graph G has a spanning tree iff G is connected.

Hence, BFS easily determines the existence of a spanning tree.

Example:- Consider a graph G



'G'

DFS and BFS spanning tree of graph are:

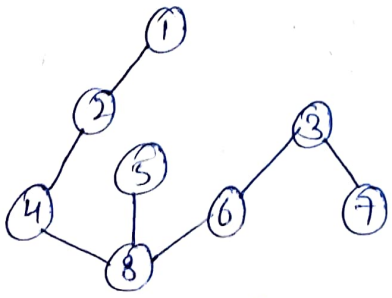


Fig: 1 DFS spanning tree

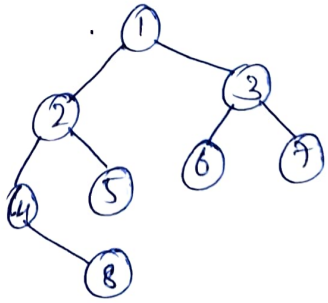


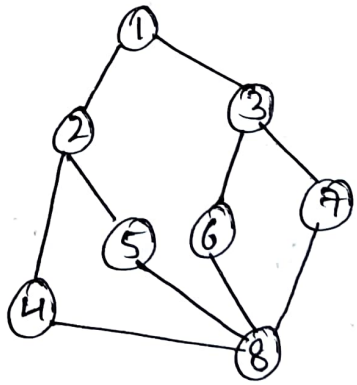
Fig: 2 BFS spanning tree

Connected components and Spanning Trees

Let us consider a graph G, G is said to be connected undirected graph if all the vertices of G will get visited on the first call to BFS (Breadth-First Search), and G is said to be not connected, if all the vertices of G will get visited for atleast two calls to BFS. Hence, BFS can be used to determine whether G is connected or not.

→ The graph G has a spanning tree iff G is connected. Hence, BFS easily determines the existence of a spanning tree.

Example:- Consider a graph G



'G'

DFS and BFS spanning tree of graph are:

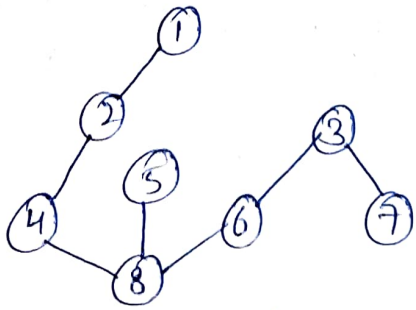


Fig: 1 DFS spanning tree

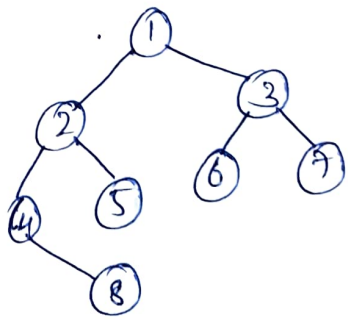


Fig: 2 BFS spanning tree

The spanning trees obtained from BFS are known as Breadth-first spanning tree.

→ The DFS spanning tree (Fig:1) of G is not only the spanning tree we get, we can get different spanning trees from graph 'G'.

Biconnected components and DFS:-

Now consider another graph G', which is undirected. A vertex 'v' in a connected graph G' is an "articulation point" if and only if the deletion of vertex v. together with all edges incident to v disconnects the graph into two or more nonempty components.

Example:- Consider a graph 'G'.

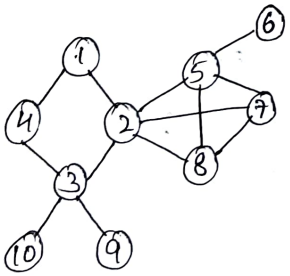


Fig: 3

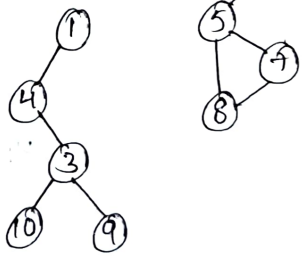


Fig: 4 : Result of deleting vertex '2'

In the above fig:3, verter '2' is an articulation point as the deletion of vertex '2' and edges (1,2), (2,3), (2,5), (2,7), and (2,8) leaves behind two disconnected nonempty

components (Fig:4). Graph 'G' of fig:3 has only two other articulation points vertex '5' and vertex '3'.

=> A graph G is "biconnected" if and only if it contains no articulation points. The graph of Fig: 3 is not biconnected, because it has articulation points.

Real-Life application of biconnected graph and articulation point.

For example, if G represents a communication network with the vertices representing communication stations and the edges communication lines, then the failure of a communication station 'i' that is an articulation point would result in the loss of communication to points other than 'i' too. On the other hand, if G has no articulation point, then if any station i fails, we can still communicate between every two stations not including station 'i'.

Example for a biconnected graph:-

Consider a graph G,

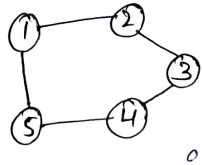


Fig: 5
Graph 'G'

This is a biconnected graph, because it has no articulation point.

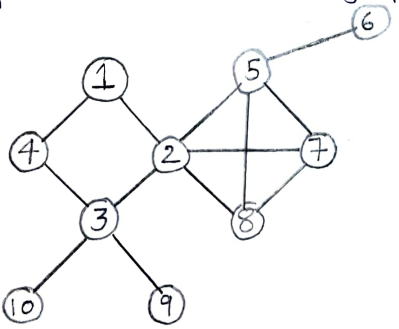
ow to identify articulation points in a Graph.

In a graph, each node i will have two new labtels.

→ DFN[i]

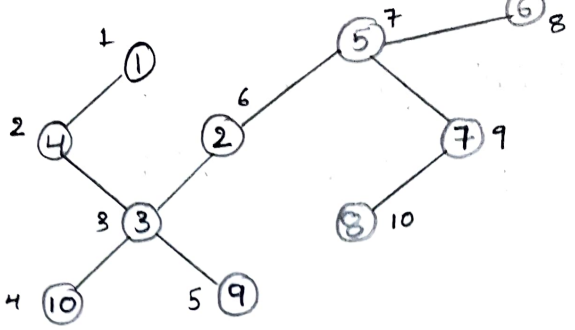
→ L[i]

- Depth first number (DFN) [i] is the time at which 'i' is visited. Thus, the first node has its DFN=1. The second node visited has a DFN=2 and so on.
- L[i] is the lowest depth first number that can be reached from i using a path of descendants followed by at most one back edge.
- If i is not the root, then i is an articulation point if and only if i has a child j such that L[j] ≥ dfn[i]
- For example, let us consider a graph



To identify dfn of vertices:

- First find the depth first spanning tree of the graph.



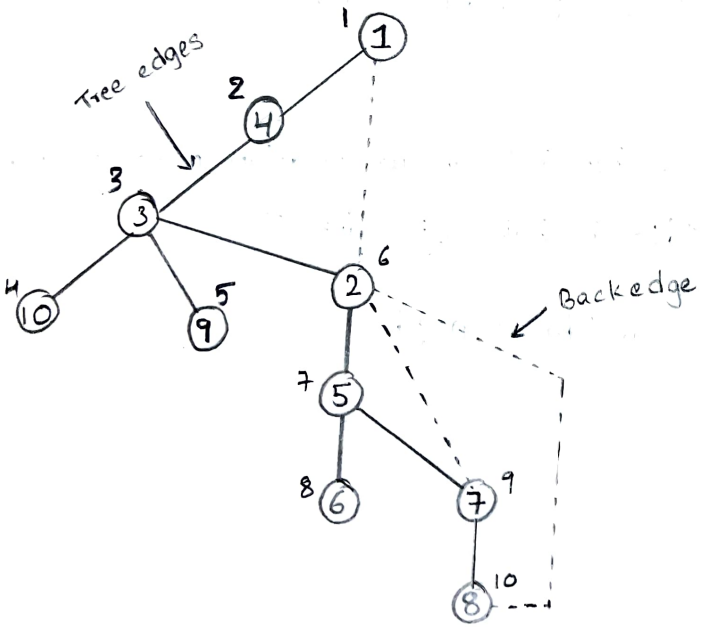
DFN[1] = 1 DFN[2] = 6

DFN[4] = 2 DFN[5] = 7

DFN[3] = 3 DFN[6] = 8

DFN[10] = 4 DFN[7] = 9

DFN[9] = 5 DFN[8] = 10



→ Solid lines represent the tree edges

→ Dotted lines represent the back edges

To identify L of the vertices:

→ For each vertex u, L[u] can be defined as:

$$L[u] = \min\{dfn[u], \min\{L[w] \mid w \text{ is child of } u\}, \min\{dfn[w] \mid (u,w) \text{ is a backedge}\}\}$$

$$L[4] = \min\{dfn[4], \min\{L[w]\}, \min\{dfn[w]\}\}$$

$$= \min\{2, 1, -\}$$

$$= 1$$

$$L[3] = \min\{3, 1, -\} = 1$$

$$L[10] = \min\{4, -, -\} = 4$$

$$L[9] = \min\{5, -, -\} = 5$$

$$L[2] = \min\{6, -, 1\} = 1$$

$$L[8] = \min\{10, -, 6\} = 6$$

$$L[7] = \min\{9, 6, 6\} = 6$$

$$L[5] = \min\{7, 6, -\} = 6$$

$$L[6] = \min\{8, -, -\} = 8$$

$$L[1] = \min\{1, 1, -\} = 1$$

$\therefore L[1:10] = \{1, 1, 1, 1, 6, 8, 6, 6, 5, 4\}$

	1	2	3	4	5	6	7	8	9	10
DFN	1	6	3	2	7	8	9	10	5	4
L	1	1	1	1	6	8	6	6	5	4

→ Vertex 3 has child 10

$u = 3$, $w = 10$

$L[10] \geq dfn[3]$

$4 \geq 3$ True

∴ 3 is an articulation point

→ Vertex 4 has child 3

$u = 4$, $w = 3$

$L[3] \geq dfn[4]$

$1 \geq 2$ False

∴ 4 is not an articulation point

→ Vertex 2 has child 5

$u = 2$, $w = 5$

$L[5] \geq dfn[2]$

$6 \geq 6$ True

∴ 2 is an articulation point

→ Vertex 5 has child 6

$u = 5$, $w = 6$

$L[6] \geq dfn[5]$

$8 \geq 7$ True

∴ 5 is an articulation point.

and so on

∴ The articulation points are : 2, 3, 5

Pseudocode to compute dfn and L

```
Algorithm Art(u,v)
{ // u is a start vertex for depth first search. v is its parent if
  // any in the depth first spanning tree. It is assumed that
  // the global array dfn is initialized to zero and that the
  // global variable num is initialized to 1. n is no.of vertices in G

  dfn[u] := num; L[u] := num; num := num+1;
  for each vertex w adjacent from u do
  {
    if (dfn[w] = 0) then
    {
      Art(w,u); // w is unvisited
      L[u] := min(L[u], L[w]);
    }
    else if (w ≠ v) then
      L[u] := min(L[u], dfn[w]);
  }
}
```

Network flow Algorithms

In optimization, network flow problems are a class of computational problems in which the input is a flow network (a graph with numerical capacities on its edges), and the goal is to construct a flow, numerical values on each edge that respect the capacity constraints and that have incoming flow equal to outgoing flow at all vertices except for certain designated terminals.

Specific types of network flow problems include:

- The maximum flow problem, in which the goal is to maximize the total amount of flow out of the source terminals and into the sink terminals.
- The minimum-cost flow problem, in which the edges have costs as well as capacities and the goal is to achieve a given amount of flow (or a max. flow) that has the minimum possible cost.
- The multi-commodity flow problem, in which one must construct multiple flows for different commodities whose total flow amounts together respect the capacities.
- Nowhere-zero flow, a type of flow studied in combinatorics in which the flow amounts are restricted to a finite set of non-zero values.

Algorithms for constructing flows include -

- Dinic's algorithm, a strongly polynomial algorithm for max. flow.
- The Edmonds-Karp algorithm, a faster algorithm

(a faster strongly polynomial algorithm) for max flow

- The Ford-Fulkerson algorithm, a greedy algorithm for max. flow that is not in general strongly polynomial.
- The network simplex algorithm, a method based on linear programming but specialized for network flow
- The out-of-kilter algorithm for min.-cost flow.
- The push-relabel max. flow algorithm, one of the most efficient known techniques for max. flow.

Flow network:-

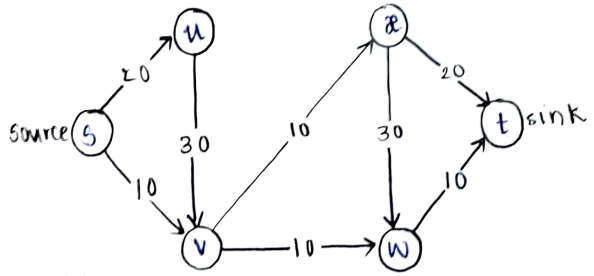
A flow network is a connected, directed graph $G=(V, E)$.

→ Each edge e has non-negative, inter capacity c_e.

→ A single source $s \in V$.

→ A single sink $t \in V$.

→ NO edge enters the source and no edge leaves the sink.



Assumptions:-

To repeat, we make these assumptions about the network:

(1) Capacities are integers.

(2) Every node has one edge adjacent to it.

(3) NO edge enters the source and no edge leaves the sink.

Flow:-

An s-t flow is a function $f: E \rightarrow R \geq 0$ that assigns a real no. to each edge.

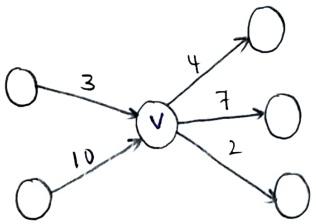
Intuitively, $f(E)$ is the amt. of material carried on the edge e.

Constraints:- (on f)

- $0 \leq f(e) \leq c_a$ for each edge e (capacity constraints).
- For each node v except s and t we have:

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ leaving } v} f(e)$$

(Balance constraints: whatever flows in, must flow out).



Notation:-

The value of flow f is:

$$v(f) = \sum_{e \text{ out of } s} f(e)$$

This is the amt. of material that s is able to send out.

→ $f^{in}(v) = \sum_{e \text{ into } v} f(e)$

→ $f^{out}(v) = \sum_{e \text{ leaving } v} f(e)$

Balance constraints becomes: $f^{in}(v) = f^{out}(v) \quad \forall\ v \notin \{s, t\}$

Maximum Flow Problem:-

Definition (value):-

The value $v(f)$ of a flow f is $f^{out}(s)$.

That is: it is the amt. of material that leaves s.

Maximum Flow Problem:-

Given a flow network G, find a flow f of max. possible value.

Residual Graph:-

We define a residual graph G_f. G_f depends on some flow f:

(1) G_f contains the same nodes as G.

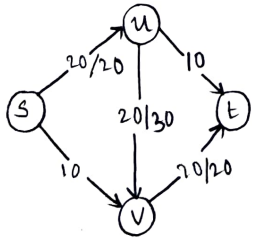
(2) **Forward edges**: For each edge $e=(u,v)$ of G for which $f(e) < c_e$ include an edge $e'=(u,v)$ in G_f with capacity $c_e - f(e)$.

(3) **Backward edges**: For each edge $e=(u,v)$ in G with $f(e) > 0$, we include an edge $e'=(v,u)$ in G_f with capacity $f(e)$.

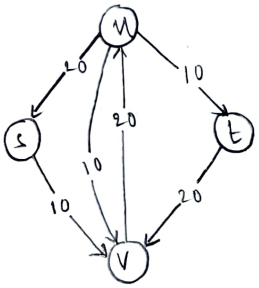
Residual Graph eg:-

(1) If $f(e) < c_e$, add edge e to G_f with capacity $c_e - f(e)$ (remaining capacity left).

(2) If $f(u,v) > 0$, add reverse edge (v,u) with capacity $f(e)$ (can erase up to $f(e)$ capacity).



with flow f



residual graph G_f

Augmenting paths:-

(1) Let P be an s-t path in the residual graph G_f.

(2) Let bottleneck(P, f) be the smallest capacity in G_f on any edge of P.

(3) If bottleneck(P, f) > 0 then we can increase the flow by sending bottleneck(P, f) along the path P.

Augmenting paths, 2:-

If bottleneck(P, f) > 0 then we can increase the flow by sending bottleneck(P, f) along the path P:

```
augment(f, P):
    b = bottleneck(P, f)
    for each edge (u,v) ∈ P:
        if e=(u,v) is a forward edge:
            increase f(e) in G by b //add some flow

        else:
            e' = (v, u)
            decrease f(e') in G by b //erase some flow

        end if
    end for
    return f.
```

Ford-Fulkerson Algorithm:-

```
MaxFlow(G):
  //initialize:
  Set f[e] = 0 for all e in G
  //while there is an s-t path in G_f:
  while P = FindPath(s, t, residual(G, f)) != none:
      f = augment(f, P)
      updateResidual(G, f)
  end while
  return f
```

After augment, we still have a flow:-

After $f' = \text{augment}(P, f)$, we still have a flow:

Capacity constraints: Let e be an edge on P:

(1) If e is a forward edge, it has capacity $c_e - f(e)$, therefore,

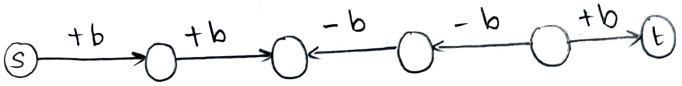
$$f'(e) = f(e) + \text{bottleneck}(P,f) \leq f(e) + c_e - f(e) \leq c_e$$

(2) If e is a backward edge, it has capacity $f(e)$, therefore,

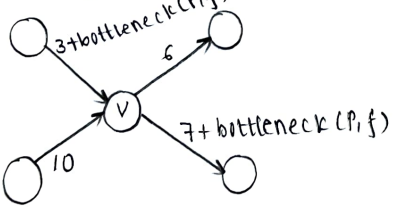
$$f'(e) = f(e) - \text{bottleneck}(P,f) \geq f(e) - f(e) = 0$$

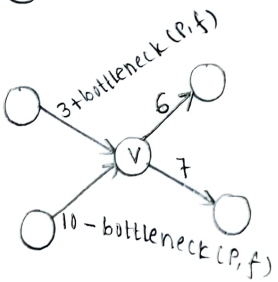
Still have flow, 2:

Balance constraints:- Any s-t path in G_f corresponds to some set of edges in G:



In other pictures,





Running time:-

(1) At every step, the flow values $f(e)$ are integers. start with ints and always add or subtract ints.

(2) At every step, we increase the amt. of flow $v(f)$ sent by atleast 1 unit.

(3) we can never send more than $C := \sum_{e \text{ leaving } s} c_e$

Theorem:-

The Ford-Fulkerson algorithm terminates in C iterations of the while loop.

Time in the while loop:-

(1) If G has m edges, G_f has $\leq 2n$ edges.

(2) Can find an s-t path in G_f in time $O(m+n)$ time with BFS or DFS.

(3) since $m \geq \frac{n}{2}$ (every node is adjacent to some edge), $O(m+n) = O(m)$.

Theorem:-

The Ford-Fulkerson algorithm runs in $O(mC)$ time.

Caveats:-

Note this is pseudo-polynomial because it depends on the size of the integers in the input.

You can remove this with slightly different algorithms. E.g.:-

(1) $O(nm^2)$: Edmonds-Karp algorithm (use BFS to find the augmenting path)

(2) $O(m^2 \log C)$ (or)

(3) $O(n^2 m)$ (or) $O(n^3)$

NP Hard and NP Complete

1. Non deterministic algorithm
2. NP hard classes
   NP complete classes
3. Cooks theorem

Polynomial time	Exponential time
Linear search n	0/1 knapsack 2^n
Binary " $\log n$	TSP 2^n
Insertion sort $- n^2$	Sum of subsets 2^n
Merge sort $- n \log n$	Graph colouring 2^n
Matrix multipli $- n^3$	Hamiltonian cycle 2^n

We need these exponential time algorithms to improve to polynomial time.

Solutions are not found for these problems yet.

* If one problem of exponential time has been solved and solution with polynomial time is obtained then all problems will also get the solution.

* when we are unable to write deterministic (Polynomial) algorithm write nondeterministic (Poly) algo.

Non Deterministic Algorithm

```
Algorithm NSearch(A, n, key)
{
    j = choice();
    if (key = A[j])
    {
        write(j);
        success();
    }
    write(0);
    failure();
}
```

→ The total time complexity is O(1)

→ Nondeterministic step which takes only O(1) time

But we don't know how to write choice, success, failure whose time complexity is O(1).

May be in future we may write a function choice() → which magically tells the position of the key automatically.

So, todays nondeterministic algo may be a deterministic algo tomorrow.

Deterministic Algo:-

Algorithms which has the property that the result of every operation is uniquely defined are termed as deterministic Algorithm

Non deterministic Algo

Algorithms which has few statements which are non deterministic.

It terminates unsuccessfully if and only if there exists no set of choices leading to success signal.

1. choice(S) :- arbitarily choose one of elements of set S
2. failure( ) :- signals an unsuccessfull completion
3. success( ) :- signals a successfull completion.

- The completion time for choice, success & failure are taken to be $O(1)$.

The machine capable of executing a nondeterministic algorithm in this way is called a non-deterministic machine

P

→ All are deterministic in polynomial time

Ex:- All sorts
All searches
Single source shortest path
Huffman code

Definition:-
P is the set of all decision problems solvable by deterministic algorithms in polynomial time.

An algorithm A is of Polynomial complexity if there exists a polynomial P() such that the computing time of A is O(P(n)) for every input of size n

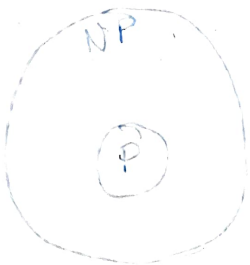
NP

→ Non deterministic
→ All non polynomial time algorithms

Ex:- TSP
0/1 knapsack
Sum of subsets
Graph coloring

Definition:-
NP is set of all decision problems solvable by non deterministic algorithms in polynomial time.

Since deterministic algorithms are a special case of non deterministic ones, we conclude P ⊆ NP



Ex:- Merge sort was previously NP now became P (deterministic algo)

NP-hard & NP-complete

Satisfiability is in P if and only if P=NP.

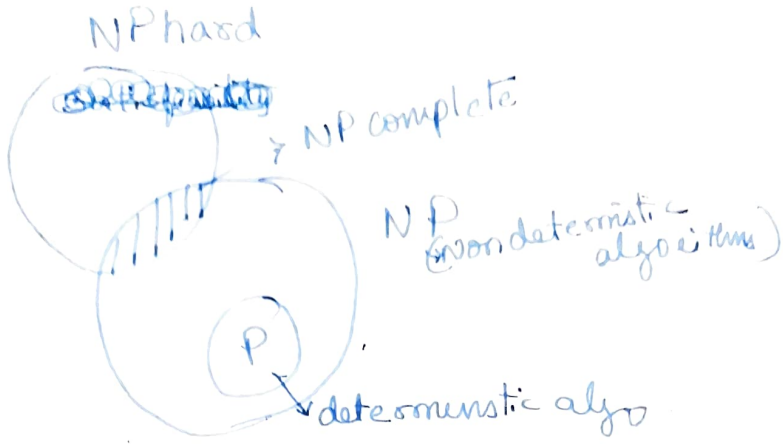
Let L_1 & L_2 be problems. Problem L_1 reduces to L_2 if and only if there is a way to solve L_1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L_2 in Polynomial time.

Transitive Reduction

NP-hard :- A problem L is NP-hard if and only if Satisfiability reduces to L.

NP complete :- A problem L is NP-complete if and only if L is NP-hard and $L \in NP$.

There are NP-hard problems that are not NP-complete. Only a decision problem can be NP-complete. However, an optimization problem may be NP-hard



⇒ After done solving of some NP problems P proves if P reaches NP

→ NP hard problems are solved then NP complete increases click &

* Knapsack & Clique problems reduces to the knapsack optimization problem

* There also exists NP-hard decision problems that are not NP-complete. The Halting problem is an example of an NP-hard decision problem that is not NP-complete.

Cook's Theorem

* Cook formulated a question "Is there any single problem in NP such that if we showed it to be in P, then that would imply that P=NP?

> Cooks theorem states that satisfiability is in P if and only if P=NP.

Proof

* We have already seen that satisfiability is in NP.
* Hence if P = NP, then satisfiability is in P.
* It remains to be shown that if satisfiability is in P, then P = NP.

→ To do this we show how to obtain from any polynomial time nondeterministic decision algorithm A and input I a formula Q(A, I) such that Q is satisfiable iff A has a successful termination with input I.

If the length of I is n and the time complexity of A is $p(n)$ for some polynomial $p()$, then the length of Q is $O(p^3(n)\log n) = O(p^4(n))$. The time needed to construct Q is also $O(p^3(n)\log n)$.

A deterministic algorithm Z to determine the outcome of A on any input I can be easily obtained.

Algorithm Z simply computes Q and then uses a deterministic algorithm for the satisfiability problem to determine whether Q is satisfiable.

If $O(q(m))$ is the time needed to determine whether a formula of length m is satisfiable, then the complexity of Z is $O(p^3(n)\log n + q(p^3(n)\log n))$

If satisfiability is in P, then $q(n)$ is a polynomial function of n and the complexity of Z becomes $O(r(n))$ for some polynomial $r()$.

Hence if satisfiability is in P, then for every nondeterministic algorithm A in NP we can obtain a deterministic Z in P.

So above construction shows that if satisfiability is in P, then $\boxed{P = NP}$

Extra notes to understand in depth and easily

NP-Completeness

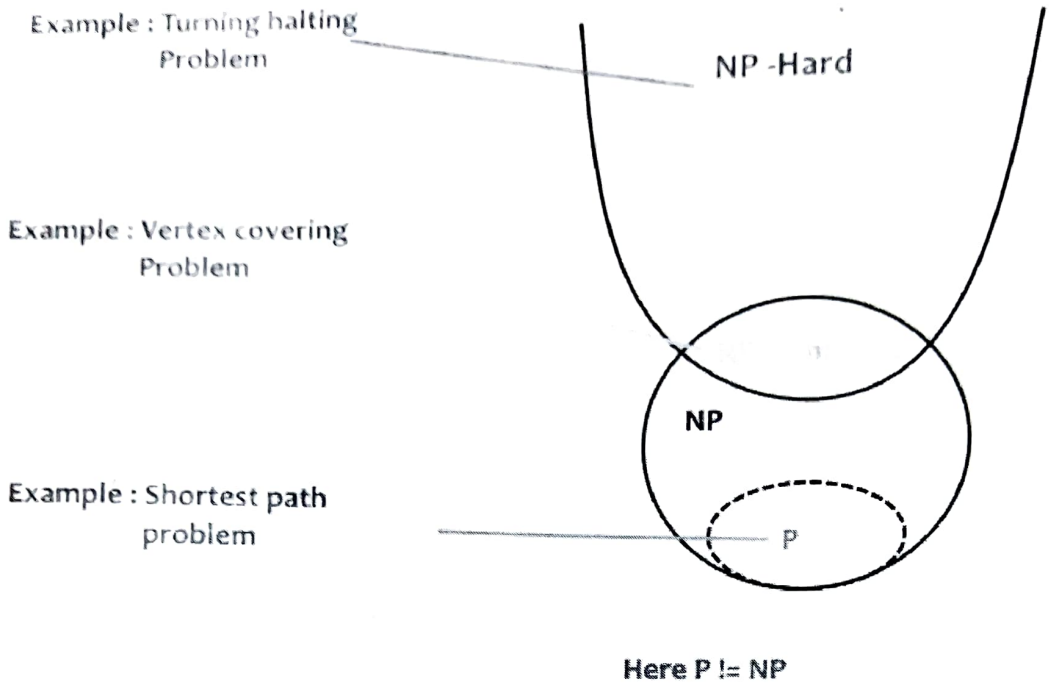
We have been writing about efficient algorithms to solve complex problems, like shortest path, Euler graph, minimum spanning tree, etc. Those were all success stories of algorithm designers. In this post, failure stories of computer science are discussed.

Can all computational problems be solved by a computer? There are computational problems that can not be solved by algorithms even with unlimited time. For example Turing Halting problem (Given a program and an input, whether the program will eventually halt when run with that input, or will run forever). Alan Turing proved that general algorithm to solve the halting problem for all possible program-input pairs cannot exist. A key part of the proof is, Turing machine was used as a mathematical definition of a computer and program (Source HaltingProblem).

Status of NP Complete problems is another failure story, NP complete problems are problems whose status is unknown. No polynomial time algorithm has yet been discovered for any NP complete problem, nor has anybody yet been able to prove that no polynomial-time algorithm exist for any of them. The interesting part is, if any one of the NP complete problems can be solved in polynomial time, then all of them can be solved.

Whatare NP, P, NP-complete and NP-Hard problems?

P is set of problems that can be solved by a deterministic Turing machine in **P**olynomial time.
NP is set of decision problems that can be solved by a **N**on-deterministic Turing Machine in **P**olynomial time. P is subset of NP (any problem that can be solved by deterministic machine in polynomial time can also be solved by non-deterministic machine in polynomial time).
Informally, NP is set of decision problems which can be solved by a polynomial time via a "Lucky Algorithm", a magical algorithm that always makes a right guess among the given set of choices (Source Ref 1).
NP-complete problems are the hardest problems in NP set. A decision problem L is NP-complete if:
1) L is in NP (Any given solution for NP-complete problems can be verified quickly, but there is no efficient known solution
2) Every problem in NP is reducible to L in polynomial time (Reduction is defined below)
A problem is NP-Hard if it follows property 2 mentioned above, doesn't need to follow property 1. Therefore, NP-Complete set is also a subset of NP-Hard set.



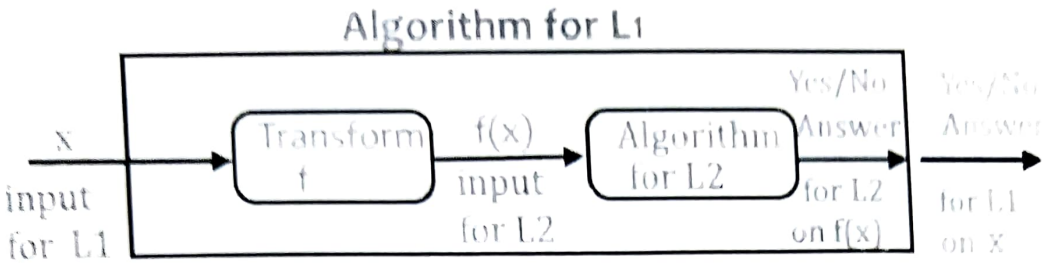
Decision vs Optimization Problems

NP-completeness applies to the realm of decision problems. It was set up this way because it's easier to compare the difficulty of decision problems than that of optimization problems. In reality, though, being able to solve a decision problem in polynomial time will often permit us to solve the corresponding optimization problem in polynomial time (using a polynomial number of calls to the decision problem). So, discussing the difficulty of decision problems is often really equivalent to discussing the difficulty of optimization problems. (Source Ref2).

For example, consider the vertex cover problem (Given a graph, find out the minimum sized vertex set that covers all edges). It is an optimization problem. Corresponding decision problem is, given undirected graph G and k, is there a vertex cover of size k?

What is Reduction?

Let L_1 and L_2 be two decision problems. Suppose algorithm A_2 solves L_2. That is, if y is an input for L_2then algorithm A_2 will answer Yes or No depending upon whether y belongs to L_2 or not. The idea is to find a transformation from L_1 to L_2 so that the algorithm A_2 can be part of an algorithm A_1 to solve L_1.



Learning reduction in general is very important. For example, if we have library functions to solve certain problem and if we can reduce a new problem to one of the solved problems, we save a lot of time. Consider the example of a problem where we have to find minimum product path in a given directed graph where product of path is multiplication of weights of edges along the path. If we have code for Dijkstra's algorithm to find shortest path, we can take log of all weights and use Dijkstra's algorithm to find the minimum product path rather than writing a fresh code for this new problem.

How to prove that a given problem is NP complete?

From the definition of NP-complete, it appears impossible to prove that a problem L is NP-Complete. By definition, it requires us to that show every problem in NP is polynomial time reducible to L. Fortunately, there is an alternate way to prove it. The idea is to take a known NP-Complete problem and reduce it to L. If polynomial time reduction is possible, we can prove that L is NP-Complete by transitivity of reduction (If a NP-Complete problem is reducible to L in polynomial time, then all problems are reducible to L in polynomial time).

What was the first problem proved as NP-Complete?

There must be some first NP-Complete problem proved by definition of NP-Complete problems. SAT (Boolean satisfiability problem) is the first NP-Complete problem proved by Cook (See CLRS book for proof).

It is always useful to know about NP-Completeness even for engineers. Suppose you are asked to write an efficient algorithm to solve an extremely important problem for your company. After a lot of thinking, you can only come up exponential time approach which is impractical. If you don't know about NP-Completeness, you can only say that I could not come with an efficient algorithm. If you know about NP-Completeness and prove that the problem as NP-complete, you can proudly say that the polynomial time solution is unlikely to exist. If there is a polynomial time solution possible, then that solution solves a big problem of computer science many scientists have been trying for years.

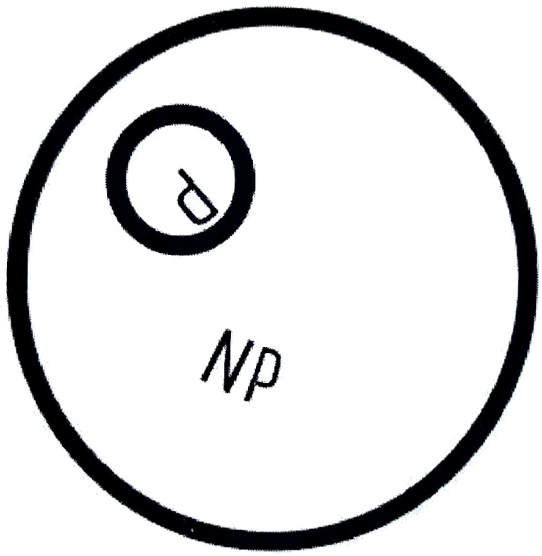
SIMPLE Understanding

P - Problems that can be **solved** in polynomial time.

NP - Problems whose solution can be **verified** in polynomial time.

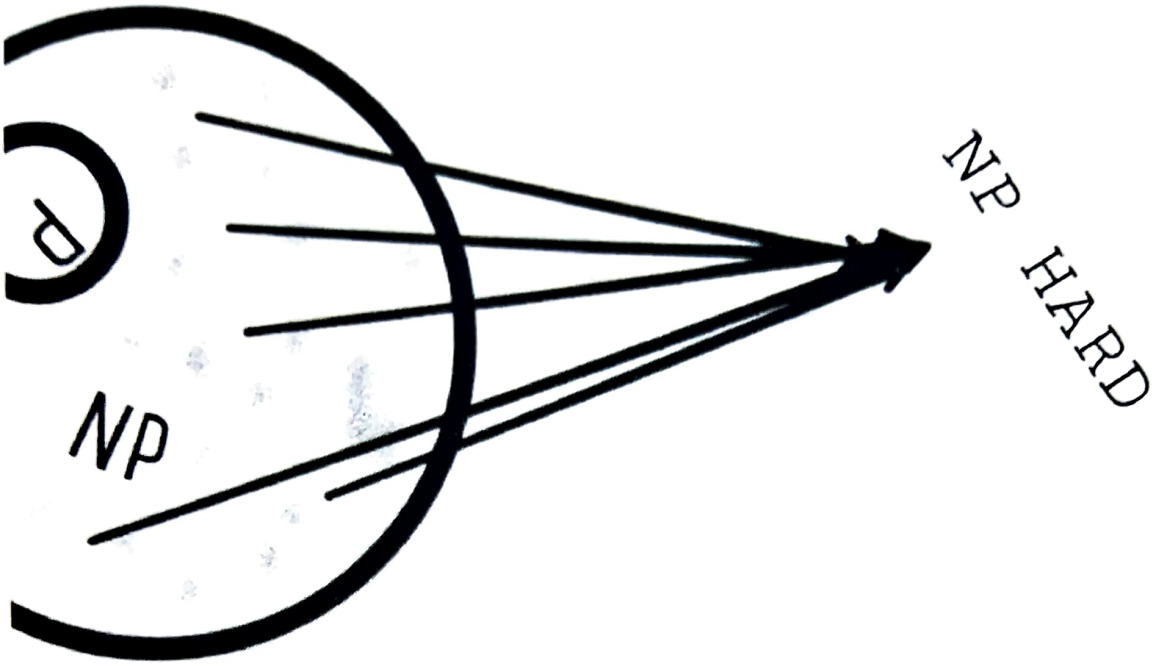
Therefore, Every P problem is also an NP as every P problem's solution can also be verified in polynomial time, but vice-versa is not true because every NP problem cannot be solved in polynomial time.

Now here is the Venn diagram representation of the same,

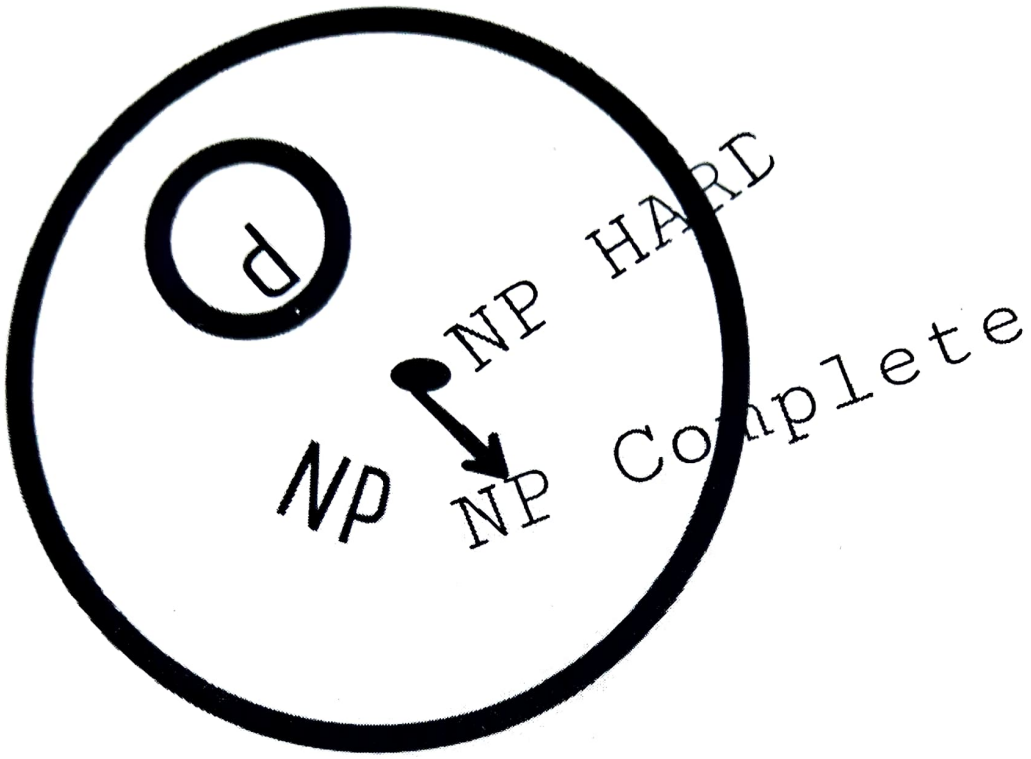


NP-Hard - If an even harder problem is reducible to all the problems in NP set (at least as hard as any NP-problem) then that problem is called as NP-hard.

Here is the Venn diagram,



NP-Complete: If an NP-hard problem is inside the set of NP problems then that is NP-complete,


d
NP HARD
NP
NP Complete

Randomized Algorithms ①

* Randomized algorithm is a different design approach taken by the standard algorithms where few random bits are added to a part of their logic.

* RA use some randomness in their logic to improve efficiency, time complexity, or may be the total memory used.

* RA are a class of algorithms in computer science that uses randomness or randomness in combination with deterministic steps to solve computational problems.
These algo. introduce randomness intentionally into their calculations to achieve improved efficiency or increasing the likelihood of finding a correct solution.
Ex:- Quantum algorithms are in principle randomized

* Most useful in situations like:
  - ① Finding an exact solution to a problem is computationally expensive or impractical.
  - ② Providing approximate solutions with a control level of error.

* Need for Randomized algo
  1. Improved efficiency.
  2. Complex problems can be handled effectively
  3. Worst case scenarios can be avoided.
  4. Cryptography.

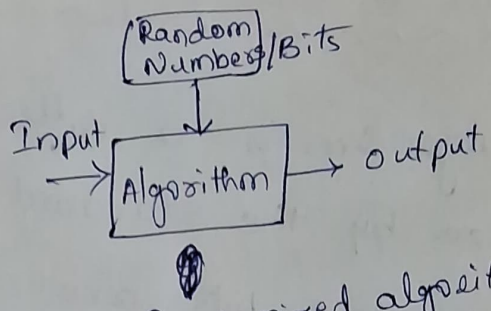


Fig:- Randomized algorithm Flowchart

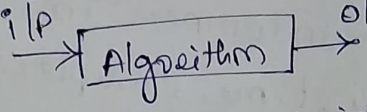


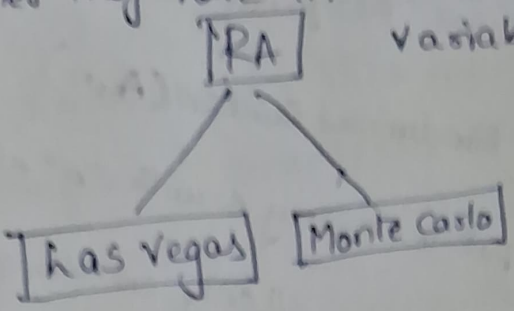
Fig:- Deterministic algorithm structure.

Unlike deterministic algorithms, randomized algorithms consider randomized bits of the logic along with the input that in turn contribute towards obtaining the output.

However, the probability of randomized algorithms providing incorrect output cannot be ruled out either. This process called as amplification is performed to reduce the likelihood of these erroneous outputs.

Amplification is an algorithm that is applied to execute some parts of the randomized algorithms multiple times to increase the probability of correctness. However too much amplification can also exceed the time constraints making the algorithm ineffective.

→ Classification of Randomized algorithms is based on whether they have time constraints as the random variable or deterministic value. (3)



① Las Vegas:- This algorithm executes in a predetermined period of time. It is predictable that if it runs out of time and doesn't find any solutions, but if it finds one within that window, it will be precisely correct.

So, this is randomized algorithm that consistently yields the right answer or fails. The time complexity depends on input. Virtually every search result contains a las vegas algo.

Ex:- Randomized quicksort

→ Quicksort is a common las vegas randomized sorting that uses no additional memory and sorts elements in place. Since, this is comparision based, the worst case will happen when doing a pairwise comparision which takes $O(n^2)$ time complexity.

→ However, with Randomization, this algorithms worst case time complexity can be lowered to $O(n \log(n))$. The two ways to randomize quicksort

(i) Randomly shuffling inputs.

(ii) Randomly choosing Pivot element.

The algorithm exactly follows the standard algorithm except it randomizes the pivot selection.

```
Algorithm RandomizedQuickSort(A, n)
{
// Input is Array A with "n" elements
if n = 1                    // only one element so already sorted
    return A
Else
{
    i = Random number in range(1, n)
                            // select pivot randomly.
    X = A[i]
    Partition A in to elements <X, ==X, >X
}
Apply QuickSort on A[1 to i-1] and A[i+1 to n]
Combine all data to obtain sorted array
}
```

If pivot is randomly selected at first or final element in array, worst case scenario is it takes $O(n^2)$

If random has selected the pivot which is neither smallest nor largest number, then it takes $O(n \log n)$

(2) Monte Carlo Algorithms

To approximate numerical values or solve issues probabilistically, Monte Carlo uses randomization with a certain degree of assurance, they offer an approximate solution.

ex:- Calculating value of π,
Resolving optimization problems.
Doing probabilistic simulations.

The goal of Monte carlo approach for randomized algorithms is to complete the execution within the allotted time limit. This method's running time is therefore predictable. For example, for performing string matching, Monte carlo starts procedure over from the last error its encountered. Thus time is saved.

The deterministic algorithm o/p is always anticipated to be right. whereas Monte carlo techniques cannot guarantee this. These algorithms are typically categorized as either false-biased or true biased for decision issues.

* When a false-biased Monte carlo algorithm delivers false, it is always correct.
* A true biased method always yields true.

Ex:- Approximating $\pi(P_i)$

Relation b/w Monte carlo and las Vegas

By executing a las vegas algo for a predetermined amount of time and producing a random response when it fails to terminate, it can be transformed in to a Monte carlo algorithm.

	Monte carlo	las Vegas
Correctness	Probabilistic	Certain
Running Time	Certain [with in given time constraint]	Probabilistic
example	Approximate π Kargers Minimum cut algo	Randomized quicksort
working	In string matching, if error occurs, it restarts from same point.	In string matching, if error occurs, it starts from beginning

Advantages

1. Simplicity
2. efficiency
3. Approximate solutions [when exact solutions are hard & unnecessary]
4. Probabilistic Correctness
5. Versatility
6. Parallelism
7. Privacy
8. Robustness

Limitations

1. Probabilistic output
2. Analysis complexity
3. Deterministic alternatives
4. Difficulty in Reproduction
5. Resource usage
6. Not always suitable
7. Error control

Note: For algorithms like mergesort the time complexity does not depend on the input, even if algorithm is randomized the time complexity will always remain same. So, randomization is only applied on algorithms whose complexity depends on input.

Four randomized algorithms complexity cases:

1. RP (Randomized complexity)
2. Co RP (complement of RP)
3. BPP (Bounded error probabilistic polynomial time)
4. ZPP (Zero error probabilistic Polynomial time)