**Stack :** <https://www.hackerearth.com/practice/data-structures/stacks/basics-of-stacks/tutorial/>

**Queue:** <https://www.hackerearth.com/practice/data-structures/queues/basics-of-queues/tutorial/>

**Selection sort:** <https://www.hackerearth.com/practice/algorithms/sorting/selection-sort/tutorial/>

**Radix Sort :** <https://www.hackerearth.com/practice/algorithms/sorting/radix-sort/tutorial/>

**Quick Sort :** <https://www.hackerearth.com/practice/algorithms/sorting/quick-sort/tutorial/>

**Branch-and-Bound**

**Branch and bound** is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. The Branch and Bound Algorithm technique solves these problems relatively quickly. The branch and bound algorithm is similar to backtracking but is used for optimization problems. It performs a graph transversal on the space-state tree, but general searches BFS instead of DFS. During the search bounds for the objective function on the partial solution are determined. At each level the best bound is explored first, the technique is called best bound first.  If a complete solution is found then that value of the objective function can be used to prune partial solutions that exceed the bounds. The difficult of designing branch and bound algorithm is finding good bounding function. The bounding the function should be inexpensive to calculate but should be effective at selecting the most promising partial solution.

**TSP Problem**

A branch-and-bound algorithm consists of a systematic enumeration of all candidate solutions, where large subsets of fruitless candidate s are discarded ,by using upper and lower estimated bounds of the quantity being optimized. The Branch and Bound strategy divides a problem to be solved into a number of sub-problems. It is a system for solving a sequence of subproblems each of which may have multiple possible solutions and where the solution chosen for one sub-problem may affect the possible solutions of later sub-problems. Suppose it is required to minimize an objective function. Suppose that we have a method for getting a lower bound on the cost of any solution among those in the set of solutions represented by some subset. If the best solution found so far costs less than the lower bound for this subset, we need not explore this subset at all. Let S be some subset of solutions. L(S)=a lower bound on the cost of any solution belonging to S Let C=cost of the best solution found so far If C ≤ L(S),there is no need to explore S because it does not contain any better solution. If C > L(S),then we need to explore S because it may contain a better solution.

**Note : Example of this is discussed in class**

**Backtracking**

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree).

**4 - Queen's problem**

In **4- queens problem**, we have 4 queens to be placed on a 4\*4 chessboard, satisfying the constraint that no two queens should be in the same row, same column, or in same diagonal.

The solution space according to the external constraints consists of 4 to the power 4, 4-tuples i.e., **Si = {1, 2, 3, 4}** and **1<= I <=4**, whereas according to the internal constraints they consist of **4!** solutions i.e., permutation of **4**.

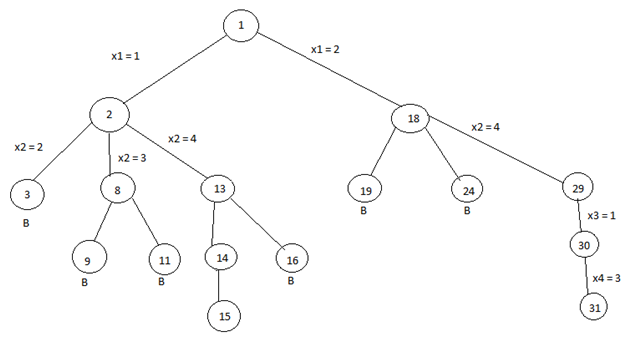
Solution of 4 – queen’s with the help of backtracking

We can solve 4-queens problem through backtracking by taking it as a bounding function .in use the criterion that if (x1, x2, ……., xi) is a path to a current E-node, then all the children nodes with parent-child labelings x (i+1) are such that (x1, x2, x3, ….., x(i+1)) represents a chessboard configuration in which no queens are attacking.

So we start with the root node as the only live node. This time this node becomes the E-node and the path is (). We generate the next child. Suppose we are generating the child in ascending order. Thus the node number 2 is generated and path is now 1 i.e., the queen 1 is placed in the first row and in the first column.

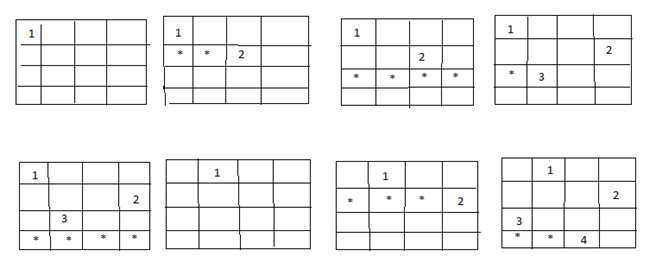
Now, node 2 becomes the next E-node or line node. Further, try the next node in the ascending nodes i.e., the node 3 which is having x2 = 2 means queen 2 is placed in the second column but by this the queen 1 and 2 are on the same diagonal, so node 3 becomes dead here so we backtrack it and try the next node which is possible.

Here, the x2 = 3 means the queen 2 is placed in the 3rd column. As it satisfies all the constraints so it becomes the next live node.

After this try for next node 9 having x3 = 2 which means the queen 3 placed in the 2nd column, but by this the 2 and 3 queen are on the same diagonal so it becomes dead. Now we try for next node 11 with x3 = 4, but again the queens 2 and 3 are on the same diagonal so it is also a dead node.  


**\*** The B denotes the dead node.

We try for all the possible positions for the queen 3 and if not any position satisfy all the constraints then backtrack to the previous live node.

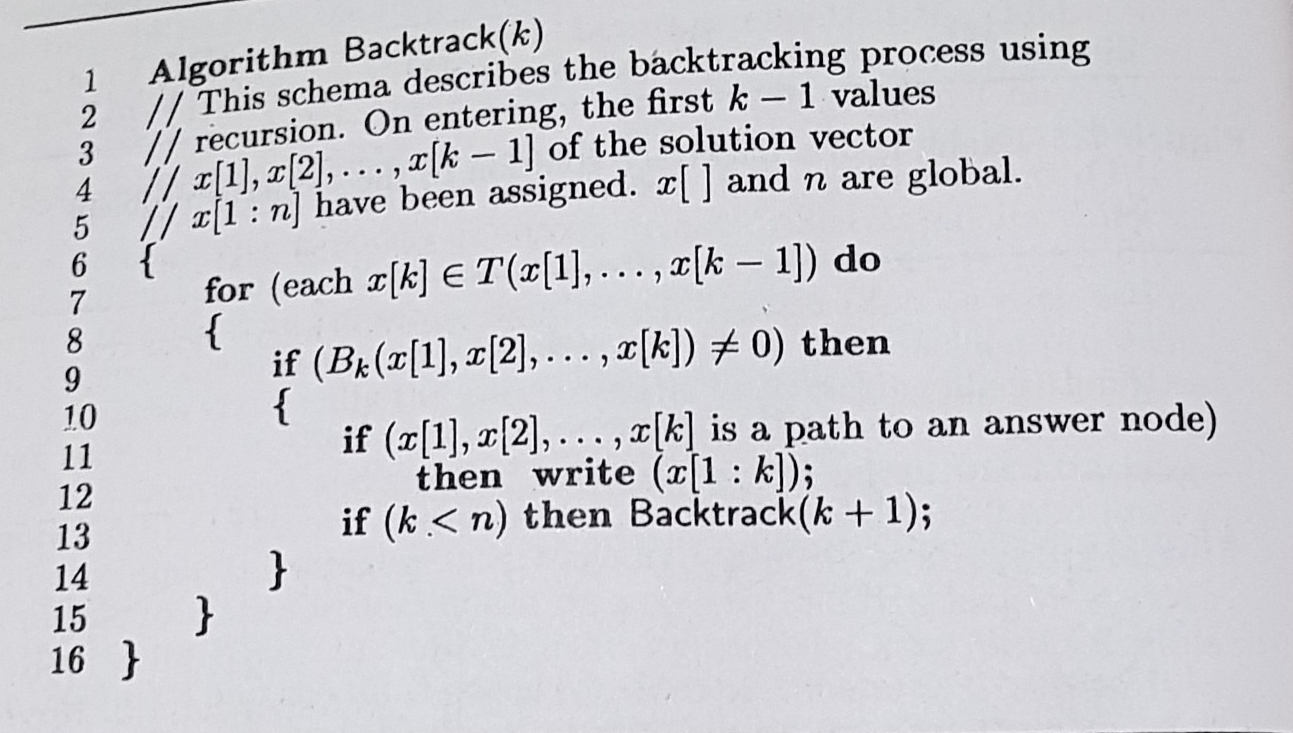


Now, the node13 become the new live node with x2 = 4, means queen 2 is placed in the 4th column. Move to the next node 14. It becomes the next live node with x3 = 2 means the queen 3 is placed in the 2nd column. Further, we move to the next node 15 with x4 = 3 as the live node. But this makes the queen 3 and 4 on the same diagonal resulting this node 15 is the dead node so we have to backtrack to the node 14 and then backtrack to the node 13 and try the other possible node 16 with x3 = 3 by this also we get the queens 2 and 3 on the same diagonal so the node is the dead node.

So we further backtrack to the node 2 but no other node is left to try so the node 2 is killed so we backtrack to the node 1 and try another sub-tree having x1 = 2 which means queen 1 is placed in the 2nd column.

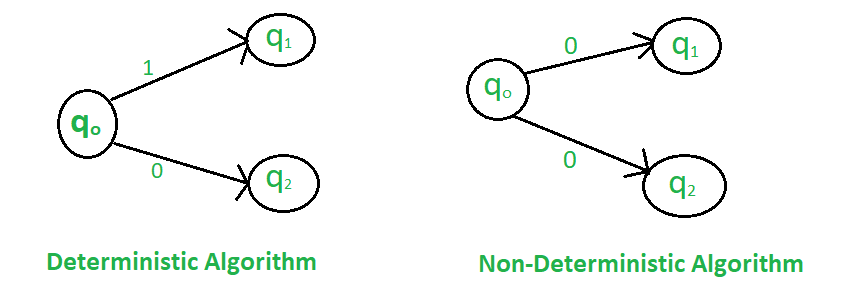
Now again with the similar reason, nodes 19 and 24 are killed and so we try for the node 29 with x2 = 4 means the queen 2 is placed in the 4th column then we try for the node 30 with x3 = 1 as a live node and finally we proceed to next node 31 with x4 = 3 means the queen 4 is placed in 3rd column.

Here, all the constraints are satisfied, so the desired result for 4 queens is {2, 4, 1, 3}.



**Deterministic and Non deterministic algorithm**

In deterministic algorithm, for a given particular input, the computer will always produce the same output going through the same states but in case of non-deterministic algorithm, for the same input, the compiler may produce different output in different runs. In fact non-deterministic algorithms can’t solve the problem in polynomial time and can’t determine what is the next step. The non-deterministic algorithms can show different behaviors for the same input on different execution and there is a degree of randomness to it.



To implement a non-deterministic algorithm, we have a couple of languages like Prolog but these don’t have standard programming language operators and these operators are not a part of any standard programming languages.

**Some of the terms related to the non-deterministic algorithm are defined below**:

* **choice(X) :** chooses any value randomly from the set X.
* **failure() :** denotes the unsuccessful solution.
* **success() :** Solution is successful and current thread terminates.

**Example :**

***Problem Statement :****Search an element x on A[1:n] where n>=1, on successful search return j if a[j] is equals to x otherwise return 0.*

***Non-deterministic Algorithm for this problem :***

*1.j= choice(a, n)*

*2.if(A[j]==x) then*

*{*

*write(j);*

*success();*

*}*

*3.write(0); failure();*

| **DETERMINISTIC ALGORITHM** | **NON-DETERMINISTIC ALGORITHM** |
| --- | --- |
| 1. For a particular input the computer will give always same output. | 1. For a particular input the computer will give different output on different execution. |
| 1. Can solve the problem in polynomial time. | 1. Can’t solve the problem in polynomial time. |
| 1. Can determine the next step of execution. | 1. Cannot determine the next step of execution due to more than one path the algorithm can take. |

**P , NP, NP HARD AND NP COMPLETE**

P is set of problems that can be solved by a deterministic Turing machine in Polynomial time. Problems which can be solved in polynomial time, which take time like O(n), O(n2), O(n3). Eg: finding maximum element in an array or to check whether a string is palindrome or not. so there are many problems which can be solved in polynomial time

NP is set of decision problems that can be solved by a Non-deterministic Turing Machine in Polynomial 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). **NP**- Non deterministic Polynomial time solving. Problem which can't be solved in polynomial time like TSP( travelling salesman problem) or An easy example of this is subset sum: given a set of numbers, does there exist a subset whose sum is zero?. but NP problems are **checkable** in polynomial time means that given a solution of a problem , we can check that whether the solution is correct or not in polynomial time.

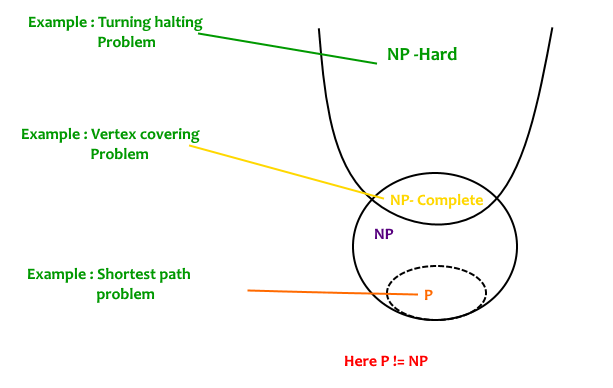
**NP-hard**-- Now suppose we found that A is reducible to B, then it means that B is at least as hard as A.

**NP-Complete** -- The group of problems which are both in NP and NP-hard are known as NP-Complete problem.

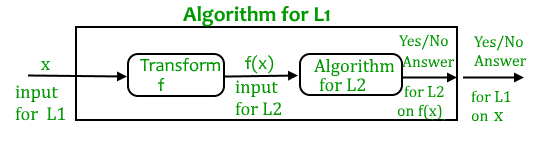
Now suppose we have a NP-Complete problem R and it is reducible to Q then Q is at least as hard as R and since R is an NP-hard problem. therefore Q will also be at least NP-hard , it may be NP-complete also.  
Informally, NP is set of decision problems which can be solved by a polynomial time via a “Lucky Algorithm”, a magical algorithm that always makes a right guess among the given set of choices

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.   
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 L1 and L2 be two decision problems. Suppose algorithm A2 solves L2. That is, if y is an input for L2 then algorithm A2 will answer Yes or No depending upon whether y belongs to L2 or not.  
The idea is to find a transformation from L1 to L2 so that the algorithm A2 can be part of an algorithm A1 to solve L1.  
  
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.

We will soon be discussing more NP-Complete problems and their proof for NP-Completeness.