**Find if there is a path of more than k length from a source**

Given a graph, a source vertex in the graph and a number k, find if there is a simple path (without any cycle) starting from given source and ending at any other vertex such that the distance from source to that vertex is atleast ‘k’ length.

**Example:**

Input : Source s = 0, k = 58

Output : True

There exists a simple path 0 -> 7 -> 1

-> 2 -> 8 -> 6 -> 5 -> 3 -> 4

Which has a total distance of 60 km which

is more than 58.

Input : Source s = 0, k = 62

Output : False

In the above graph, the longest simple

path has distance 61 (0 -> 7 -> 1-> 2

-> 3 -> 4 -> 5-> 6 -> 8, so output

should be false for any input greater

than 61.

One important thing to note is, simply doing BFS or DFS and picking the longest edge at every step would not work. The reason is, a shorter edge can produce longer path due to higher weight edges connected through it.  
The idea is to use Backtracking. We start from given source, explore all paths from current vertex. We keep track of current distance from source. If distance becomes more than k, we return true. If a path doesn’t produces more than k distance, we backtrack.  
How do we make sure that the path is simple and we don’t loop in a cycle? The idea is to keep track of current path vertices in an array. Whenever we add a vertex to path, we check if it already exists or not in current path. If it exists, we ignore the edge.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/backtracking/FindPathMoreThanKLengthFromSource.java>

**Tug of War**

Given a set of n integers, divide the set in two subsets of n/2 sizes each such that the difference of the sum of two subsets is as minimum as possible. If n is even, then sizes of two subsets must be strictly n/2 and if n is odd, then size of one subset must be (n-1)/2 and size of other subset must be (n+1)/2.  
For example, let given set be {3, 4, 5, -3, 100, 1, 89, 54, 23, 20}, the size of set is 10. Output for this set should be {4, 100, 1, 23, 20} and {3, 5, -3, 89, 54}. Both output subsets are of size 5 and sum of elements in both subsets is same (148 and 148).   
Let us consider another example where n is odd. Let given set be {23, 45, -34, 12, 0, 98, -99, 4, 189, -1, 4}. The output subsets should be {45, -34, 12, 98, -1} and {23, 0, -99, 4, 189, 4}. The sums of elements in two subsets are 120 and 121 respectively.  
The following solution tries every possible subset of half size. If one subset of half size is formed, the remaining elements form the other subset. We initialize current set as empty and one by one build it. There are two possibilities for every element, either it is part of current set, or it is part of the remaining elements (other subset). We consider both possibilities for every element. When the size of current set becomes n/2, we check whether this solutions is better than the best solution available so far. If it is, then we update the best solution.

**Time Complexity:** O(2^n)

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/backtracking/TugOfWar.java>

**The Knight’s tour problem | Backtracking-1**

Backtracking is an algorithmic paradigm that tries different solutions until finds a solution that “works”. Problems that are typically solved using the backtracking technique have the following property in common. These problems can only be solved by trying every possible configuration and each configuration is tried only once. A Naive solution for these problems is to try all configurations and output a configuration that follows given problem constraints. Backtracking works incrementally and is an optimization over the Naive solution where all possible configurations are generated and tried.  
For example, consider the following [Knight’s Tour](http://en.wikipedia.org/wiki/Knight%27s_tour) problem.

**Problem Statement:**  
Given a N\*N board with the Knight placed on the first block of an empty board. Moving according to the rules of chess knight must visit each square exactly once. Print the order of each cell in which they are visited.

**Example:**

Input :

N = 8

Output:

0 59 38 33 30 17 8 63

37 34 31 60 9 62 29 16

58 1 36 39 32 27 18 7

35 48 41 26 61 10 15 28

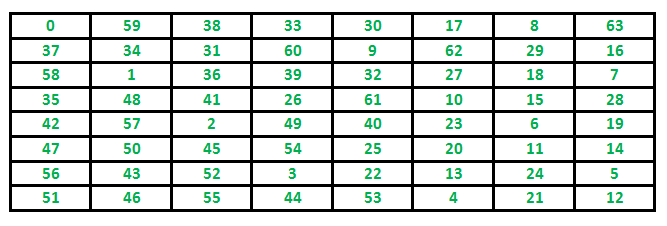
42 57 2 49 40 23 6 19

47 50 45 54 25 20 11 14

56 43 52 3 22 13 24 5

51 46 55 44 53 4 21 12

**The path followed by Knight to cover all the cells**  
Following is a chessboard with 8 x 8 cells. Numbers in cells indicate the move number of Knight.



Let us first discuss the Naive algorithm for this problem and then the Backtracking algorithm.

**Naive Algorithm for Knight’s tour**   
The Naive Algorithm is to generate all tours one by one and check if the generated tour satisfies the constraints.

while there are untried tours

{

generate the next tour

if this tour covers all squares

{

print this path;

}

}

**Backtracking**works in an incremental way to attack problems. Typically, we start from an empty solution vector and one by one add items (Meaning of item varies from problem to problem. In the context of Knight’s tour problem, an item is a Knight’s move). When we add an item, we check if adding the current item violates the problem constraint, if it does then we remove the item and try other alternatives. If none of the alternatives works out then we go to the previous stage and remove the item added in the previous stage. If we reach the initial stage back then we say that no solution exists. If adding an item doesn’t violate constraints then we recursively add items one by one. If the solution vector becomes complete then we print the solution.

**Backtracking Algorithm for Knight’s tour**

Following is the Backtracking algorithm for Knight’s tour problem.

If all squares are visited

print the solution

Else

a) Add one of the next moves to solution vector and recursively

check if this move leads to a solution. (A Knight can make maximum

eight moves. We choose one of the 8 moves in this step).

b) If the move chosen in the above step doesn't lead to a solution

then remove this move from the solution vector and try other

alternative moves.

c) If none of the alternatives work then return false (Returning false

will remove the previously added item in recursion and if false is

returned by the initial call of recursion then "no solution exists" )

Following are implementations for Knight’s tour problem. It prints one of the possible solutions in 2D matrix form. Basically, the output is a 2D 8\*8 matrix with numbers from 0 to 63 and these numbers show steps made by Knight.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/backtracking/KnightTourProblem.java>

**ime Complexity :**  
There are N2 Cells and for each, we have a maximum of 8 possible moves to choose from, so the worst running time is O(8N^2).

**Auxiliary Space:**O(N2)

**Important Note:**  
No order of the xMove, yMove is wrong, but they will affect the running time of the algorithm drastically. For example, think of the case where the 8th choice of the move is the correct one, and before that our code ran 7 different wrong paths. It’s always a good idea a have a heuristic than to try backtracking randomly. Like, in this case, we know the next step would probably be in the south or east direction, then checking the paths which lead their first is a better strategy.

Note that Backtracking is not the best solution for the Knight’s tour problem. See the below article for other better solutions. The purpose of this post is to explain Backtracking with an example.   
[Warnsdorff’s algorithm for Knight’s tour problem](https://www.geeksforgeeks.org/warnsdorffs-algorithm-knights-tour-problem/)

**Rat in a Maze | Backtracking-2**

We have discussed Backtracking and Knight’s tour problem in [Set 1](https://www.geeksforgeeks.org/backtracking-set-1-the-knights-tour-problem/). Let us discuss Rat in a [Maze](http://en.wikipedia.org/wiki/Maze)as another example problem that can be solved using Backtracking.

A Maze is given as N\*N binary matrix of blocks where source block is the upper left most block i.e., maze[0][0] and destination block is lower rightmost block i.e., maze[N-1][N-1]. A rat starts from source and has to reach the destination. The rat can move only in two directions: forward and down.

In the maze matrix, 0 means the block is a dead end and 1 means the block can be used in the path from source to destination. Note that this is a simple version of the typical Maze problem. For example, a more complex version can be that the rat can move in 4 directions and a more complex version can be with a limited number of moves.

**Following is an example maze.**

Gray blocks are dead ends (value = 0).



Following is a binary matrix representation of the above maze.

{1, 0, 0, 0}

{1, 1, 0, 1}

{0, 1, 0, 0}

{1, 1, 1, 1}

Following is a maze with highlighted solution path.



Following is the solution matrix (output of program) for the above input matrix.

{1, 0, 0, 0}

{1, 1, 0, 0}

{0, 1, 0, 0}

{0, 1, 1, 1}

All entries in solution path are marked as 1.

[Recommended: Please solve it on “***PRACTICE***” first, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/rat-in-a-maze-problem/1)

[**Backtracking Algorithm**](https://www.geeksforgeeks.org/backtracking-algorithms/)**:** Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally. Solving one piece at a time, and 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) is the process of backtracking.

**Approach:**Form a recursive function, which will follow a path and check if the path reaches the destination or not. If the path does not reach the destination then backtrack and try other paths.

**Algorithm:**

1. Create a solution matrix, initially filled with 0’s.
2. Create a recursive function, which takes initial matrix, output matrix and position of rat (i, j).
3. if the position is out of the matrix or the position is not valid then return.
4. Mark the position output[i][j] as 1 and check if the current position is destination or not. If destination is reached print the output matrix and return.
5. Recursively call for position (i+1, j) and (i, j+1).
6. Unmark position (i, j), i.e output[i][j] = 0.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/backtracking/RatInMaze.java>

**Output:**   
The 1 values show the path for rat

1 0 0 0

1 1 0 0

0 1 0 0

0 1 1 1

**Complexity Analysis:**

* **Time Complexity:** O(2^(n^2)).   
  The recursion can run upper-bound 2^(n^2) times.
* **Space Complexity:** O(n^2).   
  Output matrix is required so an extra space of size n\*n is needed.

**Count number of ways to reach destination in a Maze**

Given a maze with obstacles, count the number of paths to reach the rightmost-bottommost cell from the topmost-leftmost cell. A cell in the given maze has a value of -1 if it is a blockage or dead-end, else 0.  
From a given cell, we are allowed to move to cells (i+1, j) and (i, j+1) only.

**Examples:**

Input: maze[R][C] = {{0, 0, 0, 0},

{0, -1, 0, 0},

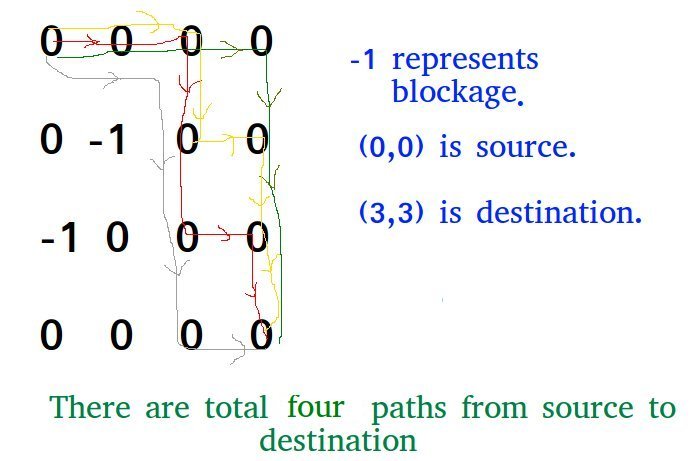
{-1, 0, 0, 0},

{0, 0, 0, 0}};

Output: 4

There are four possible paths as shown in

below diagram.



This problem is an extension of the below problem.

[Backtracking | Set 2 (Rat in a Maze)](https://www.geeksforgeeks.org/backttracking-set-2-rat-in-a-maze/)  
In this post, a different solution is discussed that can be used to solve the above Rat in a Maze problem also.  
The idea is to modify the given grid[][] so that **grid[i][j] contains count of paths to reach (i, j) from (0, 0) if (i, j) is not a blockage, else grid[i][j] remains -1.**

We can recursively compute grid[i][j] using below

formula and finally return grid[R-1][C-1]

// If current cell is a blockage

if (maze[i][j] == -1)

maze[i][j] = -1; // Do not change

// If we can reach maze[i][j] from maze[i-1][j]

// then increment count.

else if (maze[i-1][j] > 0)

maze[i][j] = (maze[i][j] + maze[i-1][j]);

// If we can reach maze[i][j] from maze[i][j-1]

// then increment count.

else if (maze[i][j-1] > 0)

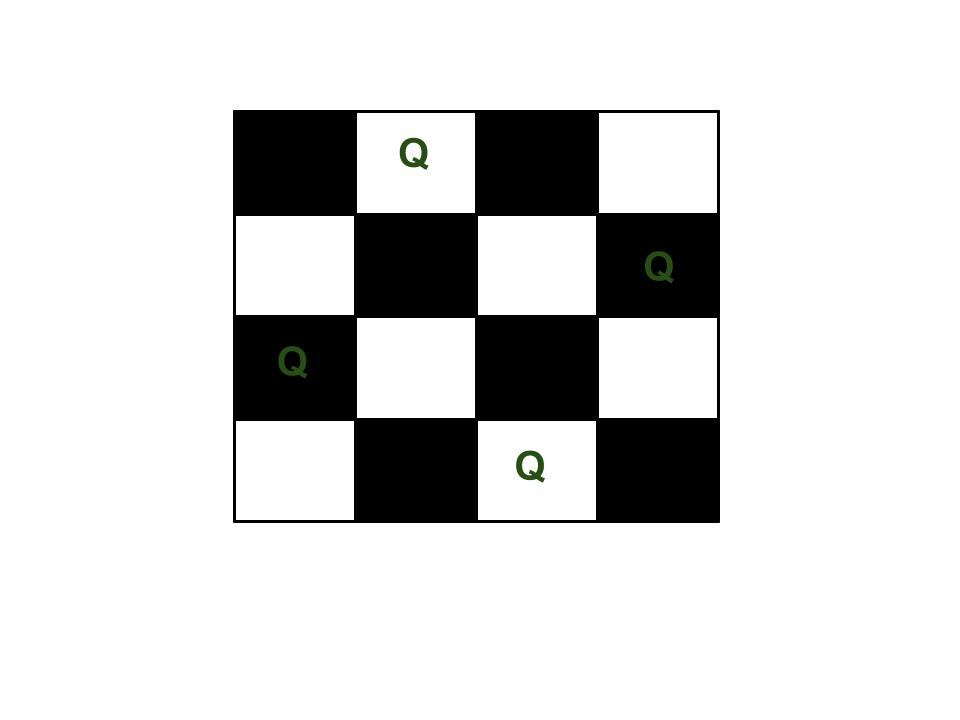
maze[i][j] = (maze[i][j] + maze[i][j-1]);

Below is the implementation of the above idea.

**N Queen Problem | Backtracking-3**

We have discussed Knight’s tour and Rat in a Maze problems in [Set 1](https://www.geeksforgeeks.org/backtracking-set-1-the-knights-tour-problem/) and [Set 2](https://www.geeksforgeeks.org/backttracking-set-2-rat-in-a-maze/) respectively. Let us discuss N Queen as another example problem that can be solved using Backtracking.

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.

[](https://media.geeksforgeeks.org/wp-content/uploads/N_Queen_Problem.jpg)

The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example, following is the output matrix for above 4 queen solution.

{ 0, 1, 0, 0}

{ 0, 0, 0, 1}

{ 1, 0, 0, 0}

{ 0, 0, 1, 0}

[Recommended: Please solve it on “***PRACTICE*** ” first, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/n-queen-problem/0)

**Naive Algorithm**  
Generate all possible configurations of queens on board and print a configuration that satisfies the given constraints.

while there are untried configurations

{

generate the next configuration

if queens don't attack in this configuration then

{

print this configuration;

}

}

**Backtracking Algorithm**  
The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes then we backtrack and return false.

1) Start in the leftmost column

2) If all queens are placed

return true

3) Try all rows in the current column.

Do following for every tried row.

a) If the queen can be placed safely in this row

then mark this [row, column] as part of the

solution and recursively check if placing

queen here leads to a solution.

b) If placing the queen in [row, column] leads to

a solution then return true.

c) If placing queen doesn't lead to a solution then

unmark this [row, column] (Backtrack) and go to

step (a) to try other rows.

3) If all rows have been tried and nothing worked,

return false to trigger backtracking.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/backtracking/NQueenProblem.java>

**m Coloring Problem | Backtracking-5**

Given an undirected graph and a number m, determine if the graph can be coloured with at most m colours such that no two adjacent vertices of the graph are colored with the same color. Here coloring of a graph means the assignment of colors to all vertices.

**Input-Output format:**

***Input:***

1. A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is an adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.
2. An integer m is the maximum number of colors that can be used.

***Output:***   
An array color[V] that should have numbers from 1 to m. color[i] should represent the color assigned to the ith vertex. The code should also return false if the graph cannot be colored with m colors.

**Example:**

**Input:**

graph = {0, 1, 1, 1},

{1, 0, 1, 0},

{1, 1, 0, 1},

{1, 0, 1, 0}

**Output:**

Solution Exists:

Following are the assigned colors

1 2 3 2

**Explanation:** By coloring the vertices

with following colors, adjacent

vertices does not have same colors

**Input:**

graph = {1, 1, 1, 1},

{1, 1, 1, 1},

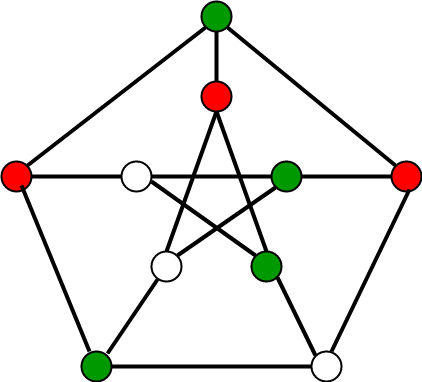
{1, 1, 1, 1},

{1, 1, 1, 1}

**Output:** Solution does not exist.

**Explanation:** No solution exits.

**Following is an example of a graph that can be coloured with 3 different colours.** 



**Method 2:**[**Backtracking**](https://www.geeksforgeeks.org/backtracking-algorithms/)**.**

Approach: The idea is to assign colors one by one to different vertices, starting from the vertex 0. Before assigning a color, check for safety by considering already assigned colors to the adjacent vertices i.e check if the adjacent vertices have the same color or not. If there is any color assignment that does not violate the conditions, mark the color assignment as part of the solution. If no assignment of color is possible then backtrack and return false.

Algorithm:

1. Create a recursive function that takes the graph, current index, number of vertices, and output color array.
2. If the current index is equal to the number of vertices. Print the color configuration in output array.
3. Assign a color to a vertex (1 to m).
4. For every assigned color, check if the configuration is safe, (i.e. check if the adjacent vertices do not have the same color) recursively call the function with next index and number of vertices
5. If any recursive function returns true break the loop and return true.
6. If no recursive function returns true then return false.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/backtracking/MColoringProblem.java>

**Hamiltonian Cycle | Backtracking-6**

[Hamiltonian Path](http://en.wikipedia.org/wiki/Hamiltonian_path) in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in the graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then prints the path. Following are the input and output of the required function.  
*Input:*   
A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.  
*Output:*   
An array path[V] that should contain the Hamiltonian Path. path[i] should represent the ith vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.  
For example, a Hamiltonian Cycle in the following graph is {0, 1, 2, 4, 3, 0}.

(0)--(1)--(2)

| / \ |

| / \ |

| / \ |

(3)-------(4)

And the following graph doesn’t contain any Hamiltonian Cycle.

(0)--(1)--(2)

| / \ |

| / \ |

| / \ |

(3) (4)

[Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/hamiltonian-path/0)

Naive Algorithm   
Generate all possible configurations of vertices and print a configuration that satisfies the given constraints. There will be n! (n factorial) configurations.

while there are untried conflagrations

{

generate the next configuration

if ( there are edges between two consecutive vertices of this

configuration and there is an edge from the last vertex to

the first ).

{

print this configuration;

break;

}

}

Backtracking Algorithm   
Create an empty path array and add vertex 0 to it. Add other vertices, starting from the vertex 1. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return false.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/backtracking/HamiltonianCycle.java>