Backtracking Algorithms

[**Learn more about Backtracking Algorithms in DSA Self Paced Course**](https://practice.geeksforgeeks.org/courses/dsa-self-paced?utm_source=geeksforgeeks&utm_medium=articles+backtracking_lp+header_link_click&utm_campaign=dsa+course+tracker)

[**Practice Problems on Backtracking Algorithms**](https://practice.geeksforgeeks.org/topics/backtracking/?utm_source=geeksforgeeks&utm_medium=articles+backtracking_lp+header_link_click&utm_campaign=practice+tracker)

[**Recent Articles on Backtracking Algorithms**](https://www.geeksforgeeks.org/category/algorithm/backtracking/)

**What is Backtracking?**

***Backtracking****can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.*

**What is Backtracking Algorithm?**

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).

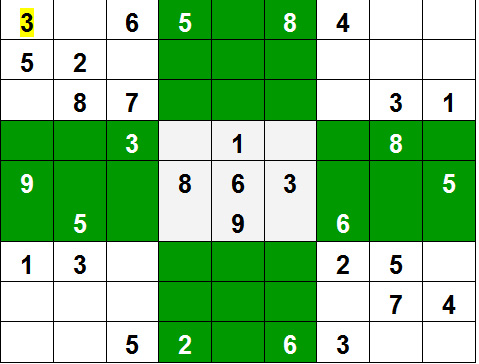
**Types of Backtracking Algorithm**

There are three types of problems in backtracking

1. Decision Problem – In this, we search for a feasible solution.
2. Optimization Problem – In this, we search for the best solution.
3. Enumeration Problem – In this, we find all feasible solutions.

**When can be Backtracking Algorithm used?**

For example, consider the SudoKo solving Problem, we try filling digits one by one. Whenever we find that current digit cannot lead to a solution, we remove it (backtrack) and try next digit. This is better than naive approach (generating all possible combinations of digits and then trying every combination one by one) as it drops a set of permutations whenever it backtracks.

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

**Topics :**

* [Introduction](https://www.geeksforgeeks.org/backtracking-algorithms/#introduction)
* [Standard Problems on Backtracking](https://www.geeksforgeeks.org/backtracking-algorithms/#standard)
* [Some Practice problems on Backtracking](https://www.geeksforgeeks.org/backtracking-algorithms/#practice)
* [Quick Links](https://www.geeksforgeeks.org/backtracking-algorithms/#quick)

**Introduction:**

1. [Introduction to Backtracking – Data Structure and Algorithm Tutorials](https://www.geeksforgeeks.org/introduction-to-backtracking-data-structure-and-algorithm-tutorials/)
2. [Difference between Backtracking and Branch-N-Bound technique](https://www.geeksforgeeks.org/difference-between-backtracking-and-branch-n-bound-technique/)
3. [What is the difference between Backtracking and Recursion?](https://www.geeksforgeeks.org/what-is-the-difference-between-backtracking-and-recursion/)

**Standard Problems on Backtracking:**

1. [The Knight’s tour problem](https://www.geeksforgeeks.org/the-knights-tour-problem/)
2. [Rat in a Maze](https://www.geeksforgeeks.org/rat-in-a-maze/)
3. [N Queen Problem | Backtracking-3](https://www.geeksforgeeks.org/n-queen-problem-backtracking-3/)
4. [Subset Sum problem](https://www.geeksforgeeks.org/subset-sum-problem/)
5. [m Coloring Problem](https://www.geeksforgeeks.org/m-coloring-problem/)
6. [Hamiltonian Cycle](https://www.geeksforgeeks.org/hamiltonian-cycle/)
7. [Sudoku | Backtracking-7](https://www.geeksforgeeks.org/sudoku-backtracking-7/)
8. [Magnet Puzzle](https://www.geeksforgeeks.org/magnet-puzzle/)
9. [Remove Invalid Parentheses](https://www.geeksforgeeks.org/remove-invalid-parentheses/)
10. [A backtracking approach to generate n bit Gray Codes](https://www.geeksforgeeks.org/backtracking-approach-generate-n-bit-gray-codes/)
11. [Write a program to print all permutations of a given string](https://www.geeksforgeeks.org/write-a-c-program-to-print-all-permutations-of-a-given-string/)

**Some Practice problems on Backtracking:**

* **Easy:**
  1. [Backtracking to find all subsets](https://www.geeksforgeeks.org/backtracking-to-find-all-subsets/)
  2. [Check if a given string is sum-string](https://www.geeksforgeeks.org/check-given-string-sum-string/)
  3. [Count all possible paths between two vertices](https://www.geeksforgeeks.org/count-possible-paths-two-vertices/)
  4. [Find all distinct subsets of a given set](https://www.geeksforgeeks.org/find-distinct-subsets-given-set/)
  5. [Find if there is a path of more than k length from a source](https://www.geeksforgeeks.org/find-if-there-is-a-path-of-more-than-k-length-from-a-source/)
  6. [Print all paths from a given source to a destination](https://www.geeksforgeeks.org/find-paths-given-source-destination/)
  7. [Print all possible strings that can be made by placing spaces](https://www.geeksforgeeks.org/print-possible-strings-can-made-placing-spaces/)
* **Medium:**
  1. [Tug of War](https://www.geeksforgeeks.org/tug-of-war/)
  2. [8 queen problem](https://www.geeksforgeeks.org/8-queen-problem/)
  3. [Combinational Sum](https://www.geeksforgeeks.org/combinational-sum/)
  4. [Warnsdorff’s algorithm for Knight’s tour problem](https://www.geeksforgeeks.org/warnsdorffs-algorithm-knights-tour-problem/)
  5. [Find paths from corner cell to middle cell in maze](https://www.geeksforgeeks.org/find-paths-from-corner-cell-to-middle-cell-in-maze/)
  6. [Find Maximum number possible by doing at-most K swaps](https://www.geeksforgeeks.org/find-maximum-number-possible-by-doing-at-most-k-swaps/)
  7. [Rat in a Maze with multiple steps or jump allowed](https://www.geeksforgeeks.org/rat-in-a-maze-with-multiple-steps-jump-allowed/)
  8. [N Queen in O(n) space](https://www.geeksforgeeks.org/n-queen-in-on-space/)
* **Hard:**
  1. [Power Set in Lexicographic order](https://www.geeksforgeeks.org/powet-set-lexicographic-order/)
  2. [Word Break Problem using Backtracking](https://www.geeksforgeeks.org/word-break-problem-using-backtracking/)
  3. [Partition of a set into K subsets with equal sum](https://www.geeksforgeeks.org/partition-set-k-subsets-equal-sum/)
  4. [Longest Possible Route in a Matrix with Hurdles](https://www.geeksforgeeks.org/longest-possible-route-in-a-matrix-with-hurdles/)
  5. [Find shortest safe route in a path with landmines](https://www.geeksforgeeks.org/find-shortest-safe-route-in-a-path-with-landmines/)
  6. [Print all palindromic partitions of a string](https://www.geeksforgeeks.org/print-palindromic-partitions-string/)
  7. [Printing all solutions in N-Queen Problem](https://www.geeksforgeeks.org/printing-solutions-n-queen-problem/)
  8. [Print all longest common sub-sequences in lexicographical order](https://www.geeksforgeeks.org/print-longest-common-sub-sequences-lexicographical-order/)

**Quick Links :**

* [**Learn Data Structure and Algorithms | DSA Tutorial**](https://www.geeksforgeeks.org/learn-data-structures-and-algorithms-dsa-tutorial?utm_source=Website&utm_medium=Landing+Page+Click&utm_campaign=DSA+Page+Tracker&utm_id=DSA-Page-Tracker&utm_term=DSA+Page+Promo&utm_content=Course+Page)
* [Top 20 Backtracking Algorithm Interview Questions](https://www.geeksforgeeks.org/top-20-backtracking-algorithm-interview-questions/)
* [‘Practice Problems’ on Backtracking](https://practice.geeksforgeeks.org/topics/backtracking/)
* [‘Quiz’ on Backtracking](https://www.geeksforgeeks.org/algorithms-gq/backtracking-gq/)
* [‘Videos’ on Backtracking](https://www.youtube.com/playlist?list=PLqM7alHXFySFbuucq7lC8ecRoWZxq1qS5)

**Easy Questions:**

**Tug of War**

Given a set of n integers, divide the set in two subsets of n/2 sizes each such that the absolute 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.

Following is the implementation for Tug of War problem. It prints the required arrays.

# Python3 program for above approach

# function that tries every possible

# solution by calling itself recursively

**def** TOWUtil(arr, n, curr\_elements, no\_of\_selected\_elements,

            soln, min\_diff, Sum, curr\_sum, curr\_position):

    # checks whether the it is going

    # out of bound

**if** (curr\_position **==** n):

**return**

    # checks that the numbers of elements

    # left are not less than the number of

    # elements required to form the solution

**if** ((int(n **/** 2) **-** no\_of\_selected\_elements) >

                          (n **-** curr\_position)):

**return**

    # consider the cases when current element

    # is not included in the solution

    TOWUtil(arr, n, curr\_elements, no\_of\_selected\_elements,

            soln, min\_diff, Sum, curr\_sum, curr\_position **+** 1)

    # add the current element to the solution

    no\_of\_selected\_elements **+=** 1

    curr\_sum **=** curr\_sum **+** arr[curr\_position]

    curr\_elements[curr\_position] **=** True

    # checks if a solution is formed

**if** (no\_of\_selected\_elements **==** int(n **/** 2)):

        # checks if the solution formed is better

        # than the best solution so far

**if** (abs(int(Sum **/** 2) **-** curr\_sum) < min\_diff[0]):

            min\_diff[0] **=** abs(int(Sum **/** 2) **-** curr\_sum)

**for** i **in** range(n):

                soln[i] **=** curr\_elements[i]

**else**:

        # consider the cases where current

        # element is included in the solution

        TOWUtil(arr, n, curr\_elements, no\_of\_selected\_elements,

                soln, min\_diff, Sum, curr\_sum, curr\_position **+** 1)

    # removes current element before returning

    # to the caller of this function

    curr\_elements[curr\_position] **=** False

# main function that generate an arr

**def** tugOfWar(arr, n):

    # the boolean array that contains the

    # inclusion and exclusion of an element

    # in current set. The number excluded

    # automatically form the other set

    curr\_elements **=** [None] **\*** n

    # The inclusion/exclusion array

    # for final solution

    soln **=** [None] **\*** n

    min\_diff **=** [999999999999]

    Sum **=** 0

**for** i **in** range(n):

        Sum **+=** arr[i]

        curr\_elements[i] **=** soln[i] **=** False

    # Find the solution using recursive

    # function TOWUtil()

    TOWUtil(arr, n, curr\_elements, 0,

            soln, min\_diff, Sum, 0, 0)

    # Print the solution

    print("The first subset is: ")

**for** i **in** range(n):

**if** (soln[i] **==** True):

            print(arr[i], end **=** " ")

    print()

    print("The second subset is: ")

**for** i **in** range(n):

**if** (soln[i] **==** False):

**print**(arr[i], end **=** " ")

# Driver Code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    arr **=** [23, 45, **-**34, 12, 0, 98,

**-**99, 4, 189, **-**1, 4]

    n **=** len(arr)

    tugOfWar(arr, n)

# This code is contributed by PranchalK

**Output:**

The first subset is: 45 -34 12 98 -1  
The second subset is: 23 0 -99 4 189 4

**Time Complexity:** O(2^n)

**SpaceComplexity:** O(n)

**8 queen problem**

The eight queens problem is the problem of placing eight queens on an 8×8 chessboard such that none of them attack one another (no two are in the same row, column, or diagonal). More generally, the n queens problem places n queens on an n×n chessboard. There are different solutions for the problem. [Backtracking | Set 3 (N Queen Problem)](https://www.geeksforgeeks.org/backtracking-set-3-n-queen-problem/) [Branch and Bound | Set 5 (N Queen Problem)](https://www.geeksforgeeks.org/branch-and-bound-set-4-n-queen-problem/) You can find detailed solutions at <http://en.literateprograms.org/Eight_queens_puzzle_(C)>

**Explanation:**

* This **pseudocode uses a backtracking algorithm**to find a solution to the 8 Queen problem, which consists of placing 8 queens on a chessboard in such a way that no two queens threaten each other.
* The algorithm starts by placing a queen on the first column, then it proceeds to the next column and places a queen in the**first safe** row of that column.
* If the algorithm reaches the 8th column and al**l queens are placed in a safe position**, it prints the board and returns true.  
  If the algorithm is unable to place a queen in a safe position in a certain column, it backtracks to the previous column and tries a different row.
* The “isSafe” function **checks**if it is safe to place a queen on a certain row and column by checking if there are any queens in the same row, diagonal or anti-diagonal.
* It’s worth to notice that this is just a high-level **pseudocode**and it might need to be adapted depending on the specific implementation and language you are using.

**Here is an example of pseudocode for solving the 8 Queen problem using backtracking:**

N **=** 8 # (size of the chessboard)

**def** solveNQueens(board, col):

**if** col **==** N:

**print**(board)

**return** True

**for** i **in** range(N):

**if** isSafe(board, i, col):

            board[i][col] **=** 1

**if** solveNQueens(board, col **+** 1):

**return** True

            board[i][col] **=** 0

**return** False

**def** isSafe(board, row, col):

**for** x **in** range(col):

**if** board[row][x] **==** 1:

**return** False

**for** x, y **in** zip(range(row, **-**1, **-**1), range(col, **-**1, **-**1)):

**if** board[x][y] **==** 1:

**return** False

**for** x, y **in** zip(range(row, N, 1), range(col, **-**1, **-**1)):

**if** board[x][y] **==** 1:

**return** False

**return** True

board **=** [[0 **for** x **in** range(N)] **for** y **in** range(N)]

**if not** solveNQueens(board, 0):

**print**("No solution found")

**Output**

[[1, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 1], [0, 1, 0, 0, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 0, 0, 1, 0, 0], [0, 0, 1, 0, 0, 0, 0, 0]]

**Time Complexity : O((m + q) log^2 n)**

**Space Complexity : O((m + q) log n)**

**Combinational Sum**

Given an array of positive integers **arr[]** and an integer **x**, The task is to find all unique combinations in arr[] where the sum is equal to x.

The same repeated number may be chosen from arr[] an unlimited number of times. Elements in a combination (a1, a2, …, ak) must be printed in non-descending order. (ie, a1 <= a2 <= … <= ak). If there is no combination possible print “Empty”.

**Examples:**

***Input:****arr[] = 2, 4, 6, 8, x = 8*

***Output:***

*[2, 2, 2, 2]*

*[2, 2, 4]*

*[2, 6]*

*[4, 4]*

*[8]*

Recommended Problem

Combination Sum

**Approach:**

*Recursively find all combinations and if the current combination sums up to give****X****then add this combination in the final set of combinations.*

Follow the below steps to implement the idea:

* Sort the array **arr[]**and remove all the duplicates from the **arr[]**then create a temporary vector **r**.to store every combination and a vector of vector **res**.
* Recursively follow:
* If at any time sub-problem **sum == 0** then add that array to the **res** (vector of vectors).
* Run a while loop till the sum – arr[I] is not negative and i is less than arr.size().
* Push arr[i] in **r**and recursively call for **i**and sum-arr[i] ,then increment i by **1**.
* pop r[i] from back and backtrack.

Below is the implementation of the above approach.

# Python3 program to find all combinations that

# sum to a given value

**def** combinationSum(arr, sum):

    ans **=** []

    temp **=** []

    # first do hashing nothing but set{}

    # since set does not always sort

    # removing the duplicates using Set and

    # Sorting the List

    arr **=** sorted(list(set(arr)))

    findNumbers(ans, arr, temp, sum, 0)

**return** ans

**def** findNumbers(ans, arr, temp, sum, index):

**if**(sum **==** 0):

        # Adding deep copy of list to ans

        ans.append(list(temp))

**return**

    # Iterate from index to len(arr) - 1

**for** i **in** range(index, len(arr)):

        # checking that sum does not become negative

**if**(sum **-** arr[i]) >**=** 0:

            # adding element which can contribute to

            # sum

            temp.append(arr[i])

            findNumbers(ans, arr, temp, sum**-**arr[i], i)

            # removing element from list (backtracking)

            temp.remove(arr[i])

# Driver Code

arr **=** [2, 4, 6, 8]

sum **=** 8

ans **=** combinationSum(arr, sum)

# If result is empty, then

**if** len(ans) <**=** 0:

**print**("empty")

# print all combinations stored in ans

**for** i **in** range(len(ans)):

**print**("(", end**=**' ')

**for** j **in** range(len(ans[i])):

**print**(str(ans[i][j])**+**" ", end**=**' ')

    print(")", end**=**' ')

# This Code Is Contributed by Rakesh(vijayarigela09)

**Output**

( 2 2 2 2 ) ( 2 2 4 ) ( 2 6 ) ( 4 4 ) ( 8 )

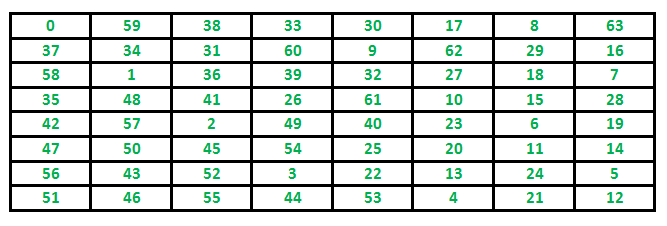
**Time Complexity:** O(k\*(2^n)) where n is the size of array, and k is average length

**Auxiliary Space:**O(k\*x) where is x is number of possible combinations

**Warnsdorff’s algorithm for Knight’s tour problem**

**Problem :** A knight is placed on the first block of an empty board and, moving according to the rules of chess, must visit each square exactly once.

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



We have discussed [Backtracking Algorithm for solution of Knight’s tour](https://www.geeksforgeeks.org/backtracking-set-1-the-knights-tour-problem/). In this post [Warnsdorff’s heuristic](https://en.wikipedia.org/wiki/Knight%27s_tour" \l "Warnsdorf.27s_rule) is discussed.

**Warnsdorff’s Rule:**

1. We can start from any initial position of the knight on the board.
2. We always move to an adjacent, unvisited square with minimal degree (minimum number of unvisited adjacent).

This algorithm may also more generally be applied to any graph.

**Some definitions:**

* A position Q is accessible from a position P if P can move to Q by a single Knight’s move, and Q has not yet been visited.
* The accessibility of a position P is the number of positions accessible from P.

**Algorithm:**

1. Set P to be a random initial position on the board
2. Mark the board at P with the move number “1”
3. Do following for each move number from 2 to the number of squares on the board:

* let S be the set of positions accessible from P.
* Set P to be the position in S with minimum accessibility
* Mark the board at P with the current move number

1. Return the marked board — each square will be marked with the move number on which it is visited.

Below is implementation of above algorithm.

# Python program to for Knight's tour problem using

# Warnsdorff's algorithm

**import** random

**class** Cell:

**def** \_\_init\_\_(self, x, y):

        self.x **=** x

        self.y **=** y

N **=** 8

# Move pattern on basis of the change of

# x coordinates and y coordinates respectively

cx **=** [1, 1, 2, 2, **-**1, **-**1, **-**2, **-**2]

cy **=** [2, **-**2, 1, **-**1, 2, **-**2, 1, **-**1]

# function restricts the knight to remain within

# the 8x8 chessboard

**def** limits(x, y):

**return** ((x >**=** 0 **and** y >**=** 0) **and** (x < N **and** y < N))

# Checks whether a square is valid and empty or not

**def** isempty(a, x, y):

**return** (limits(x, y)) **and** (a[y **\*** N **+** x] < 0)

# Returns the number of empty squares adjacent to (x, y)

**def** getDegree(a, x, y):

    count **=** 0

**for** i **in** range(N):

**if** isempty(a, (x **+** cx[i]), (y **+** cy[i])):

            count **+=** 1

**return** count

# Picks next point using Warnsdorff's heuristic.

# Returns false if it is not possible to pick

# next point.

**def** nextMove(a, cell):

    min\_deg\_idx **= -**1

    c **=** 0

    min\_deg **=** (N **+** 1)

    nx **=** 0

    ny **=** 0

    # Try all N adjacent of (\*x, \*y) starting

    # from a random adjacent. Find the adjacent

    # with minimum degree.

    start **=** random.randint(0, 1000) **%** N

**for** count **in** range(0, N):

        i **=** (start **+** count) **%** N

        nx **=** cell.x **+** cx[i]

        ny **=** cell.y **+** cy[i]

        c **=** getDegree(a, nx, ny)

**if** ((isempty(a, nx, ny)) **and** c < min\_deg):

            min\_deg\_idx **=** i

            min\_deg **=** c

    # IF we could not find a next cell

**if** (min\_deg\_idx **== -**1):

**return** None

    # Store coordinates of next point

    nx **=** cell.x **+** cx[min\_deg\_idx]

    ny **=** cell.y **+** cy[min\_deg\_idx]

    # Mark next move

    a[ny **\*** N **+** nx] **=** a[(cell.y) **\*** N **+** (cell.x)] **+** 1

    # Update next point

    cell.x **=** nx

    cell.y **=** ny

**return** cell

# displays the chessboard with all the legal knight's moves

**def** printA(a):

**for** i **in** range(N):

**for** j **in** range(N):

**print**("%d\t" **%** a[j **\*** N **+** i], end**=**"")

**print**()

# checks its neighbouring squares

# If the knight ends on a square that is one knight's move from the beginning square,then tour is closed

**def** neighbour(x, y, xx, yy):

**for** i **in** range(N):

**if** ((x **+** cx[i]) **==** xx) **and** ((y **+** cy[i]) **==** yy):

**return** True

**return** False

#  Generates the legal moves using warnsdorff's heuristics. Returns false if not possible

**def** findClosedTour():

    # Filling up the chessboard matrix with -1's

    a **=** [**-**1] **\*** N **\*** N

    # initial position

    sx **=** 3

    sy **=** 2

    # Current points are same as initial points

    cell **=** Cell(sx, sy)

    a[cell.y **\*** N **+** cell.x] **=** 1  # Mark first move.

    # Keep picking next points using Warnsdorff's heuristic

    ret **=** None

**for** i **in** range(N **\*** N **-** 1):

        ret **=** nextMove(a, cell)

**if** ret **==** None:

**return** False

    # Check if tour is closed (Can end at starting point)

**if not** neighbour(ret.x, ret.y, sx, sy):

**return** False

    printA(a)

**return** True

# Driver Code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    # While we don't get a solution

**while not** findClosedTour():

**pass**

# This code is contributed by Tapesh(tapeshdua420)

**Output:**

59 14 63 32 1 16 19 34   
62 31 60 15 56 33 2 17   
13 58 55 64 49 18 35 20   
30 61 42 57 54 51 40 3   
43 12 53 50 41 48 21 36   
26 29 44 47 52 39 4 7   
11 46 27 24 9 6 37 22   
28 25 10 45 38 23 8 5

***The Hamiltonian path problem is NP-hard in general. In practice, Warnsdorff’s heuristic successfully finds a solution in linear time.***

**Do you know?**

“On an 8 × 8 board, there are exactly 26,534,728,821,064 directed closed tours (i.e. two tours along the same path that travel in opposite directions are counted separately, as are rotations and reflections). The number of undirected closed tours is half this number, since every tour can be traced in reverse!”

**Find paths from corner cell to middle cell in maze**

Given a square maze containing positive numbers, find all paths from a corner cell (any of the extreme four corners) to the middle cell. We can move exactly n steps from a cell in 4 directions i.e. North, East, West and South where **n is value of the cell**,

We can move to mat[i+n][j], mat[i-n][j], mat[i][j+n], and mat[i][j-n] from a cell mat[i][j] where n is value of mat[i][j].

Example

Input: 9 x 9 maze  
[ **3**, 5, 4, **4**, 7, 3, 4, **6**, 3 ]  
[ 6, 7, **5**, 6, 6, 2, 6, **6**, 2 ]  
[ 3, **3**, 4, 3, **2**, 5, 4, 7, 2 ]  
[ 6, 5, 5, **1**, **2**, 3, 6, 5, 6 ]  
[ 3, 3, 4, 3, 0, 1, 4, 3, 4 ]  
[ 3, 5, **4**, 3, **2**, 2, 3, 3, 5 ]  
[ 3, 5, 4, **3**, 2, 6, 4, **4**, 3 ]  
[ 3, **5**, 1, 3, 7, 5, 3, **6**, 4 ]  
[ 6, 2, 4, 3, 4, 5, 4, 5, 1 ]

Output:  
(0, 0) -> (0, 3) -> (0, 7) ->   
(6, 7) -> (6, 3) -> (3, 3) ->   
(3, 4) -> (5, 4) -> (5, 2) ->   
(1, 2) -> (1, 7) -> (7, 7) ->  
(7, 1) -> (2, 1) -> (2, 4) ->   
(4, 4) -> MID

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

The idea is to use backtracking. We start with each corner cell of the maze and recursively checks if it leads to the solution or not. Following is the Backtracking algorithm –

If destination is reached

1. print the path

Else

1. Mark current cell as visited and add it to path array.
2. Move forward in all 4 allowed directions and recursively check if any of them leads to a solution.
3. If none of the above solutions work then mark this cell as not visited and remove it from path array.

Below is the implementation of the above approach:

# Python program to find a path from corner cell to

# middle cell in maze containing positive numbers

# Rows and columns in given maze

N **=** 9

# check whether given cell is a valid cell or not.

**def** isValid(visited, pt):

    # check if cell is not visited yet to

    # avoid cycles (infinite loop) and its

    # row and column number is in range

**return** (pt[0] >**=** 0) **and** (pt[0] < N) **and** (pt[1] >**=** 0) **and** (pt[1] < N) **and** (pt **not in** visited)

# Function to print path from source to middle coordinate

**def** printPath(path):

**for** i **in** path:

**print**("({}, {}) -> ".format(i[0], i[1]), end**=**"")

    print("MID")

    print()

# For searching in all 4 direction

row **=** [**-**1, 1, 0, 0]

col **=** [0, 0, **-**1, 1]

# Coordinates of 4 corners of matrix

\_row **=** [0, 0, N**-**1, N**-**1]

\_col **=** [0, N**-**1, 0, N**-**1]

**def** findPathInMazeUtil(maze, path, visited, curr):

    # If we have reached the destination cell.

    # print the complete path

**if** curr[0] **==** N **//** 2 **and** curr[1] **==** N **//** 2:

        printPath(path)

**return**

    # consider each direction

**for** i **in** range(4):

        # get value of current cell

        n **=** maze[curr[0]][curr[1]]

        # We can move N cells in either of 4 directions

        x **=** curr[0] **+** row[i]**\***n

        y **=** curr[1] **+** col[i]**\***n

        next **=** (x, y)

        # if valid pair

**if** isValid(visited, next):

            # mark cell as visited

            visited.append(next)

            # add cell to current path

            path.append(next)

            # recurse for next cell

            findPathInMazeUtil(maze, path, visited, next)

            # backtrack

            # remove cell from current path

            path.pop()

            visited.remove(next)

# Function to find a path from corner cell to

# middle cell in maze containing positive numbers

**def** findPathInMaze(maze):

    # list to store complete path

    # from source to destination

    path **=** []

    # to store cells already visited in current path

    visited **=** []

    # Consider each corners as the starting

    # point and search in maze

**for** i **in** range(4):

        x **=** \_row[i]

        y **=** \_col[i]

        pt **=** (x, y)

        # mark cell as visited

        visited.append(pt)

        # add cell to current path

        path.append(pt)

        findPathInMazeUtil(maze, path, visited, pt)

        # backtrack

        # remove cell from current path

        path.pop()

        visited.remove(pt)

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    maze **=** [

        [3, 5, 4, 4, 7, 3, 4, 6, 3],

        [6, 7, 5, 6, 6, 2, 6, 6, 2],

        [3, 3, 4, 3, 2, 5, 4, 7, 2],

        [6, 5, 5, 1, 2, 3, 6, 5, 6],

        [3, 3, 4, 3, 0, 1, 4, 3, 4],

        [3, 5, 4, 3, 2, 2, 3, 3, 5],

        [3, 5, 4, 3, 2, 6, 4, 4, 3],

        [3, 5, 1, 3, 7, 5, 3, 6, 4],

        [6, 2, 4, 3, 4, 5, 4, 5, 1]

    ]

    findPathInMaze(maze)

# This code is contributed by Vikram\_Shirsat

**Output**

(0, 0) -> (0, 3) -> (0, 7) -> (6, 7) -> (6, 3) -> (3, 3) -> (3, 4) -> (5, 4) -> (5, 2) -> (1, 2) -> (1, 7) -> (7, 7) -> (7, 1) -> (2, 1) -> (2, 4) -> (4, 4) -> MID

**Output :**

(0, 0) -> (0, 3) -> (0, 7) ->   
(6, 7) -> (6, 3) -> (3, 3) ->   
(3, 4) -> (5, 4) -> (5, 2) ->   
(1, 2) -> (1, 7) -> (7, 7) ->  
(7, 1) -> (2, 1) -> (2, 4) ->   
(4, 4) -> MID

**A better approach:**

# Python program to find a path from corner cell to

# middle cell in maze containing positive numbers

**def** printPath(maze, i, j, ans):

    # If we reach the center cell

**if** (i **==** len(maze) **//** 2 **and** j **==** len(maze) **//** 2):

        # Make the final answer, Print

        # final answer and Return

        ans **+=** "(" **+** str(i) **+** ", " **+** str(j) **+** ") -> MID";

        print(ans);

**return**;

    # If the element at the current position

    # in maze is 0, simply Return as it has

    # been visited before.

**if** (maze[i][j] **==** 0):

**return**;

    # If element is non-zero, then note

    # the element in variable 'k'

    k **=** maze[i][j];

    # Mark the cell visited by making the

    # element 0. Don't worry, the element

    # is safe in 'k'

    maze[i][j] **=** 0;

    # Make recursive calls in all 4

    # directions pro-actively i.e. if the next

    # cell lies in maze or not. Right call

**if** (j **+** k < len(maze)):

        printPath(maze, i, j **+** k, ans **+** "(" **+** str(i) **+** ", " **+** str(j) **+** ") -> ");

    # down call

**if** (i **+** k < len(maze)):

        printPath(maze, i **+** k, j, ans **+** "(" **+** str(i) **+** ", " **+** str(j) **+** ") -> ");

    # left call

**if** (j **-** k > 0):

        printPath(maze, i, j **-** k, ans **+** "(" **+** str(i) **+** ", " **+** str(j) **+** ") -> ");

    # up call

**if** (i **-** k > 0):

        printPath(maze, i **-** k, j, ans **+** "(" **+** str(i) **+** ", " **+** str(j) **+** ") -> ");

    # Unmark the visited cell by substituting

    # its original value from 'k'

    maze[i][j] **=** k;

    # Driver code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    # Creating the maze

    maze **=** [[ 3, 5, 4, 4, 7, 3, 4, 6, 3 ],[ 6, 7, 5, 6, 6, 2, 6, 6, 2 ],[ 3, 3, 4, 3, 2, 5, 4, 7, 2 ],

            [ 6, 5, 5, 1, 2, 3, 6, 5, 6 ],[ 3, 3, 4, 3, 0, 1, 4, 3, 4 ],[ 3, 5, 4, 3, 2, 2, 3, 3, 5 ],

            [ 3, 5, 4, 3, 2, 6, 4, 4, 3 ],[ 3, 5, 1, 3, 7, 5, 3, 6, 4 ],[ 6, 2, 4, 3, 4, 5, 4, 5, 1 ]] ;

    # Calling the printPath function

    printPath(maze, 0, 0, "");

# This code contributed by gauravrajput1

**Output**:

(0, 0) -> (0, 3) -> (0, 7) ->   
(6, 7) -> (6, 3) -> (3, 3) ->  
(3, 4) -> (5, 4) -> (5, 2) ->  
(1, 2) -> (1, 7) -> (7, 7) ->   
(7, 1) -> (2, 1) -> (2, 4) ->   
(4, 4) -> MID

**Find Maximum number possible by doing at-most K swaps**

Given two positive integers **M** and **K**, find the maximum integer possible by doing at-most **K** swap operations on its digits.

**Examples:**

***Input:****M = 254, K = 1*

***Output:****524*

***Explanation:****Swap 5 with 2 so number becomes 524*

***Input:****M = 254, K = 2*

***Output:****542*

***Explanation:****Swap 5 with 2 so number becomes 524, Swap 4 with 2 so number becomes 542*

***Input****: M = 68543, K = 1*

***Output:****86543*

***Explanation:****Swap 8 with 6 so number becomes 86543*

***Input:****M = 7599, K = 2*

***Output:****9975*

***Explanation:****Swap 9 with 5 so number becomes 7995, Swap 9 with 7 so number becomes 9975*

***Input:****M = 76543, K = 1*

***Output:****76543*

***Explanation:****No swap is required.*

***Input:****M = 129814999, K = 4*

***Output:****999984211*

***Explanation:****Swap 9 with 1 so number becomes 929814991, Swap 9 with 2 so number becomes 999814291, Swap 9 with 8 so number becomes 999914281, Swap 1 with 8 so number becomes 999984211*

Recommended Practice

[Largest number in K swaps](https://practice.geeksforgeeks.org/problems/largest-number-in-k-swaps-1587115620/1/)

[Try It!](https://practice.geeksforgeeks.org/problems/largest-number-in-k-swaps-1587115620/1/)

**Naive solution for the Largest number in K swaps:**

*The idea is to consider every digit and swap it with digits following it one at a time and see if it leads to the maximum number. The process is repeated K times. The code can be further optimized, if the current digit is swapped with a digit less than the following digit.*

Follow the below steps to Implement the idea:

1. Create a global variable that will store the maximum string or number.
2. Define a recursive function that takes the string as a number and value of k
3. Run a nested loop, the outer loop from 0 to the length of string -1, and the inner loop from i+1 to the end of the string.
4. Swap the ith and jth characters and check if the string is now maximum and update the maximum string.
5. Call the function recursively with parameters: string and k-1.
6. Now again swap back the ith and jth character.

Below is the Implementation of the above approach:

# Python3 program to find maximum

# integer possible by doing at-most

# K swap operations on its digits.

# utility function to swap two

# characters of a string

**def** swap(string, i, j):

**return** (string[:i] **+** string[j] **+**

            string[i **+** 1:j] **+**

            string[i] **+** string[j **+** 1:])

# function to find maximum integer

# possible by doing at-most K swap

# operations on its digits

**def** findMaximumNum(string, k, maxm):

    # return if no swaps left

**if** k **==** 0:

**return**

    n **=** len(string)

    # consider every digit

**for** i **in** range(n **-** 1):

        # and compare it with all digits after it

**for** j **in** range(i **+** 1, n):

            # if digit at position i is less than

            # digit at position j, swap it and

            # check for maximum number so far and

            # recurse for remaining swaps

**if** string[i] < string[j]:

                # swap string[i] with string[j]

                string **=** swap(string, i, j)

                # If current num is more than

                # maximum so far

**if** string > maxm[0]:

                    maxm[0] **=** string

                # recurse of the other k - 1 swaps

                findMaximumNum(string, k **-** 1, maxm)

                # backtrack

                string **=** swap(string, i, j)

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    string **=** "129814999"

    k **=** 4

    maxm **=** [string]

    findMaximumNum(string, k, maxm)

    print(maxm[0])

# This code is contributed

# by vibhu4agarwal

**Output**

999984211

**Time Complexity:** O((N2)k). For every digit, **N2** recursive calls are generated until the value of k is 0 Thus O((N2)k).

**Auxiliary Space:**O(N). This is the space required to store the output string.

**Find the Maximum number possible by doing at-most K swaps by swapping with the maximum element on the right:**

*It can be observed that to make the maximum string, the maximum digit is shifted to the front. So, instead of trying all pairs, try only those pairs where one of the elements is the maximum digit that is not yet swapped to the front.*

Follow the below steps to Implement the idea::

1. Create a global variable that will store the maximum string or number.
2. Define a recursive function that takes the string as a number, the value of k, and the current index.
3. Find the index of the maximum element in the range current index to end.
4. if the index of the maximum element is not equal to the current index then decrement the value of k.
5. Run a loop from the current index to the end of the array
6. If the ith digit is equal to the maximum element
7. Swap the ith and element at the current index and check if the string is now maximum and update the maximum string.
8. Call the function recursively with parameters: string and k.
9. Now again swap back the ith and element at the current index.

Below is the Implementation of the above approach:

# Python3 program to find maximum

# integer possible by doing at-most

# K swap operations on its digits.

# function to find maximum integer

# possible by doing at-most K swap

# operations on its digits

**def** findMaximumNum(string, k, maxm, ctr):

    # return if no swaps left

**if** k **==** 0:

**return**

    n **=** len(string)

    # Consider every digit after

    # the cur position

    mx **=** string[ctr]

**for** i **in** range(ctr**+**1,n):

        # Find maximum digit greater

        # than at ctr among rest

**if** int(string[i]) > int(mx):

            mx**=**string[i]

    # If maxm is not equal to str[ctr],

    # decrement k

**if**(mx!**=**string[ctr]):

        k**=**k**-**1

    # search this maximum among the rest from behind

    # first swap the last maximum digit if it occurs more than 1 time

    # example str= 1293498 and k=1 then max string is 9293418 instead of 9213498

**for** i **in** range(ctr,n):

        # If digit equals maxm swap

        # the digit with current

        # digit and recurse for the rest

**if**(string[i]**==**mx):

            # swap str[ctr] with str[j]

            string[ctr], string[i] **=** string[i], string[ctr]

            new\_str **=** "".join(string)

            # If current num is more than

            # maximum so far

**if** int(new\_str) > int(maxm[0]):

                  maxm[0] **=** new\_str

            # recurse of the other k - 1 swaps

            findMaximumNum(string, k , maxm, ctr**+**1)

            # backtrack

            string[ctr], string[i] **=** string[i], string[ctr]

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    string **=** "129814999"

    k **=** 4

    maxm **=** [string]

    string **=** [char **for** char **in** string]

    findMaximumNum(string, k, maxm, 0)

**print**(maxm[0])

# This code is contributed Aarti\_Rathi

**Output**

999984211

**Time Complexity:** O(Nk), For every recursive call N recursive calls are generated until the value of k is 0, Thus O((Nk).

**Auxiliary Space:** O(N). The space required to store the output string.

**Exercise:**

* [Find the minimum integer possible by doing at least K swap operations on its digits.](https://www.geeksforgeeks.org/find-maximum-number-possible-by-doing-at-most-k-swaps/)
* [Find the maximum/minimum integer possible by doing exactly K swap operations on its digits](https://www.geeksforgeeks.org/find-maximum-number-possible-by-doing-at-most-k-swaps/).

**Rat in a Maze with multiple steps or jump allowed**

This is the variation of [Rat in Maze](https://write.geeksforgeeks.org/javapythoncc-code-for-article-backtracking-set-2-rat-in-a-maze/)

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 destination. The rat can move only in two directions: forward and down.

In the maze matrix, 0 means the block is dead end and non-zero number means the block can be used in the path from source to destination. The non-zero value of mat[i][j] indicates number of maximum jumps rat can make from cell mat[i][j].

In this variation, Rat is allowed to jump multiple steps at a time instead of 1.

**Examples:**

**Input :** { {2, 1, 0, 0},  
 {3, 0, 0, 1},  
 {0, 1, 0, 1},  
 {0, 0, 0, 1}  
 }  
**Output :** { {1, 0, 0, 0},  
 {1, 0, 0, 1},  
 {0, 0, 0, 1},  
 {0, 0, 0, 1}  
 }

**Explanation**   
Rat started with M[0][0] and can jump upto 2 steps right/down.   
Let's try in horizontal direction -   
M[0][1] won't lead to solution and M[0][2] is 0 which is dead end.   
So, backtrack and try in down direction.   
Rat jump down to M[1][0] which eventually leads to solution.

**Input :** {   
 {2, 1, 0, 0},  
 {2, 0, 0, 1},  
 {0, 1, 0, 1},  
 {0, 0, 0, 1}  
 }  
**Output :** Solution doesn't exist

**Naive Algorithm:**

The Naive Algorithm is to generate all paths from source to destination and one by one check if the generated path satisfies the constraints.

while there are untried paths  
{  
 generate the next path  
 if this path has all blocks as non-zero  
 {  
 print this path;  
 }  
}

**Backtracking Algorithm:**

If destination is reached  
 print the solution matrix  
Else  
 a) Mark current cell in solution matrix as 1.   
 b) Move forward/jump (for each valid steps) in horizontal direction   
 and recursively check if this move leads to a solution.   
 c) If the move chosen in the above step doesn't lead to a solution  
 then move down and check if this move leads to a solution.   
 d) If none of the above solutions work then unmark this cell as 0   
 (BACKTRACK) and return false.

**Implementation of Backtracking solution**

""" Python3 program to solve Rat in a

Maze problem using backtracking """

# Maze size

N **=** 4

""" A utility function to print solution matrix

sol """

**def** printSolution(sol):

**for** i **in** range(N):

**for** j **in** range(N):

            print(sol[i][j], end **=** " ")

        print()

""" A utility function to check if

x, y is valid index for N\*N maze """

**def** isSafe(maze, x, y):

    # if (x, y outside maze) return false

**if** (x >**=** 0 **and** x < N **and** y >**=** 0 **and**

         y < N **and** maze[x][y] !**=** 0):

**return** True

**return** False

""" This function solves the Maze problem using

Backtracking. It mainly uses solveMazeUtil() to

solve the problem. It returns false if no path

is possible, otherwise return True and prints

the path in the form of 1s. Please note that

there may be more than one solutions,

this function prints one of the feasible solutions."""

**def** solveMaze(maze):

    sol **=** [[0, 0, 0, 0],

           [0, 0, 0, 0],

           [0, 0, 0, 0],

           [0, 0, 0, 0]]

**if** (solveMazeUtil(maze, 0, 0, sol) **==** False):

        print("Solution doesn't exist")

**return** False

    printSolution(sol)

**return** True

""" A recursive utility function

to solve Maze problem """

**def** solveMazeUtil(maze, x, y, sol):

    # if (x, y is goal) return True

**if** (x **==** N **-** 1 **and** y **==** N **-** 1) :

        sol[x][y] **=** 1

**return** True

    # Check if maze[x][y] is valid

**if** (isSafe(maze, x, y) **==** True):

        # mark x, y as part of solution path

        sol[x][y] **=** 1

        """ Move forward in x direction """

**for** i **in** range(1, N):

**if** (i <**=** maze[x][y]):

                """ Move forward in x direction """

**if** (solveMazeUtil(maze, x **+** i,

                                  y, sol) **==** True):

**return** True

                """ If moving in x direction doesn't give

                solution then Move down in y direction """

**if** (solveMazeUtil(maze, x,

                                  y **+** i, sol) **==** True):

**return** True

        """ If none of the above movements work then

        BACKTRACK: unmark x, y as part of solution

        path """

        sol[x][y] **=** 0

**return** False

**return** False

# Driver Code

maze **=** [[2, 1, 0, 0],

        [3, 0, 0, 1],

        [0, 1, 0, 1],

        [0, 0, 0, 1]]

solveMaze(maze)

# This code is contributed by SHUBHAMSINGH10

**Output:**

1 0 0 0   
1 0 0 1   
0 0 0 1   
0 0 0 1

**N Queen in O(n) space**

Given n, of a n x n chessboard, find the proper placement of queens on chessboard.

**Previous Approach :** [N Queen](https://www.geeksforgeeks.org/branch-and-bound-set-4-n-queen-problem/)

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

**Algorithm :**

Place(k, i)  
// Returns true if a queen can be placed  
// in kth row and ith column. Otherwise it  
// returns false. X[] is a global array  
// whose first (k-1) values have been set.  
// Abs( ) returns absolute value of r  
{  
 for j := 1 to k-1 do

// Two in the same column  
 // or in the same diagonal  
 if ((x[j] == i) or  
 (abs(x[j] – i) = Abs(j – k)))  
 then return false;

return true;  
}

**Algorithm nQueens(k, n) :**

// Using backtracking, this procedure prints all   
// possible placements of n queens on an n×n   
// chessboard so that they are nonattacking.  
{  
 for i:= 1 to n do  
 {  
 if Place(k, i) then  
 {  
 x[k] = i;  
 if (k == n)  
 write (x[1:n]);  
 else   
 NQueens(k+1, n);  
 }  
 }  
}

**Implementation:**

# Python code to for n Queen placement

**class** GfG:

**def** \_\_init\_\_(self):

        self.MAX **=** 10

        self.arr **=** [0] **\*** self.MAX

        self.no **=** 0

**def** breakLine(self):

        print("\n------------------------------------------------")

**def** canPlace(self, k, i):

        # Helper Function to check

        # if queen can be placed

**for** j **in** range(1, k):

**if** (self.arr[j] **==** i **or**

               (abs(self.arr[j] **-** i) **==** abs(j **-** k))):

**return** False

**return** True

**def** display(self, n):

        # Function to display placed queen

        self.breakLine()

        self.no **+=** 1

**print**("Arrangement No.", self.no, end **=** " ")

        self.breakLine()

**for** i **in** range(1, n **+** 1):

**for** j **in** range(1, n **+** 1):

**if** self.arr[i] !**=** j:

                    print("\t\_", end **=** " ")

**else**:

**print**("\tQ", end **=** " ")

**print**()

        self.breakLine()

**def** nQueens(self, k, n):

        # Function to check queens placement

**for** i **in** range(1, n **+** 1):

**if** self.canPlace(k, i):

                self.arr[k] **=** i

**if** k **==** n:

                    self.display(n)

**else**:

                    self.nQueens(k **+** 1, n)

# Driver Code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    n **=** 4

    obj **=** GfG()

    obj.nQueens(1, n)

# This code is contributed by vibhu4agarwal

**Output:**

---------------------------------  
Arrangement No. 1  
---------------------------------  
 \_ Q \_ \_  
 \_ \_ \_ Q  
 Q \_ \_ \_  
 \_ \_ Q \_

---------------------------------

---------------------------------  
Arrangement No. 2  
---------------------------------  
 \_ \_ Q \_  
 Q \_ \_ \_  
 \_ \_ \_ Q  
 \_ Q \_ \_

---------------------------------

**Medium Questions:**

**Hard Questions:**