Design and analysis of Algorithm

-Vishal Nagargoje (181210059)
-Vikas Paliwal (181210058)

TABLE OF CONTENT

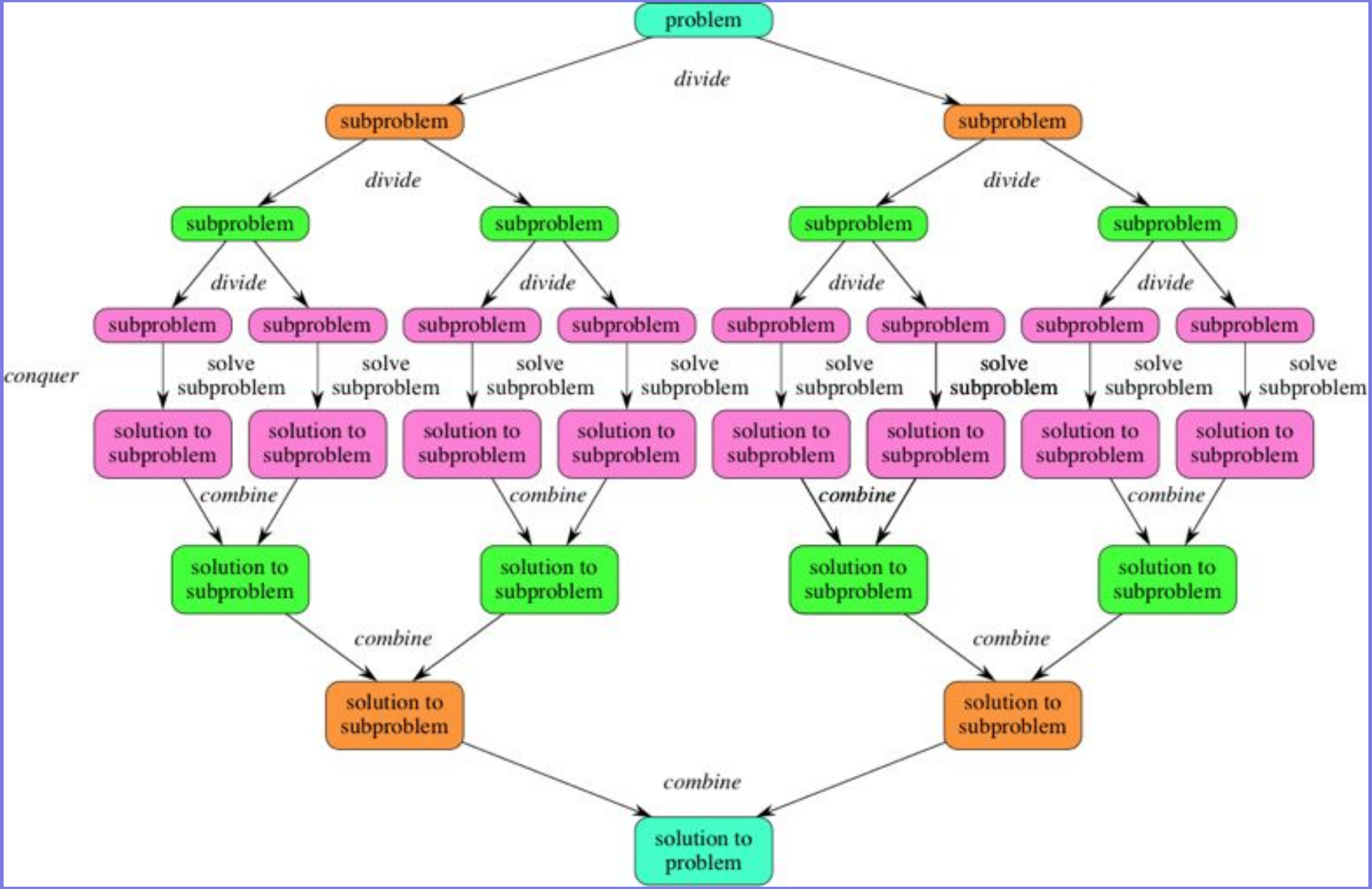
DIVIDE AND CONQUER ALGORITHM

In computer science, **divide and conquer** is an algorithm design paradigm based on multi-branched recursion. A divide-and-conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

Use of Divide and conquer

This divide-and-conquer technique is the basis of efficient algorithms for all kinds of problems, such as:

- sorting (e.g., quicksort, merge sort)
- multiplying large numbers (e.g. the Karatsuba algorithm)
- finding the closest pair of points
- syntactic analysis (e.g., top-down parsers)
- computing the discrete Fourier transform (FFT)


problem
divide
subproblem
subproblem
divide
divide
subproblem
subproblem
subproblem
subproblem
divide
divide
divide
divide
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
conquer
solve subproblem
solution to subproblem
combine
solution to subproblem
combine
solution to subproblem
combine
solution to problem

Algorithm

```
DAC(a, i, j)
{
    if(small(a, i, j))
      return(Solution(a, i, j))
    else
      m = divide(a, i, j)
      b = DAC(a, i, mid)
      c = DAC(a, mid+1, j)
      d = combine(b, c)
   return(d)
 }
```

Real life Example

Finding Cube Root of a Number

C++ PROGRAM TO FIND CUBIC ROOT OF A NUMBER

```
// using Binary Search
#include <bits/stdc++.h>
using namespace std;
// Returns the absolute value of n-mid*mid*mid
double diff(double n,double mid)
{
        if (n > (mid*mid*mid))
        return (n-(mid*mid*mid));
        else
        return ((mid*mid*mid) - n);
}

// Returns cube root of a no n
double cubicRoot(double n)
{
        // Set start and end for binary search
        double start = 0, end = n;

   // Set precision
        double e = 0.0000001;
while (true)
        {
        double mid = (start + end)/2;
        double error = diff(n, mid);

        // If error is less than e then mid is
        // our answer so return mid
        if (error <= e)
        return mid;

        // If mid*mid*mid is greater than n set
        // end = mid
        if ((mid*mid*mid) > n)
        end = mid;

        // If mid*mid*mid is less than n set
        // start = mid
        else
        start = mid;
        }
}

// Driver code
int main()
{
        double n = 3;
        printf("Cubic root of %lf is %lf\n",
        n, cubicRoot(n));
        return 0;
    }
```

BACKTRACKING ALGORTIHM

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

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).

Use of Backtracking algorithm

- Backtracking is an important tool for solving constraint satisfaction problems, such as crosswords, verbal arithmetic, Sudoku, and many other puzzles.
- It is often the most convenient (if not the most efficient) technique for parsing, for the knapsack problem and other combinatorial optimization problems
- It is also the basis of the so-called logic programming languages such as Icon, Planner and Prolog.

Given an instance of any computational problem and data corresponding to the instance, all the constraints that need to be satisfied in order to solve the problem are represented by .

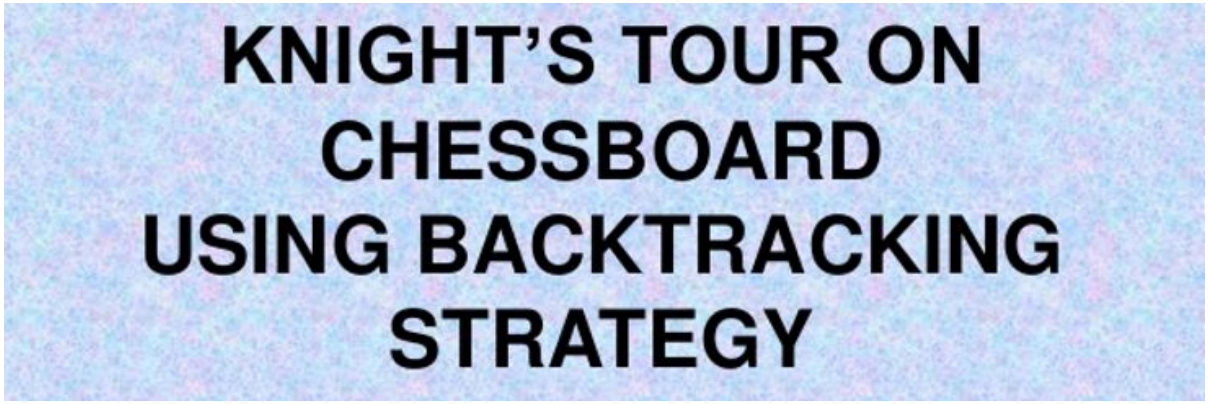
A backtracking algorithm will then work as follows:

The Algorithm begins to build up a solution, starting with an empty solution set . **S = {}**

1. Add to the first move that is still left (All possible moves are added to one by one). This now creates a new sub-tree in the search tree of the algorithm.
2. Check if satisfies each of the constraints in .
    - ○ If Yes, then the sub-tree is "eligible" to add more "children".
    - ○ Else, the entire subtree is useless, so recurs back to step 1 using argument .
3. In the event of "eligibility" of the newly formed sub-tree , recurs back to step 1, using argument .
4. If the check for returns that it is a solution for the entire data . Output and terminate the program.

    If not, then return that no solution is possible with the current and hence discard it.

Real life example

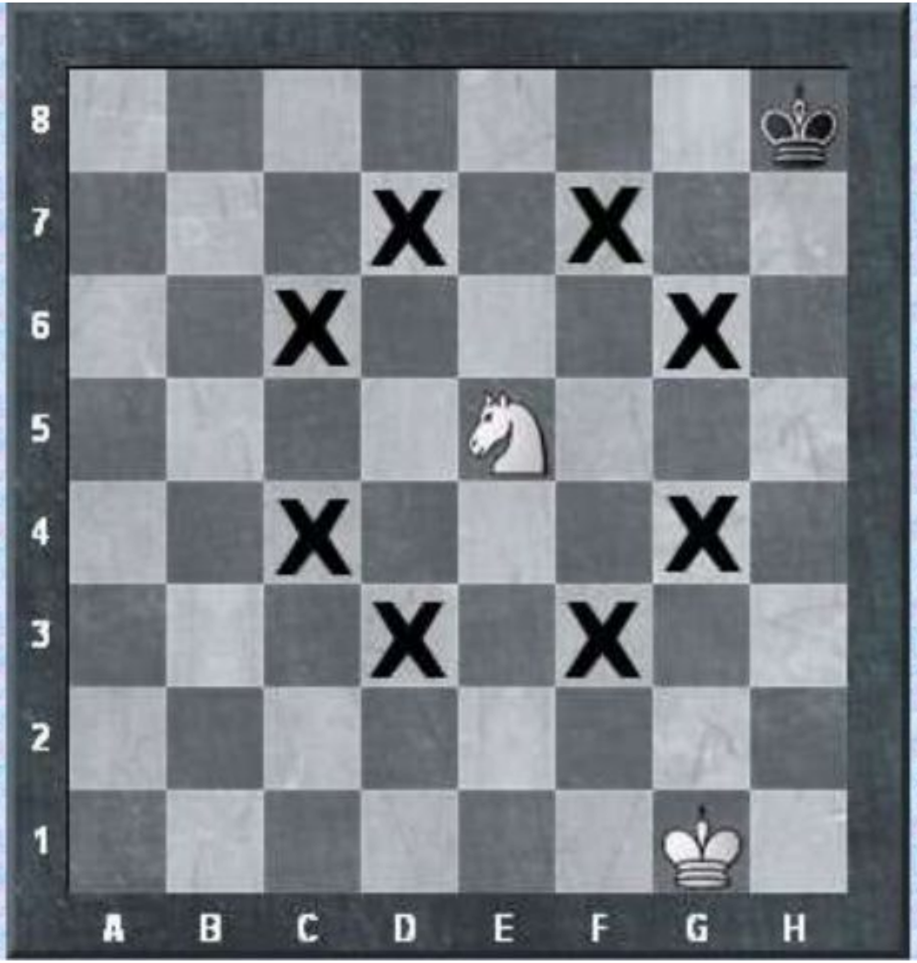

KNIGHT'S TOUR ON CHESSBOARD
USING BACKTRACKING STRATEGY

What is Knights Tour?

- The problem is that if” *The knight is placed on any block of an empty chess board and, moving according to the rules of chess, must visit each square exactly once.”*
- *This Problem is an application of Hamiltonian Path or cycle*

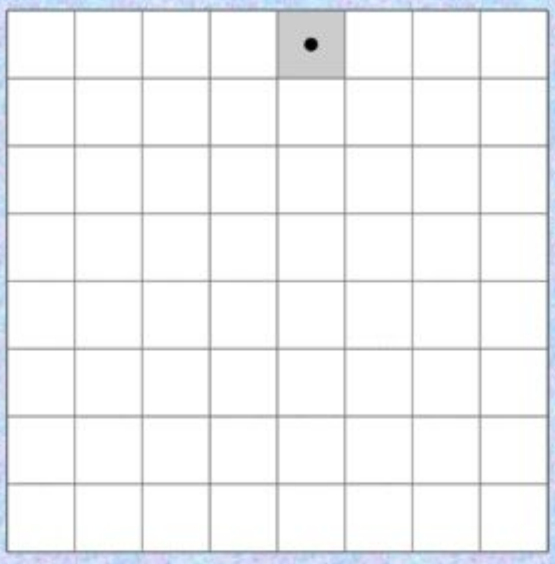
POSSIBLE MOVES

FOR KNIGHT



- The number of moves possible depends on the position of the knight on the board are :

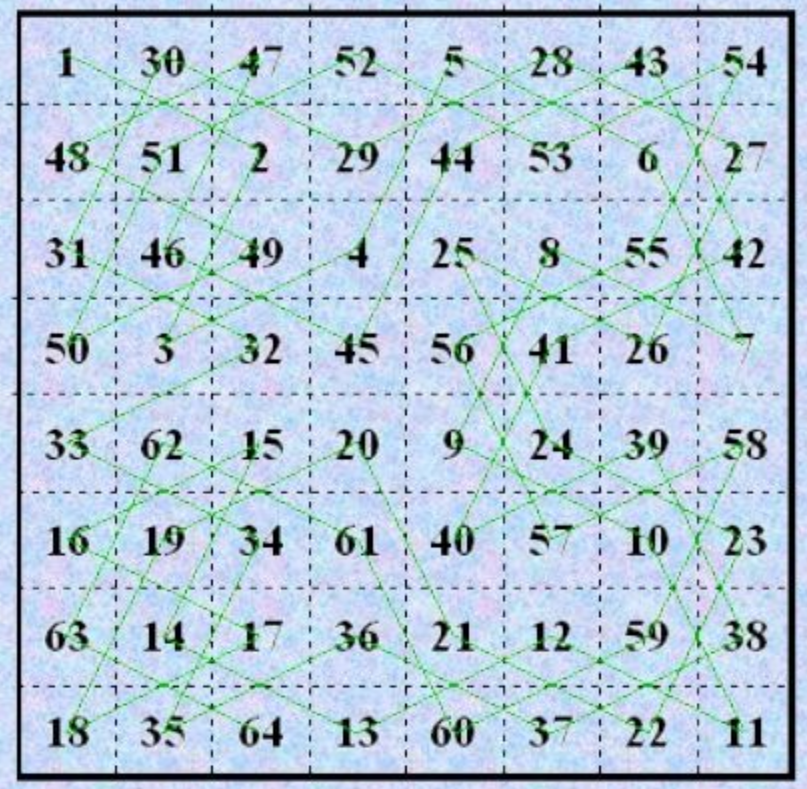
In the corners there are only two legal moves, on the squares adjacent to the corners there are three and in the middle of the board there are eight.



Knights tour

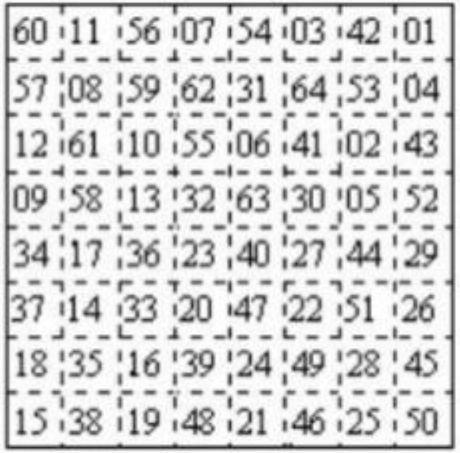
This is the pictorial representation of the knights tour on the chessboard

Open Tour

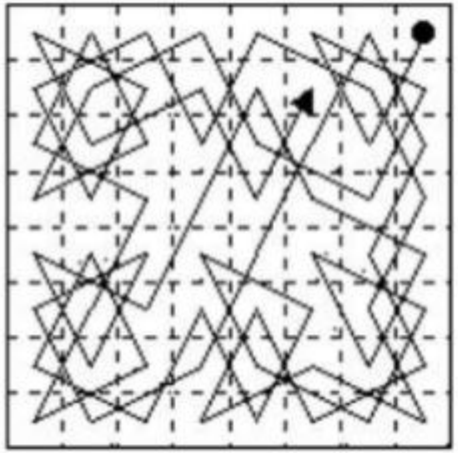


1	30	47	52	5	28	43	54
48	51	2	29	44	53	6	27
31	46	49	4	25	8	55	42
50	3	32	45	56	41	26	7
33	62	15	20	9	24	39	58
16	19	34	61	40	57	10	23
63	14	17	36	21	12	59	38
18	35	64	13	60	37	22	11

Closed tour



60	11	56	07	54	03	42	01
57	08	59	62	31	64	53	04
12	61	10	55	06	41	02	43
09	58	13	32	63	30	05	52
34	17	36	23	40	27	44	29
37	14	33	20	47	22	51	26
18	35	16	39	24	49	28	45
15	38	19	48	21	46	25	50



26,534,728,821,064

Are the total no of possibilities for closed tour on 8*8 Chessboard.

Algorithm

Naïve approach to Knight's tour problem

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;
    }
}
```

```
SOLVEKT()
   initialise  sol[N][N]
   for  x = 0 to N-1
      for y = 0 to N-1
         sol[x][y] = -1

   initialise  xMove[8] = {  2, 1, -1, -2, -2, -1,  1,  2 }
   initialise  yMove[8] = {  1, 2,  2,  1, -1, -2, -2, -1 }

   sol[0][0]  = 0

   if   SOLVEKTUTIL(0, 0, 1, sol, xMove, yMove) =false
      print  Solution does not exist
      return false
   else
      printSolution(sol)

   return true
```

```
SOLVEKTUTIL(x, y, movei, sol[N][N],
            xMove[N], yMove[N])
  int k, next_x, next_y
  if movei = N*N
     return true

  for k = 0 to 8
     next_x = x + xMove[k]
     next_y = y + yMove[k]
     if isSafe(next_x, next_y, sol)=true
        sol[next_x][next_y] = movei
        if   SOLVEKTUTIL(next_x, next_y, movei+1, sol,
                   xMove, yMove) = true
                   return true
        else
                   sol[next_x][next_y] = -1

      return false
isSafe(int x, int y, int sol[N][N] )
     return (x>=0 && x<N && y>=0 && Y<N &&sol[x][y]==-1)
```

TIME COMPLEXITY

- The reason for this is that the knight's tour problem as we have implemented it so far is an exponential algorithm of size O(K^N), where N is the number of squares on the chessboard, where k is a small constant.
- Best case : In any step no backtracking is found necessary, then Time complexity is O(N), in an n*n chessboard. (N is no of squares on chessboard).

Greedy algorithm

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So the problems where choosing locally optimal also leads to global solution are best fit for Greedy.

A **greedy algorithm** is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum. In many problems, a greedy strategy does not usually produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

Use of greedy Algorithm

In mathematical optimization, greedy algorithms optimally solve combinatorial problems having the properties of matroids, and give constant-factor approximations to optimization problems with submodular structure.

Real life example of Greedy algorithm

Array element moved by k using single moves

Given a list of n integers containing numbers 1-n in a shuffled way and a integer K. N people are standing in a queue to play badminton. At first, the first two players in the queue play a game. Then the loser goes to the end of the queue, and the one who wins plays with the next person from the line, and so on. They play until someone wins k games consecutively.

This player becomes the winner.

Examples

Input: arr[] = {2, 1, 3, 4, 5}

k = 2

Output: 5

Explanation:

2 plays with 1, 1 goes to end of queue.

2 plays with 3, 3 wins, 2 goes to end of queue.

3 plays with 4, so 3 goes to the end of the queue.

5 plays with everyone and wins as it is the

largest of all elements.

Input: arr[] = {3, 1, 2}

k = 2

Output: 3

Explanation :

3 plays with 1. 3 wins. 1 goes to the end of the line.

3 plays with 2. 3 wins. 3 wins twice in a row.

A **naive approach** is to run two nested for loops and check for every element which one is more from i to n being the first loop and the second being from i+1 to n and then from 0-20*. Array element moved by k using single moves* to n-1 and count the number of continuous smaller elements and get the answer. This will not be efficient enough as it takes **O(n*n)** .

An **efficient approach** will be to run a loop from 1 to n and keep track of best (or maximum element) so far and number of smaller elements than this maximum. If current best loose, initialize the greater value to the best and the count to 1, as the winner won 1 time already.

If at any step it has won k times, you get your answer. But if k >= n-1, then the maximum number will be the only answer as it will the most number of times being the greatest. If while iterating you don’t find any player that has won k times, then the maximum number which is in the list will always be our answer.

Below is the implementation to the above approach

C++ Implementation

```
#include <iostream>
using namespace std;

int winner(int a[], int n, int k)
{
    // if the number of steps is more then
    // n-1,
    if (k >= n - 1)
    return n;
    // initially the best is 0 and no of
    // wins is 0.
    int best = 0, times = 0;
    // traverse through all the numbers
    for (int i = 0; i < n; i++) {
    // if the value of array is more
    // then that of previous best
    if (a[i] > best) {

    // best is replaced by a[i]
    best = a[i];

    // if not the first index
    if (i)
        times = 1; // no of wins is 1 now
    }
    else
    times += 1; // if it wins

   // if any position has more then k wins

    // then return

    if (times >= k)

    return best;}

    // Maximum element will be winner because

    // we move smaller element at end and repeat

    // the process.

    return best;}

// driver program to test the above function

int main()

{

    int a[] = { 2, 1, 3, 4, 5 };

    int n = sizeof(a) / sizeof(a[0]);

    int k = 2;

    cout << winner(a, n, k);

    return 0;}
```

Dynamic Programming

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.

Example

Memoization

The memoized program for a problem is similar to the recursive version with a small modification that it looks into a lookup table before computing solutions. We initialize a lookup array with all initial values as NIL. Whenever we need the solution to a subproblem, we first look into the lookup table. If the precomputed value is there then we return that value, otherwise, we calculate the value and put the result in the lookup table so that it can be reused later.

Following is the memoized version for nth Fibonacci Number.

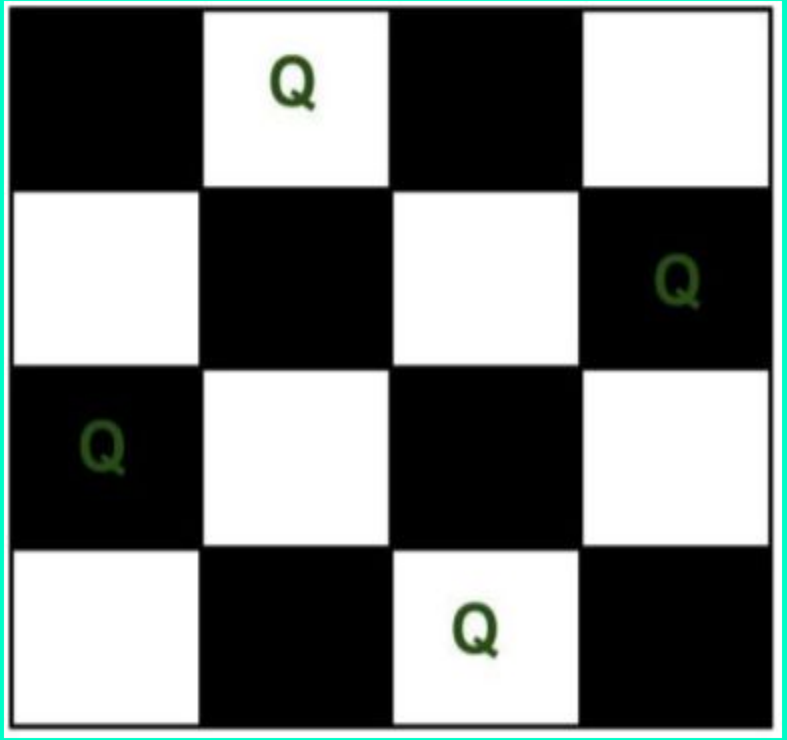
```
#include<stdio.h>
#define NIL -1
#define MAX 100
int lookup[MAX];
/* Function to initialize NIL values in lookup table
*/
void _initialize()
{ int i;
  for (i = 0; i < MAX; i++)
          lookup[i] = NIL;}
/* function for nth Fibonacci number */
int fib(int n)
{ if (lookup[n] == NIL)
   { if (n <= 1)
           lookup[n] = n;
           else
           lookup[n] = fib(n-1) + fib(n-2);
   }

   return lookup[n];
}
int main ()
{
  int n = 40;
  _initialize();
  printf("Fibonacci number is %d ", fib(n));
  return 0;
}
```

Branch and Bound

Branch-and-bound is a general technique for improving the searching process by systematically enumerating all candidate solutions and disposing of obviously impossible solutions. Branching is the process of spawning subproblems, and bounding refers to ignoring partial solutions that cannot be better than the current best solution. To this end, lower and upper bounds L and U are maintained. Since global control values on the solution quality improve over time, branch-and-bound is effective in solving optimization problems, in which a cost-optimal assignment to the problem variables has to be found.

Problem statement

N Queen problem 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



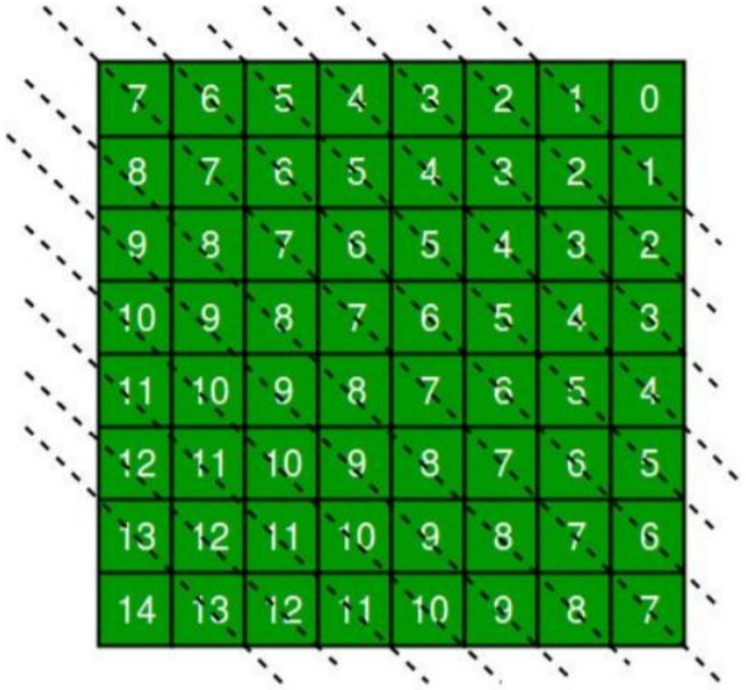
Approach

In Branch and Bound solution, after building a partial solution, we figure out that there is no point going any deeper as we are going to hit a dead end. We need to figure out an efficient way of keeping track of which cells are under attack. In previous solution we kept an 8 -by -8 Boolean matrix and update it each time we placed a queen, but that required linear time to update as we need to check for safe cells. Basically, we have to ensure 4 things: 1. No two queens share a column. 2. No two queens share a row.

3. No two queens share a top-right to left-bottom diagonal.

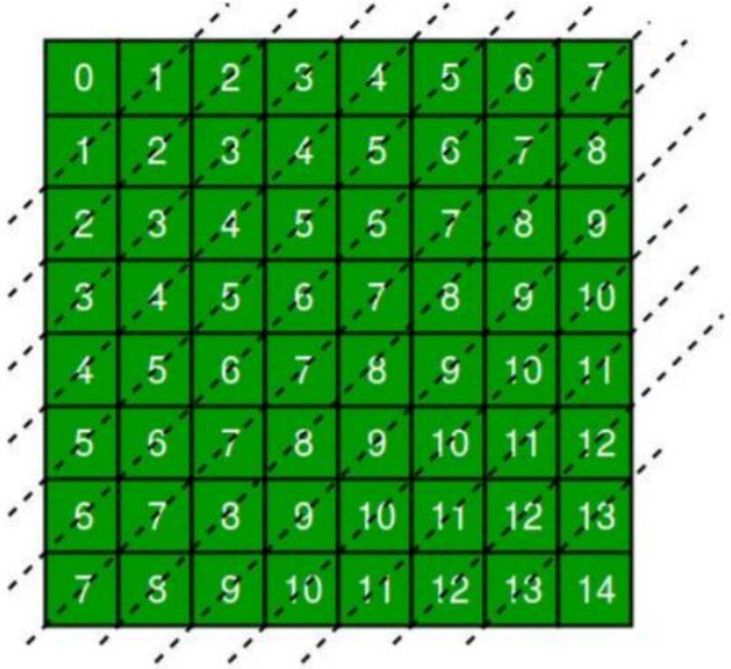
4. No two queens share a top-left to bottom-right diagonal. Let's create two N x N matrix one for / diagonal and other one for \ diagonal. Let's call them slashCode and backslashCode respectively. The trick is to fill them in such a way that two queens sharing a same / diagonal will have the same value in matrix slashCode, and if they share same \ diagonal, they will have the same value in backslashCode matrix.

Using above algo we will get given matrices -



7	6	5	4	3	2	1	0
8	7	6	5	4	3	2	1
9	8	7	6	5	4	3	2
10	9	8	7	6	5	4	3
11	10	9	8	7	6	5	4
12	11	10	9	8	7	6	5
13	12	11	10	9	8	7	6
14	13	12	11	10	9	8	7

r -c + 7



0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	9	10	11	12	13	14

r + c

The 'N – 1' in the backslash code is there to ensure that the codes are never negative because we will be using the codes as indices in an array. Now before we place queen i on row j, we first check whether row j is used (use an array to store row info). Then we check whether slash code (j + i) or backslash code (j – i + 7) are used (keep two arrays that will tell us which diagonals are occupied).

If yes, then we have to try a different location for queen i. If not, then we mark the row and the two diagonals as used and recurse on queen i + 1. After the recursive call returns and before we try another position for queen i, we need to reset the row, slash code and backslash code as unused again.

Python program to implement N- Queen problem

```python
N = 8
def printSolution(board):
    for i in range(N):
        for j in range(N):
            print(board[i][j], end = " ")
        print()
def isSafe(row, col, slashCode, backslashCode,
           rowLookup, slashCodeLookup, backslashCodeLookup):
    if (slashCodeLookup[slashCode[row][col]] or
            backslashCodeLookup[backslashCode[row][col]] or
            rowLookup[row]):
        return False
    return True
def solveNQueensUtil(board, col, slashCode, backslashCode,
                     rowLookup, slashCodeLookup,
                     backslashCodeLookup):
    if(col >= N):
        return True
```

```
for i in range(N):
        if(isSafe(i, col, slashCode, backslashCode,
                    rowLookup, slashCodeLookup,
                    backslashCodeLookup)):
            board[i][col] = 1
            rowLookup[i] = True
            slashCodeLookup[slashCode[i][col]] = True
            backslashCodeLookup[backslashCode[i][col]] =
True
            if(solveNQueensUtil(board, col + 1,
slashCode, backslashCode,
                                rowLookup,
slashCodeLookup,
                                backslashCodeLookup)):
                return True
            board[i][col] = 0
            rowLookup[i] = False
```

```
 slashCodeLookup[slashCode[i][col]] = False
            backslashCodeLookup[backslashCode[i][col]] =
False
    return False
def solveNQueens():
    board = [[0 for i in range(N)]
                for j in range(N)]
    slashCode = [[0 for i in range(N)]
                    for j in range(N)]
    backslashCode = [[0 for i in range(N)]
                        for j in range(N)]
    rowLookup = [False] * N
    x = 2 * N - 1
    slashCodeLookup = [False] * x
    backslashCodeLookup = [False] * x
    for rr in range(N):
        for cc in range(N):
            slashCode[rr][cc] = rr + cc
            backslashCode[rr][cc] = rr - cc + 7
```

```
if(solveNQueensUtil(board, 0, slashCode,
backslashCode,
                          rowLookup,
slashCodeLookup,
                          backslashCodeLookup) ==
False):
        print("Solution does not exist")
        return False
    printSolution(board)
    return True
solveNQueens()
```

THANK YOU