

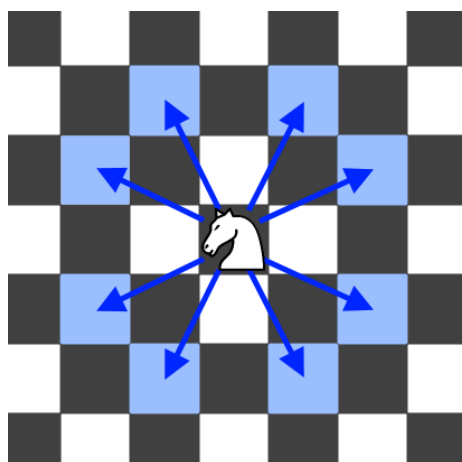
## Day 28 / 100 :

### Topic - Dynamic Problem

#### 1 Problem statement: [Knight Probability in chess](#) (Medium)

On an  $n \times n$  chessboard, a knight starts at the cell (row, column) and attempts to make exactly  $k$  moves. The rows and columns are 0-indexed, so the top-left cell is (0, 0), and the bottom-right cell is ( $n - 1$ ,  $n - 1$ ).

A chess knight has eight possible moves it can make, as illustrated below. Each move is two cells in a cardinal direction, then one cell in an orthogonal direction.



Each time the knight is to move, it chooses one of eight possible moves uniformly at random (even if the piece would go off the chessboard) and moves there.

The knight continues moving until it has made exactly  $k$  moves or has moved off the chessboard.

Return the probability that the knight remains on the board after it has stopped moving.

#### Example 1:

Input:  $n = 3$ ,  $k = 2$ , row = 0, column = 0

Output: 0.06250

**Explanation:** There are two moves (to (1,2), (2,1)) that will keep the knight on the board.

From each of those positions, there are also two moves that will keep the knight on the board.

The total probability the knight stays on the board is 0.0625.

**Example 2:**Input:  $n = 1$ ,  $k = 0$ , row = 0, column = 0

Output: 1.00000

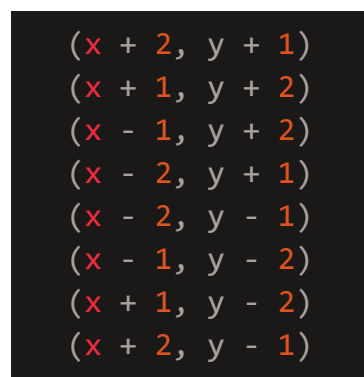
**Solutions :****Approach 1 - Iterative Approach**

## Explanation

The idea is, we have a knight and it has to move in 8 directions and it has  $k$  moves in total. Also, in question it is given that, it will continue moving until  $k$  becomes 0 or knight moves off the chessboard. And, we need to find the probability that knight remains on the board after it has stopped moving.

Now, If you look at the problem, it generally talks about the path and exploration and stuff like that. Whenever, we talk related to exploring all the path then we should think about recursion.

Now, we have decided that we are going to use recursion and we will be exploring all the paths. So, possible path for the knight is going to be given below, when the knight is at  $(x, y)$  from there it will move,



```
(x + 2, y + 1)
(x + 1, y + 2)
(x - 1, y + 2)
(x - 2, y + 1)
(x - 2, y - 1)
(x - 1, y - 2)
(x + 1, y - 2)
(x + 2, y - 1)
```

in these many directions.

Now, let's talk about the probability, Since knight has to move in 8 directions, for one move of knight the probability will be  $1/8$ .

Also, let's talk about the base case:

Check if the position is valid or not if not return 0.

Check if  $k == 0$  or not, if  $k$  is 0 then there is no movement from knight and Probability of knight on the board is 1.

**Solutions :****Approach 2 - Recursive**

```

class Solution {
public:
    double knightProbability(int N, int K, int r, int c) {
        return helper(N, K, r, c);
    }

    double helper(int N, int K, int row, int col){

        if(row < 0 || col < 0 || row >= N || col >= N) return 0.0;

        if(K == 0) return 1.0;

        double ans = helper(N, K-1, row+2, col+1) + helper(N, K-1,
row+1, col+2) +
                    helper(N, K-1, row-1, col+2) + helper(N, K-1,
row-2, col+1) +
                    helper(N, K-1, row-2, col-1) + helper(N, K-1,
row-1, col-2) +
                    helper(N, K-1, row+1, col-2) + helper(N, K-1,
row+2, col-1);

        double result = ans / 8.0;
        return result;
    }
};

```

**Solutions :****Approach 2 - Recursive + Memoization - using matrix**

```

class Solution {
public:
    double knightProbability(int N, int K, int r, int c) {
        vector<vector<vector<double>>> memo(N+1,
vector<vector<double>>(N+1, vector<double>(K+1, -1)));
        return helper(N, K, r, c, memo);
    }
};

```

```

    }

    double helper(int N, int K, int row, int col,
vector<vector<vector<double>>>& memo){

        if(row < 0 || col < 0 || row >= N || col >= N) return 0.0;

        if(K == 0) return 1.0;

        if(memo[row][col][K] != -1) return memo[row][col][K];

        double ans =
            helper(N, K-1, row+2, col+1, memo) + helper(N, K-1,
row+1, col+2, memo) +
            helper(N, K-1, row-1, col+2, memo) + helper(N, K-1,
row-2, col+1, memo) +
            helper(N, K-1, row-2, col-1, memo) + helper(N, K-1,
row-1, col-2, memo) +
            helper(N, K-1, row+1, col-2, memo) + helper(N, K-1,
row+2, col-1, memo);

        double result = ans / 8.0;
        memo[row][col][K] = result;
        return result;
    }
};

```

## Solutions :

### Approach 1 - Using Map

```

class Solution {
public:
    unordered_map<string, double> mp;
    double knightProbability(int N, int K, int r, int c) {
        mp.clear();
        return helper(N, K, r, c);
    }
}

```

```

double helper(int N, int K, int row, int col){

    if(row < 0 || col < 0 || row >= N || col >= N) return 0.0;

    if(K == 0) return 1.0;

    string key = to_string(K) + "#" + to_string(row) + "#" +
to_string(col);
    if(mp.find(key) != mp.end()) return mp[key];

    double ans = helper(N, K-1, row+2, col+1) + helper(N, K-1,
row+1, col+2) +
                helper(N, K-1, row-1, col+2) + helper(N, K-1,
row-2, col+1) +
                helper(N, K-1, row-2, col-1) + helper(N, K-1,
row-1, col-2) +
                helper(N, K-1, row+1, col-2) + helper(N, K-1,
row+2, col-1);

    double result = ans / 8.0;
    mp[key] = result;
    return result;
}
};

```

## 2 Problem statement: [3Sum](#) (Medium)

Given an integer array `nums` of length `n` and an integer `target`, find three integers in `nums` such that the sum is closest to `target`. Return the sum of the three integers. You may assume that each input would have exactly one solution.

Example 1:

Input: `nums = [-1,2,1,-4]`, `target = 1`

Output: 2

Explanation: The sum that is closest to the target is 2.  $(-1 + 2 + 1 = 2)$ .

Example 2:

Input: `nums = [0,0,0]`, `target = 1`

Output: 0 Explanation: The sum that is closest to the target is 0.  $(0 + 0 + 0 = 0)$ .

**Solutions :****Approach 1 - Using Sort**

Sort the vector and then no need to run  $O(N^3)$  algorithm as each index has a direction to move.

The code starts from this formation.

```

-----
^  ^                               ^
|  |                               |
| +- second                        third
+-first

```

if  $\text{nums}[\text{first}] + \text{nums}[\text{second}] + \text{nums}[\text{third}]$  is smaller than the target, we know we have to increase the sum. so only choice is moving the second index forward.

```

-----
^  ^                               ^
|  |                               |
| +- second                        third
+-first

```

if the sum is bigger than the target, we know that we need to reduce the sum. so only choice is moving 'third' to backward. of course if the sum equals to target, we can immediately return the sum.

```

-----
^  ^                               ^
|  |                               |
| +- second                        third
+-first

```

when second and third cross, the round is done so start next round by moving 'first' and resetting second and third.

```

-----
^   ^               ^
|   |               |
|   +- second      third
+-first

```

while doing this, collect the closest sum of each stage by calculating and comparing delta. Compare `abs(target-newSum)` and `abs(target-closest)`. At the end of the process, the three indexes will eventually be gathered at the end of the array.

```

-----
      ^   ^   ^
      |   |   ^- third
      |   +- second
      +-first

```

if no exactly matching sum has been found so far, the value in closest will be the answer.

```

class Solution {
public:
int threeSumClosest(vector<int>& nums, int target) {
    if(nums.size() < 3) return 0;
    int closest = nums[0]+nums[1]+nums[2];
    sort(nums.begin(), nums.end());
    for(int first = 0 ; first < nums.size()-2 ; ++first)
    {
        if(first > 0 && nums[first] == nums[first-1])
        continue;
        int second = first+1;
        int third = nums.size()-1;
        while(second < third) {

```

```
        int curSum =  
nums[first]+nums[second]+nums[third];  
        if(curSum == target) return curSum;  
        if(abs(target-curSum)<abs(target-closest)) {  
            closest = curSum;  
        }  
        if(curSum > target) {  
            --third;  
        } else {  
            ++second;  
        }  
    }  
}  
return closest;  
}  
};
```