Brute force & Backtracking

Q $1 \le N \le 99$ $\{a,b,c,d,e,f,g,h,i,j\}$ → Variables

Brute force: $\frac{abcde}{fghij} = N$

↓ one to one

$\{0,1,2,3,4,5,6,7,8,9\}$

Constraint ② ; Constraint ① ie one to one correspondence

$[a \to 3, b \to 5, \ldots][a \to 2, b \to 1, \ldots]$ - - -

⇒ Meth ① 1> Generate (Constraint ①) → $10!$ 2> Check ② valid or not — $10!$ loop

Meth ② 1> Check ② 2> generate ①

```
for( fghij ∈ [1 ... 99999])
    abcde = N * fghij
    check abcdefghij are distinct(); → O(10)
```

10^5 loop.

☆ Every brute force problem has constraints, you have to choose tighter constraint for generate & looser (bigger) for check.

☆ Pruning - use extra information to reduce search space

```
void solve1()
{   vector<int> arr = {0,1,2,3,4,5,6,7,8,9,10}
    do { int abcde = arr[0]*10^5 + arr[1]*10^4 + ... arr[4]*10^0;
         int fghij = arr[5]*10^5 + arr[6]*10^4 + ... arr[9]*10^0;
         if( abcde/fghij == N) cout << abcde << " " << fghij << endl;
    } while(next_permutation(arr.begin(), arr.end()));
}
```

```
Void solve2(){ int n; cin>>n;
  for( int fghij = 1234 ; fghij <= 98765/n ; fghij++)
    { int abcde = n* fghij;
  } } if( all_distinct()) cout<< abcde <<" " << fghij <<endl;
```

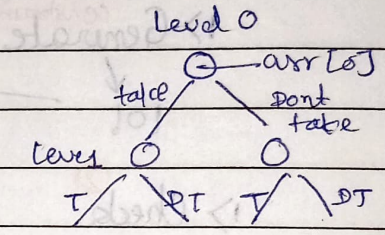
Backtracking Framework.

- level → the things we can iterate on.
- choice → make choice on that level
- check → can we make this choice or not
- move → if yes, make choice & move to next level.

Q1 Given an array, find all the subsets of array of size k.

⇒ L1 L2 L3 . .

{ 1, 2, 3, 4, 5 . . }



check() - checks if no. of elements is less than k. in sol.

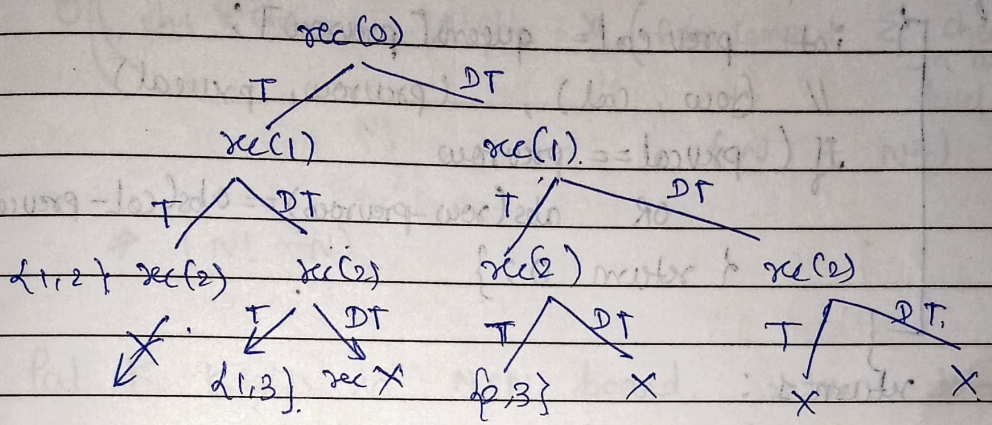
```
int n, k;
vector<int> arr, sol;
void rec (int level)
{   // base case
    if (level == n)
      { if (sol.size() >= k) print sol\n; }

    // recursive case
    if (sol.size() < k)  // check
      { sol.push_back(arr[level]); // take
        rec(level+1);  // move
        sol.pop_back();
      }
    // don't take
    rec(level+1);  // move
}
```

⇒ in main call rec(0);

if don't write check ⇒ TC = $O(2^n)$.

if write check ⇒ TC = ${}^nC_k$

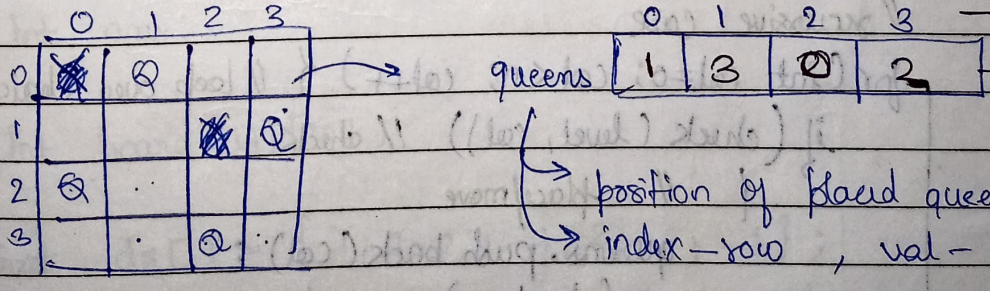
N=3 , {1,2,3}, K=2



Q2 N-Queen, place N queen on nxn board s.t. no two queens attack each other.

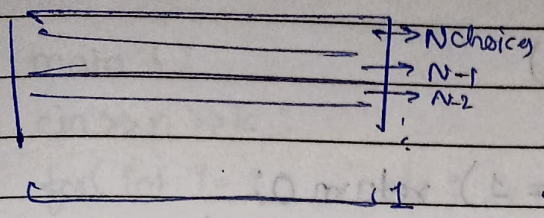
⇒ as we have n queens & nxn board, each row must should have one queen exactly.

each row is a level, and each level has col as choices

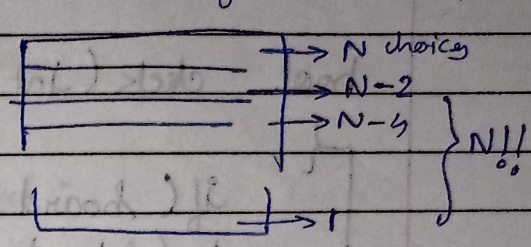


```
int n;
vector<int> queens;

bool check( int row, int col). // check if we can place Queen at (r,c)
{ for( int prevrow=0; prevrow < row; prevrow++)
    { int prevcol = queens[prevrow];
      // (row, col), (prevrow, prevcol)
      if ( prevcol == prevrow
           OR  abs(row-prevrow) == abs(col-prevcol))
          { return 0; }
    }
  return 1;
}

void rec ( int level). // level is row.
{   // base case.
    if( level == n) // means we placed n queens
      { if ( queens.size() == n)
          pr(quien);
      }

    // recursive case
    for(int col=0; col<n; col++). { // loop over choices
        if (check (level, col)) // check
          { // place/move
            queens.push_back(col);
            rec (level+1);
            queens.pop_back();
          } }
}
```

Every queen blocks 1 col



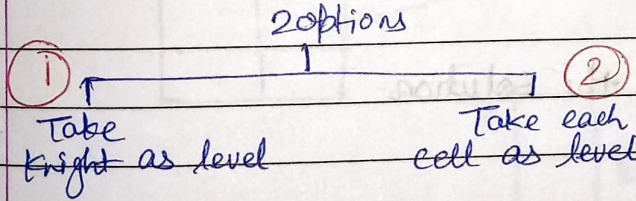
Consider digonal also:

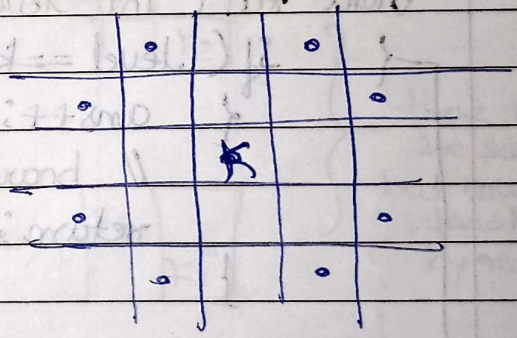


$O\left(\left(\underbrace{ch_1 * ch_2 - - - * ch_n}_{N!} * (\text{Base comp} + \underbrace{\Sigma \text{ checks in all levels}}_{(N + N^2)})\right)\right)$

$\Rightarrow (N! * N)$

Qs Put K knights in N×N board, return total no. of ways of putting K knights on board st no knights attack each other.

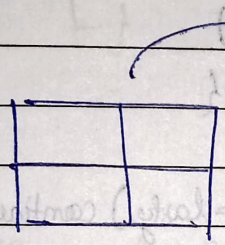




```
int n, k;
int ans;

int board[14][14];

int dx[] = {1, 2, 2, 1, -1, -2, -2, -1};
int dy[] = {2, 1, -1, -2, -2, -1, 1, 2};
```

```
bool check(int i, int j)
{
    if(board[i][j] == 1) return 0;
    for(int pos=0 ; pos<8; pos++)
    {   int nx = i+dx[pos];
        int ny = j+dy[pos];
        // (nx, ny) -> are the cells getting attacked.
        if(nx<n && nx>=0 && ny<n && ny>=0
                && board[nx][ny]==1) return 0;
    }
    return 1;
}

void rec(int level) // level = no of knights placed.
{   if(level==k)
    {   ans++;
        // board gives the solution
        return;
    }

    // choices
    for(int i=0 ; i<n ; i++)
    {   for(int j=0 ; j<n ; j++)
        {   // check
            if(check(i,j))
            {   // move
                board[i][j]=1;
                rec(level+1);
                board[i][j]=0;
            }
        }
    }
}
```

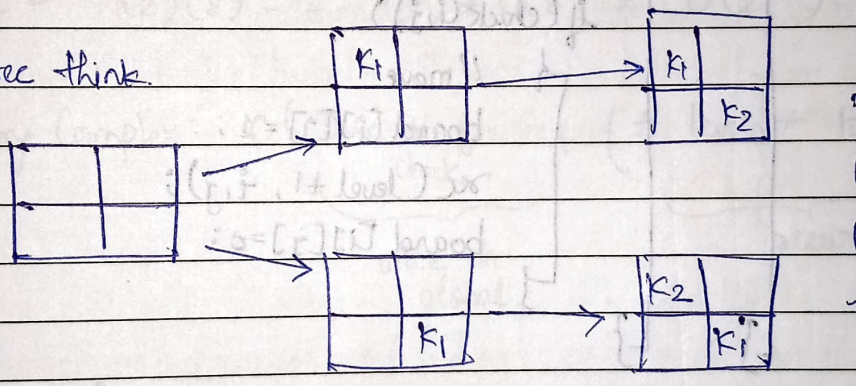
```
int main ()
{   cin>>n>>k;
    for(int i=1; i<=k; i++) f=f*i;
    rec(0);

    cout<< ans/f << endl;
}
```



for N=2×2
K=2

$^4C_2$ — i.e 6 ways.

but rec think.



these cases are same but counted separately by recursion

So each arrangement of K knights is calculated K! times
So we have to divide ans by K! to remove the repetition.

$$TC = [N^2 \cdot (N^2-1) \cdot (N^2-2) \cdots (N^2-K+1)]$$

$$\hookrightarrow \frac{(N^2)!}{(N^2-K)!} \rightarrow {}^NP_K$$

$$\& \text{ total solns} = \frac{(N^2)!}{(N^2-K)! \times (K!)} \rightarrow {}^NC_K$$

General TC: O((#choices at Level 1) * (#ch at L2) * ---- (#ch at Lk) * (Avg check))
↳ O(# of ans * Avg check).

to remove repetition

```
void rec(int level, lastx, lasty)
{   if(level==k)
    { ans++;
      return;
    }

// choices
for(int i=lastx; i<n; i++)
{   for(int j=0; j<n; j++){
      // check
      if(i==lastx && j<=lasty) continue;
      if(check(i,j))
      {  // move
         board[i][j]=1;
         rec(level+1, i, j);
         board[i][j]=0;
      }
}}
}
```

Take / don't take method

```
void rec(int level, int knights_left)
{   if(level==n*n)
    {  if(knights_left==0) ans++;
       return;
    }

int row = level/n;
int col = level%n;
```

```
// don't place a knight at (row, col)
rec(level+1, knights);

// place a knight.
if(knight>0 && check(row, col))
{
    board[row][col] = 1;
    rec(level+1, knights left-1);
    board[row][col] = 0;
}
}
```

$TC = (2)(2)\cdots(2) * O(8) = O(2^{N^2})$

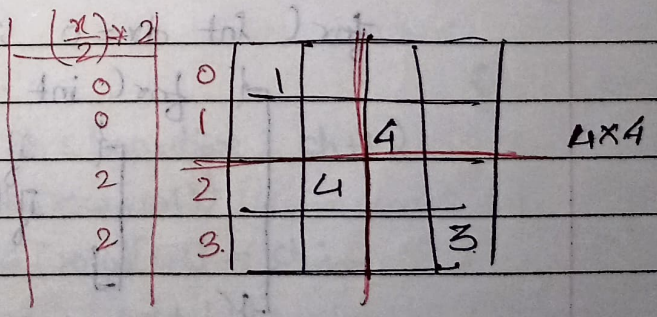
Memory Complex: $O($ Data str. for check $+$ (# levels $*$ Parameters memory $))$

Data str. for check → once in global.

levels * Parameters memory → recursion stack

Q4 Sudoku Solver NXN

⇒ level → cell
choice → [1-4]
check → valid
move → place



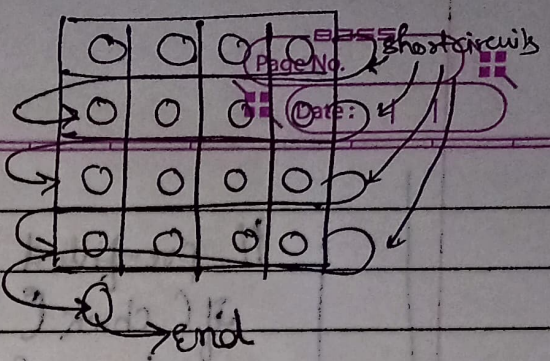
for checking squares $(x,y) \rightarrow \left(\left(\frac{x}{2}\right)*2, \left(\frac{y}{2}\right)*2\right)$.

↳ gives topleft corner of ~~each~~ cell.

as $x = \left(\frac{x}{p}\right)*p + (x\%p)$

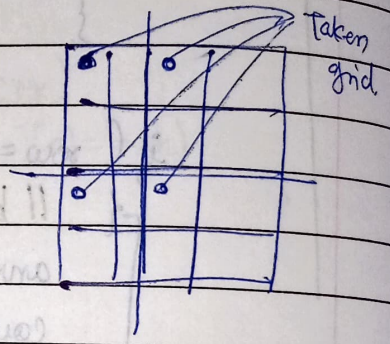
Quotient, remainder

```
const int BoardSize = 4;        // 4X4
const int Cellsize = 2;         // 2X2


bool check (int ch, int row, int col)
{
    // check in row
    for (int c=0; c< BoardSize ; c++)
    {  if (c != col && board[row][c]==ch)
        { return false; }
    }

    // check in col
    for (int r=0; r< BoardSize ; r++)
    {  if (r!=row && board[r][col]==ch)
        { return false; }
    }

    // check in square
    int str = (row/cellsize) * cellsize;
    int stc = (col/ cellsize) * cellsize;
    for (int dx=0 ; dx< 2 ; dx++)
    {  for (int dy =0 ; dy < 2 ; dy++)
        {  if (str+dx == row && stc+dy == col) continue;
           if (board [str+dx][stc+dy] == ch) return 0;
        }
    }

    return true;
}
```




```
int ans=0;

void rec (int row, int col)
{   if( col == BoardSize)  // short circuit
    { rec(row+1, 0);
      return;
    }

    if ( row == BoardSize)
    { // base case
      ans++;
      cout<< "Answer" << ans << endl;
      for ( int i=0; i< BoardSize; i++)
      { for( int j=0; j< BoardSize; j++)
        { cout<< board[i][j]<<" ";
        } cout<< endl;
      }
      return;
    }
// choices
if ( board[row][col] ==0)
{ // we need to fill
  for ( int ch=1; ch<= BoardSize; ch++)
  { if (check (ch, row, col))
    { board[row][col] = ch;
      rec (row, col+1);
      board[row][col]=0;
    }
  }
} else {
```

```
// pre-filled
if( check( board[row][col], row, col))
    { rec(row, col+1);
    }
}
```



Optimization using bitmasks

```
int takenRow[BoardSize];
int takenCol[BoardSize];
int takenGrid[cellSize][cellSize];
int log2[1000];
int getchoices( int row, int col)
 {
   int taken = ( takenRow[row] | takenCol[col] | takenGrid[row/cellsize]
                                                 [col/cellsize]);
   int nottaken = ((1<<(BoardSize+1))-1)^taken;
   // OR nottaken = (~taken) & ((1<<(BoardSize+1))-1);
   if ( nottaken & 1) nottaken^=1;
   return nottaken;
 }

int makemove ( int ch, int row, int col)
  {  board[row][col] = ch;
     takenRow[row] |= (1<<ch);
     takenCol[col] |= (1<<ch);
     takengrid[row/cellsize][col/cellsize] |= (1<<ch);
  }
```

```
int revertmove ( int ch, int row, int col):
{   board[row][col] = 0;
    takenRow ^= (1<<ch);
    takenCol ^= (1<<ch);
    takenGrid[row/cellsize][col/cellsize] ^= (1<<ch);

void rec ( ____ )
{
    ____
    ____

    // choices.
    if( board[row][col] == 0)
    { // we need to fill
        int chmask = getchoices(row, col);
        for( int ch=1 ; ch<=BoardSize ; ch++)
        { if ( chmask & (1<<ch))
            { makemove (ch, row, col);
              rec (row, col+1);
              revertmove (ch, row, col);
            }
        }
    } else {
        // prefilled
        rec(row, col+1);
    }
}

void Solve ()
{   for (int i=0 ; i<BoardSize ; i++)
    { for (int j=0 ; j< BoardSize ; j++)
      { int ch ;
```

```
        cin>>ch;
        makemove(ch,i,j);
      }
    }
    rec(0,0);
  }
```

// There one more optimization

Q5- Generate all permutations of a set of numbers in sorted order

```
int n;
vector< vector<int>> all_sol;
vector<int> curr_per;
map<int,int> mp;
void rec(int level)  // level = how many elements placed in curr_permutation
{
  if(level == n) // base case
  {
    all_sol.push_back(curr_per);
    return;
  }

  // choices
  for(auto v: mp)
  {
    if(v.second != 0)
    {
      mp[v.first]--;
      curr_per.push_back(v.first);
      rec(level+1);
      curr_per.pop_back();
      mp[v.first]++;
    }
  }
}
```

```
void solve()
{   cin>>n;
    int arr[n];
    for( int i=0 ; i<n ; i++)
    {   cin>> arr[i];
        mp[arr[i]]++;
    }
    see(0);
    pr(allsol);
}
```

Thinking:

→ {1, 2, 2}
→ {2, 1, 2}
→ {2, 2, 1}

⟶ map ⟶ [{1→1, 2→2}]

{1→1, 2→2, 1}

OR

1 | 2 —

↑ level 0

{1→1, 2→2, 1}

2 | — —

↑ level 0 ↑ level 1

Meet in the Middle

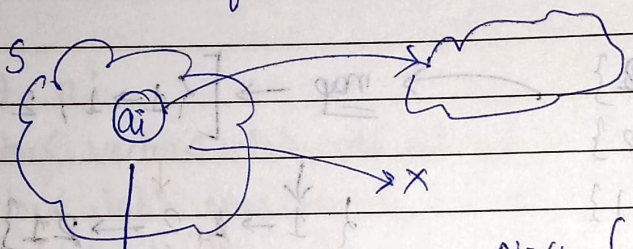
Q1 Form a set of N Given Numbers, find the no. of subsets having sum of elements <= X for a given X.

N = {1, 2, 3}

X = 5

{} → 0
{1} → 1
{2} → 2
{3} → 3
{1, 2} → 3
{2, 3} → 5
{1, 3} → 4
~~{1, 2, 3} → 6~~

if N <= 20, we have $2^{20} = 10^6$ subsets each of length N, so bruteforce works



Bitmask

N=4 (3 2 1 0)

(arr[3] + arr[2] + arr[0])

Coding Sol^n.

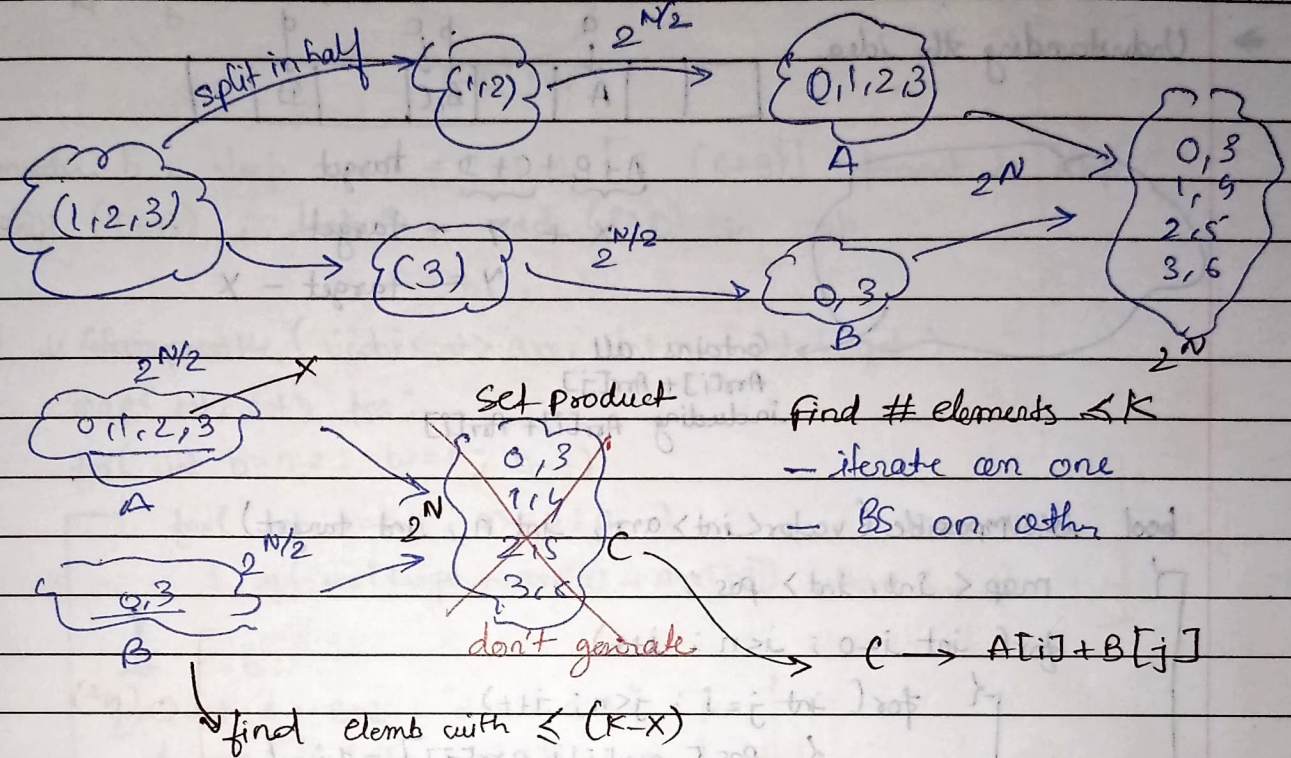
```
vector<int> generate (vector<int> arr).
{ int n = arr.size();
  vector <int> subVals;
  for (int mask = 0 ; mask < (1 << n); mask++)
    { int sum = 0;
      for (int j=0; j<n ; j++)
        { if (mask >> j) & 1 ) sum += arr[j];
        }
      subVals.push_back(sum);
    }
  sort (subVals.begin() , subVals.end ()); ⟶ O(2^N log 2^N)
  return subVals;
}
```

O(N) inside O(2^N . N)

∴ TC = $O(N \cdot 2^N)$

★ Hint for meet in the middle i.e $N \geq 30$ or 40

For $N <= 40$ — $2^{40} = 10^{12}$ → TLE




```
long long count(vector<in> arr, int X)
{
    long long ans = 0;
    vector<int> newarr[2];

    for(int i=0; i<arr.size(); i++)
    {
        newarr[i%2].push_back(arr[i]);
    }                                          } O(N)

    sub0 = generate(newarr[0]);
    sub1 = generate(newarr[1]);                } O(2 · n/2 · 2^{N/2})

    for(auto v: sub0)                          — 2^{N/2}
    {
        ans += ( upper_bound
            (sub1.beg(), sub1.end(), X-v)
            - sub1.begin) — log(2^{N/2})
    }
    return ans;
}
```

```
vector<int> generate(vector<int> arr)
{
    int n = arr.size();
    vector<int> subvals;
    for(int mask=0; mask < (1<<n); mask++)
    {
        int sum = 0;
        for(int j=0; j<n; j++)
        {
            if((mask>>j)&1) sum += arr[j];
        }
        subvals.push_back(sum);
    }
    sort(subvals.beg(), subvals.end());
    return subvals;
}
```

Overall TC = $O(n \cdot 2^{N/2})$

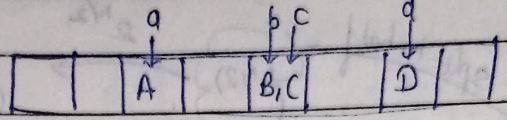
VARIANT-1

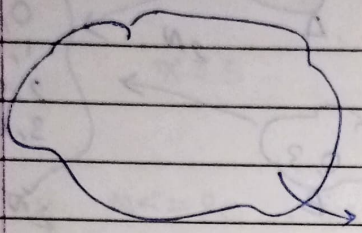
Q2 From an array of N numbers Arr, and given a target (TARGET) : Does their exists 4 values a,b,c,d s.t

(Arr[a] + Arr[b] + Arr[c] + Arr[d] == TARGET)

1 <= N <= 1000 , 0 <= a,b,c,d <= N-1 , a can be equal to b.

⇒ Understanding the idea.





A+B+C+D = target

X + Y = target

Y = target - X

Contains all Arr[i] + Arr[j] including Arr[i] + Arr[i]

```
bool isItPossible(vector<int> arr, int n, int target)
{
    map<int, int> pos;
    for(int i=0; i<n; i++)
    {
        for(int j=i; j<n; j++)
        {
            pos[arr[i] + arr[j]] = 1;
        }
    }
    for(int i=0; i<n; i++)
    {
        for(int j=i; j<n; j++)
        {
            if(pos[target - (arr[i] + arr[j])]) return 1;
        }
    }
    return 0;
}
```

O(N^2) → logN

O(logN) N^2

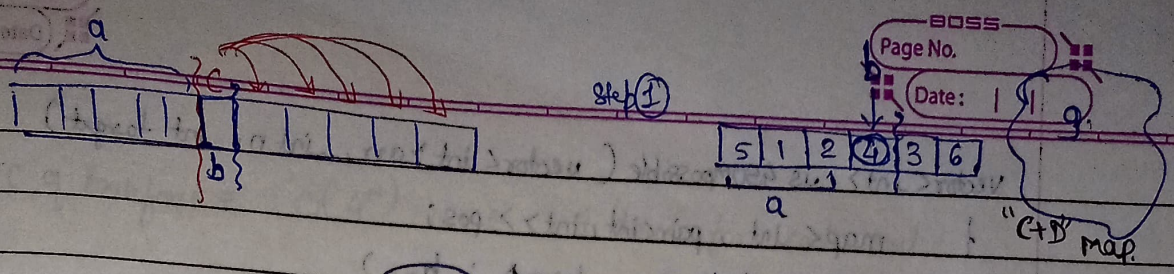
⇒ TC = O(N^2 logN)

VARIANT - 2

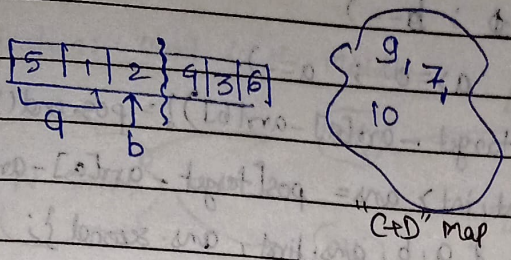
From an array of N numbers Arr, and - - - .

(Arr[a] + Arr[b] + Arr[c] + Arr[d] == TARGET)

1 <= N <= 1000 , 0 <= a < b < c < d <= N-1



Step 2



① consider b, loop over A, check [T-(A+B)] present in map.
② shift (C=B), loop D, add (C+D) in map.

```
bool is4SumPossible(vector<int> arr, int n, int target)
{
    map<int, int> pos;
    for(int b=n-2; b>=1; b--)
    {
        for(int a=b-1; a>=0; a--)
            { if(pos[target - arr[a] - arr[b]]) return 1; }
    }
    int c=b;
    for(int d = c+1; d<n; d++)
    {
        pos[arr[c]+ arr[d]] = 1;
    }
    return 0;
}
```

$n^2 \log n$

$n^2 \log n$

$TC = O(n^2 \log n)$

★ General meet in the middle 4 sum problem, pattern, i.e maintain loop on one subhalf of the problem and maintain a map of other half.

VARIANT 3

find the 4 values. s.t Arr[a] + Arr[b] + Arr[c] + Arr[d] == TARGET.

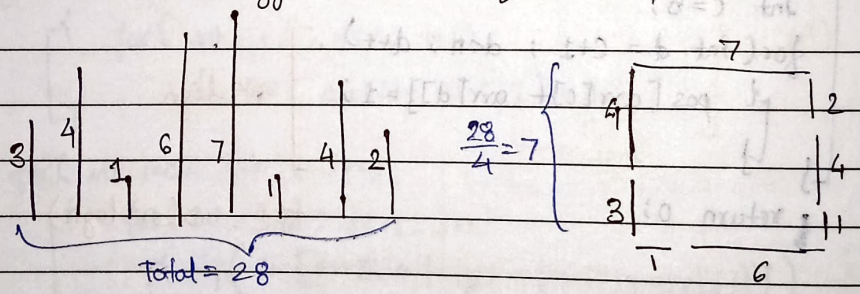
$1 <= N <= 1000$, $0 <= a < b < c < d <= N-1$

⇒ pos[C+D] → index pair {c, d}.

```
vector<int> isSumpossible(vector<int> arr, int n, int target)
{
    map<int, pair<int,int>> pos;
    for(int b=n-2; b>=1; b--)
    {
        for(int a=b-1; a>=0; a--)
        {
            if(pos.find(target-arr[a]-arr[b])!=pos.end())
            {
                pair<int,int> ans = pos[target-arr[a]-arr[b]];
                return {a,b,ans.first,ans.second};
            }
        int c=b;
        for(int d=c+1; d<n; d++)
            pos[arr[c]+arr[d]] = {c,d};
        }
    }
```

TC = $O(N^2 \log N)$

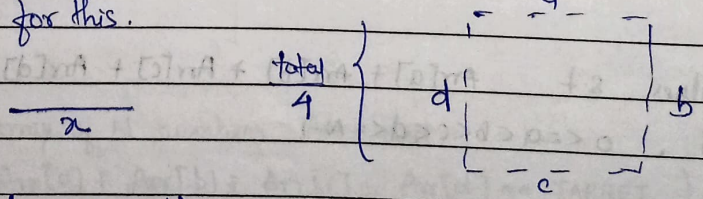
Q3 - Given N rods with their lengths in array Arr[]. Can you use all the rods to combine them into bigger rods and form a square?



⇒ Observations: ① → on same side, order does not matter

② → sq we form has side len = total/4

Bruteforce for this.



generate(i, a, b, c, d)

↳ (i+1, a+arr[i], b, c, d)

↳ (i+1, a, b+arr[i], c, d)

↳ (i+1, a, b, c+arr[i], d)

↳ (i+1, a, b, c, d+arr[i])

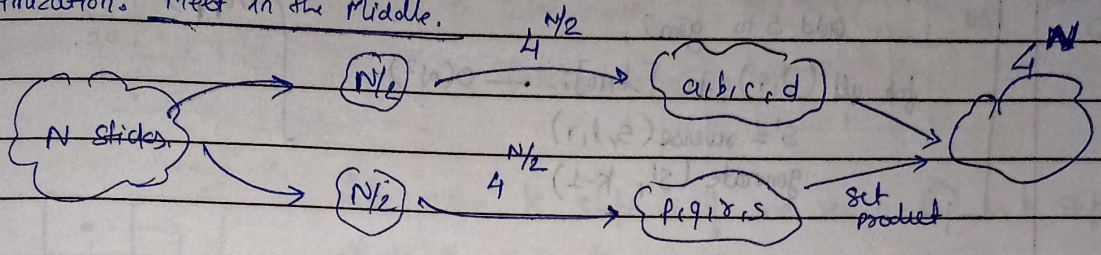
(i==N)

↳ a, b, c, d

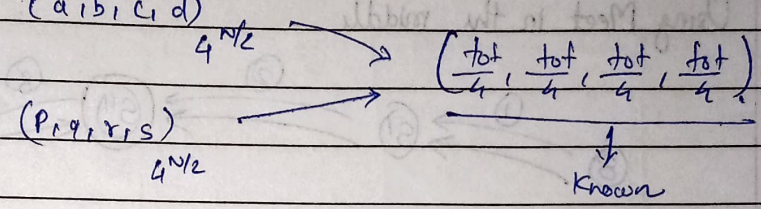
↳ a=b=c=d → square!

TC of Bruteforce = $O(4^N)$ i.e 4 choice for each rod.

Optimization: Meet in the Middle.



How to combine,



PSEUDOCODE:

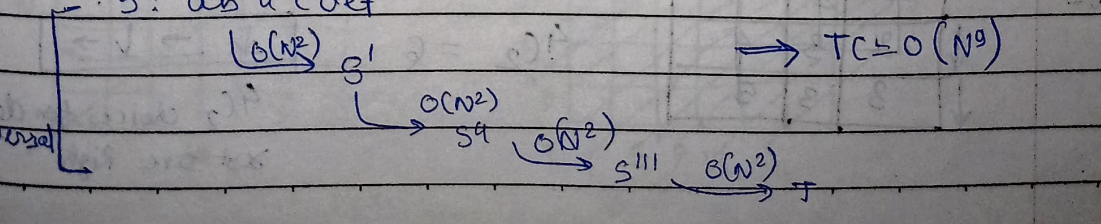
```
N, Arr = { }
Arr1 , Arr2 = split in almost half.(Arr);        // O(N)
Pos1 = generate (Arr1, 0,0,0,0,0)   // possible quadraples from 1st half.
                  i a b c d
                                                  O(4^N/2 . N)
Pos2 = generate (Arr2, 0,0,0,0,0)   // possible quadraples from 2nd half.
                                                  // O(4^N/2 . N)
x = tot/4;   // if tot %4 != 0 ⇒ Not possible   // O(1)
for{a,b,c,d} in Pos1:                           // O(4^N/2)
    if ( { x-a, x-b, x-c, x-d } in Pos2): return 1;   // O(N)
return 0.
```

Q4. From a string S, can we create the string T using 4 substring reversals?

S (abacde) → ① abedca → ② acdeba → ③ dcaeba → ④ beacda

T (beacda)

⇒ Brute-force : $O(N^2)$ no. of choices for reversal of substrings.

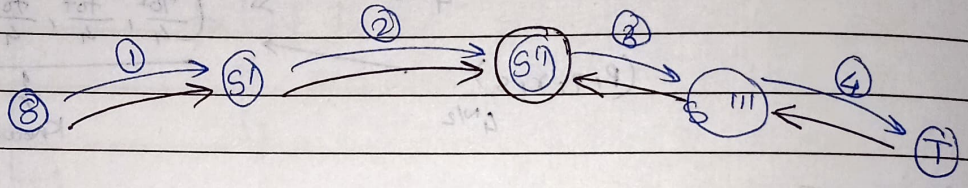


Code:

```
generate (string S, int K)
    if (K==0):
        add S to gen;
    for all (l,r) in [l,n]:  — O(N^2)
        S' = reverse(S,l,r)
        generate (S', K-1)
```

O(N^3)

Using Meet in the middle



generate (S, 2)
- → gen1
- → gen2

loop over one set & find match in other

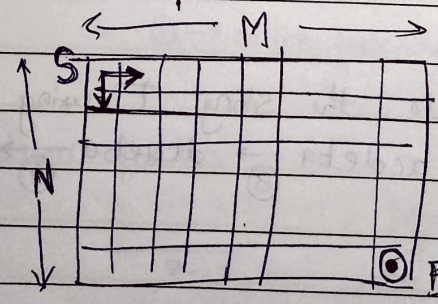
generate (T, 2)

O(N4 . N . logN)
- for matching
- set

TC: $O((N^2)^K \cdot N)$
$= O(N^5)$

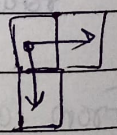
There are O(N4) elements in gen1 & gen2

Q5- Grid XOR paths.



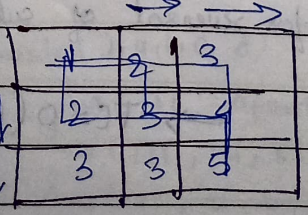
$0 \le N+M \le 20$

$0 \le a_{ij} \le 10^9$.

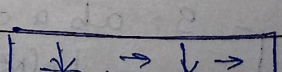


find no. of paths from S to E s.t the XOR of elements in path is equal to K.

Ex-



$^4C_2 = 6$



$^4C_2$ choices for down are Right

BOSS
Page No.
Date: / /

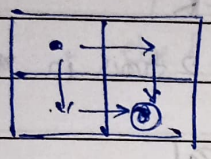
1 ^ 2 ^ 3 ^ 4 ^ 5 = 1

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

N-1 → Down moves
M-1 → Right moves

⇒ Total moves = $^{N+M-2}C_{N-1}$ OR $^{N+M-2}C_{M-1}$

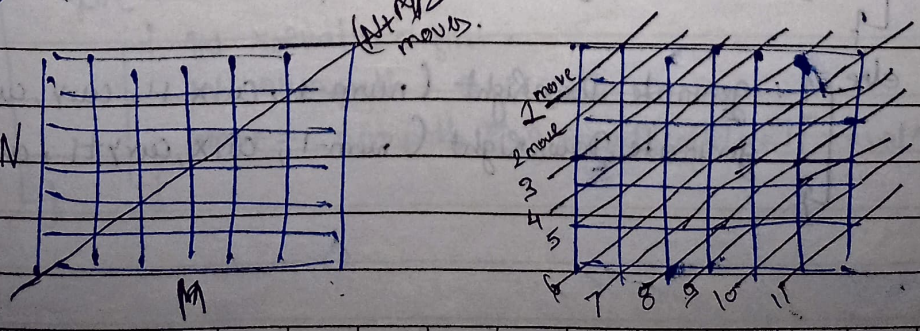
Total moves ≤ 2^{N+M-2}

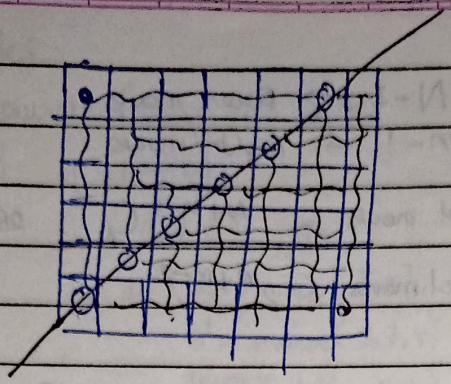


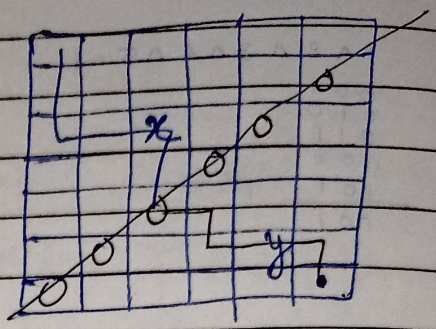
Lets move K steps, if we reach (E) in k steps check the XOR = K or not.

```
#include<bits/stdc++.h>
using namespace std;

int n,m,k;
int arr[100][100];

map<pair<int,int>, map<int,int>> generateDownRightValues;
void generateDownRight(int num, int currx, int curry, int curxor)
{   curxor = curxor ^ arr[currx][curry];
    if(currx>=n || curry>=m) return;
    if(num==0)
    {   generateDownRightValues[makepair(currx,curry)][curxor]+=1;
        // the no. of ways to reach [currx][curry] with xor value curxor
    }
    else
    {   generateDownRight(num-1, currx+1, curry, curxor);
        generateDownRight(num-1, currx, curry+1, curxor);
    }
}
```

If, $0 \le N+M \le 40$

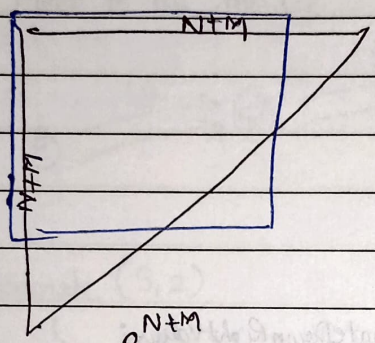


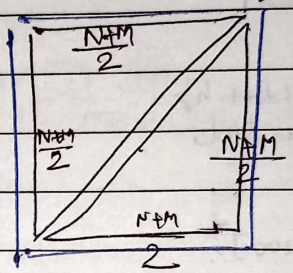




arr[pos](x ∧ y) = K

↳ counted 2 times in x & y.






```
#include<bits/stdc++.h>
using namespace std;

int n,m,k;
int arr[100][100];

map<pair<int,int>, map<int,int>> generateDownRightValues;

void generateDownRight(int num, int curx, int cury, int curXOR)
{   if(curx>=n || cury>=m || curx<0 || cury<0) return;
    curXOR ^= arr[curx][cury];
    if(num==0)
    {   generateDownRightValues[make_pair(curx,cury)][curXOR]+=1;
    }
    else{   generateDownRight(num-1, curx+1, cury, curXOR);
            generateDownRight(num-1, curx, cury+1, curXOR);
    }
}
```

```
map< pair<int, int> , map<int, int>> generateUpleftValues;

void generateUpleft( int num, int currx, int curry, int curXOR)
    { if( currx>=n || currx>=m || currx<0 || curry<0) return;
      curxor ^= arr[currx][curry];
      if(num==0)
        { generateUpleftvalues[{currx, curry}][curXOR]+=1;
        }
      else { generateUpleft(num-1, currx-1, curry, curXOR);
             generateUpleft( num-1, currx, curry-1, curXOR);
           }
    }

int main()
{ cin>>n>>m>>k;
  for( int i=0; i<n; i++)
    { for( int j=0; j<m; j++).
        { cin>> arr[i][j];
        }
    }

int total_move = n-1+m-1;
generateDownRight( totalmoves/2, 0,0,0);
generateUpleft (totalmoves/2, n-1,m-1,0);
int ans=0;
for( auto v: generateDownRightValues)
  { pair<int,int> pos= v.first;
    for( auto u: v.second)
      { int xorval = u.first;
        int numpath = u.second;
        ans+= numpath * generateUpleft[pos][xorval^k^arr[pos.first]
                                              [pos.second]];
      }
  }
}
```