Function calling itself.

Use → Solve the problem using solution of subproblems.

Q → Find sum of first N natural numbers. $\frac{N*(N+1)}{2}$

N = 4 sum(4) = 1 + 2 + 3 + 4 = 10

N = 5 sum(5) = 1 + 2 + 3 + 4 + 5 = 15

sum(4) + 5

sum(N) = sum(N-1) + N

How to use recursion?

1) Define exactly what the function do.
2) Identify how to use subproblems to get the answer.
3) Define base case i.e. smallest subproblem.

```
int sum(N) {
    if (N == 1) return 1
    return sum(N-1) + N
}
```

Function call tracing

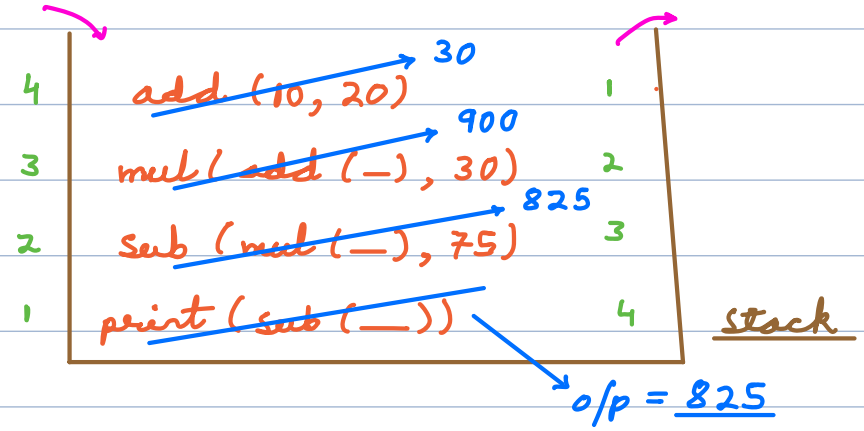
```
int add(x, y) {
    return x+y
}
```

```
int sub(x, y) {
    return x-y
}
```

```
int mul (x, y) {
    return x * y
}
```

x = 10 y = 20

```
print ( sub (mul ( add (x, y), 30), 75))
```



Q → Find factorial of N using recursion.

N = 5 fact (5) = 1 * 2 * 3 * 4 * 5 = 120

fact (4)

1) Define the fuction → int fact (N) {...}
2) Use of subproblem → fact (N) = N * fact (N-1)
3) Base case → fact (0) = 1

```
int fact (N) {
    if (N == 0) return 1
    return N * fact (N-1)
}
```

TC = O (N * 1) = O(N)

SC = O(N)

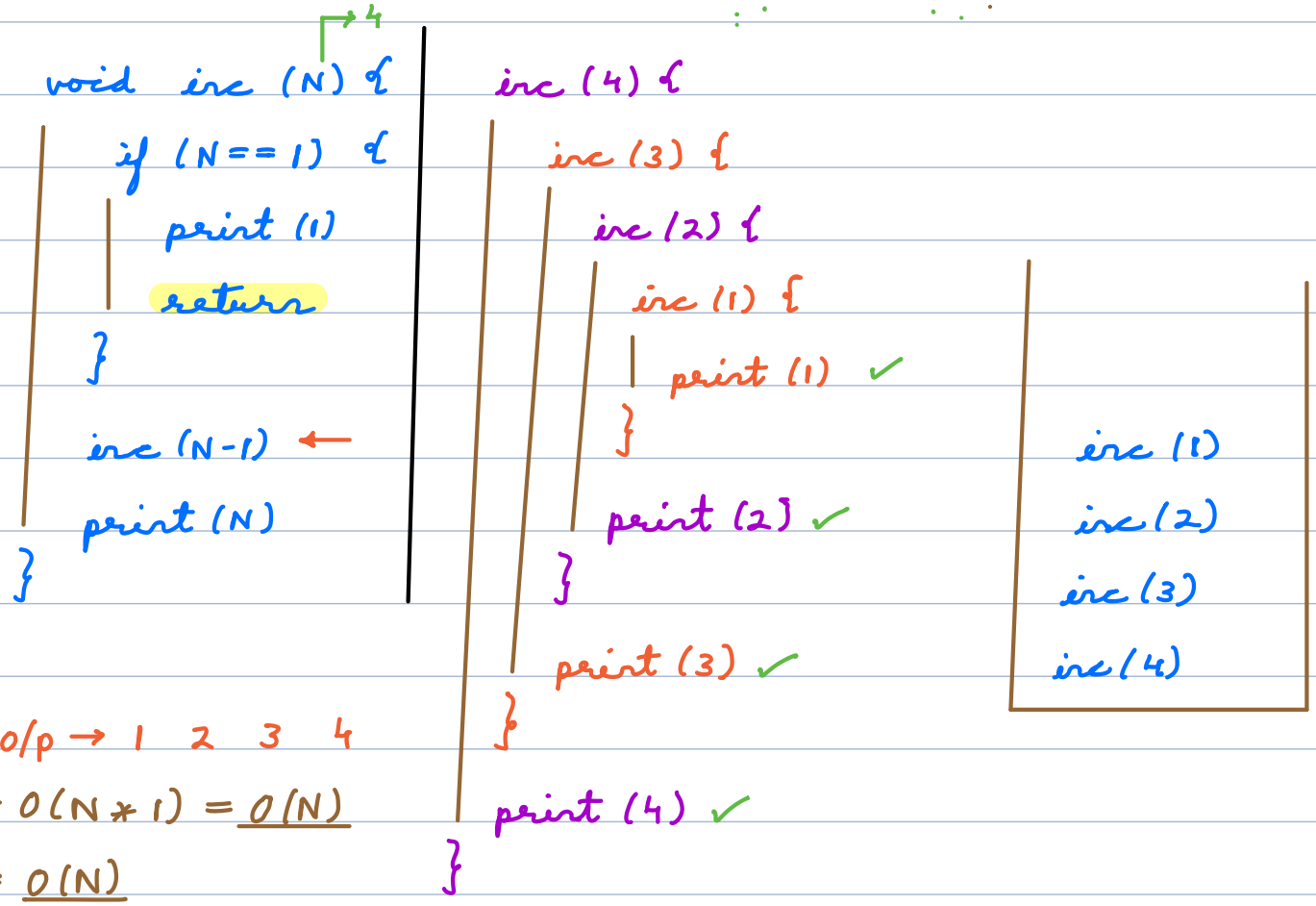
```
fact (3) {                    6
    return 3 * fact (2) {     2
        return 2 * fact (1) { 1
            return 1 * fact (0) { 1
                return 1
            }
        }
    }
}
```

}

Q → Print numbers 1 to N in increasing order.

N = 3 o/p → 1 2 3

1) Define the fuction → void inc (N) {...}
2) Use of subproblem → inc (N) → inc (N-1) print (N)
3) Base case → inc (1) → print (1)



o/p → 1 2 3 4

TC = O(N * 1) = O(N)

SC = O(N)

Q → Print numbers N to 1.

dec (3) → 3 2 1

dec (N) → print (N)

dec (N-1)

```
void dec (N) {
    if (N == 1) {
        print (1)
        return
    }
    print (N)
    dec (N-1)
}
```

```
dec (3) {
    print (3) ✓
    dec (2) {
        print (2) ✓
        dec (1) {
            print (1) ✓
        }
    }
}
```

o/p → 3 2 1

Time Complexity → O ((# function calls) * (time per function call))

Space Complexity → O (Max stack size at any point + space in function call)

Q → Given an integer N, print numbers N to 1 followed by 1 to N using recursion.

N = 3 o/p → 3 2 1 1 2 3

1) Define the fuction → void decInc (N) { ... }

2) Use of subproblem → decInc (N) → print (N)
decInc (N-1)
print (N)

3) Base case →
decInc (0) → return

```
void decInc(N) {
    if (N == 0) return
    print (N)
    decInc (N-1)
    print (N)
}
```

TC = O(N)

SC = O(N)

```
decInc(2) {
    print (2)
    decInc(1) {
        print (1)
        decInc(0) { }
        print (1)
    }
    print (2)
}
```

Fibonacci Numbers

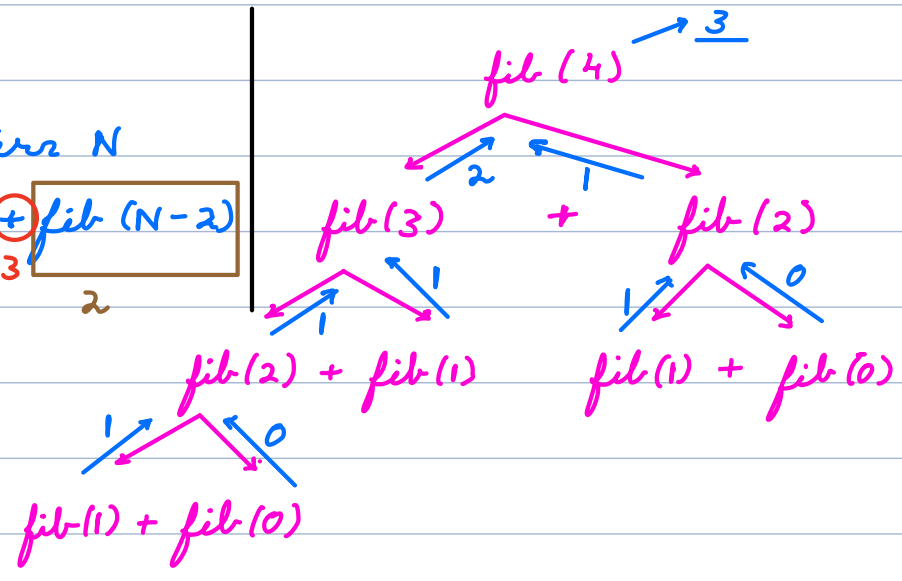
N =	0	1	2	3	4	5	6	7	...
f(N) =	0	1	1	2	3	5	8	13	...

1) Define the fuction → int fib (N) {...}

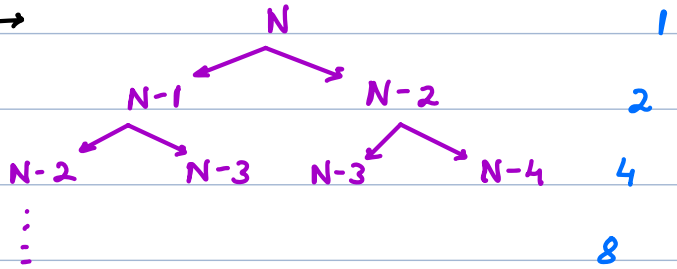
2) Use of subproblem → fib (N) = fib (N-1) + fib (N-2)

3) Base case → if (N <= 1) return N

```
int fib (N) {
    if (N <= 1) return N
    return fib (N-1) + fib (N-2)
}
```



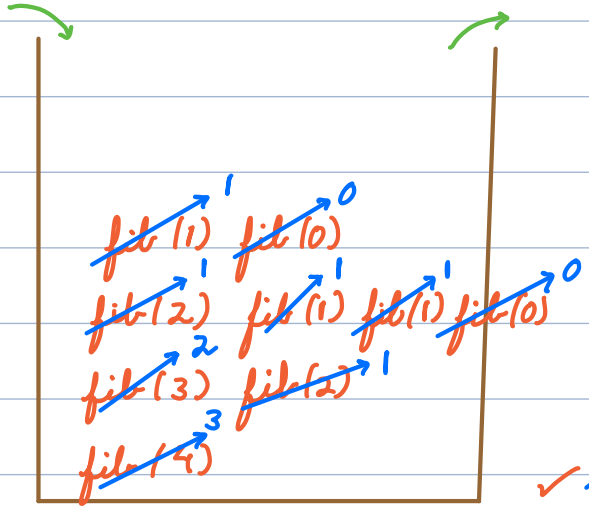
Time Complexity →

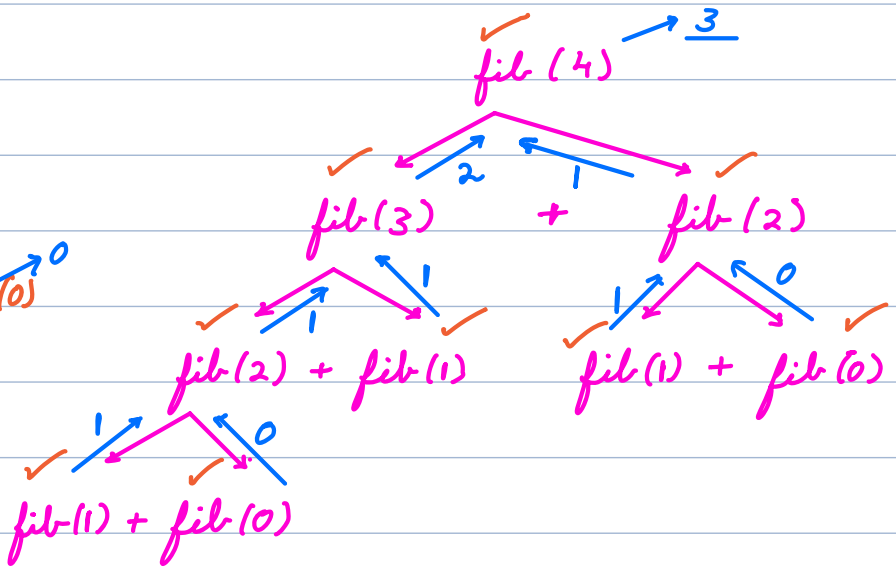


$$\frac{a(r^n - 1)}{r - 1} \longrightarrow \frac{1(2^{N+1} - 1)}{2 - 1} \rightarrow 2^{N+1} - 1$$

$O(2^N)$

Space Complexity →





SC = $O(N)$

> Max size of recursion stack at any point = Height of recursion tree

A = [1 0 1] (0 1 2)

OR → any 1 ⇒ 1

1 → 1
1 0 → 1
1 0 1 → 1
0 → 0
0 1 → 1
1 → 1

5

Ans = (# subarrays) − (# subarrays with all 0's)

7

A = [1 0 1 1 0 0 0 1]

1

6

consecutive 0's = K ⇒ # subarrays = $\frac{K*(K+1)}{2}$

A = [1 3 5 2] 1 — 6

[1 3 5 2 1 2 3 (4) 5 (6)]

Find 2 unique numbers where others are repeating twice.

Q → Find a^b using recursion.

$a = 2 \quad b = 3 \quad 2^3 = 8$

↳ $2 * 2 * 2$

$x^n = x * x * x \ldots$ (n times)

$2^4 = 2 * \underbrace{2 * 2 * 2}_{2^3} * 2$

$a^b = a * a^{b-1}$

```
int pow (a, b) {
    if (b == 0) return 1      // a^0 = 1
    return a * pow (a, b-1)
}
```

TC = O(b) SC = O(b)

$2^6 = 2^3 * 2^3$

$2^7 = 2 * 2^3 * 2^3$

$x^n \Rightarrow x^{n/2} * x^{n/2}$, n → even

↳ $x * x^{n/2} * x^{n/2}$, n → odd

2^{10} → $x^n = x * x^{n-1}$ ⇒ 10 steps

$2^{10} = 2^5 * 2^5 = 1024$ (32, 32)

$2^5 = 2 * 2^2 * 2^2 = 32$ (4, 4)

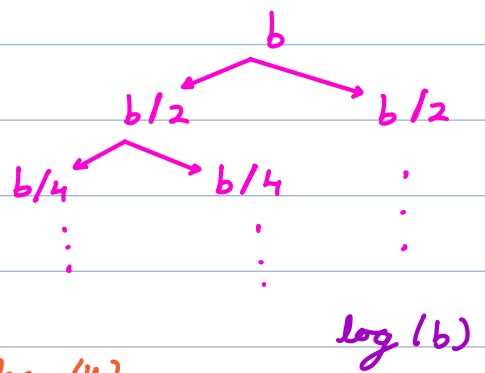
$2^2 = 2^1 * 2^1 = 4$ (2, 2)

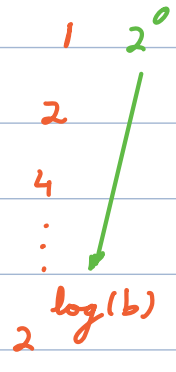
$2^1 = 2 * 2^0 * 2^0 = 2$ (1, 1)

4 steps

```
int pow (a, b) {
    if (b == 0) return 1
    if (b % 2 == 0)
```

```
    return pow(a, b/2) * pow(a, b/2)
  else
    return a * pow(a, b/2) * pow(a, b/2)
}
```





$[0 \quad \log(b)]$

$$\frac{1\,(2^{\log b + 1} - 1)}{2 - 1}$$

$$= 2 * 2^{\log b} - 1$$

$$= 2 * b - 1$$

$TC = O(b)$

$x^{\log_x(y)} = y$

$2^{\log_2(8)} = 8$

$SC = O(\log(b))$ ✓

```
int pow(a, b) {                    // Fast Power  v. imp.
  if (b == 0) return 1
  p = pow(a, b/2)
  if (b % 2 == 0)   return p * p
  else  return a * p * p
}
```

HW → Try iterative code.

$TC = O(\log(b))$

$SC = O(\log(b))$

```
pow(2, 9) {   // 512
  p = pow(2, 4) {   // 16
    p = pow(2, 2) {   // 4
      p = pow(2, 1) {   // 2
        p = pow(2, 0) {   // 1
          return 1
        }
```

return 2*1*1 = 2
}
return 2*2 = 4
}
return 4*4 = 16
}
return 2*16*16 = 512
}

$N \to N/2 \to N/4 \to \dots \frac{N}{2^k} = 1$

$\Rightarrow N = 2^k$

$= k = \log(N)$

Q → Print array elements using recursion.

i
0 1 2 3
A = [3 5 1 8] o/p → 3 5 1 8

0

```
void printArray (A[], i) {
    if (i == A.length) return
    print (A[i])
    printArray (A, i+1)
}
```

TC = O(A.length)

SC = O(A.length) → O(1)

Tail Recursion → If recursive call is last step (some languages) then function calls are not stored.

Q → Find max element of array using recursion.

i
0 1 2 3
A = [3 5 1 8] o/p → 8

```
int maxArray(A[], i) {        // i → 0
    if (i == A.length) return Int_Min
    return max(A[i], maxArray(A, i+1))
}
```

TC = O(A.length)

SC = O(A.length)

Q → check if the given string is palindrome.

str = reverse(str)

Eg → racecar Ans = true

race Ans = false

```
boolean isPalindrome(s, l, r) {        // l → 0, r → N-1
    if (l >= r) return true
    if (s[l] != s[r]) return false
    return isPalindrome(s, l+1, r-1)
}
```

TC = O(N) SC = O(N) → O(1)

(tail recursion)

Tower of Hanoi

Given 3 towers (A, B & C) & N disks of different sizes.

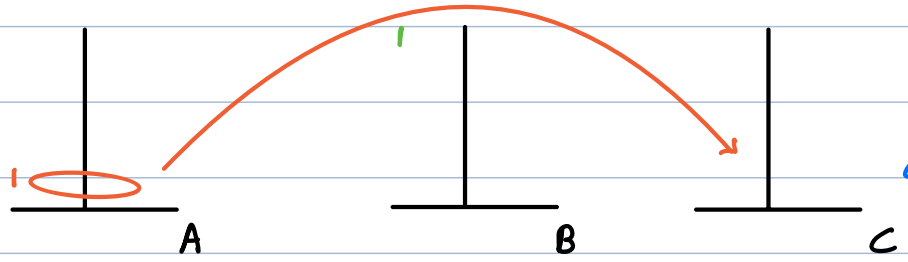
Move all disks from A to C.

Constraints → 1) In step only 1 disk can be moved.

2) large disk cannot be placed over small disk.

Goal → Use minimum steps.

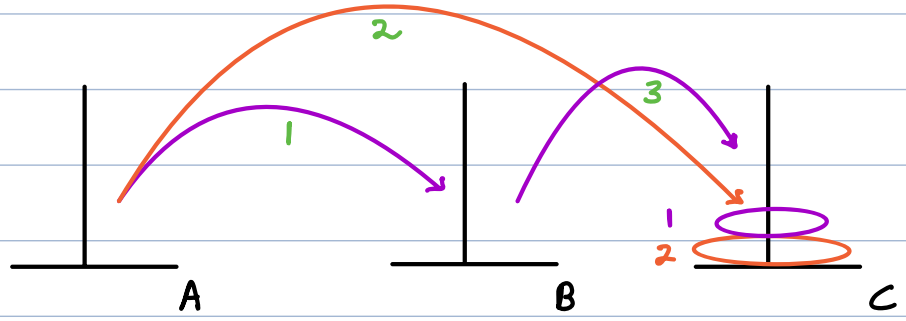
N = 1



steps = 1

o/p = 1, A → C

N = 2



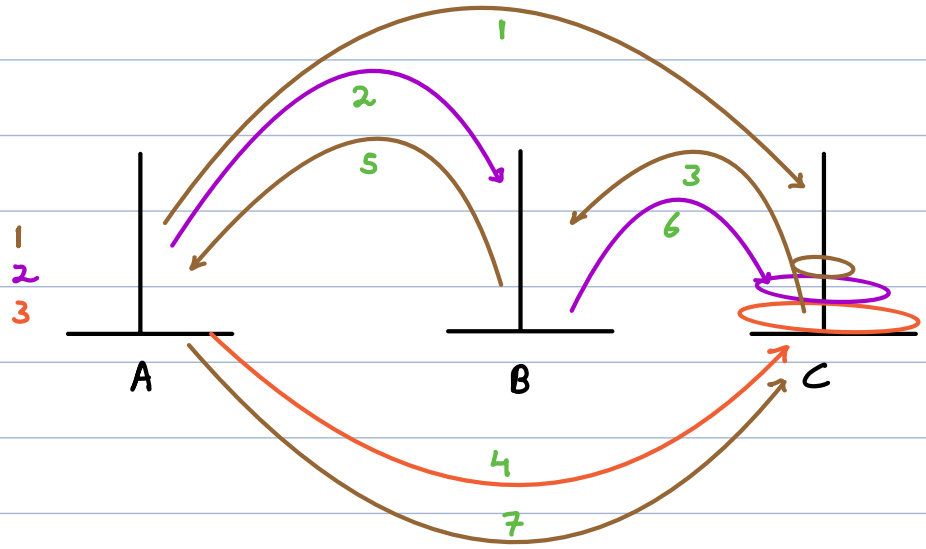
steps = 3

o/p = 1, A → B

2, A → C

1, B → C

N = 3



steps = 7

o/p → 1, A → C

2, A → B

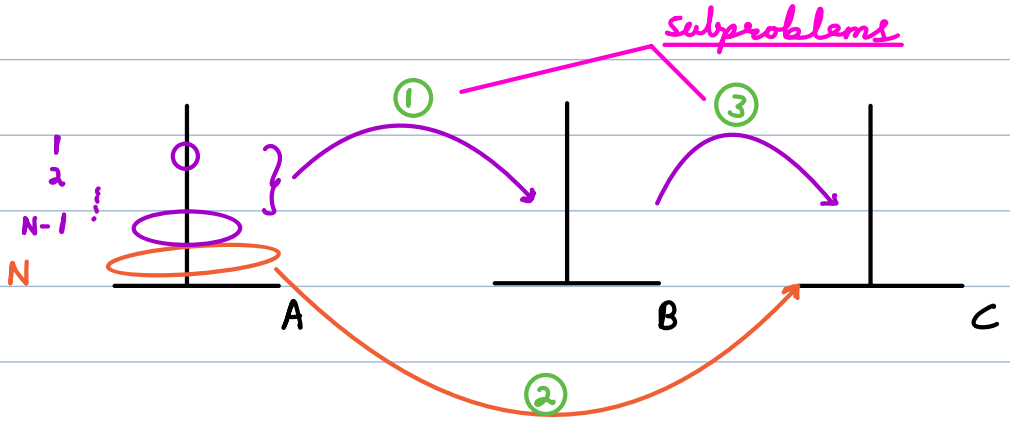
1, C → B

3, A → C

1, B → A

2, B → C

1, A → C

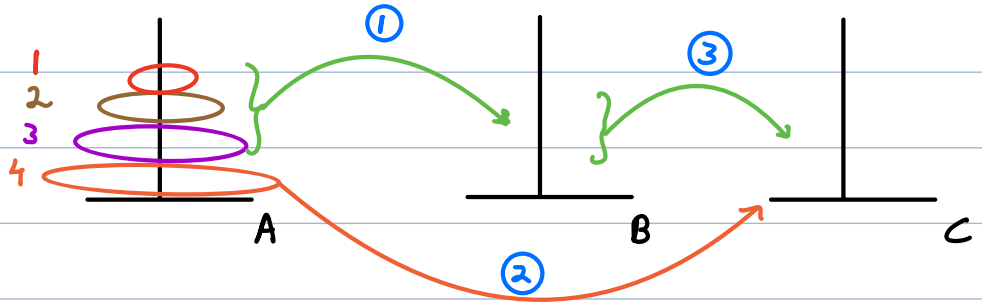


Move (N-1) disks from A → B

Move N^{th} disk from A → C

Move (N-1) disks from B → C

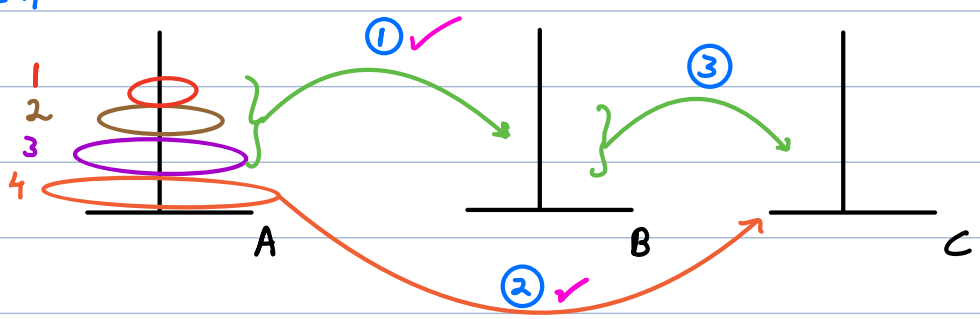
N = 4



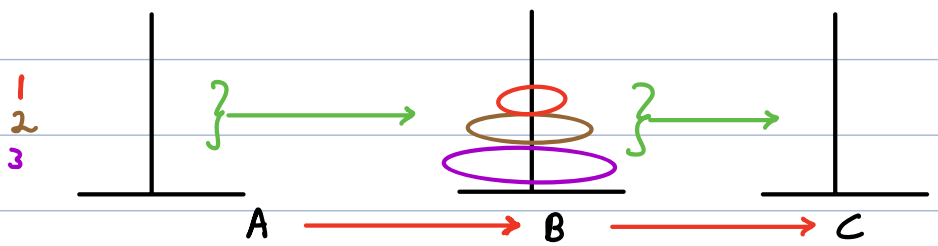

```
void toh (N, T1, T2, T3) {        T1 → T3
    if (N == 1) {
        print (N, T1 → T3)
        return
    }
    toh (N-1, T1, T3, T2)   // A C B  (A → B)
    print (N, T1 → T3)      // N, A → C
    toh (N-1, T2, T1, T3)   // B A C  (B → C)
}
```

(A B C)

N = 4



N = 3



steps = 7 # steps = 7

N → # steps = $2^N - 1$

TC = $O(2^N)$

SC = $O(N)$

N	# steps		
1	1		$2^1 - 1$
2	3	(1 + 1 + 1)	$2^2 - 1$
3	7	(3 + 1 + 3)	$2^3 - 1$
4	15	(7 + 1 + 7)	$2^4 - 1$
⋮	⋮		

SCALER

Backtracking

TABLE OF CONTENTS

Output based questions

```
int magicfun (int N){
    if (N==0) {return 0}
    else{
      return magicfun(N/2) * 10 + (N%2);
    }
}
```

10

mf (10) {
  return 10%2 + 10 * mf(5) {
    return 5%2 + 10 * mf(2) {
      return 2%2 + 10 * mf(1) {
        return 1%2 + 10 * mf(0) {
          return 0
        }
      }
    }
  }
}

1010

0 + 10 * 101 = 1010
1 + 10 * 10 = 101
0 + 10 * 1 = 10
1 + 10 * 0 = 1

```
void solve (N) {
    if (N == 0) return
    print (N)
    solve (N-1)
    print (N)
}
```

2

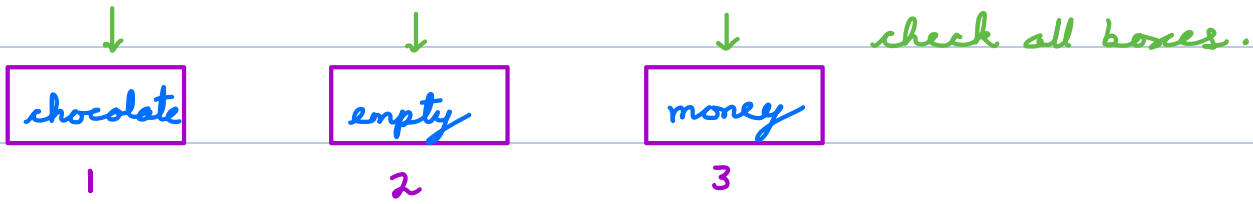
o/p → 2 1 1 2

solve (2) {
  print (2) ✓
  solve (1) {
    print (1) ✓
    solve (0) { return }
    print (1) ✓
  }
  print (2) ✓
}

Backtracking

Try all possibilities using recursion.

closed box



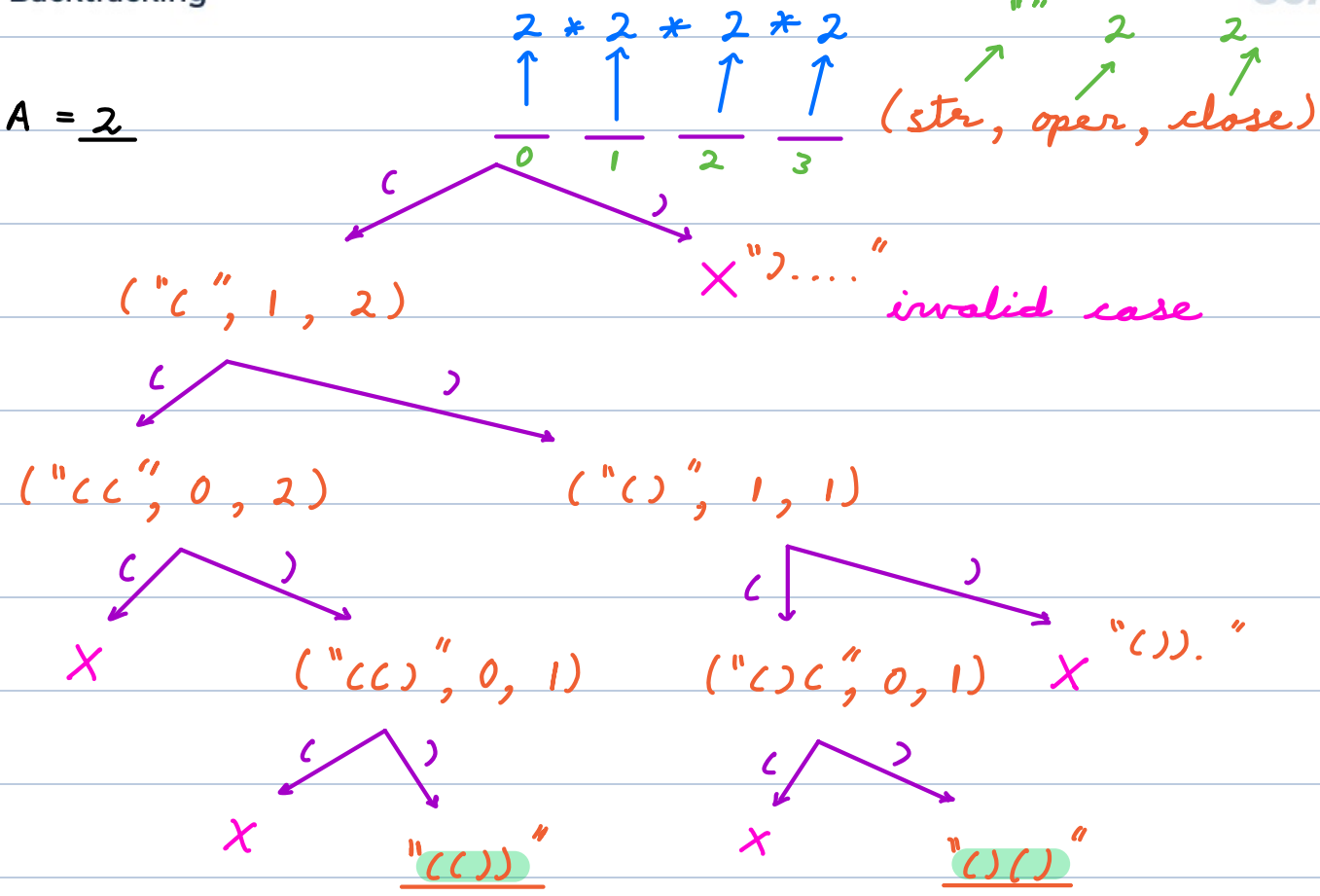
< *Question* > : Given an integer A pairs of parentheses, write a function to generate all combinations of well-formed parentheses of length 2*A.

A = 1 ()

A = 2 ()() , (()) ()()(X)

A = 3 ((())) , (()()) , (())() , ()(()) , ()()()

A = 2



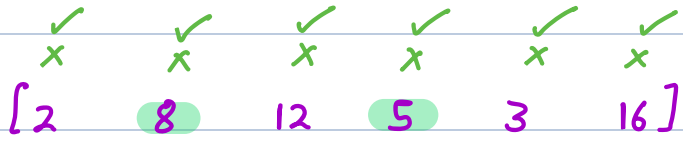

```
void solve(str, oper, close) {
    if (oper == 0 && close == 0) {
        print(str)
        return
    }
    if (oper > 0)   solve(str + '(', oper - 1, close)
    if (close > oper)  solve(str + ')', oper, close - 1)
}
```

(str: "", oper: A, close: A)

TC < $O(2^N)$ SC = $O(N)$

(small constraints)

Definition of Subset and Subsequences



subset = 2^N

same subsets

[8 5] → Subset → May or maynot be continuous.

[2 5 16] (Any possible selection)

[12]

[ ]

[5 8]

X

Subsequence → Subset where order of selected elements is maintained.

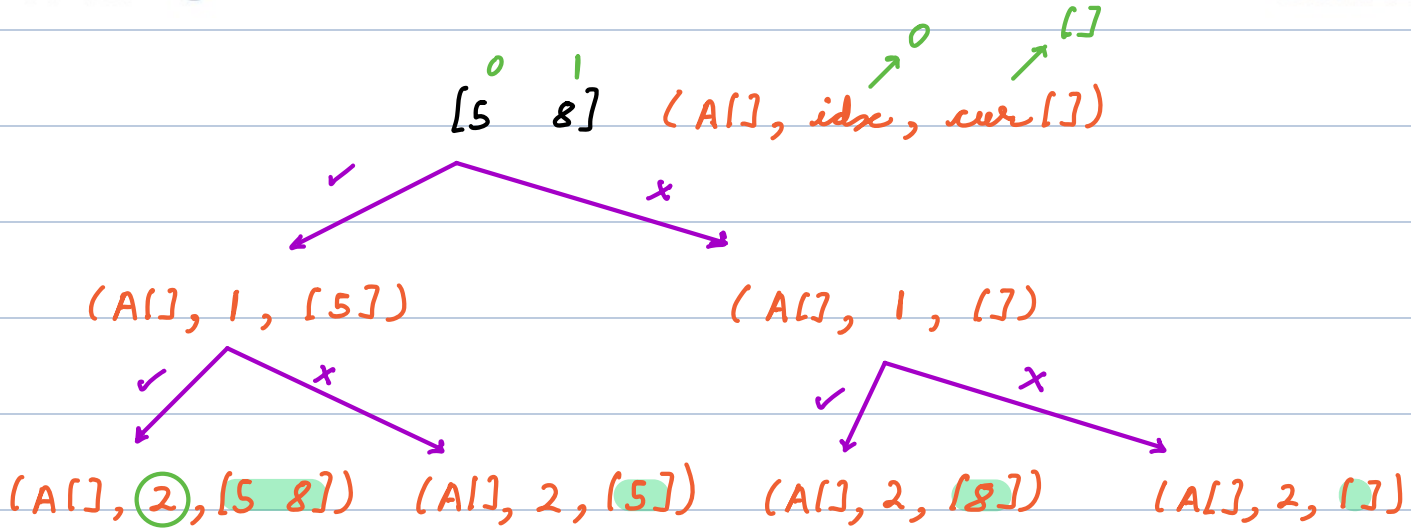
Q → Given an array of distinct integers, print all subset of the array.

A = [5 8] [ ] [5] [8] [5 8]

A = [1 2 3] [ ] [1] [2] [3]

[1 2] [1 3] [2 3]

[1 2 3]




```
Arraylist < Arraylist <Integer> > ans ;
void subset (A[], idx, cur[]) {
    if ( idx == A.length ) {   // basecase
        copy = new Arraylist (cur)
        ans . add (copy)
        return
    }
    cur . add (A[idx])          // do
    subset (A, idx+1, cur)      // select
    cur . removelast ()         // undo
    subset (A, idx+1, cur)      // reject
}
```

TC = $O(2^N)$ SC = $O(N)$

Data in every subset to be sorted → sort i/p.

H.W → Return ans in sorted order.

Problem

A popular Fitness app **FitBit**, is looking to make workouts more exciting for its users. The app has noticed that people get bored when the same exercises are shown in the same order every time they work out. To mix things up, **FitBit** wants to show all the different ways the exercises can be arranged so that each workout feels new.

Your challenge is to write a program for **FitBit** that takes a string **A** as input, where each character in the string represents a different exercise. Your program should then find and display all possible arrangements of these exercises.

Example:
A = Push-ups
B = Squats
C = Burpees
D = Planks

Then different ways of doing the exercise includes -
ABCD
ACBD
ADBC
ADCB
.

a b c

a b c
a c b
b a c
b c a
c a b
c b a

a b c d

↓ ↓ ↓ ↓

4 * 3 * 2 * 1 = 24 (4!)

Permutations

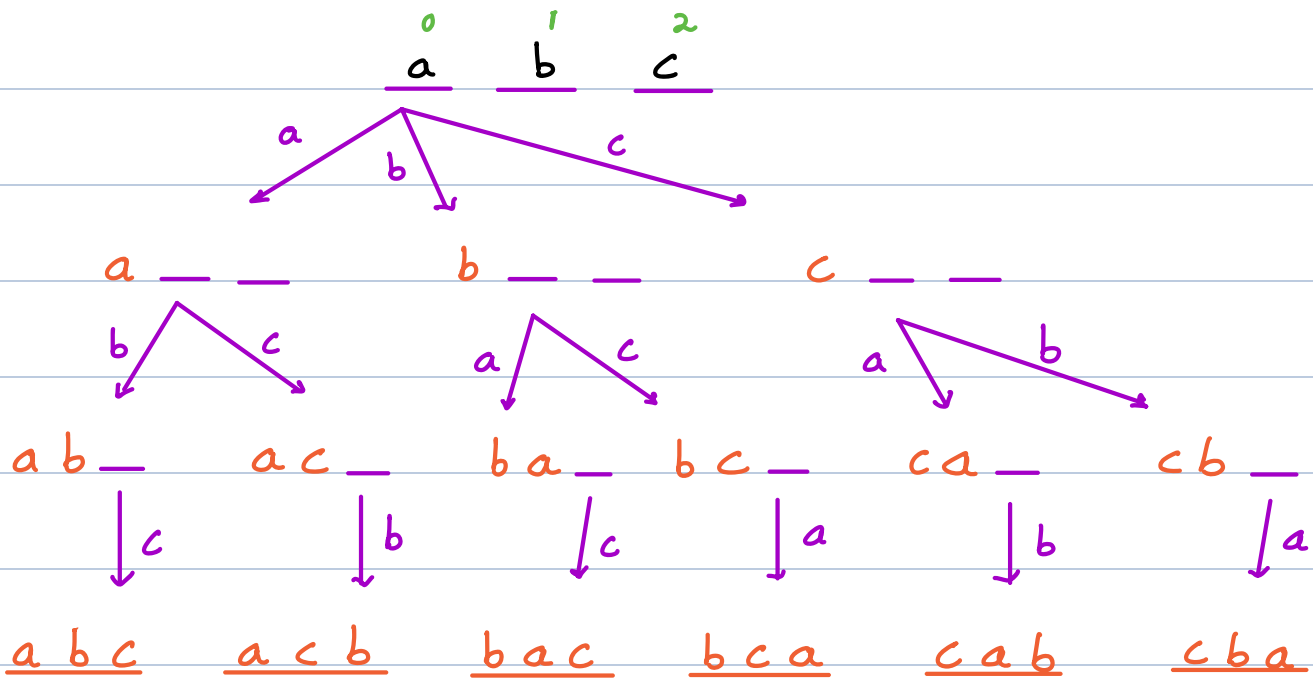
< *Question* > : Given a character array with distinct elements, print all permutations of it without modifying it.

A → [a b c]

a b c a c b

b a c b c a

c a b c b a



```
void permutation(A[], idx, cur[], vst[]) {        // idx: 0, cur: empty[], vst: ∀i, false
    if (idx == A.length) {                        // Base Case
        print(cur)
        return
    }
    for i → 0 to (N-1) {                          // All Possibilities
        if (!vst[i]) {                            // Valid Possibility
            vst[i] = true                         // Do
            cur[idx] = A[i]
            permutation(A, idx+1, cur, vst)       // Recursion
            vst[i] = false                        // Undo
        }
    }
}
```

TC = O(N!) SC = O(N)

Backtracking 2

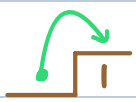
Agenda

1. Print paths in staircase problem
2. Print all paths from source to destination
3. Shortest path in a matrix with huddles

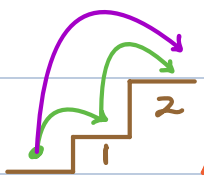
Q → climb N stairs s.t. in each step only 1 or 2 stairs can be climbed. Find # distinct ways to do the task & return all possibilities in lexicographical order.

N = 1

Ans = 1

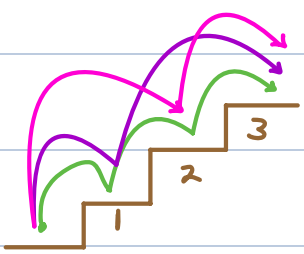


N = 2



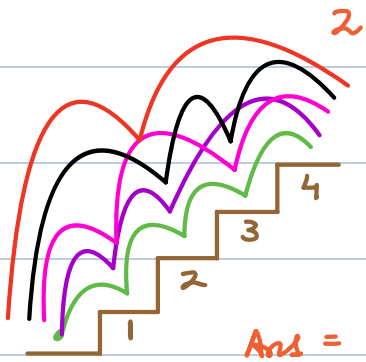
Ans = 1, 1

N = 3

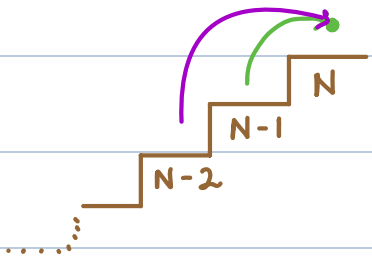


Ans = 1, 1, 1
1, 2
2, 1

N = 4



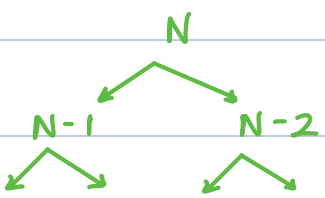
Ans = 1, 1, 1, 1
1, 1, 2
1, 2, 1
2, 1, 1
2, 2



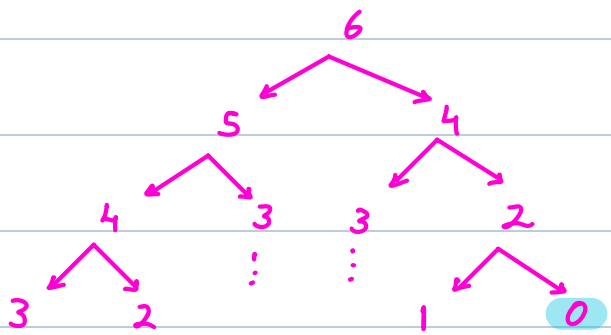
empty list

```
void paths (N, cur[]) {
    if (N == 0) {          // Base case
        print (cur)
        return
    }
    if (N >= 1) {
        cur.add (1)        // Do
        paths (N-1, cur)   // Recursion
        cur.removelast()   // Undo
    }
    if (N >=2) {
        cur.add (2)        // Do
        paths (N-2, cur)   // Recursion
        cur.removelast()   // Undo
    }
}
```

```
void paths (N, cur[]) {
    if (N < 0) return
    if (N == 0) {          // Base case
        print (cur)
        return
    }
    cur.add (1)            // Do
    paths (N-1, cur)       // Recursion
    cur.removelast()       // Undo
    cur.add (2)            // Do
    paths (N-2, cur)       // Recursion
    cur.removelast()       // Undo
}
```



TC < $O(2^N)$ SC = $O(N)$



Q → Given a N * M board.

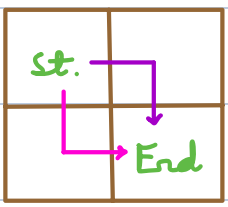
Print all the possible paths from top left to bottom right corner.

Move → Down ('D')

Right ('R')

Print paths in lexicographical order.

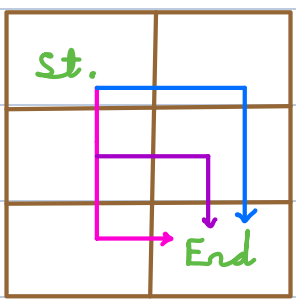
'D' < 'R'



2 * 2

Ans = D R

R D



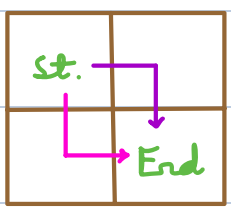
3 * 2

Ans = D D R

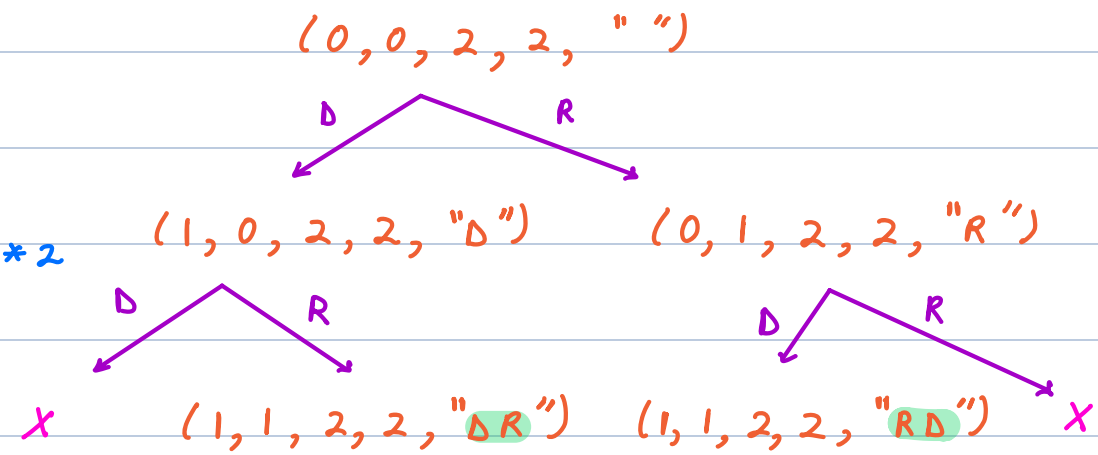
D R D

R D D

```
void maze(r, c, N, M, cur) {
    if (r >= N || c >= M) return
    if (r == N-1 && c == M-1) {
        print(cur)
        return
    }
    maze(r+1, c, N, M, cur + 'D')
    maze(r, c+1, N, M, cur + 'R')
}
```



2*2



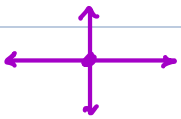
TC < $O(2^{(N+M)})$

SC = $O(N+M)$

Q → Given a N * M board.

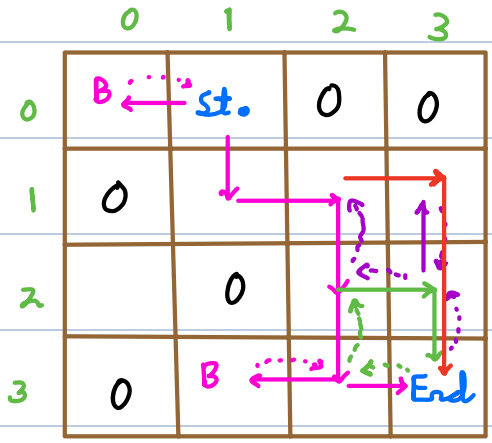
Find the length of **shortest path** from source to destination. If not possible ans = -1.

Move →

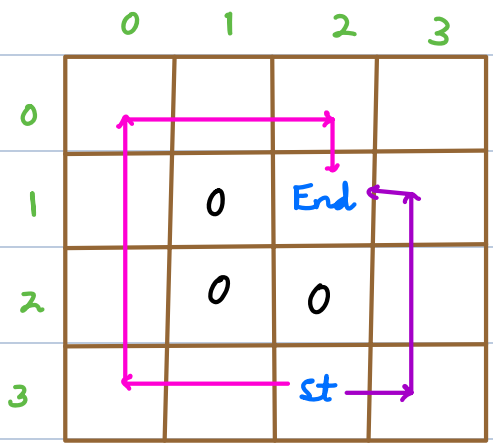


A[i][j] → 0 Blocked cell

→ 1 Empty cell



Ans = 5



Ans = 4

(r-1, c)

(r, c-1) ← (r, c) → (r, c+1)

(r+1, c)

↓ ↑ ← →

dx = [1 -1 0 0]

dy = [0 0 -1 1]

// N, M

ans = ∞

i/p — source — mark vst before calling — 0 — ∀i,j = false

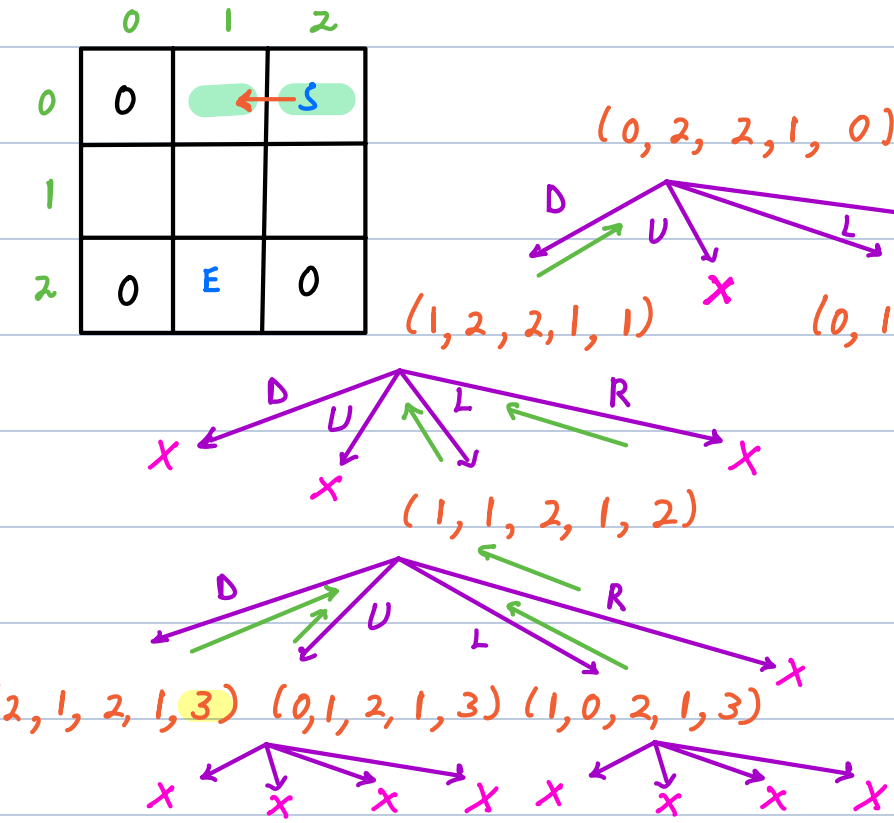
```
void distance ( A[][], r, c, dr, dc, dist, vst[][]) {

    if ( r == dr && c == dc ) {
        ans = min (ans, dist)
        return
    }

    for i → 0 to 3 {
        nr = r + dx[i]
        nc = c + dy[i]
        if ( nr < 0 || nr >= N ||
           nc < 0 || nc >= M || A[nr][nc] == 0 ||
           vst[nr][nc]) continue
        vst[nr][nc] = true
        distance ( A, nr, nc, dr, dc, dist+1, vst)
        vst[nr][nc] = false
    }

}
```

TC < $O(4^{(N*M)})$ SC < $O(N*M)$


0 1 2
0
1
2
0
S
0
E
0
(0, 2, 2, 1, 0)
D
U
L
R
X
(1, 2, 2, 1, 1)
(0, 1, 2, 1, 1)
(1, 1, 2, 1, 2)
(2, 1, 2, 1, 3)
(0, 1, 2, 1, 3)
(1, 0, 2, 1, 3)