29/04/24

Recursion - I

Recursion → A function that calls itself.

E.g → $f(f(x))$

If $f(x) = x^x$, then $f(f(x)) = f(x^2)$.

Let's say $x = 2$

$f(x) = 2^2 = 4$.

$f(f(x)) = f(4) = 4^4 = 16$.

Q) Print numbers from 5 to 1.

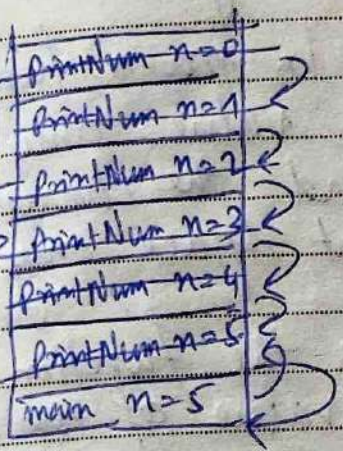
Iteration

```
for (int i=5; i>0; i--)
{
    Sys.o (i);
}
```

Recursion

```
Public Static Void PrintNum(int n)
{
    if (n==20)        // base case
        return;
    S.o.Pln(n);       // Print
    PrintNum(n-1);    // recursion.
}
```

b) What happens in memory?



1. When main function calls PrintNum and n=5 is passed, a space is created in the stack.

2. With PrintNum calling itself again & again, everytime a new memory is created with the new value of n. n=5, 4, 3, ... and so on.

3. When the base condition is found it returns the control to the prev. Stack memory and then to the next and so on. In this example, it checks whether it has to execute any statement after PrintNum(n-1) when n=1, if not then it gives the control to n=2, ... similarly n=3, n=4, and so on.

Every time, the control is given to the previous Stack memory, the memory is deleted. In this manner, it finally gives the control to n=5 of main function.

✓ Memory affection while performing recursion :-

✓ When we define a variable in main function and increases / decreases it's value, only 1 memory space is occupied and the change of value takes place in that space only.

✓ Whereas in recursion, every the function is called with the different value of the same parameter, a new memory space is occupied. So, in recursion it occupies a lot of space.

✓ If the base condition is never satisfied, such as in case of infinity (or infinite loop), too much memory space is occupied and the program throws an error. Usually, then we encounter the problem of stack overflow.

Q) using recursion, print sum of first n natural numbers.

```
→ class Recursion
  {
    public static void printSum(int i, int n, int sum)
    {
      if (i == n)
      {
        sum += i;
        System.out.println(sum);
        return;
      }
      sum += i;
      printSum(i+1, n, sum);
    }
```

```
Public static void main (String[] args)
{
    printSum (1, 5, 0);
}
```

Output : 15

Memory :-

Print 15

~~P.S, i=5, n=5, S=15~~
~~P.S, i=4, n=5, S=10~~
~~P.S, i=3, n=5, S=6~~
~~P.S, i=2, n=5, S=3~~
~~P.S, i=1, n=5, S=1~~
main i=1 n=5 Sum=0

After printing the output (15), it gives the control to the previous step, and deleting each memory every time, until the control finally comes to the main.

If a slight change is made in the code and we print 'i' after the recursive function call, then the following takes place =

```
sum += i;
printSum (i+1, n, sum);
S.O.Pln (i);
```

Output :-

15
4
3
2
1

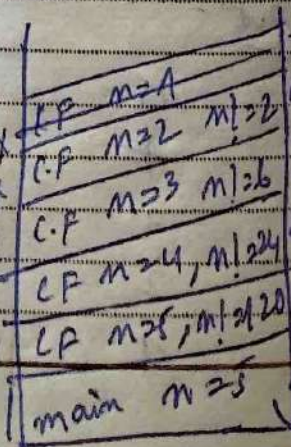
Because, when the control is given to the previous stack memory, it print the value of 'i' and then given the control to the next previous stack memory.

8) Print factorial of number n.

$n! = n \times (n-1) \times (n-2) \times (n-3) \times \dots \times 1$.

```
→ Public class Recursion1
{
    Public Static int calcfactorial (int n)
    {
        if(n==1 || n==0)
        {
            return 1;
        }
        int fact_nm1 = calcfactorial (n-1);
        int fact_n = n * fact_nm1;
        return fact_n;
    }
    Public Static void main (String args[])
    {
        int n=5;
        int ans = calcfactorial (n);
        System.out.println (ans);
    }
}.
```

Output: 120.



8) Print the fibonacci series till nth term (Recurssion).

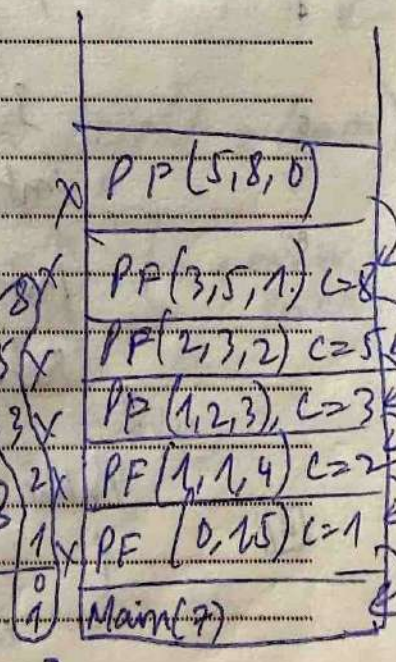
(i) given :- 1st term (a) = 0, 2nd term (b) = 1.

(ii) work :- create the next term (c = a+b).

(iii) base case :- nth term.

⇒ class Recurssion 2
{
Public Static void PrintFib (int a, int b, int n)
{
int c = a+b;
System.out.println(c);
PrintFib(b, c, n-1);
}
Public Static void main (String[] args)
{ int a = 0, b = 1;
System.out.println(a);
System.out.println(b);
int n = 7;
PrintFib(a, b, n-2);
}
}

output :- 0 1 1 2 3 5 8



8) Using Recursion, Print x^n (Stack height = n)

① input → x, n

② work → $x^n = x * x^{n-1}$

③ basecase $x^0 = 1$

$x = 0 \rightarrow 0$

→ Class Recursion 3

{

Public Static int calcPower (int x, int n)

{

if (n == 0)

{

Return 1;

}

if (x == 0) {

Return 0;

}

int xpownm1 = calcPower (x, n-1);

int xpown = x * xpownm1;

} Return xpown;

Public static void main (String args[])

{

int x = 2, n = 5;

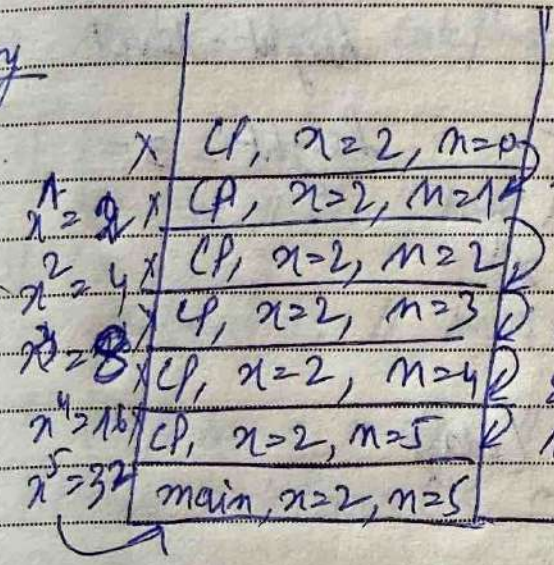
int ans = calcPower (x, n);

S.o.pln (ans);

}}

Output :- If x=2, n=5, ans = 32

Memory



Here, level = 6, but n = 5.
But in asymptotic notation, we can ignore the difference between the stack height & level.

for. eg, if level = 100000, n = 99999, then the difference is negligible and we can consider both as almost equal. Keeping in mind that concept only, we have solved the problem.

8) using recursion, print x^n (stack height = log n)

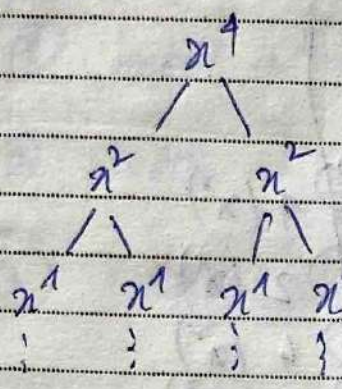
→ x^n (n → odd Power)

$x^{n/2}$ $x^{n/2}$
$1 * x^{n/2} * x^{n/2}$

x^n (n → even Power)

$x^{n/2}$ $x^{n/2}$
$x^{n/2} * x^{n/2}$

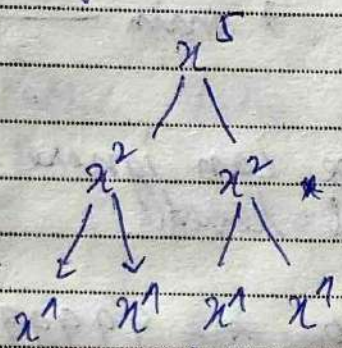
Let's say, $n = 4$. (even)



height = level

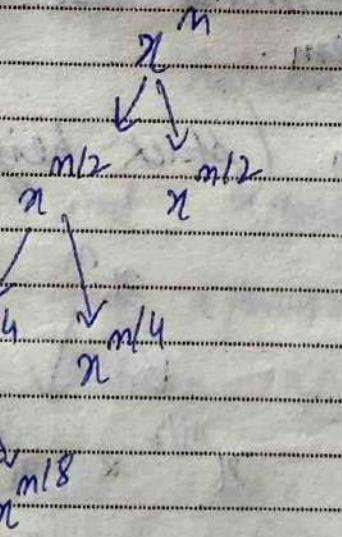
height = 3.

Let's say, $n = 5$ (odd)



height = 3

Let's generalise it,



$\rightarrow q = 0,\ 2^q = 1,\ \frac{n}{1}$

$\rightarrow q = 1,\ 2^q = 2,\ \frac{n}{2}$

$\rightarrow q = 2,\ 2^q = 4,\ \frac{n}{4}$

$\rightarrow q = 3,\ 2^q = 8,\ \frac{n}{8}$.

The process ends where $\frac{n}{2^q} = 1$.

$\therefore n = 2^q$

$\Rightarrow \boxed{q = \log_2 n}$, where q = height of the stack.

Class Recursion4

```
{
Public static int calcPower (int x, int n)
{
    if (n == 0)
    return 1; // base case 1
    if (x == 0)
      return 0; // base case 2

    if (n % 2 == 0) // if n is even
    return calcPower (x, n/2) * calcPower(x, n/2);

    else // if n is odd
    return calcPower (x, n/2) * calcPower (x, n/2) * x;

}
Public static void main (String args[])
{
    int x = 2, n = 5;

    int ans = calcPower (x, n);
    S.o.pln (ans);
}
}
```

output :- 32

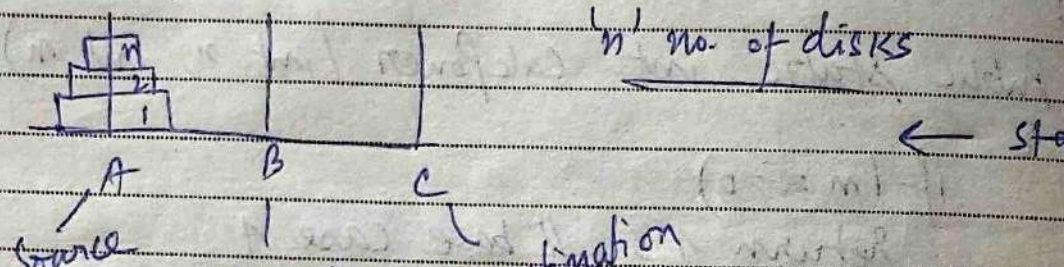
* note → In this case, the height of the stack is reduced to log n, so memory is less consumed and the code is optimized.

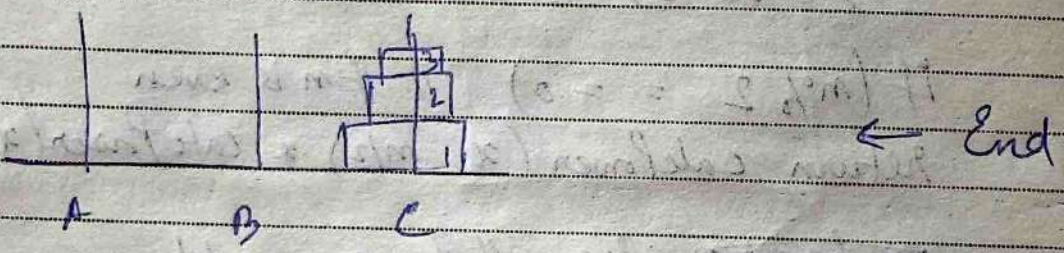
29/4/24

NOTES

Recursion - II

Q) Tower of Hanoi





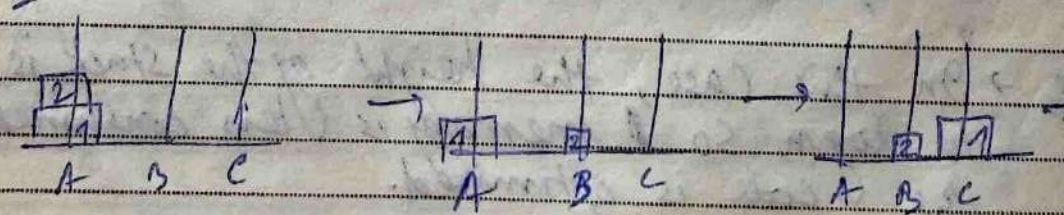
Rules :-

1. only one disk transferred in 1 step.
2. Smaller disks are always kept on top of larger disk.

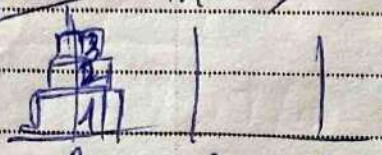
Let's see Some Examples :-

Example 1 n = 1

Example 2 n = 2



Example 3 $n = 3$

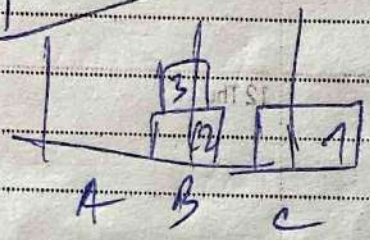


Steps

1: Disk 3 → A → C
2: Disk 2 → A → B
3: Disk 3 → C → B
4: Disk 1 → A → C
5: Disk 3 → B → A
6: Disk 2 → B → C
7: Disk 3 → A → C

Steps = 7

After step 4



After step 4, we can follow the example 2 by interchanging the Source and the helper.

Generalization

Step 1:- Let's say there are 'n' disks. 1st step is to transfer (n-1) disks from S → H.

Step 2:- Transfer 1 disk (largest) S → D.

Step 3:- Transfer (n-1) disks from H → D by interchanging source and helper. So, in this step initial helper will be treated as Source and Source shall be used as helper.

2022 JUNE

May 19th Wk • 122-243

02 Monday

code

```
class Recursion 5
{
Public static void towerofHanoi (int n, String src,
                                 String helper, String dest)
{
    if (n == 1)
    {
        S.o.pln ("transfer disk" + n + "from" + src + "to" + dest);
        return;
    }
towerofHanoi (n-1, src, dest, helper);
S.o.pln ("transfer disk" + n + "from" + src + "to" + dest);
towerofHanoi (n-1, helper, src, dest);
}

Public static void main (String args[])
{
int n=3;
towerofHanoi (n, "S", "H", "D");
}
}
```

Time complexity = $O(2^n - 1) \approx O(2^n)$

Mathematical proof :-

$T(n-1) = 2T(n-2) + 1$ —(i)

$T(n-2) = 2T(n-3) + 1$ —(ii)

$T(1) = 1$

Now, $T(n) = 2 \cdot T(n-1) + 1$

Substituting value of eqn (i),

$T(n) = 2[2 \cdot T(n-2) + 1] + 1$

$\Rightarrow T(n) = 4T(n-2) + 2 + 1$

Substituting value of eqn (ii),

$T(n) = 4(2 \cdot T(n-3) + 1) + 2 + 1$

$\Rightarrow T(n) = 8T(n-3) + 4 + 2 + 1$.

we need to do it until we find $T(n-x) = 1$.

So, we can write,

$T(n) = 2^{n-1} \cdot T(n-x) + 2^{n-2} + 2^{n-3} + \cdots$

$\Rightarrow T(n) = 2^{n-1} \cdot T(n-x) + 2^{n-1}$

So, $T(n) = 2^{n-1} + 2^{n-1} = 2^n$

[ignore 1 for larger 'n' values].

$\therefore$ Time complexity, $O(2^{n-1}) \approx O(2^n)$.

Q) Print a String in Reverse,

"abcd" → "dcba"

index (idx) = str. length - 1.

```
class Recursion 6
{
Public static void printRev (String str, int idx)
{
  if (idx == 0)
  {
    System.out.println (str.charAt (idx));
    return;
  }
  System.out.println (str.charAt (idx));
  printRev (str, idx-1);
}

Public static void main (String args[]) {
  String str = "abcd";
  printRev (str, str.length() -1);
}
}
```

Time Complexity :- $O(n)$, where n = String length.

8) Find the 1st and last occurence of an element in string.

E.g → "abaacdaefaah"

start index = 0

end index = 10

```
→ class Recursion7
  {
      Public static int first = -1;
      Public static int last = -1;

      Public static void findOccurance(String str, int idx,
                                       char element)
      {
          if (idx == str.length())
          {
              System.out.println(first);
              System.out.println(last);
              return;
          }
          char currChar = str.charAt(idx);
          if (currChar == element)
          {
              if (first == -1)
                  first = idx;
              else
                  last = idx;
          }
          findOccurance(str, idx+1, element);
      }
```

```
Public static void main ( String args[])
{
    String str = "abaacdaefaah";
    findOccurence (str, 0, 'a');
}
```

Time complexity → O(n), where n = String length. Here, we have traversed the total length of the String only once, hence it's O(n)

8) check if an array is sorted (strictly increasing)

```
class Recursion8
{
Public static boolean isSorted (int arr[], int idx)
{
    if ( idx == arr.length - 1)
    {
        return true;
    }
    if (arr[idx] < arr[idx + 1])
    {
        return isSorted (arr, idx+1);
    }
    else
    {
        return false;
    }}
Public static void main (String args[])
{
    int arr[] = {1, 3, 5};
    System.out.println (isSorted (arr, 0));
}
}
```

(I/P → {1,3,5}, output: true)

* time complexity → O(n) → array length

Q) Move all 'x' to the end of the string

E.g → i/p = "axbcxxd", o/p → "abcdxxx"

```
class Recursion9
{
Public static void moveAllx (String str, int idx, int count,
                             String newString)
{
    if (idx == str.length)
    { for (int i=0; i<count; i++) { newString += 'x'; }
    System.out.println (newString);
    return;
    }
    char currChar = str.charAt (idx);
    if (currChar == 'x')
    {
      count ++;
      moveAllx (str, idx+1, count, newString);
    }
      else
      {
      newString += currChar;
      moveAllx (str, idx+1, count, newString);
      }
    }
Public static void main (String args[])
    {
      String str = "axbxcxxd";
      moveAllx (str, 0, 0, "");
    }
}
```

Time complexity = $O(n+count)$, $\approx O(n+n) = O(2n)$.
For asymptotic notations, $O(2n) \approx O(n)$.

8) Remove duplicates in a string.

E.g → i/p = "aabbccda", o/p → "abcd".

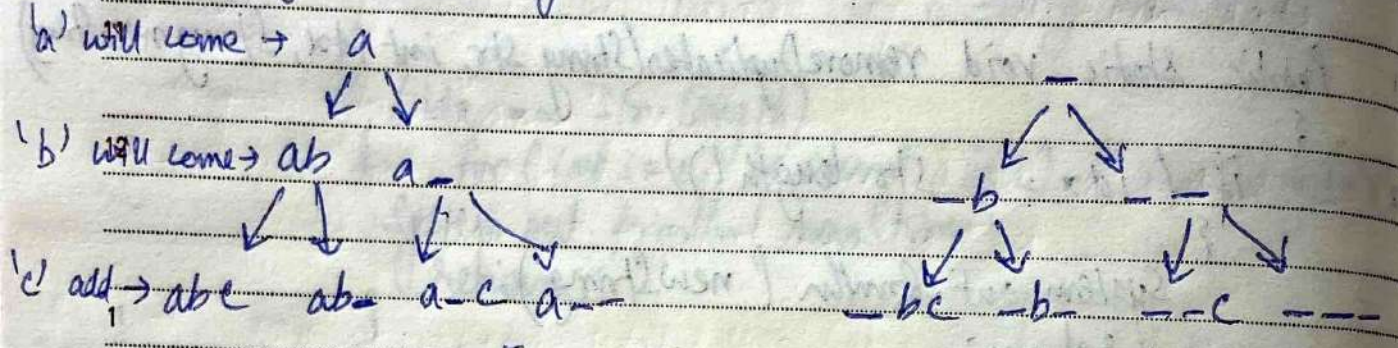
```
class Recursion10
{
Public static boolean[] map = new boolean[26];

Public Static void removeDuplicates(String str, int idx, String newString)
{
if (idx == str.length())
{
System.out.println(newString);
return;
}
char curChar = str.charAt(idx);
if (map[curChar - 'a'] == true)
{
removeDuplicates(str, idx+1, newString);
}
else
{
newString += curChar;
map[curChar - 'a'] = true;
removeDuplicates(str, idx+1, newString);
}
}
Public void static main(String args[])
{
String str = "abbccda";
removeDuplicates(str, 0, "");
}
}
```

Time complexity = O(n).

Q> Print all subsequences of a String.

E.g → "abc".

Subsequence of a string → the subsequences of the string can come or not come. However, if they come, they have to follow the same order as in the original string.



Subsequences = [abc, ab, ac, a, bc, b, c, _]

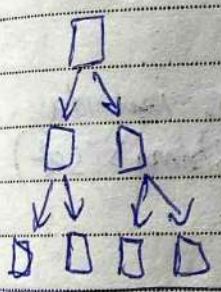
= [a, b, c, ab, ac, bc, abc, _].

* ca cannot happen, ba cannot happen and similarly bac, bca, cab, cba also cannot happen. The order of appearance in the original string must be followed. 'b' has to come before 'c' and 'a' has to come before 'b' & 'c'.

→ class Recursion11
{
Public Static void subsequences (String str, int idx, String newString)
{
if (idx == str.length())
{
System.out.println(newString);
return;
}

```
char currChar = str.charAt(idx);
subsequences(str, idx+1, newString+currChar);
subsequences(str, idx+1, newString);
}
Public static void main(String args[])
{
    String str = "abc";
    subsequences(str, 0, "");
}
```

*Calculating the time complexity,



last level = 2^n nodes
2nd last level = 2^{n-1} nodes
3rd last level = 2^{n-2} nodes
⋮
1 node.

So, total no. of nodes in the recursive tree =
$2^n + 2^{n-1} + 2^{n-2} + \dots + 1$

total subsequences = 2^n,
where, n = string length.

This is actually a geometric progression or G.P series.
So, the formula is, $\frac{a(r^n - 1)}{(r-1)}$.

Here, $a = 1$, r (common diff) $= 2$, $n = n+1$

$\therefore$ Time complexity $= \frac{(2^{n+1} - 1)}{(2-1)} = O(2^{n+1}) \approx O(2^n)$.

Another variation

8 8) Print all the unique subsequences of a string (that don't get repeated).

9

✓ Here, we shall use a special data structure called 10 HashSet. HashSet includes a lot of data within itself but keeps only the unique copy of each data.

11 E.g → i/p → "aaa"

12 without HashSet → o/p → aaa, aa, aa, aa, a, a, a, _

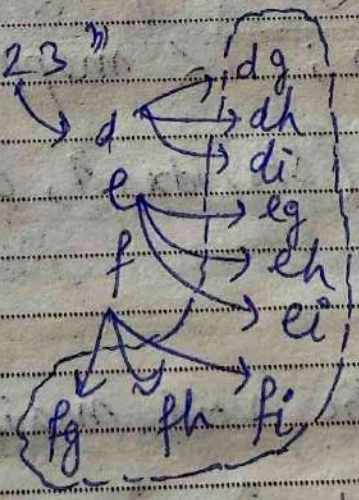
with Hashset → i/p → aaa, aa, a, _

```
1 → import java.util.HashSet;
2   Class Recursion12
    {
3   Public static void Subsequences(String str, int idx,
                String newString, HashSet<String> set)
4   {
      if (idx == str.length())
      {
5       if (set.contains(newString))
        {
6         return;
        }
        else
        {
          System.out.println(newString);
          set.add(newString);
          return;
        }
    }
```

```
char currChar = str.charAt(idx);

subsequences(str, idx+1, newString+currChar, set);
subsequences(str, idx+1, newString, set);
}
public static void main(String args[])
{
String str = "abc";
HashSet<String> str = new HashSet<>();
subsequences(str, 0, "", set);
}
}
```

Q) Print Keypad combination

0 → .
1 → abc
2 → def
3 → ghi
4 → jkl
5 → mno
6 → pqrs
7 → tu
8 → vwx
9 → yz



here, in each level, the number of choices have becomes variable.

→ class Recursion 13

```
{
Public static String[] keypad = {"", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tu", "vwx", "yz"};

Public static void printComb (String str, int idx, String combi)
{
    if (idx = str.length())
    {
        System.out.println(combi);
        return;
    }
    char currChar = str.charAt(idx);
    String mapping = keypad[currChar - '0'];
    for (int i=0; i< mapping.length(); i++)
    {
        printComb(str, idx+1, combi + mapping.charAt(i));
    }
}
Public static void main (String args[])
{
    String str = "23";
    printComb(str, 0, "");
}
}
```

Time Complexity = $O(4^n)$

Here, worst case is "6666". Therefore, we are considering the worst case time complexity.

Recursion - III

Q) Print all permutations of a String.

e.g → i/p = "abc"
o/p = "abc", "acb", "bac", "bca", "cab", "cba".

no. of permutation = n!, here 3! = 6.

```
class Recursion 14
{
Public static void printPerm (String str, String Permutation)
{
  if (str.length() == 0) {
  System.out.println (Permutation);
  return; }
for (int i=0; i < str.length(); i++)
{
  char currChar = str.charAt(i);
  String newStr = str.substring(0,i) + str.substring(i+1);
  printPerm (newStr, Permutation + currChar);
}
}
Public static void main (String args[])
{
  String str = "abc";
  printPerm (str, "");
}
}
```

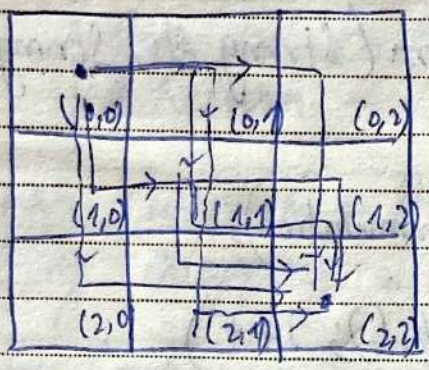
Time Complexity = $n * (n-1) * (n-2) * \dots$

$= O(n!)$.

8) count total paths in a maze to move from (0,0) to (n, m).

E.g → Let's say n = 3, m = 3.
Also, following conditions are given —

(I) movement in the right direction
(II) movement in the downward direction.



total Paths = 6.

(i, j)
(i+1, j) (i, j+1)

So, total count → count (i+1, j) + count (i, j+1)
Also, we will go from (0,0) to (n-1, m-1)

→ class Recursion 15
{
Public static int countPaths (int i, int j, int n, int m)
{
if (i == n || j == m) // base case 1
return 0;
if (i == n-1 && j == m-1) // base case 2
return 1;

int downPaths = countPaths (i+1, j, n, m);
int rightPaths = countPaths (i, j+1, n, m);

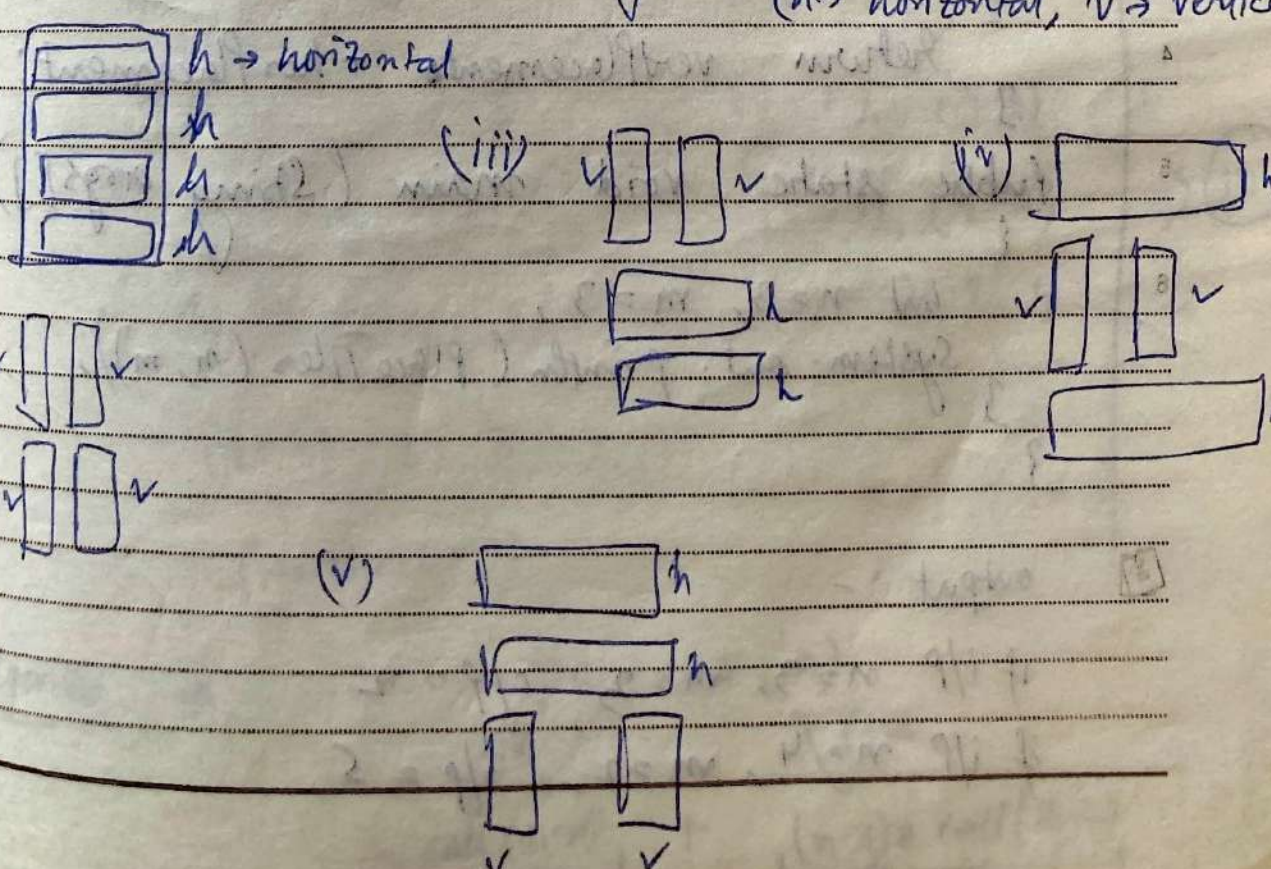
return downPaths + rightPaths;
}

```
Public static void main (String args[])
{
    int n=3, m=3;
    int totalPaths = countPaths(0, 0, n, m);
    System.out.println(totalPaths);
}
}
```

Note :- We have used 'Backtracking' method to solve the problem. Backtracking basically means we take a path to solve an issue, if it's correct we can proceed or else we shall come back to the prev. path and try a different way.

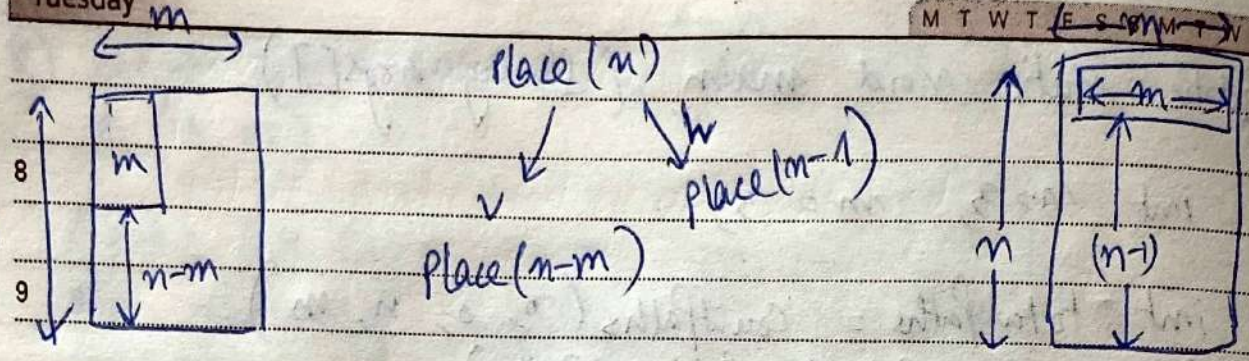
Q) Place tiles of size 1×m in a floor size of n×m. (E.g → n=4, m=2).

→ now there are multiple ways to place the tiles. (h → horizontal, v → vertical)



✓ When n = m → we have 2 options, either place all 'h' or all 'v'

✓ When n < m → Place all tiles 'h'.




```
class Recursion 16
{
Public Static int placeTiles (int n, int m)
{
if (n == m)
return 2;

if (n < m)
return 1;

int vertPlacement = placeTiles (n-m, m);
int horPlacement = placeTiles (n-1, m);

return vertPlacement + horPlacement;
}
public static void main (String args[])
{
int n = 3, m = 3;
System.out.println (placeTiles (n, m));
}
}
```

output :-

if i/p n = 3, m = 3, o/p = 2

if i/p n = 4, m = 2, o/p = 5

Q) Find the number of ways in which you can invite 'n' people to your party, single or in pairs.

e.g. (n=4)

if n=1, ⇒ (1)

if n=2, (1, 2) (1–2) ⇒ (2)

if n=3, 1, 2, 3 — (1)

(1–2), 3

(1–3), 2 — 3

(2–3), 1

⇒ (4)

if n=4,

1, 2, 3, 4 — 1–2, 3, 4

1, (2–3), 4 — (1–2), (3–4)

1, 2, (3–4) — (1–3), (2–4) ⇒ (10)

1, (2–4), 3 — (1–3), 2, 4

(1–4), 2, 3

(1, 4), (2–3)

algorithm ⇒ Guest call (n)

single / pair

call (n–1) + (n–1) * call (n–2)

19 May 2022
Thursday
21st Wk • 139-226

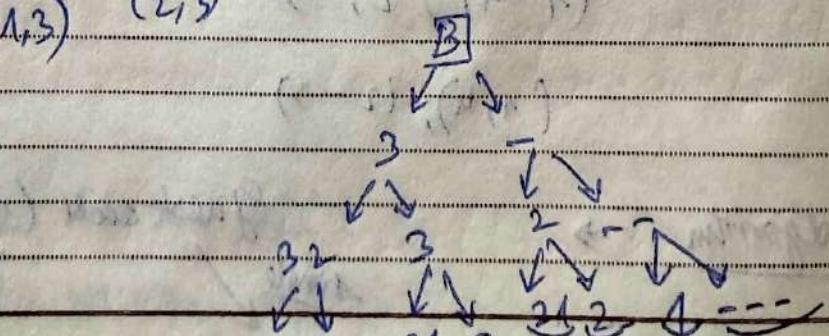
→ class Recursion17
{
Public static int callGuests (int n)
{
if (n <= 1)
return 1;

int ways1 = callGuests (n-1); // single
int ways2 = (n-1) * callGuests (n-2); // pair

return ways1 + ways2;
}
Public static void main (String args[])
{
int n = 4;
System.out.println (callGuests (n));
}
}

Q) Print all the subsets of a set of first N natural numbers.

n=3 (1, 2, 3)
→ (-) → empty subset
(1) (2) (3)
(1,2) (1,3) (2,3)
(1,2,3)



(3,2,1), (3,2), (3,1), (3),
(2,1), (2), (1), (-).

import java.util.*;

→ class Recursion18
{
Public static void printSubset(ArrayList<Integer> subset)
{
for(int i=0; i<subset.size(); i++)
{
System.out.print(subset.get(i) + " ");
}
System.out.println();
}
Public static void findSubsets(int n, ArrayList<Integer> subset)
{
if(n==0)
{
PrintSubset(subset);
return;
}
// if added
subset.add(n);
findSubsets(n-1, subset);

// if not added
subset.remove(subset.size()-1);
findSubsets(n-1, subset);
}
Public static void main(String args[])
{
int n=3;
ArrayList<Integer> subset = new ArrayList<>();
findSubsets(n, subsets);
}
}

Time complexity → $O(2^n)$ → bcz every element has a choice - to be added or not at every level.