Introduction to Recursion

* A function calling itself called as Recursion.

```
Void print(){
    print();
}
```

* If we write return statement in a method it ends the method irrespective of lines below return statement.

* PMI: Principle of Mathematical Induction → Sum of first n natural numbers for any given no n

To prove: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = Sum(n)$

$\sum$ represents $1+2+\dots+n$

i.e $f(x) = x+2$

we can also write like this as $f(n)$ or $Sum(n)$

Assumption: $\sum_{i=1}^{K} i = \frac{K(K+1)}{2} = Sum(K)$ → Assumption

proof: 1: $\sum_{i=1}^{K+1} = \frac{(K+1)(K+2)}{2}$ → self work

2: $n = 1 \rightarrow Sum(1) = \frac{1(1+1)}{2} = 1$ → Base case

proof1: $\sum_{i=1}^{K+1} i = 1+2+3\dots K + (K+1)$

$\sum_{i=1}^{K+1} = \sum_{i=1}^{K} + K+1$

$= \frac{K(K+1)}{2} + K + 1$

$= K+1\left(\frac{K}{2}+1\right)$

$= \frac{(K+1)(K+2)}{2}$

LHS = RHS

problem - WAP to print all natural numbers from 1 to n using recursion

```
Static Void printIncreasing(int n){ // 1 2 3 .. n-1, n
    if(n==1){                       // Base case
        Sout(n);
        return;
    }
    printIncreasing(n-1); // Assumption // 1 2 3 .. n-1
    Sout(n);              // self work  // n
}
```

PI(5) = 1 2 3 4 5 ⇒ PI(n) = PI(n-1) n

PI(4) 5

PI(4) = 1 2 3 4

PI(3) = 1 2 3

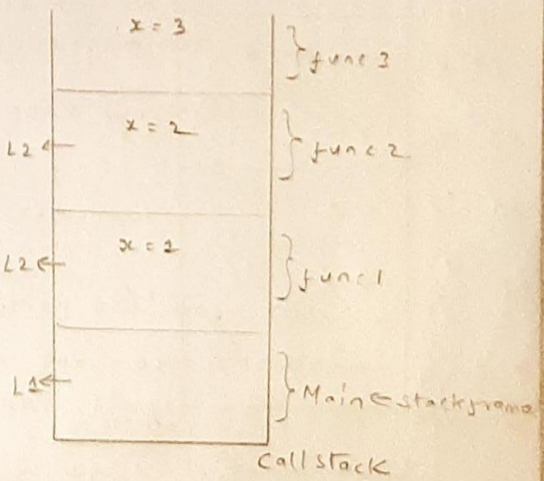
PI(2) = 1 2

PI(1) = 1

LIFO

```
* void func1(){
      int x = 1; L1
      func2(); L2
      Sout(x); L3
  }
  Void func2(){
      int x = 2; L1
      func3(); L2
      Sout(x); L3
  }
  Void func3(){
      int x = 3; L1
      Sout(x) L2
  }
  psvm(){
      func1(); L1
  }
```

o/p - 3 2 1



→ Stack frame stores local variables of current line which is executing

→ Let in func2 line 2 is called i.e func3 after completing the func3 line 3 will be executed in func2.

→ Stack frame of func will be removed after completing the func execution

Print 1 to n natural Numbers

* Static void printIncreasing(int n) {

```
L1  if(n==1){
L2      Sout(1);
L3      return;
L4  }
L5  printIncreasing(n-1);
L6  Sout(n);
L7  }

psvm(){
L1  int n=4;
L2  printIncreasing(n);
}
```

if we have not written this code PI(1) → PI(0) → PI(-1) → stack overflow infinite recursion occurs. this condition helps to prevent infinite recursion so it is called halting condition / Base case (smallest problem we already know the ans.)

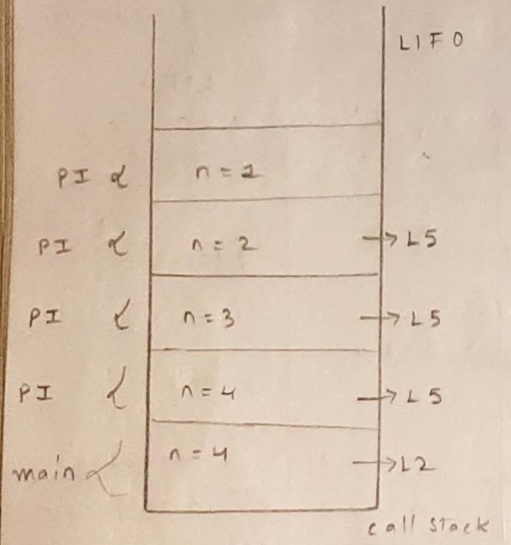
// 1, 2, 3 (expectation)

→ Subproblem/smaller problem

// 4 → Self work

olp - 1 2 3 4

Dry run in call stack:



main → PI(4) → PI(3) → PI(2) → PI(1)

printing is done in this order (LIFO)

olp → 1 2 3 4

Recursion - A function calling itself with different parameters & a base case to terminate the infinite loop

* In Recursion we solve a bigger problem by solving smaller sub problems

* We represent the problem in terms of functions

i.e PI(n) ← Problem

PI(n-1) ← subproblem

→ A function calls itself to solve its subproblems.

Steps for solving Recursion problem

The Recursion Spell (High level Thinking)

1 Identifying the Smaller problem → let recursion solve it (show full faith in Recursion (The Recursive Leap of Faith) do not doubt it.)

2 Self Work - Do your work (solve remaining)

1 2 3 4 5

Recursion work — Self Work

3 Base Case (Identify the base case) → Smallest problem for which the answer is very obvious and simple. (and doesn't depend on any subproblem)

Note: Identify first which is to be done self work / Recursive work

Working of Recursive function

Syntax:

```
methodName (N Parameters)
{
    if (haltCondition) {
        // base case
        return result
    }
    return methodName (N Parameters)
    // recursive work
    }
    // self work
```

problem - WAP to print all Natural No's from n to 1 using recursion. $n \geq 1$

Self work: PD(5) → 5, 4, 3, 2, 1 ← Recursive work

PD(n) → n, n-1, . . . 1

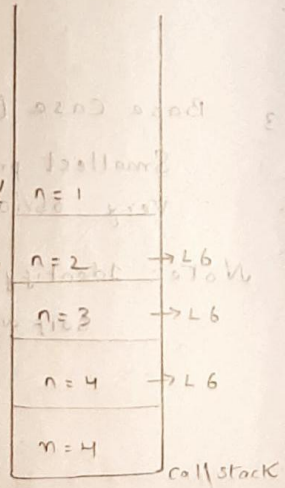
1. Smaller problem : PD(n-1)
2. Self work : Sout(n)
3. Base case : n==1

```
static void printDecreasing(int n) {
L1      if (n==1) {
L2          Sout(1);
L3          return;
L4      }
L5      Sout(n);
L6      printDecreasing(n-1);
L7  }
```

n = 4

expected o/p - 4 3 2 1

o/p - 4 3 2 1



main → P(4) → P(3) → P(2) → P(1)

Solving factorial & fibonacci problems by Recursion

problem - Find the value of factorial 5 n, where $n \geq 0$

Subproblem → f(n-1)

f(5) → 5 × 4 × 3 × 2 × 1

f(4)

f(n) → n × n-1 × . . 1

f(n-1)

self work → n × f(n-1) = f(n) → This term/eq is called Recurrence Relation

Base case → if (n == 0) { return 1; }

5 * factorial(4)

5 * 4 * factorial(3)

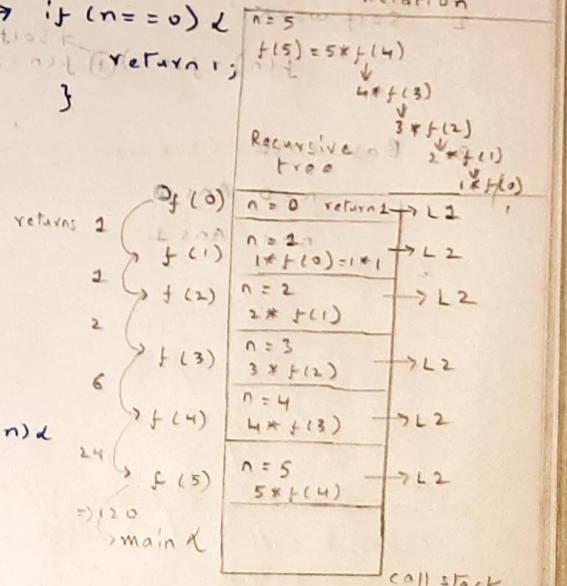
5 * 4 * 3 * factorial(2)

5 * 4 * 3 * 2 * factorial(1)

5 * 4 * 3 * 2 * 1

```
static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    int fact = n * factorial(n-1);
    return fact;
}
```

```
int smallAns = factorial(n-1);
return n * smallAns;
}
```



expected o/p - 5 × 4 × 3 × 2 × 1 = 120

o/p - 120

Problem - To find n^{th} fibonacci number

term =	0	1	2	3	4	5	6	7
	0	1	1	2	3	5	8	13

current term = (current term -1) + (current term -2)

1. Subproblem

   f(n-1) } find
   f(n-2) } using recursion

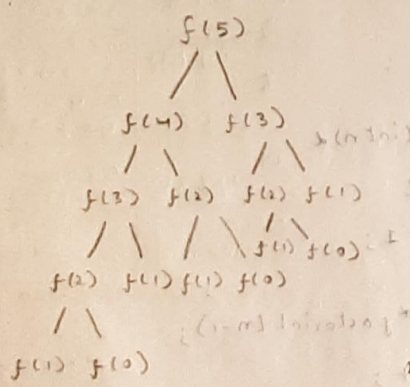
2. Self work

   f(n) = f(n-1) + f(n-2) (+ → Self work (add))

3. Base case (Subproblem doesn't exists for these cases)

   n = 0 Ans 0

   n = 1 Ans 1



L1 (level 1)

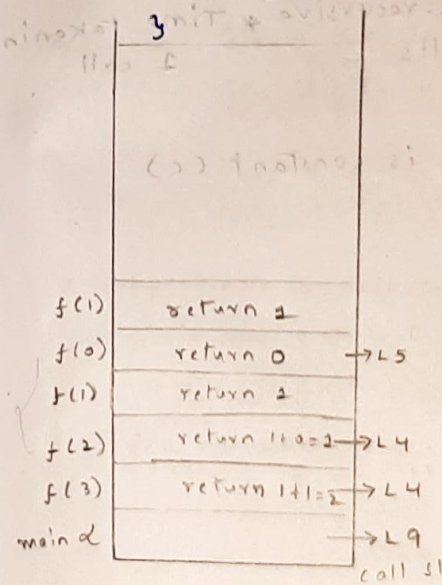
L2

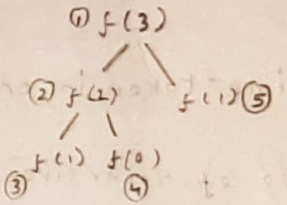
L3

L4

L5

```
static int fib(int n) {
    L1 if(n==0 || n==1) {
    L2     return n;
    L3 }
    L4 int prev = fib(n-1);
    L5 int prevPrev = fib(n-2);
    L6 return prev + prevPrev;
} L7
L8 psvm(String[] args){
    L9 Sout(fib(3));
```





→ Factorial

Time complexity = No of recursive calls * Time taken in 1 call

= (n+1) ≅ n * c (constant)

= nc ≅ O(n)

• n+1 because if n = 5 f(5) → f(4) → f(3) → f(2) → f(1) → f(0)

Space complexity → O(n) because in stack call frame if the n=4 the stack frames are 4.

* Basically after seeing the code of recursive functions we can't say the space complexity until after seeing the call stack memory how many stack frames are used

→ Fibonacci

Time Complexity = no. of. recursive calls * Time taken in 1 call

- Time taken in one call is constant (c)
- No. of. recursive calls

level 1 — f(5) — 1 → 2^0 → 2^{1-1}

level 2 — f(4) f(3) — 2 → 2^1 → 2^{2-1}

level 3 — f(3) f(2) f(2) f(1) — 4 → 2^2 → 2^{3-1}

level 4 — f(2) f(1) f(1) f(0) f(1) f(0) — 8 (let) → 2^3 → 2^{4-1}

level 5 — f(1) f(0) — 16 (let) → 2^4 → $2^{(5-1)}$

K^{th} level → $2^{K-1} = 2^{n-1}$ (level no)

*we need to find Total no. of calls i.e (add all)

$2^0 + 2^1 + 2^2 + 2^3 \cdots + 2^{n-1}$ (G.P) Geometric progression

$\frac{a(r^n - 1)}{r - 1} = \frac{2^0(2^n - 1)}{2 - 1} = 2^{n-1} \simeq 2^n$

Tc = no. of. recursive calls * Time taken in 1 call

$Tc = 2^n * c \simeq 2^n$

$Tc = O(2^n)$

Space Complexity → O(n)

33

Sum of Digits and Power of Number using Recursion

Problem: Given an Integer, find out the sum of its digits using recursion

n = 513 => 5+1+3 = 9

n = 1234 => 1+2+3+4 = 10

* Subproblem & selfwork

n = 1 2 3 4 → last digit (selfwork)

(Sum) 6 [(d-1) digits] (subproblem)

let the given number have d digits eg n=1234 / d=4

→ Recursively find (d-1) digits Sum (smallAns)

→ Ans = smallAns + lastDigit

* If we want to find the lastdigit of an number use % (modulo) i.e

1234 % 10 = 4

* If we divide the number / 10 we get the (n-1) digits i.e 1234 / 10 = 123

f(1234) = f(123) + f(4)

bigproblem — Smallerproblem — selfwork

f(n) = f(n/10) + n % 10

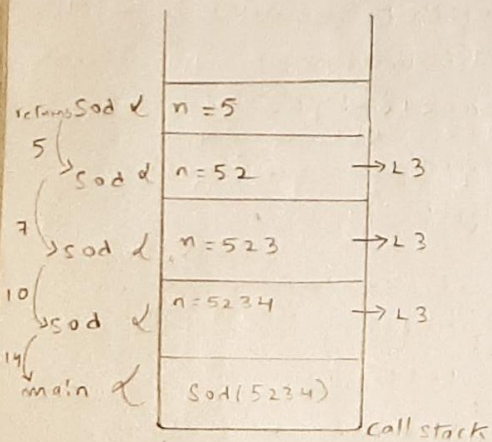
Sum of digits of n — Sum of all digits except last digit — last digit

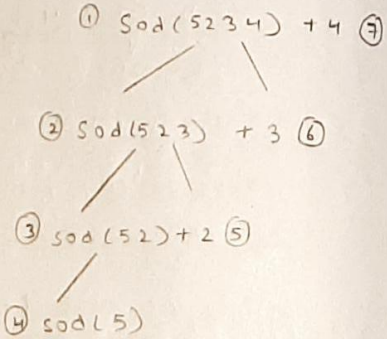
* Base case

```
if (n>=0 && n<=9)
    return n;
```

* sod(1234) = sod(123) + 4
  - ↳ sod(12) + 3 // 6
    - ↳ sod(1) + 2 // 3
      - ↓ returns 1

* L1 static int sod(int n) {
  L2 if (n>=1 && n<=9) return n;
  L3 return sod(n/10) + n%10;
  L4 }





n = 5234

= 14

Time complexity = No. of recursive calls * Time taken in 1 call

= d * c (c constant)

(no. of digits)

O(dc) ≅ O(d)

Space complexity = O(d)

* Follow up question

→ return the count of digits in a given number n

n = 5683 ans = 4

n = 534 ans = 3

```
Static int countofDigits(int n) {
    if (n>=0 && n<=9)
        return 1;
    return countofDigits(n/10) + 1;
}
```

* Subproblem & self work

n = 1234

cod(n/10) → return 1 (because 1 digit count is always 1)

* Base case

```
if (n>=0 && n<=9)
    return 1; // because single digit count is 1.
```

① cod(1234) + 1 ⑦

② cod(123) + 1 ⑥

③ cod(12) + 1 ⑤

④ cod(1)

n = 1234

Ans = 4

Problem:

Given two numbers p & q, find the value p^q using a recursive function

$p^q = p \times p \times p \cdots p \rightarrow q$ times

$p^q = p^{q-1} \times p$

recursive work | self work

i.e $a^b \times a^c = (a)^{b+c}$

$= p^{q-1+1}$

$= p^q$

* pow(p, q) = pow(p, q-1) * p → recurrence relation

p^q = p^{q-1} * p

p^{q-1} → recursive work

p → self work

* Base case

p = 5, 10, 523 but q = 0

$p^q = 5^0, 10^0, 523^0$

Ans = 1 1 1

$$pow(p,q) = \begin{cases} pow(p, q-1) * p & , q > 0 \\ 1 & , q = 0 \end{cases}$$

* f(5, 4) = f(5, 3) * 5

p, q

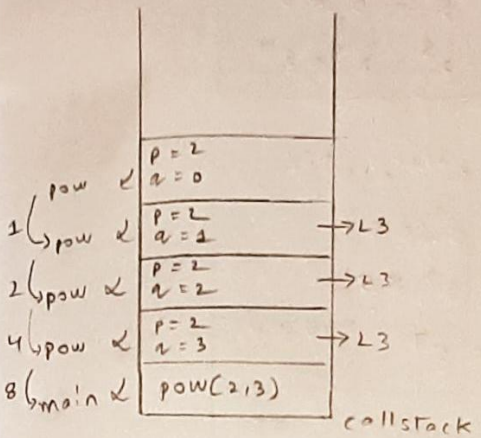
→ f(5, 2) * 5

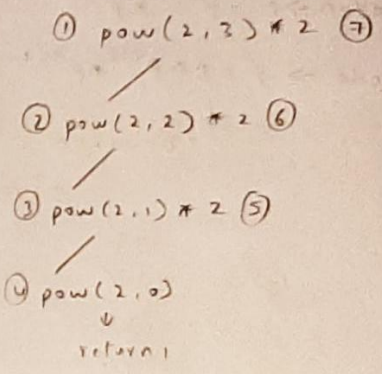
→ f(5, 1) * 5

→ f(5, 0) * 5

$5^4 = 625$

```
* L1 static int pow(int p, int q) {
  L2     if (q == 0) return 1;
  L3     return pow(p, q-1) * p;
  L4 }
```





$2^3 = 8$

* Time complexity → O(q) because no. of recursive calls are in order of q

i.e 3 - 2 - 1 - 0

Tc = No. of recursive calls * Time taken in 1 call

Tc = q * c

Tc = O(q x c) ≅ O(q)

* Space Complexity →

O(q) because there are q stack frames

// q is 3 but there are 4 stack frames

O(q+1) ≅ O(q)

Approach 2 -

1st Approach → $p^a = p^{a-1} * p$

2nd Approach → $p^a = p^{a/2} * p^{a/2}$

i.e

$5^4 = 5^2 * 5^2$

$2^8 = 2^4 * 2^4$

q even → $p^a = p^{a/2} * p^{a/2}$

q odd → $p^a = p * p^{a/2} * p^{a/2}$

↳ because

$5^7 \rightarrow 5^{7/2} * 5^{7/2}$

$= 5^3 \times 5^3$

$= 5^6 \times 5^1$ → (* p)

$= 5^7$

* Recurrence relation

$$f(p,a) = \begin{cases} f(p,\frac{a}{2}) * f(p,\frac{a}{2}) & , a \text{ is even} \\ f(p,\frac{a}{2}) * f(p,\frac{a}{2}) * p & , a \text{ is odd} \\ 1 & , a \text{ is } 0 \end{cases}$$

* 1st call $f(p,a)$ $a/2^0$

↓

2nd call $f(p,\frac{a}{2})$ $a/2^1$

↓

$f(p,\frac{a}{4})$ $a/2^2$

↓

$f(p,\frac{a}{8})$ $a/2^3$

⋮

$(k+1)^{th}$ call $f(p,\frac{a}{2^k})$ $\frac{a}{2^k}$

* If $2^k > a$, a will be 0 because we are performing the operations until a=0

i.e $\frac{2}{5} = 0$, $\frac{a}{2^k}$, $2^k > a$

$2^k > a$

$\log_2 2^k > \log a$

$k > \log a$

i.e $k \simeq \log a$

* Total no. of calls are $(k+1) \simeq k$

i.e $k \simeq \log a$

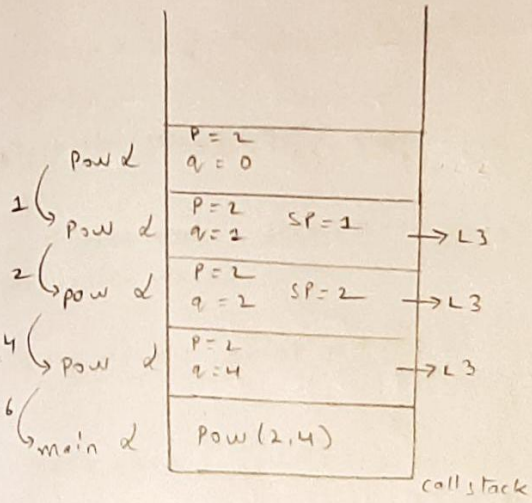
* In Approach 1 the total no. of calls are a but In Approach 2 the total no of calls are $\log a$ which is better approach than 1st Approach.

$TC = \log a \times c$

$TC = O(\log a \times c)$

$\boxed{TC = O(\log a)}$

* Dry run



* 1 static int pow(int p, int a) {

2 if (a == 0) return 1;

3 int smallPow = pow(p, a/2);

4 if(a%2==0) {

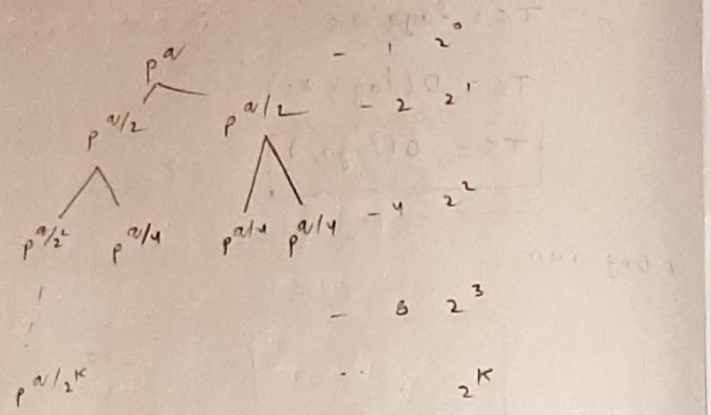
5 return smallPow * smallPow;

6 }

7 return p * smallPow * smallPow;

8 }

* Instead of previous code i.e smallPow * SmallPow if we have written (Pow(p, a/2) * Pow(p, a/2)) we are calling the function 2 times.



we need to find the total calls add all of them

$2^0 + 2^1 + 2^2 \ldots 2^K$

$$\frac{1(2^{K+1}-1)}{2-1} \approx 2^K$$

Total calls (No of) $= 2^K$

$K = \log q$

$= 2^{\log_2 q} \approx q$

i.e T.C will become O(q) so we have written in smallPow * SmallPow (i.e called the function only 1 time) for which TC is O(log q) which is better than O(q)

K Multiples & Alternate Sum Series using Recursion

Problem – Given a number num and a value K. print K multiples of num

constraints : K > 0

Input 1 : num = 12 , K = 5

o/p 1 : 12, 24, 36, 48, 60

I/p 2 - num = 3 , K = 8

o/p 2 - 3, 6, 9, 12, 15, 18, 21, 24

Big problem : f(n,K) → prints first K multiples of n

small problem: f(n, K-1) → faith (recursive work)

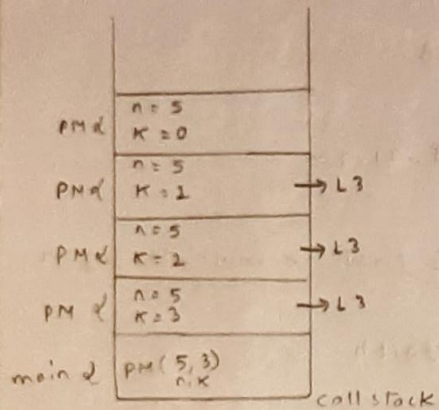
self work : print n*K

Base case : if(K==1) Sout(n) return;

f(5,4)
↓
f(5,3)
↓
f(5,2)
↓
Base case f(5,1)

(5,4) → 5 10 15 20
Recursive work | self work

```
* L1: static void printMultiples(int n, int k) {
L2          if(k==0) return;
L3          printMultiples(n, k-1);
L4          Sout(n*k);
L5  }
```



printMultiples(5,3)

5 10 15

Tc = k * c

Tc = O(k * c) ≈ O(k)

Sc = O(k)

Problem - Given a number n. Find the sum of natural numbers till n but with alternate signs.

That means if n=5 then you have to return 1-2+3-4+5 = 3 as your answer

constraints : $0 <= n <= 1e6$ (10^6)

ip1 - n=10

op1 = -5

ip2 - n=5

op2 - 3

Question - n = 5

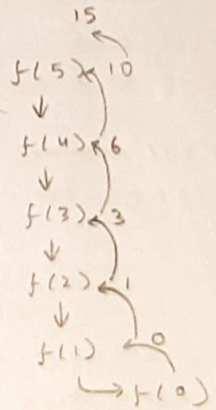
1+2+3+4+5 = 15

f(n) = f(n-1) + n

- f(n): Bigger problem — sum of 1 to n
- f(n-1): Smaller problem — recursive work 1 to n-1
- n: self work

Basecase: if(n==0) return 0;

```
* static int SeriesSum1(int n) {
        if(n==0)
          return 0;
        return SeriesSum1(n-1) + n;
  }
```



n=5 f(n) = 1-2+3-4+5 = 3

f(n) = f(n-1) + n recurrence relation

if n is odd add
if n is even subtract

n=6

f(6) = 1-2+3-4+5-6 = -3

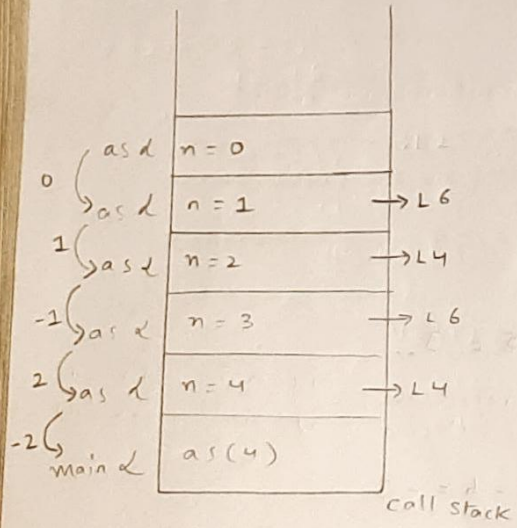
f(n) = f(n-1) - n recurrence relation

* $f(n) = \begin{cases} f(n-1)+n & , n \text{ is odd} \\ f(n-1)-n & , n \text{ is even} \end{cases}$

Base case - if n == 0
return 0;

```
* L1 static int alternateSeries(int n){
  L2     if(n==0) return 0;
  L3     if(n%2==0){
  L4         return alternateSeries(n-1) - n;
  L5     }else{
  L6         return alternateSeries(n-1) + n;
  L7     }
  L8 }
```

* Dry run



TC = n x c
TC = O(n)

GCD & Euclid's Algorithm using Recursion

problem - Given two numbers x and y. Find the greatest common divisor of x and y using recursion.

constraints : 0 <= x, y <= 1e6

ip1: x = 4, y = 9

op1: 1

* x = 24 ⇒ 1, 2, 3, 4, 6, 8, 12, 24
y = 15 ⇒ 1, 3, 5, 15

Common Divisors ⇒ 1, 3

Greatest common Divisor ⇒ 3

* x = 16 ⇒ 1, 2, 4, 8, 16
y = 12 ⇒ 1, 2, 3, 4, 6, 12

Common Divisors ⇒ 1, 2, 4

GCD ⇒ 4

Brute force Approach - Refer code in VS Code

x = 12 y = 16

gcd(x, y) ≤ min(x, y) i.e GCD will be less than 12, 16 it will not be in between 12 & 16.

* Divide both the numbers from

from ↓	12	16
12	✓	x
11	x	x
10	x	x
9	x	x
8	x	✓
7	x	x
6	✓	x
5	x	x
4	✓	✓ (both 12 & 16 can be divided by 4 which is GCD)

Time complexity will be -
O(min(x, y))

* x = 4 , y = 9

	4	9
4	✓	x
3	x	✓
2	✓	x
1	✓	✓

Approach 2 - (Long Division method) (Iterative)

(Refer code in vscode)

eg- x = 24
y = 15

divisor ↑ dividend ↑

15) 24 (1

15 ← dividend

divisor ← 9) 15 (1

9

6) 9 (1

6

3) 6 (2

6

Answer (0)

if (x < y)

gcd(x, y)

↓

gcd(y, x) ✓

swaps ✓ Automatic call ✓

i e

24) 15 (0

0

15) 24 (1

15

9

eg2- x = 12
y = 16

16) 12 (0

0

12) 16 (1

12

4) 12 (3

Ans 12

(0)

7) 9 (1

7

2) 7 (3

6

1) 2 (2

2

(0)

GCD for 2 prime numbers is always 1

Approach 3 - Euclids Algorithm

Euclids Algorithm - gcd(x, y) = gcd(y, x%y)

gcd(x, 0) = gives x

Base case

Recurrence relation

eg- gcd(24, 15) = gcd(15, 9)

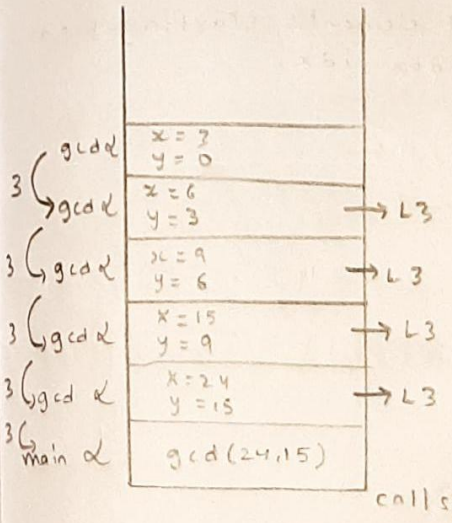
gcd(9, 6)

gcd(6, 3)

gcd(3, 0)

```
* L1 static int gcd(int x, int y) {
L2      if(y==0) return x;
L3      return gcd(y, x%y);
L4  }
```

Dry run -



x = 24
y = 15

Ans - GCD => 3

TC is in terms of log which is better Approach

* How to find LCM

W.K.T
LCM * GCD = X * Y

$$LCM = \frac{X * Y}{GCD}$$

* HW Armstrong no using recursion, Multiplication of x and Y

36
Recursion on Arrays

Question - Given an Array print all its values recursively

* while using recursion on Arrays we use a index parameter so that to traverse in the Array, we can point to current indexes inside the Array on which we are currently pointing.

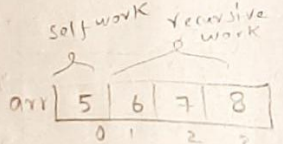
Bigproblem - f(arr, idx) → For a given array arr, print all elements starting from index idx.

Small problem - f(arr, idx+1)

Self work - Sout(arr[idx])

Basecase - empty Array

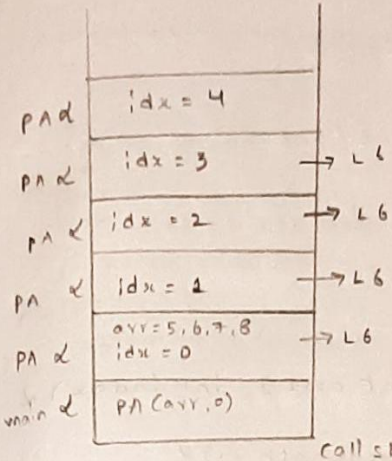
f(arr, 0) l5
↓
f(arr, 1) l6
↓
f(arr, 2) l7
↓
f(arr, 3) l8
↓
f(arr, 4) → Basecase

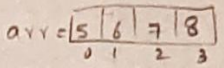


if (idx == n) i.e n=4
return;

```
*l1 static void printArray(int[] arr, int idx){
l2      if (idx == arr.length){
l3          return;
l4      }
l5      Sout(arr[idx]);
l6      printArray(arr, idx+1);
l7  }
```

* Dry Run



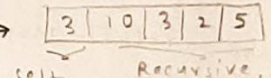


5 6 7 8

Problem - print the Max Value of the Array [3,10,3,2,5]

Bigger problem - f(arr, idx) → max value in the array starting from index idx

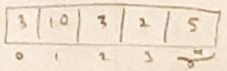
Smaller problem - f(arr, idx+1) →



Self work - Compare(smallAns, arr[idx])

Basecase - if (idx == n-1)
return arr[idx]

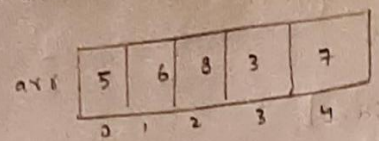
Single element

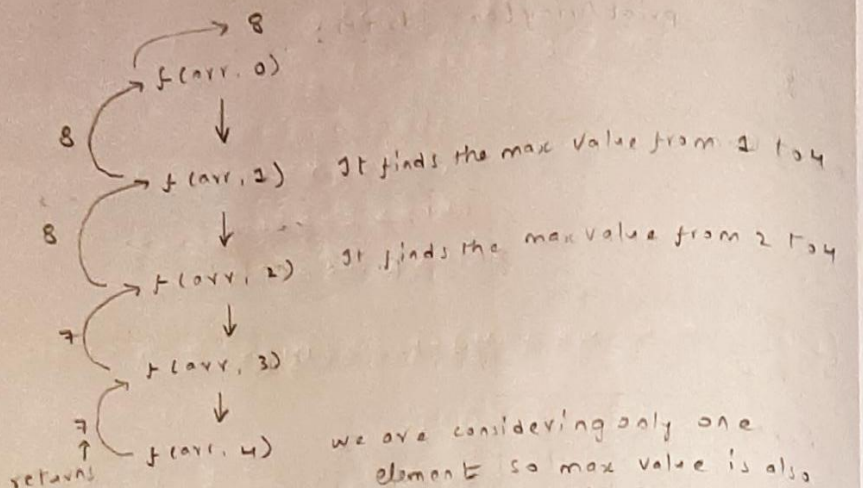


n = 5
n-1 = 4

i.e if idx == 4
it finds the max value from 4 i.e 1 to 4 but in case of 4 it is only one element so max value is also that element

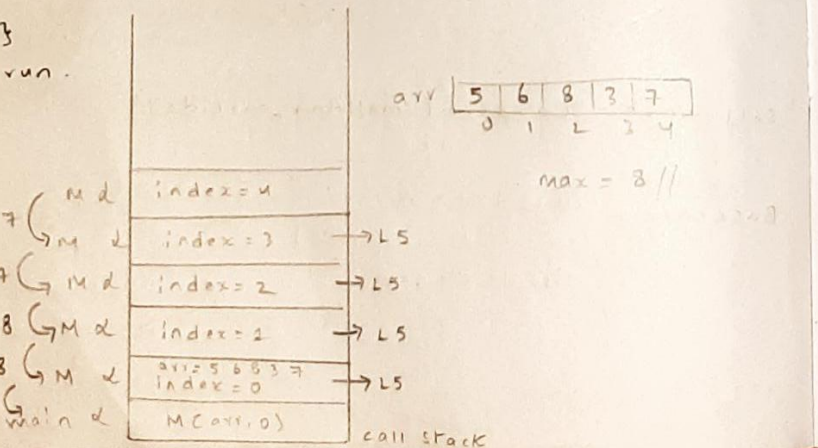
Recursive tree -






```
* L1: static int maxInArray(int arr[], int index) {
L2          if (index == arr.length - 1) {
L3              return arr[index];
L4          }
L5          int smallAns = maxInArray(arr, index+1);
L6          return Math.max(arr[index], smallAns);
L7      }
```

* Dry run.



TC → No. of calls * Time taken in one call

TC → n * c

↓ length of the Array

TC → O(n * c) ≃ O(n)

S.C → O(n) (because no. of stack frames used are same as length of Array) but if we do this problem in iterative method the space complexity will be O(1)

* problem — Find the sum of the elements of the Array [2, 3, 5, 20, 1]

Same logic as above problem.

Ans → 31

Bigger problem → f(arr, idx) ⇒ Sum of array starting from index idx

Base case → if the Array have only one element return that element i.e if (index == n-1) return arr[index];

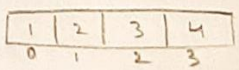
or if arr is empty return 0. (out of Array) if (index == n) return 0;

smaller problem → arr [2 | 3 | 5 | 20 | 1]

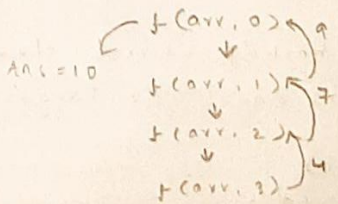
self work — recursive work

f(arr, index+1) ↓ Sum of Array starting from index idx+1

self work → add (arr[index], recursive work)

Recursive tree → 

=> 10



Dry run - Similar to maxInArray

code - refer vscode

TC → O(N)

SC → O(N)

37

Linear Search Using Recursion

Problem - Given an array of n integers and a target value x. print whether x exists in the array or not

constraints : $0 <= n <= 1e6$,
$-1e8 <= x <= 1e8$
$-1e8 <= a[i] <= 1e8$

i/p 1 : n = 8, x = 14, array = [4, 12, 54, 14, 3, 8, 6, 7]

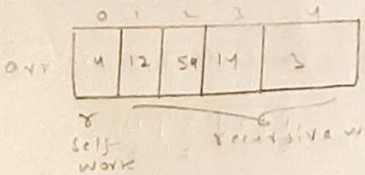
o/p 1 : Yes

i/p 2 : n = 1, x = 9, array = [2]

o/p 2 : No

Bigger problem - f(arr, target, idx)
↓
starting from index idx, if target is present in arr, return true or return false

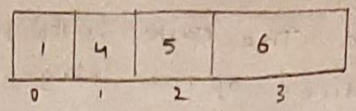
Smaller problem - f(arr, target, idx+1)
↓
starting from idx+1, if target is present in arr return true else false.



Self work - compare check if current element arr[idx] is equal to target

Base case - if (idx == n) // empty array (full array is processed but target element is not found)
return false

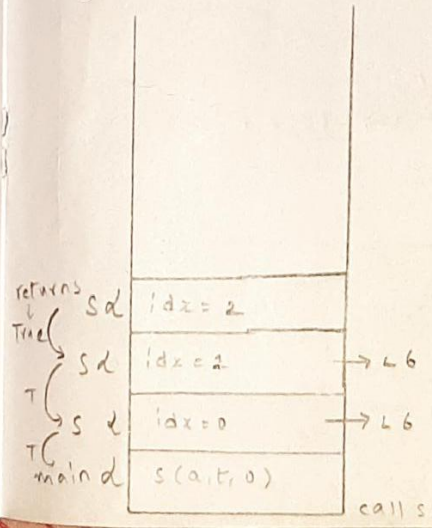
*

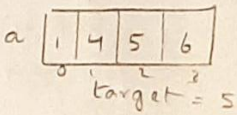


n = 4
target = 5

f(a, 5, 0)
↓
f(a, 5, 1)
↓
f(a, 5, 2) returns true
↓
f(a, 5, 3)
↓
f(a, 5, 4)
} → this calls will not be called for above array.

```
*1 static boolean search(int [] a, int target, int idx) {
2       if (idx >= a.length) {
3           return false;
4       }
5       if (a[idx] == target) return true;
6       return search(a, target, idx+1);
7   }
```





TC → In worst case no of calls are equal to n.
TC → n x c
TC → O(n)
S.C → O(n)
↓
no. of stack frames used are n in worst case.

Problem - Given an Array arr of size N and an Integer x. The task is to find all the indices of the integer x in the Array.

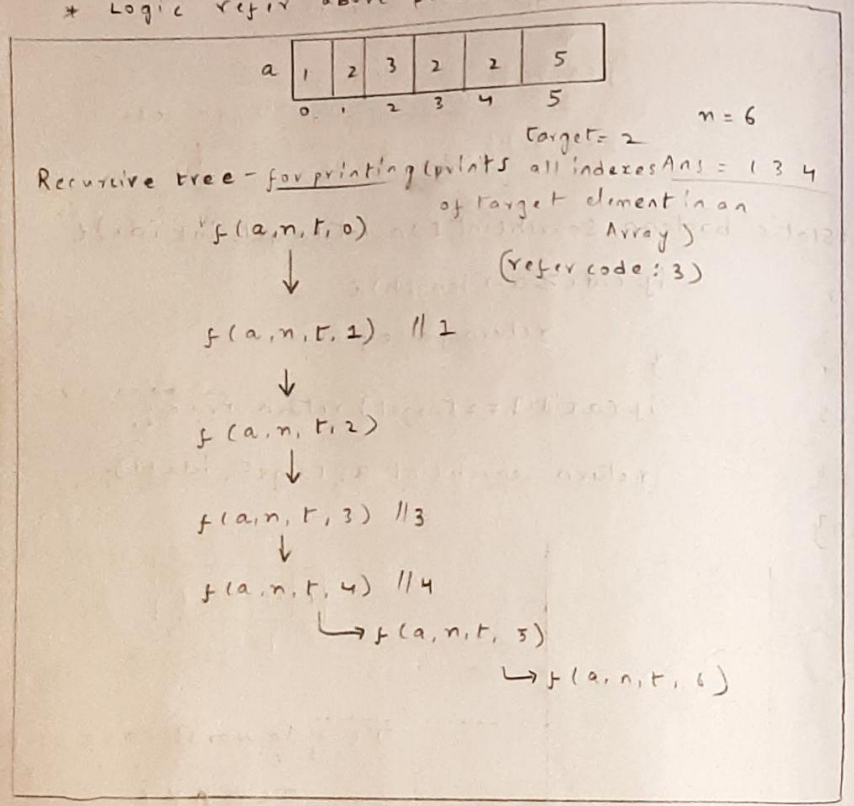
i/p1 - arr = {1, 2, 3, 2, 2, 5}, X = 2

o/p1 - 1, 3, 4

i/p2 - arr = {8, 8, 8}, X = 8

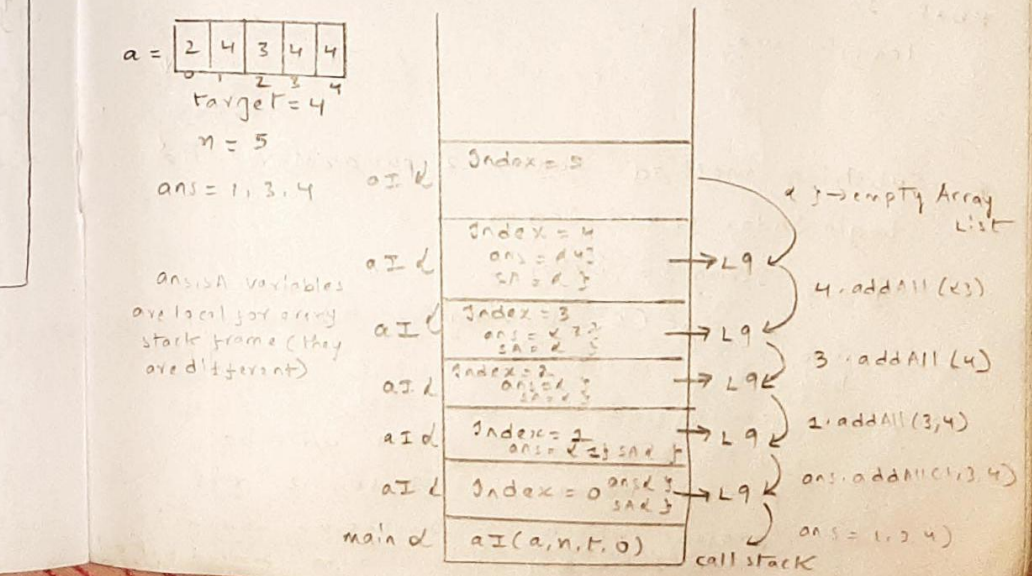
o/p2 - 0 1 2

* Logic refer above problem.



* return ArrayList containing all indexes of target element in an Array.

```
L1  static ArrayList<Integer> allIndices(int a[], int n,
                                int target, int index) {
        // base case
L2      if(index>=n) {
L3          return new ArrayList<Integer>(); // returns empty Array List
L4      } // self work
L5      ArrayList<Integer> ans = new ArrayList<>();
L6      if(arr[index]==target) {
L7          ans.add(index); // add elements stores current index value
L8      }
        //recursive work
L9      ArrayList<Integer> smallAns = allIndices(
                    /* a, n, target, index+1);
L10
        ans.addAll(smallAns); // values of smallAns will be added to Ans (concatenating).
L11     return ans;
L12 }
```



Follow up questions

1. Implement issorted method → Return boolean ^Yes or No
2. Find last index of target in arr

38
Recursion on Strings

* In case of Arrays if we want a particular element to read we use -

```
arr = {1, 2, 3}
arr[0]  // 1
```

* But in case of strings if we want to read a character we use.

```
String s = "Suraj";
char ch = s.charAt(0); // s
```

* In case of Arrays if we want to find length we use arr.length; (property)

* but in strings if we want to find the length we use.

s.length(); (method)

* Substring method takes 2 parameters i.e begin index & end index. i.e

```
String s = "Suraj";
           01234
Sout(s.substring(2, 4)); // ra
```

begin index is inclusive and end index is exclusive means in above example from 2 to 4 only ra will be printed because at index 2 r is present which is inclusive and a is at index 3 which is also be printed but not index 4.

* end index maximum value will be length of the string

* end index is also optional for substring method.

Problem - Remove all occurances of 'a' from string s = "abcax".

o/p = bcx

Iterative approach -

```
String ans = "";
for(int i=0; i<s.length(); i++){
    if(s.charAt(i) != 'a'){
        ans += s.charAt(i);
    }
}
```

Recursive Approach - Approach 1

Big problem - f(s, idx) ← current index

↓

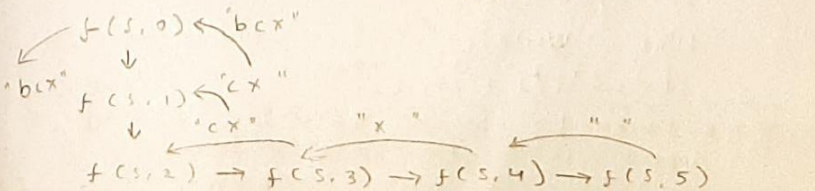
It will remove all occurences of 'a' in the string s starting from index idx

$$f(s,idx) = \begin{cases} f(s,idx+1), & s.charAt(idx) == 'a' \\ current + f(s,idx+1), & s.charAt(idx) != 'a' \end{cases}$$

self work ↓ recursive work ↓

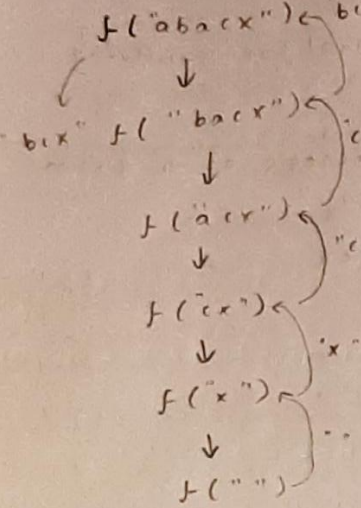
Basecase - if(idx == s.length())
return "";

Recursive tree - s = "abacx"
0 1 2 3 4 n=5



Approach 2 —
Previous problem without passing Index -

Recursive tree - S = "abacx" o/p - "bcx"



* using substring method we can solve the problem without passing Index i.e

```
s.substring(1);   // "abacx"
                  // "bacx"
```

read characters from Index 1

* Time complexity → No. of calls * Time Taken in 1 call

Approach 1 → n * (n + c)

because of (currentchar + smallAns)

i.e string concatenations are not constant they are linear (n+m) ≈ n

Tc → O(n²)

* why string concatenations are linear ?

Let say

str1 = "Hello"; ← length = n

str2 = "World"; ← length = m

str = str1 + str2

Time taken to concat both the strings will depend upon length so it is linear (n+m)

* Using stringbuilders we can read string concatenations in constant time. (In Java) but in c++ string concatenations are always linear

Approach 2 → using substring

n * 2n → 2n² ≈ n²

O(n²)

* substring methods takes linear time, while passing Index in parameters doesn't take extra time.

Problem - WAP to print reverse of a string using recursion.

S = "abcde"

o/p - "edcba"

S = "a b c d e"

self Recursion

Bigger problem → f(s, idx) (idx)

reverse the string starting from Index (idx)

SmallAns → f(s, idx+1)

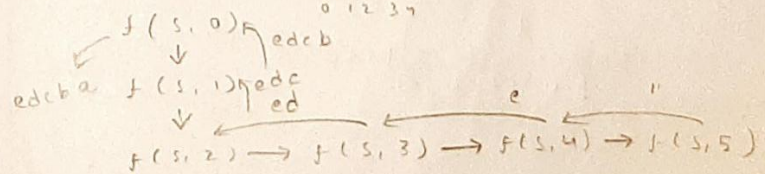
reverse of string starting from Index idx+1

Selfwork → SmallAns + currentchar

f(s,idx) → f(s,idx+1) + s.charAt(idx)

Base case → if(idx == n) return "";

Recursive tree →



Approach 2 -
using substring reverse the string without passing index
refer code.

Time Complexity → No. of calls * Time taken

Approach 1 → n * n

$O(n^2)$

Problem - WAP to check given string is Palindrome or not

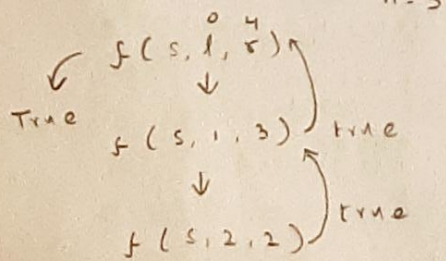
s = "dad" Yes
s = "level" Yes

s = "College" No
s = "Suraj" No
s = "Vaishu" No

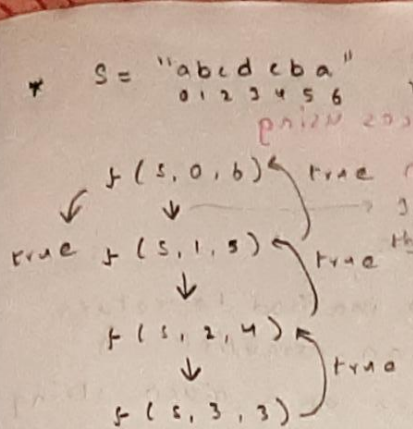
method1 - use previous problem to reverse the string check original string w reverse string using s.equals(rev)
TC → $O(n^2)$

method2 - (without reversing the string check if Palindrome ?)

*



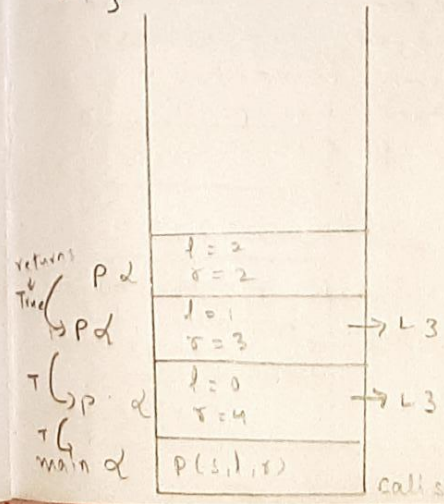
*



* f(s, l, r) = s.charAt(l) == s.charAt(r) && f(s, l+1, r-1)
self
recursive case

Basecase → a b c d c b a
l r
l < r
if l >= r
return true

* L1 static boolean isPalindrome(String s, int l, int r){
L2 if (l ≥ r) return true;
L3 return (s.charAt(l) == s.charAt(r) && isPalindrome(s, l+1, r-1));
L4 }



s = "radar";
0 1 2 3 4

TC = no. of calls * Time taken

= $\frac{n}{2} = n * c$

TC = O(n)

Follow up question -
* check if give integer is palindrome or not

39

Find Subsequences using Recursion

Problem - Return subsequence

Given a string, write a method to return all its subsequences in an arraylist. (A string is a subsequence of a given string that is generated by deleting some character of a given string without changing its order.)

Input : abc

o/p : a, b, c, ab, bc, ac, abc

"abc"	abc	abc
n = 3	ab	ab- deleted
Total subsequences	ac	a-c
$2^n = 2^3 = 8$	a	a--
	bc	-bc
	b	-b-
	c	--c
	" "	--- deleted

If we have a string of length 3, all 3 have 2 choices that they should be present and not to be present. i.e from above a is present in 4 cases but not present in another 4 cases b is present in 4 cases but not present in another 4 cases similarly c.

$2 \cdot 2 \cdot 2 = 2^3$ (choices)

* If we have a string of length n of choices 2 then $2 \cdot 2 \cdot 2 \cdot 2 \cdots = 2^n$

* f("abc") = [abc, ab, ac, a, bc, b, c, " "]

↓ Big problem

* f("bc") = [bc, b, c, " "]

↓ Subproblem

Recursive faith

* Self work

a + f(bc) = abc, ab, ac, a

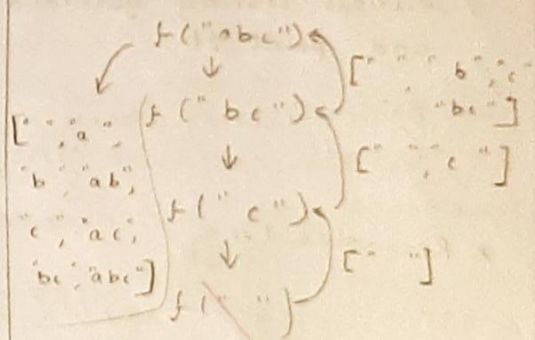
f(bc) = bc, b, c, " "

Recurrence relation -

f("abc") = a + f("bc")

[element] f("bc") + element

Recursive tree



2^n, n = 0, $2^0 = 1$ empty string SSQ is 1

Base case -

```
if (s.length() == 0)
    return "";
```

S.S = $O(2^n)$

Dry run - refer code in VS code

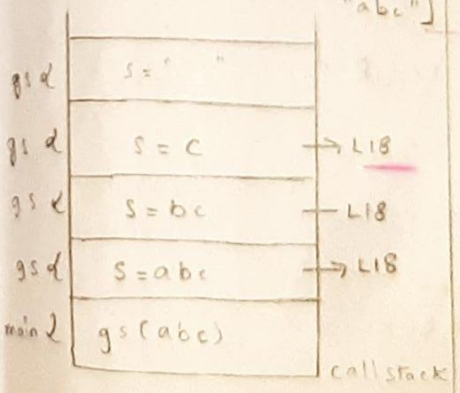
s = "abc"

ans = [" "]

ans = ["", "c"]

ans = ["", "b", "c", "bc"]

ans = ["", "a", "b", "ab", "c", "ac", "bc", "abc"]



Problem - Print Subsequences

Given a string. write a method to print all its subsequences

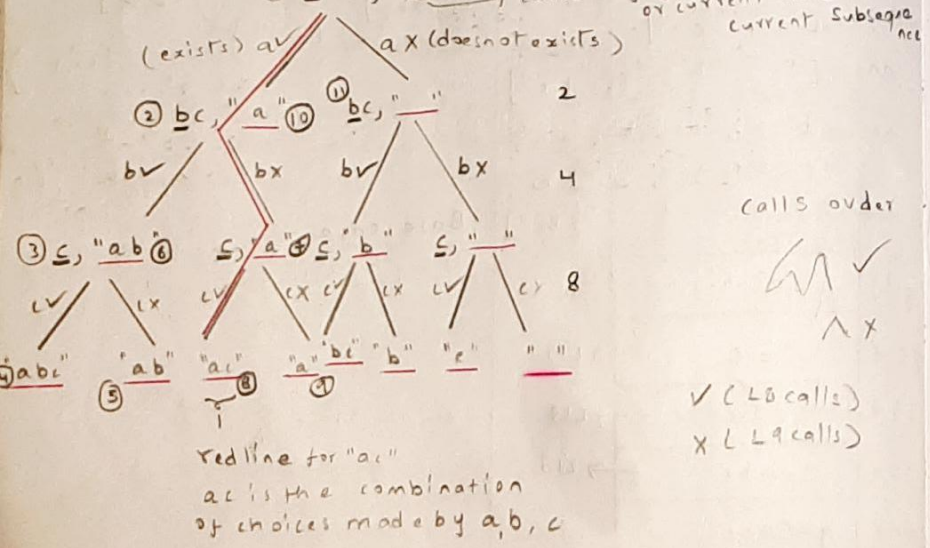
* In previous question they have said that create an arraylist and store all its subsequences. In this question they are saying that without storing them print all its subsequences.
* The size of Arraylist in previous question will be for length n there will be 2^n elements which consumes a lot of space so in this question prove that without consuming any space print the subsequences

→ without storing generate the subsequence and print it

→ Basically every character has a choice to exist or not exist in a given subsequence

(Doubt)

* Recursive tree



calls order

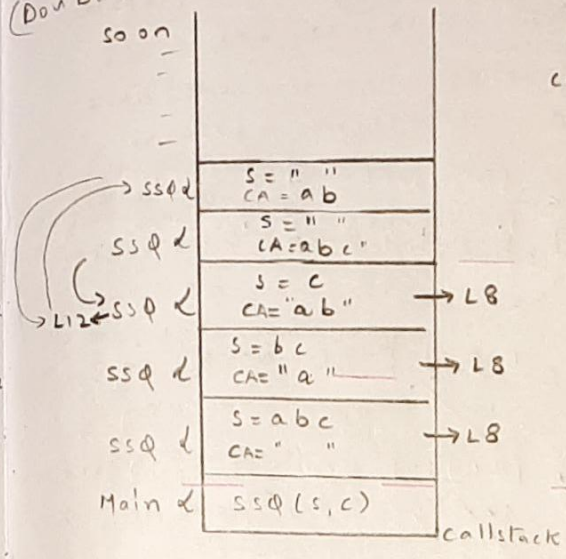
✓ (L8 calls)

✗ (L9 calls)

```
*L1 static void printSS(String s, String currentAns)
{
L2      if(s.length() == 0){
L3          Sout(currentAns);
L4          return;
L5      }
L6      char currentchar = s.charAt(0);
L7      String remstring = s.substring(1);

L8      printSS(remstring, currentAns+currentchar);
L9      printSS(remstring, currentAns);
}L10
```

(Doubt)



s = "abc"

curreAns = "" ""

abc, ab

Note: At any point Maximum no stack frames are n+1

Total space consume = No. of stack frames at any given instance * space occupied in 1 stack frame

(n+1) * n

S.C = $O(n^2)$

Problem - Given an Array of Integers, print sums of all subsets in it. output sums can be printed in any order

ilp - arr[] = {2, 3}

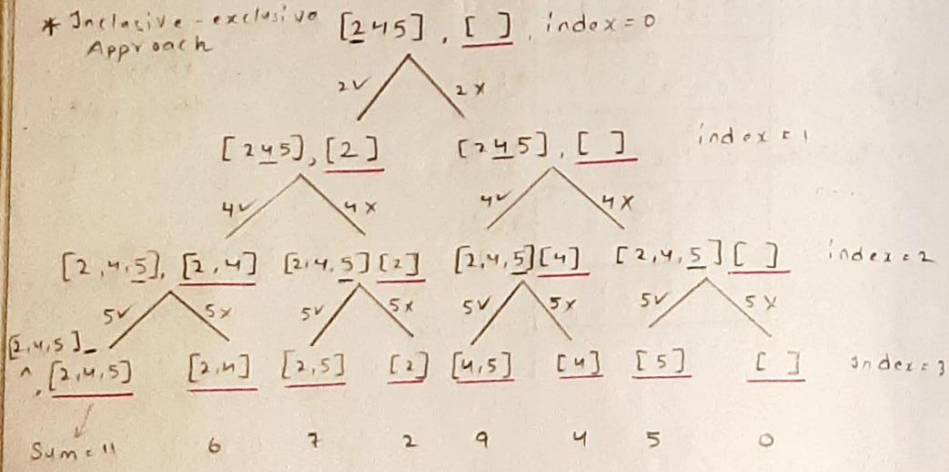
olp - 0 2 3 5

ilp - arr[] = {2, 4, 5}

olp - 0 2 4 5 6 7 9 11

	Subset	sum
* {2 4 5} =>	2 4 5	11
	2 4	6
	2 5	7
	2	2
	4 5	9
	4	4
	5	5
	[ ]	0

* Inclusive - exclusive Approach



40

Recursion on Array & Strings

Problem - Frog Jump

There are N stones, numbered 0, 1, 2 .. N-1. for each i ($0 \le i < N$), the height of stone i is h_i. There is a frog who is initially on stone 0. He will repeat the following action some number of times to reach stone N-1:

If the frog is currently on stone i, jump to stone i+1 or stone i+2.

Here, a cost of $|h_i - h_j|$ is incurred, where j is the stone to land on.

Find the minimum possible total cost incurred before the frog reaches stone N-1.

ilp n = 4

arr[] = 10 30 40 20
(0 1 2 3)

olp - 30

ways for frog to jump — total cost to reach N-1 stone

* $0 \to 1 \to 2 \to 3 = 50$

$0 \to 2 \to 3 = 50$

$0 \to 1 \to 3 = 30$

Minimum = 30

* lets think how we can solve this problem: if we are on 0th stone we can see how much that the cost of going to 1st stone or 2nd stone

* predicting the cost from 0th stone to let say 7th stone is not possible

* starting from stone 1 or 2 to reach last stone if we find the minimum cost we can find the minimum cost of frog to reach n-1 th via stone 1 or 2.

* $f(h, n, idx)$ (Bigger problem)

↓

Starting from idx minimum cost to reach (n-1) index.

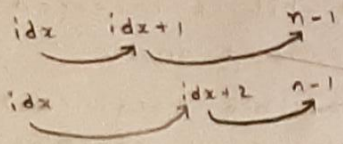
* Smaller problem.

$f(h, n, idx+1)$

Starting from idx+1 minimum cost to reach (n-1) index

$f(h, n, idx+2)$

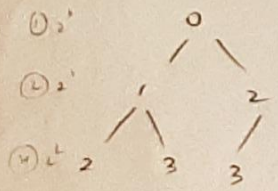
Starting from idx+2 minimum cost to reach (n-1) index



Smaller problem

* $f(h, n, idx) = \min \begin{cases} |h[idx] - h[idx+1]| + f(h, n, idx+1) & \text{op1} \\ |h[idx] - h[idx+2]| + f(h, n, idx+2) & \text{op2} \end{cases}$

Self work

* 

2^n

Basecase: If we reach the last index i.e. (n-1) → index the minimum cost to reach to index is 0 i.e minimum cost to reach n-1 to n-1 is 0.

```
if(idx == n-1)
    return 0.
```

→ and if we reach the last second index i.e n-2 we cannot find n-2 (idx+2) so return minimum cost is from op1.

Time complexity → Total no. of calls * Time taken from recursive tree

→ $1+2+4+8+\cdots 2^n$

$\frac{a(r^n-1)}{r-1}$

$\frac{1(2^{n+1}-1)}{2-1}$

$2^n \cdot 2^n - 1 = 2^n$

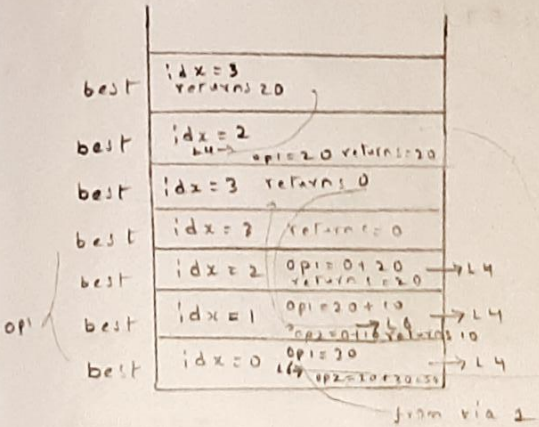
→ $2^n * c$

TC → $O(2^n)$

n = 4

h[] = [10, 30, 40, 20]

0 1 2 3



from via 1 minimum cost = 20

```
L1  *
L2  static int best(int[]h, int n, int idx){
L3      if(idx == n-1) return 0;
L4      int op1 = best(h, n, idx+1) + Math.abs(h[idx+1] - h[idx]);
L5      if(idx == n-2) return op1;
L6      int op2 = best(h, n, idx+2) + Math.abs(h[idx+2] - h[idx]);
L7      return Math.min(op1, op2);
L8  }
```

ASCII → American standard code for Information Interchange

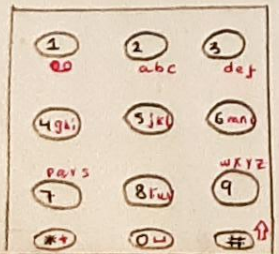
* It is an standard data encoding format for communication bwt computers
* Assigns standard values to letters, numerical, characters etc.

characters
- a-z : 97 to 122
- A-Z : 65 to 90
- 0-9 : 48 to 57

* char ch = '3';
  int chVal = ch - '0';
  = 51 - 48 → ascii code
  = 3
* To convert a given character to integer use minus (-) 0.

Problem -
Given a string containing digits from 2-9 inclusive, return all possible letter combinations that the number could represent. Return the Answer in any order.

ilp - digits = "23"
olp - ["ad","ae","af","bd","be","bf","cd","ce","cf"]

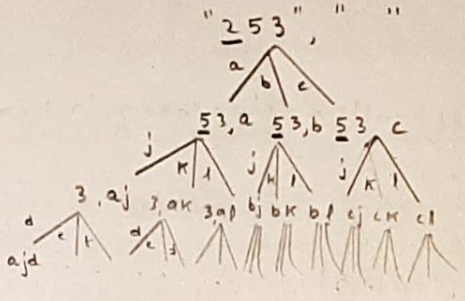


* "23"
  - 2 → a . (press 1 time)
    b .. 2
    c ... 3
  - 3 → d . 1
    e .. 2
    f ... 3

ad ae af bd be bf cd ce cf

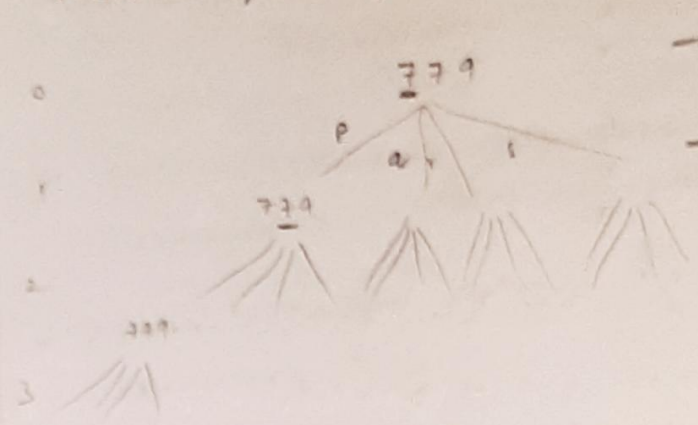
Total no. of combinations are → 3 × 3 = 9
→ 4 × 3 = 12

*



Algorithm -
* Declare string eg string digit = "23"
* Make an an Array of keypad i.e for values 2=abc, 3=def etc.
* extract the first digit which is character convert it to int. eg string digit = "23".
* current choices of currentdigit is in Array, Access it.
* Iterate a loop for current choices i.e till length of current choices.
* In loop recursive call i.e Access remaining digits using digit.substring(1) eg "23", and in current ans add first dig choices.
* Base case if (digit.length() == 0) Soul (ans) return;

Time complexity -



- 1 4^0

- 4 4^1

- 16 4^2

- 64 4^3

4^n

* levels :
how many digit
that many levels

$1 + 4 + 16 + 64 + \ldots 4^n$

$S_n = \Sigma 4^n$

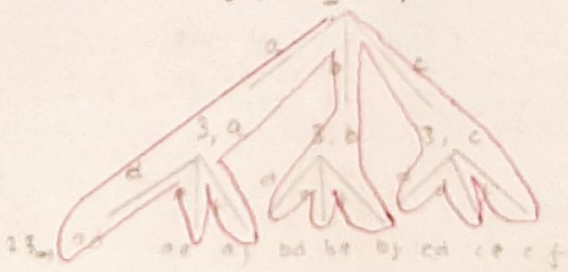
Tc = Total no. of calls * Time taken in 1 call

$= 4^n * (c+n)$

$Tc = O(n4^n)$

* Recursive Tree

S = "23", " "



2 = abc
3 = def