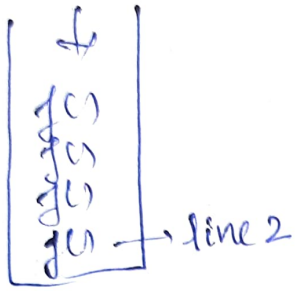
RECURSION — Striver

function call itself till a condition is reached

Prog → Functions

specified recursion

infinite Recursive → segmentation fault
↓
(Stack overflow)



last line completely executed
↓
till function call remains in the stack.

```
f()
print(count)
count++;
f();
```

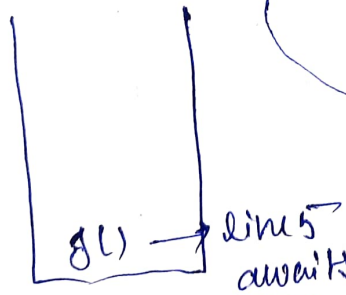
```
f() {
print(count)
count++;
f().
```

```
}
main() {
f();
```

→ modify

```
f() {
if(count == 4)
return;
print(count)
count++;
f()
}.
```

```
f() {
if(count == 3)
return;
X
X
X
}
```



Recursion Tree.

Writing function calls could be tiring so we have something called the "recursion Tree"

→ Recursive
→ Base case
→ Stack space → non-completed functions.
→ Recursion Tree
→ Segmentation fault (stack flow)
↓
accessing memory that doesn't belong to you.

Basic Recursion Problems —

Print Name N times.

```
void printName(int n, String name){
    if(n==0) →
(n==0) ← return;
    sout(name);
    printName(n-1, name);
}
```

Computer dynamic memory uses stack space.

1 → N (using i<n) (i+1, n) — using + — using −

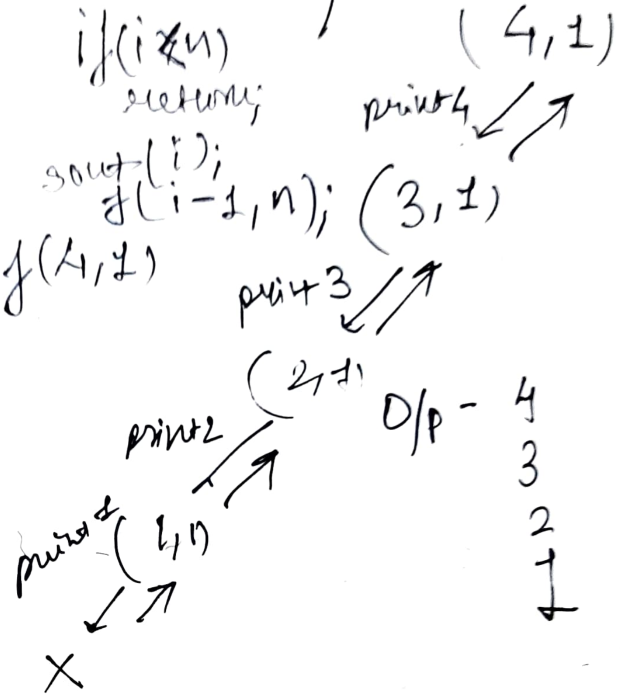
~~f(i, n) {~~
~~if (i > N)~~
~~return;~~
~~sout(i);~~
~~} f(i+1, n);~~

~~N → 1~~
~~f(i, n)~~
~~if (i < 0)~~
~~return~~
~~sout(i);~~
~~f(i−1, n);~~
~~f(4, 1)~~

N → 1

```
f(i, n) {
  if (i < n)
    return;
  sout(i);
  f(i-1, n);
f(4, 1)
```

→ **Recursive Tree**



O/p − 4
3
2
1

1 → N

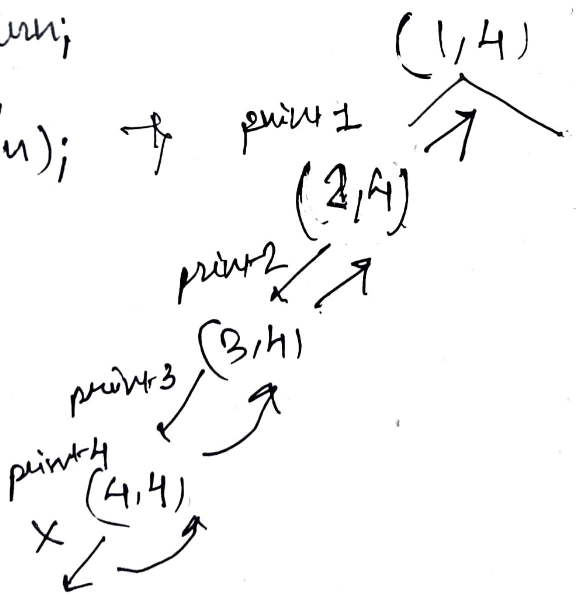
```
f(i, n) {
  if (i > n)
    return;
  sout(i);
  f(i+1, n);
```

→

f(1, 4);

O/p → 1
2
3
4

Recursive Tree



1 → N without (+) operator (BACKTRACKING)

```
f(i,n){
  if(i<N)
  return;
  f(i-1,N);
  cout(i);
}
```

i-N Recursion Tree →

(4,1) print(4);
print(3)
(3,1)
print(2)
(2,1)
print(1)
(1,1)

(1,1)
(2,1)
(3,1)
(4,1)
main()

This line will wait after the last function call call when (1,1) (0<1) then return

(3,1) print3
(2,1) print2
(1,1) print1
(0,1)

print(1) print(2)

N → 1 without (−) operator. (BACKTRACKING) → we have to do opposite (i+1,N)

```
f(i,n){
  if(i>N) return;
  f(i+1,N);
  print(i);
}
```

Call the function → print

f(1,1);
f(1,4);

(1,4)
print1
(2,4) print2
(3,4) print3;
(4,4) print 4
5,4 X

after (4,4) executed print 4,

f(5,4)
(5>4)
return

f(i,n){
print(i);
}

BACK TRACK

(1) Sum of first N numbers →

parameterised / functional

i, sum

```
f(i, sum) {
    (i<1) {
        print(sum);
        return;
    }
    f(i-1, sum+i)
}
f(5, 0);
```

f(5, 0)
f(4, 5)
f(3, 9)
f(2, 12)
f(1, 14)
f(0, 15) print sum, return;

Functional Approach →

return something rather than printing / giving back something.

```
f(n)
    if(n==0)
        ret 0;
    ret n+f(n-1);

main() {
    f(n)
```

(n=3)

3 + f(2)

2 + f(1)

1 + f(0)

3 + something

f(2) { if ✗ 2+f(1) } (3)

f(1) { if ✗ 1+f(0) } 1+0; (1)

f(0) { if ✓ return 0 }

O/p → (6) ans

2. Factorial of a number

parameterised → f(n, fact)

```
void f(n, fact)
    if(n==1)
        return 1;
    return n * f(n-1, fact);
```

f(3,1) → something.

(n, 1)
(3, 1)
3 * some
2 * some
1 * some
3 * 2 * 1
= 6 o/p

---

functional → f(n):

int → return type

```
f(n) {
    if n==1;
    return
```

Functional (3)

```
returns fact (Integer) {
    f(n) {
        if(n==1)
            return 1;
    return n * f(n-1);
}
```

3 → 2 → 1
(3x2) = 6
(2x1)
o/p.

parameterised

```
void mosty.          (3,1)
f(n, fact) {
    if(n==1) {
        print(fact);
        return;
    }
    f(n-1, fact*n);
}
```

(n==0) (ret 1) → works for 0 factorial

N==0. → 1
N==1 → 1

RECURSIVE TREE →

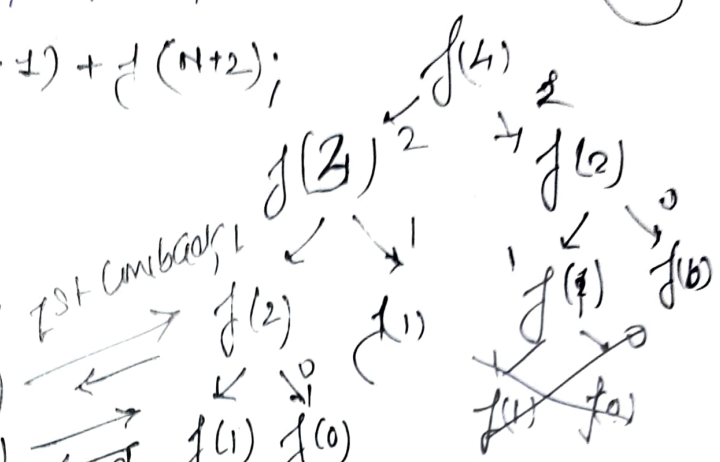
(3,1)
(2,3)
(1,6) → Ret 6 ← o/p

FIBONACCI NUMBERS —

0 1 2 3 4 5 6 ...
0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ... → (3)

$f(N) \rightarrow f(N-1) + f(N+2)$;

```
f(n) {
  if(n <= 1)
    return n;
  last  = f(n-1);
  slast = f(n-2);
  return last + slast;
}
f(4);
```



wait (·)

f(3) {
  if(); X
  last = f(2)
  slast = f(1)
  1+1
}

f(2) {
  if(); X
  last = f(1)
  slast = f(0)
  = 1+0
}

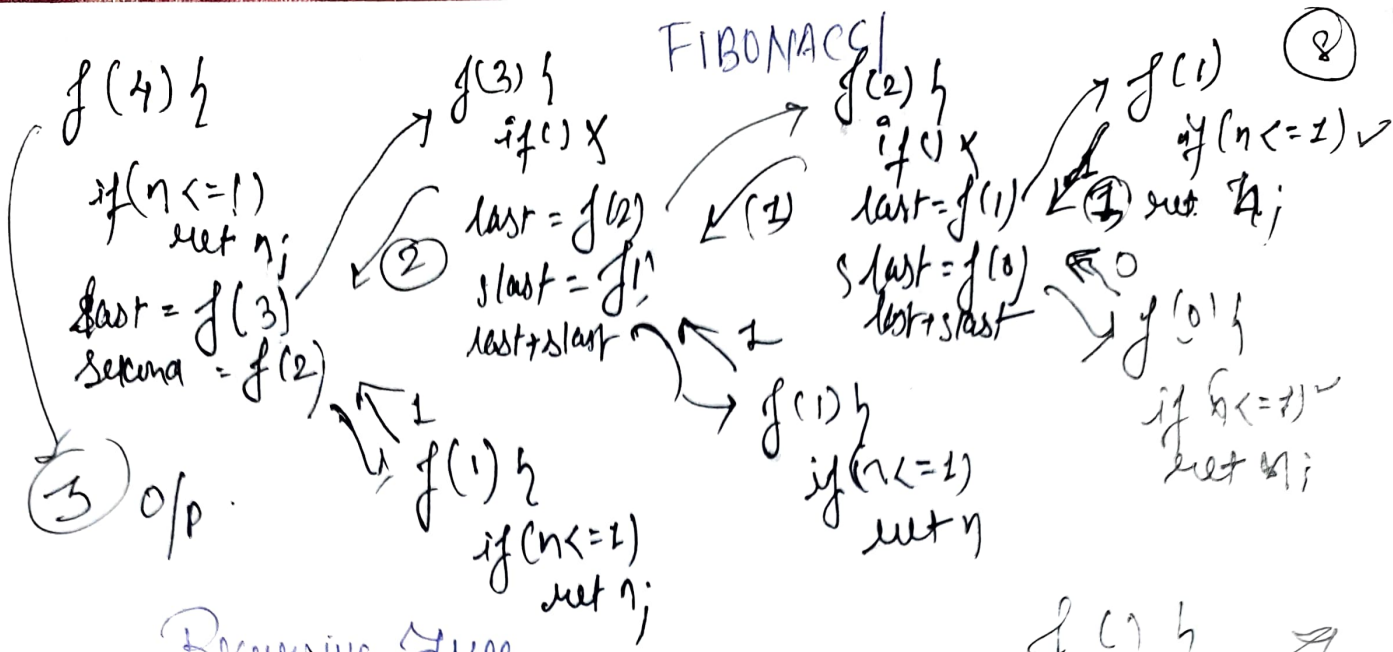
f(1) if(n<=1) return n;
1

f(0) if(n<=1) return n;
0

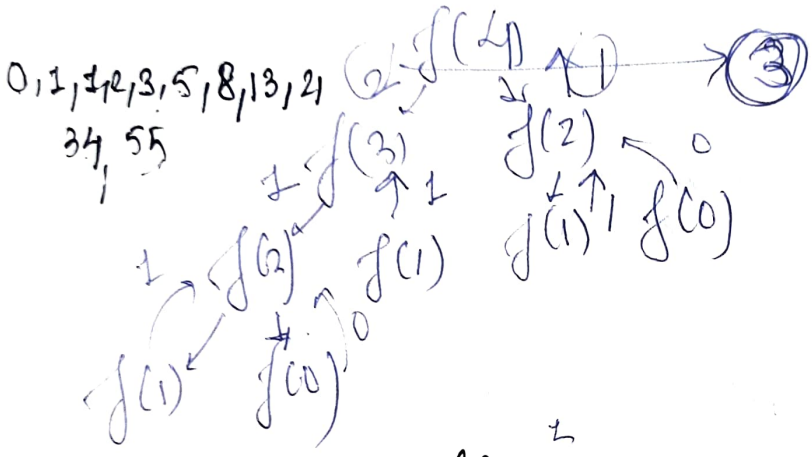
1

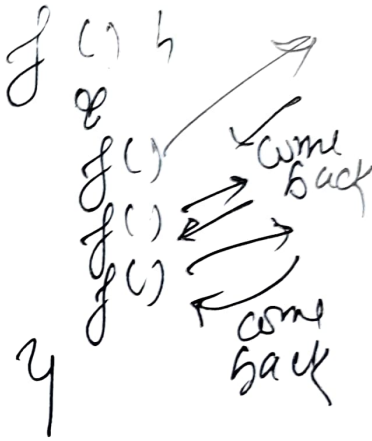
f(2

FIBONACCI



Recursive Tree





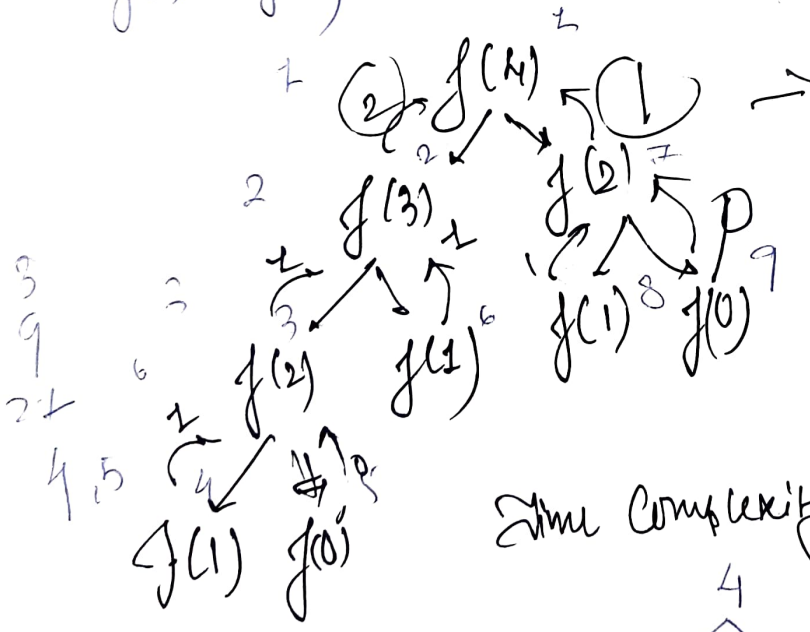
power of 3

if while(n%3==0) n=n/3;

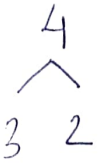
f(4)

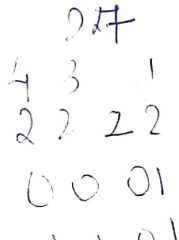
0,1,1,2,3.

(T) (1/3)



Time Complexity → (2^n) nearly Exponential





$24 = 16 = (9)$

Q. PRINT ALL SUBSEQUENCES! index changing from 0 to n

what is a subsequence?

contigous / non contigous sequence of chars which holds the order.

arr = {3,1,2}

TRIAL ROOM

f(index, [])

if(index >= n) {
print([])
return;
}

[].add(arr[i]); → take

f(index+1, []);

[].remove(arr[i]; → X take

f(index+1, [])

f(0, [3,1,2])

f(1, [3])

if (index >= n) X

[3].add[1]

f(index+1, [3])

remove arr[i]

[3,1]

f(2, [3,1])

if() X

[3,1].add[2]

f(index+1, [3,1,2])

f(3, [3,1,2])

(index >= n) ✓

print [3,1,2]

[3,1] remove(index]

f(3,1)

f(3, [3,1])

if() ✓

print [3,1]

[3]

f(2, [3])

O/P → 3,1,2

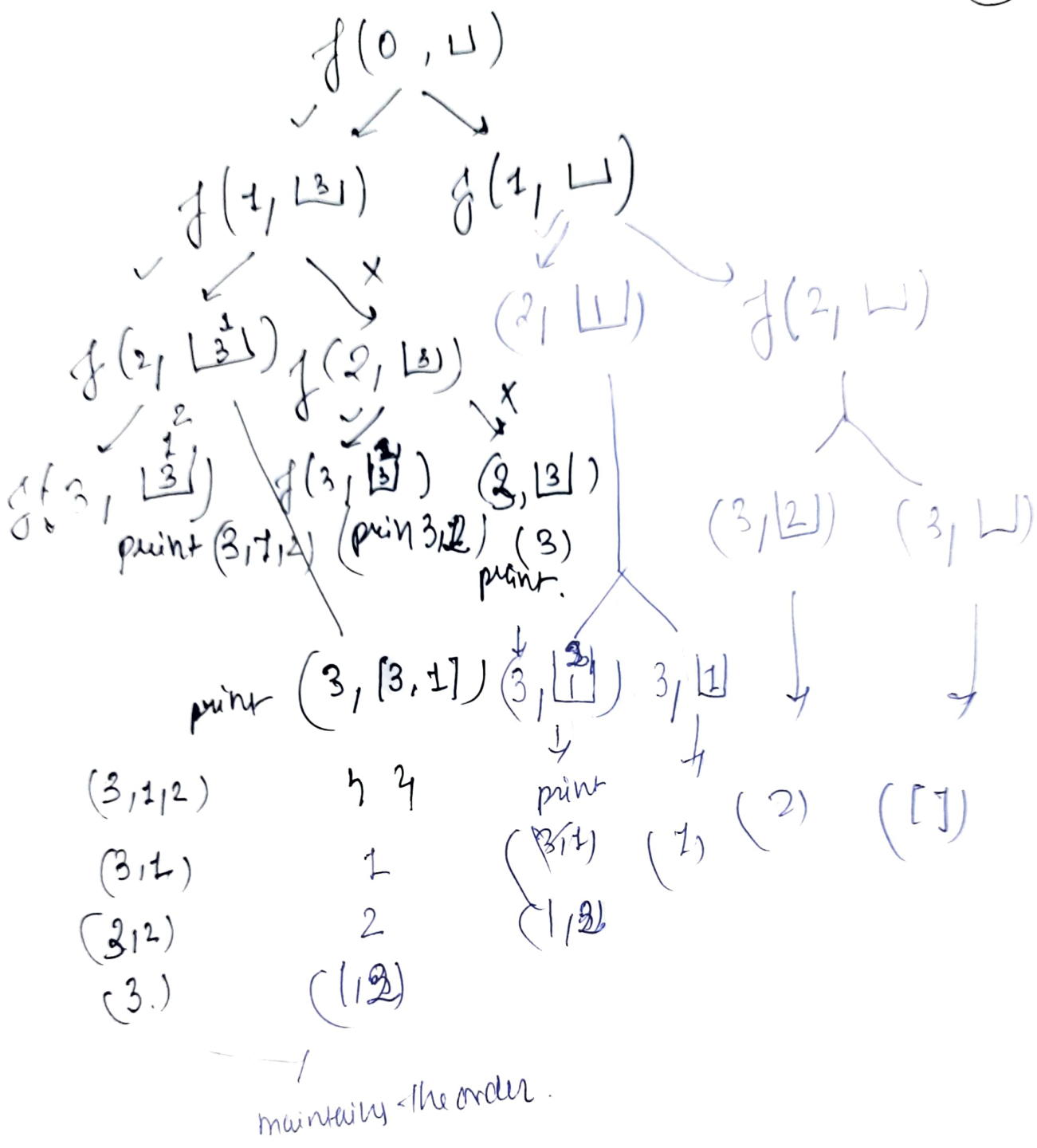
3,2

Trial Room

Pick a cloth →

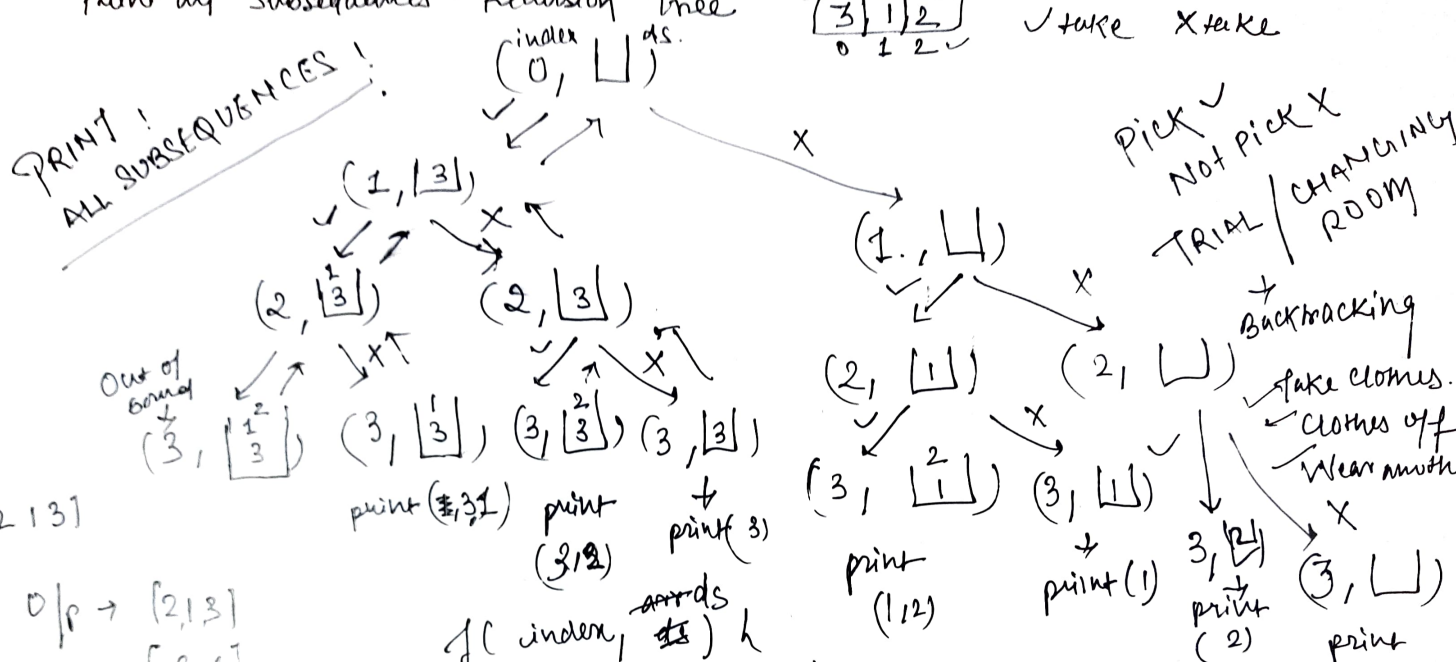
remove current clothes →

wear cloth


f(0, [])
f(1, [3])
f(1, [])
f(2, [3,1])
f(2, [3])
f(2, [1])
f(2, [])
f(3, [3])
print (3,1,2)
print 3,2
(3)
print.
print (3, [3,1])
(3, [2])
3, [1]
(3,2)
print
(3,1)
(1)
(2)
([])
(1,2)
(3,1,2)
(3,1)
(3,2)
(3.)
(1,2)
maintaining the order.

(11) Print all Subsequences Recursion Tree

PRINT ALL SUBSEQUENCES !

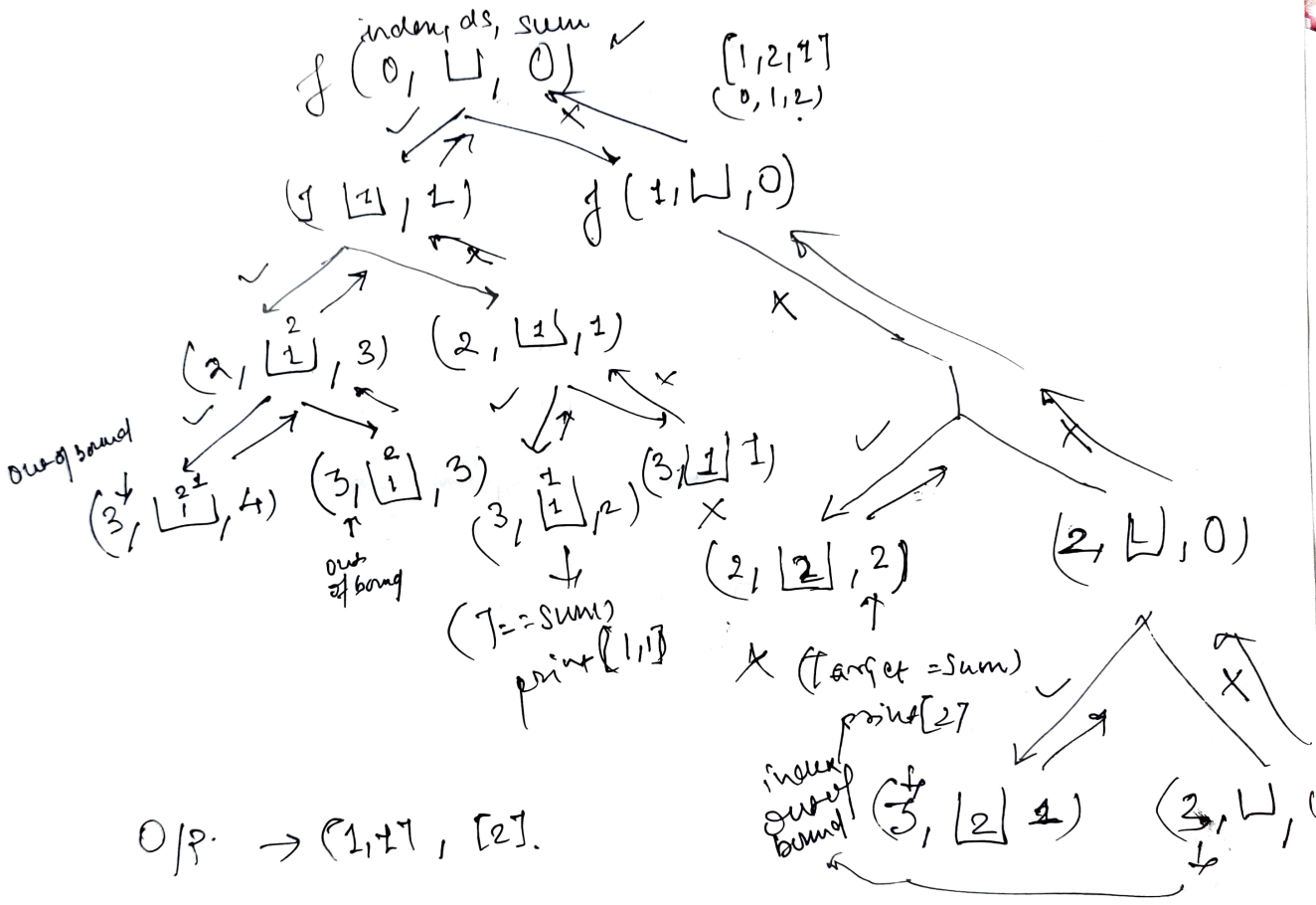


O/P → [2,1,3]
[3,1]
[3,2]
[3]
[1,2]
[1]
[2]
[]

```
f( index, ds ) {
  if ( index >= n ) {
      sout (ds);
      return
  }
  ds.add (arr[i]);
  f( index+1, ds)
  ds.remove (ds.size()-1);
  f(index+1, ds)
}
```

(index) ✗ work here

Q. PRINTING SUBSEQUENCES WITH SUM K !



O/p. → [1,1], [2].

Q. Subsequence with a sum (K) → Target

Pseudocode — [1,2,1] ∪ target ∘ specific condition

```
f(index, arr, ds, target, sum)
    // base case.
    if (index == arr.length || sum == 2) {
        print(sum);
        return;
    }
    ds.add(arr[index]); // pick the element
    sum += arr[index];
    f(index+1, arr, ds, target, sum);
    ds.remove(ds.size()-1);
    sum -= arr[index];
    f(index+1, arr, ds, target, sum);
```

Follow up: print only one subsequence (first)

Technique to print one number

```
f()
    base case
        cond ✓
        ret true;
    ret false
```

modify

```
return type to void () {
    if (condition) ✓
        return true;
```

pick → ds.add → f() == true → ret true; → ds.remove(ds.size()-1);

xpick → f() == true → ret true

if true ←

X call this ☺

else return false;
}

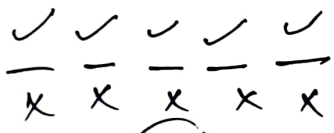
Q. Count the number subsequences with sum K.

int f(index, arr, ~~ds~~, target, sum) → 0 no need in count.

```
// base case
if (index == arr.length) {
    if (sum == target)
        return 1;
    return 0;          // else
}
// ds.add(arr[index]); // We alone have to carry ds here
sum += arr[index];
int left = f(index+1, arr, target, sum);
sum -= arr[index];
int right = f(index+1, arr, target, sum);
return left + right;
```

left call → int left = f(index+1, arr, target, sum);

right call → int right = f(index+1, arr, target, sum);



O(2^n)

O(2^n) O(2^n)

```
if (sum > target)
    return
optimisation if
array[i] > 0
```

print → parameter

print → 1 subsequence

count → ret 1; (base case) ✓

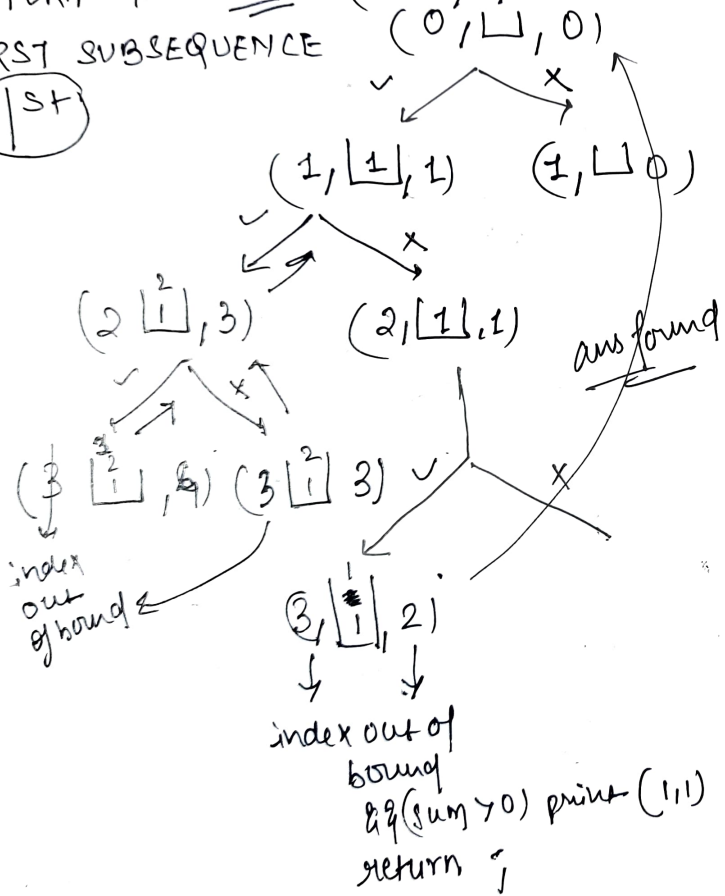
ret 0; ✗ (base case)

(15) Recursion tree | function call pseudocode

return(1 ans;) subsequence

RETURN type void

FIRST SUBSEQUENCE 1st



ans [1, 2, 1]
0, 1, 2

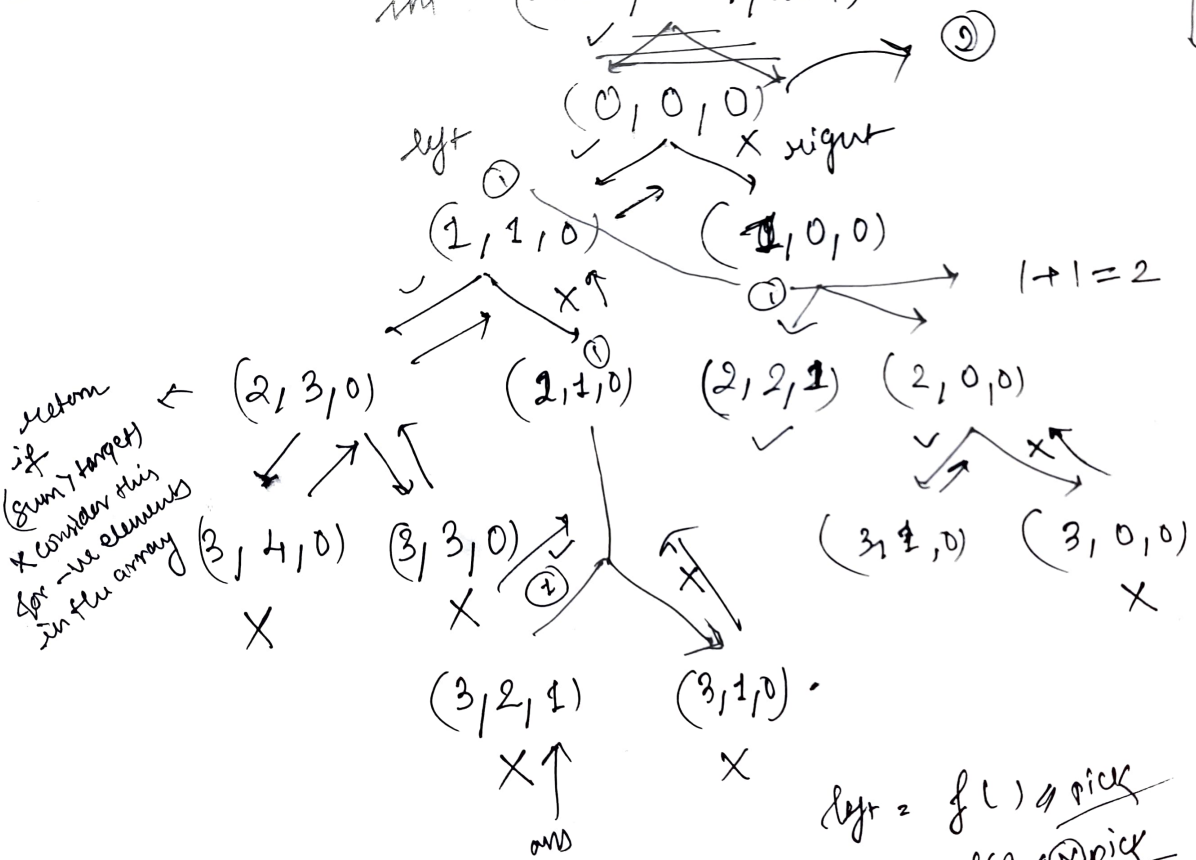
(sum == 2) return;

(16)

Recursion Tree for returning count of subsequences of sum(k) == 2

arr[1,2,1]

int (index, sum, count)



we don't have to store any thing

so no need of data structure

we want count

↓

INTEGER

USE FUNCTIONAL RECURSION.

TC - $O(2^n)$

Optimisation

if sum > target return 0;

left = f( ) // pick

right = f( ) // ~~pick~~

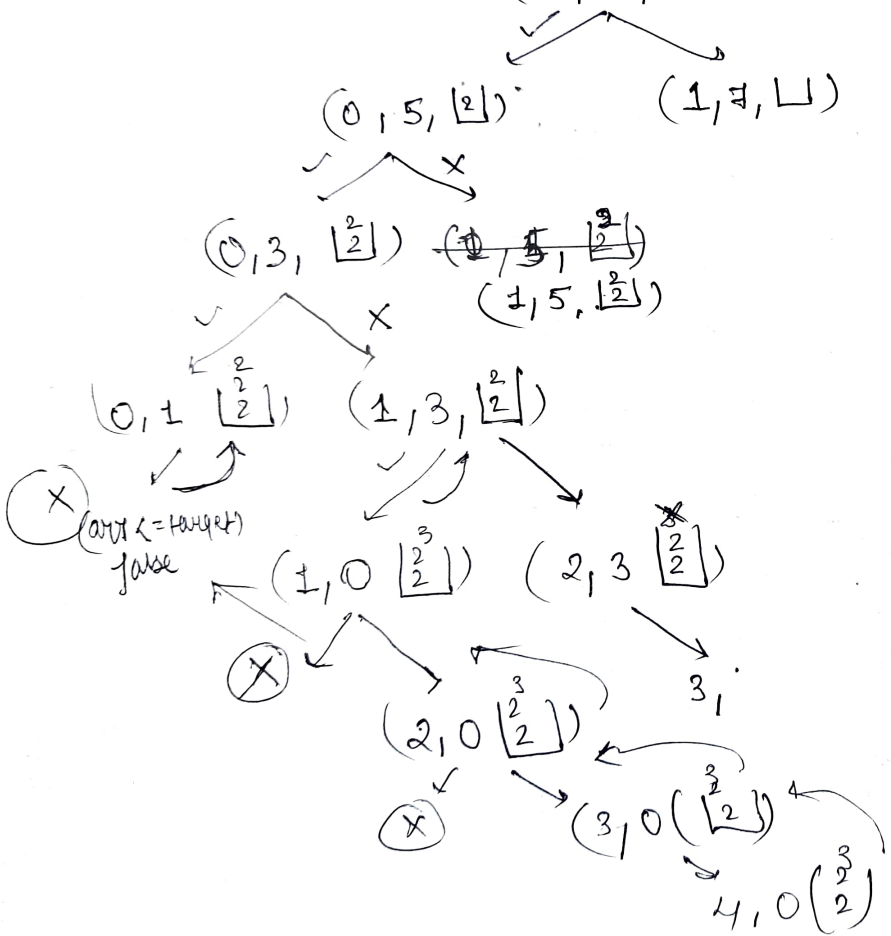
arr = {2, 3, 6, 7}

(index, target, ds)

additional list<List>.

✓ continue with same element (index)

✗ continue with next element (index)



Pseudocode for Combination Sum 1

```
recursive (int index, arr, ds, res, target) {
    if (index == arr.length) { // base case
        if (target == 0)
            res.add(ds);
        return;
    }
// pick the current element (index, target -= arr[index])
    if (arr[index] <= target) {
        ds.add(arr[index]); target -= arr[index];
        recursive(index, ds, res, target);

// not pick (pick the element at next index
//          (index+1, target)
        ds.remove(ds.size()-1);
    }
    recursive(index+1, arr, ds, res, target)
}

main() {
    List<List<Integer>> res = new AL<>();
    combSum
    recursive(0, arr, new ArrayList<>(), res, target);
    return res;
}
```

Q. COMBINATION SUM 2. SORT THE I/P ARRAY

1st Intuition

Sort the given array.

try to go for looping from 0 to → len(arr)

check for duplicate elements, as we want to sort return only unique elements.

return if (target == 0) // base case

[1,1,2,2,2]

i = 0, i = 1, i = 2, i = 3, i = 4

(index+1, target - arr[i])

(index+1, target)

```
recursive (index, target, arr, ds, res) {
    if (target == 0) {
        res.add(ds);
        return;
    }
    // run a loop for checking combinations
    for (int i = 0; i < arr.length; i++) {
        if (i > index && arr[index] == arr[index+1])
            continue; // skip duplicate elements.
        if (arr[i] > target) break;
        // pick the element
        ds.add(arr[i]);
        recursive(index+1, target - arr[index], arr, ds, res);
        // not pick the element
        ds.remove(ds.size() - 1);
    }
}
```

[1,1,2,2,2] → [1,1,2,2]

if i is first element we pick irrespective of duplicates

not consider duplicates f(call)

only use index while calling function

arr, res);

f(call) → recursive(index+1, target, arr, ds, res);

// we have coob for this,

FIND UNIQUE SUBSEQUENCES WITH SUM (k)

1. SORT (ARRAY)
2. Iterate from index 0 → N
3. call rec once in each Iteration ✓

```
f(index, target, arr, ds, res) {
    // base case
    if (target == 0) {
        ds.add
        res.add(new AL<>(ds));
        return;
    }
    for (int i = index; i < arr.length; i++)
        if (i > index && arr[i] == a[i-1])
            continue;
        if (arr[i] > target) break;
        // pick element
        ds.add(arr[i]);
f(arr) → f(index+1, target - arr[i], arr, ds, res);
    }
```

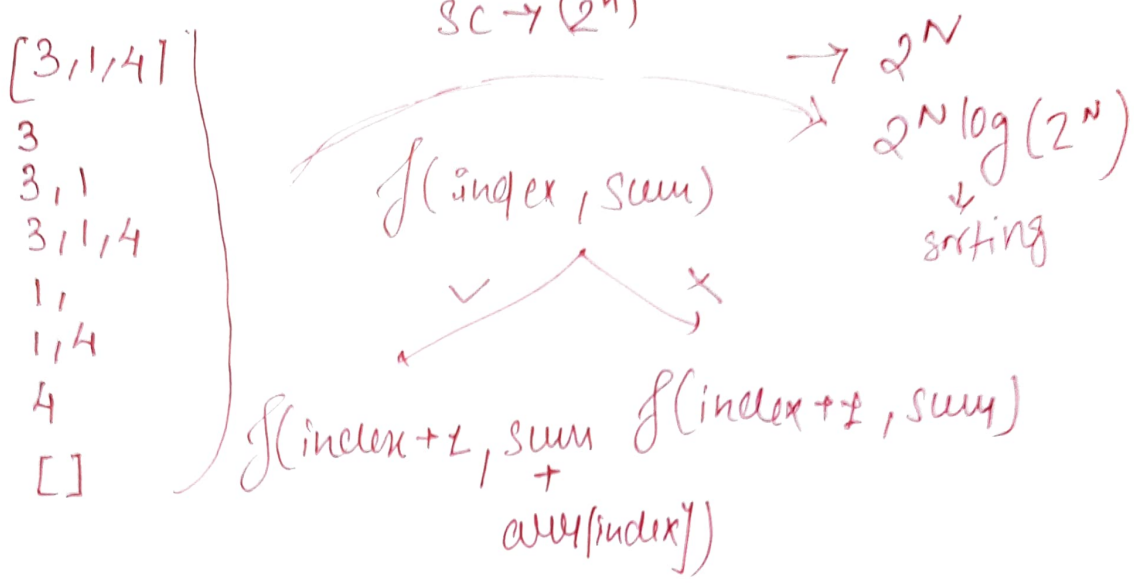
```
main()   arr = [1,2,1,2,2]   [target = 8]
func() {  Arrays.sort(candidates);
List<List<Integer>> res = new AL<>();
f(0, target, arr, new ArrayList<>(), res);
return res;
}
```

O/P → [1,1,6],
[1,2,5],
[1,7],
[2,6].

Q. Subset Sum 1 → print

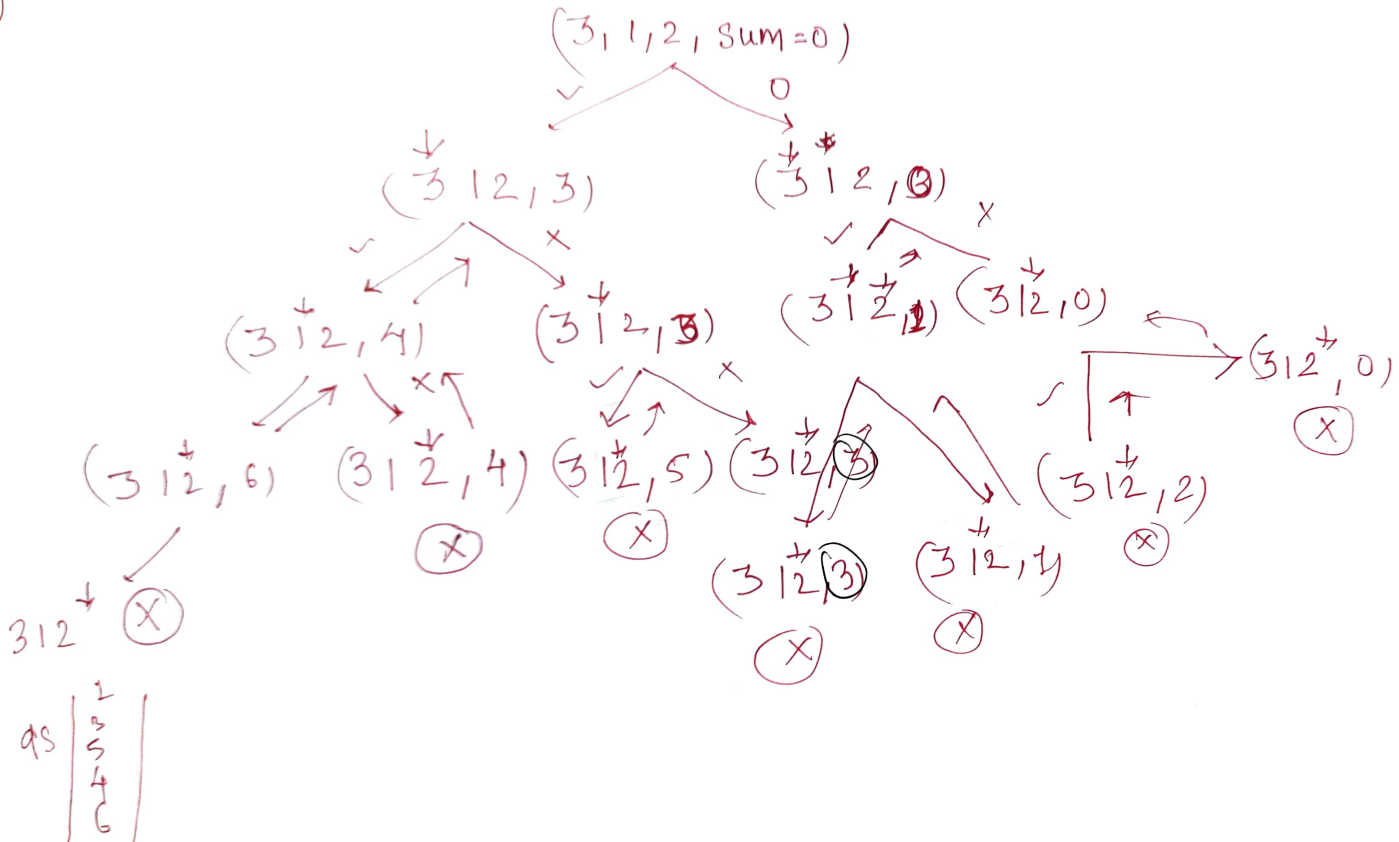
power set → $(2^n \times N)$ TC

SC → (2^n)




```
void recursive f(index, sum, arr, ds, res) {
    if (index == n) {
        res.add(sum); return;
    }

    //pick element.
    ds.add(arr[index]);
    recursive f(index+1, sum + arr[index], arr, ds, res);

    //not pick element
    ds.remove(ds.size()-1, sum, arr, ds, res);

}
    Collections.sort(res);

ret res;
```

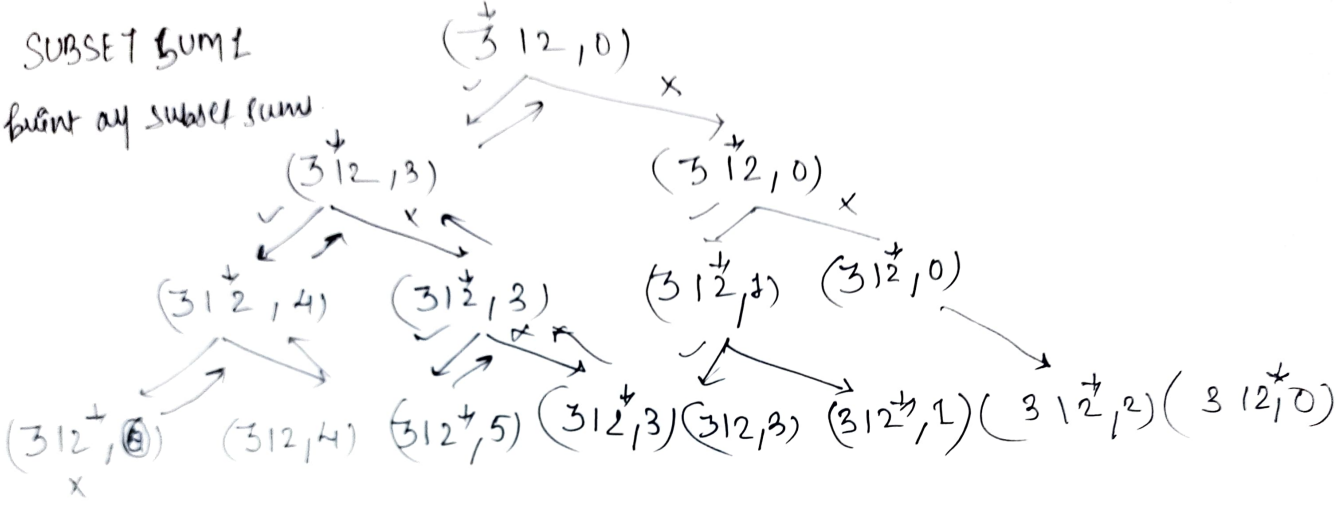
(2)


(3, 1, 2, sum=0)
(3 1 2, 3)
(3 1 2, 0)
(3 1 2, 4)
(3 1 2, 3)
(3 1 2, 1)
(3 1 2, 0)
(3 1 2, 6)
(3 1 2, 4)
(3 1 2, 5)
(3 1 2, 3)
(3 1 2, 2)
(3 1 2, 0)
(3 1 2, 3)
(3 1 2, 1)
312
qs
1
2
5
4
6

(32)

SUBSET SUM 2

Print any subset sums



0
2
1
3
3
5
4
6

2^n sorting $2^n \log 2^n + 2^n$

SC → $O(2^n)$

9. COMBINATION SUBSET 2. [1,2,2,2,3,3]

unique subsets.

```
f (index, ds)
if (index == arr.length) {
    ans.add(new AL<>(ds));
    return;
}
//pick element
for (int i = index; i < arr.length; i++) {
    if (i > index && arr[i] == arr[i-1]) continue;
    ds.add(arr[i]);
    f(index+1, ds)
    ds.remove(ds.size() - 1)
}
```

you dont need this

this is base case

pick 1st element no matter where i != index

you dont call here we already using loop.

have an arr.

(index → N)

$T \rightarrow 2^n \times n$

$SC \rightarrow 2^n \times O(n)$

Subset II Find power set of the given array having duplicate elements.

```
recursive (index, arr, ds, res) {

    for (int i = index; i < length(arr); i++) {
        if ( i != index && arr[i] == arr[i-1])
            continue; // skip duplicates.

        // pick element.
        ds.add(arr[i]);
        recursive (index+1, arr, ds, res);
        ds.remove(ds.size()-1);
    }
```

create a distinction array.

<bits/stdc++.h>

SUBSETS QUESTIONS → SUMMARY

Summary —

use Iteration inside recursion to go. (none

index → N

if array contains duplicates — possibility

consider a check

```
i != index && a[i] == a[i-1];
```

↓

consider 1st occurance, if duplicates

if not done

consider.

rest

for loop (index → N ~~-1~~)

```
ds.add(arr[i]);
func call ( index+1, arr, ds, res);
ds.remove (~~arr[i]~~ ds.size() - 1);
```

// done forget to add (ds) to result

```
res.add(new ArrayList<>(ds));
```

```
// if DUPLICATE
Arrays.sort(nums);
ELSE
```

↓ NO

TC → $2^n \times n$ → Duplicates

SC → 2^n

Unique

TC → 2^n

SC → 2^n

Count number of permutation

Two approach - with space O(n)
without space O(1)

n! = 3! = 6

[1,2,3]

base case
when (n == ds.size())

```
f(ds, map)
    (ds.size() == n) { res.add(ds); return; }
loop (0 -> N)
    if (arr[i] ! map contains)
        ds.add(arr[i])
        map[i] = 1
```

```
if(!map[i]) {
    ds.add(arr[i]);
    map[i] = true;
    recursive(arr, map, ds, res);
    ds.remove(arr[i]; // remove element from ds, Remark false in map while backtracking
    map[i] = false;
}
```

//no need of index as for loop will do the job

```
L<L<Integer>> = new AL<>();
boolean[] map = new boolean[n]   // i/p arr(len)
recursive(nums, arr, new AL<>(), res);
return res;
```

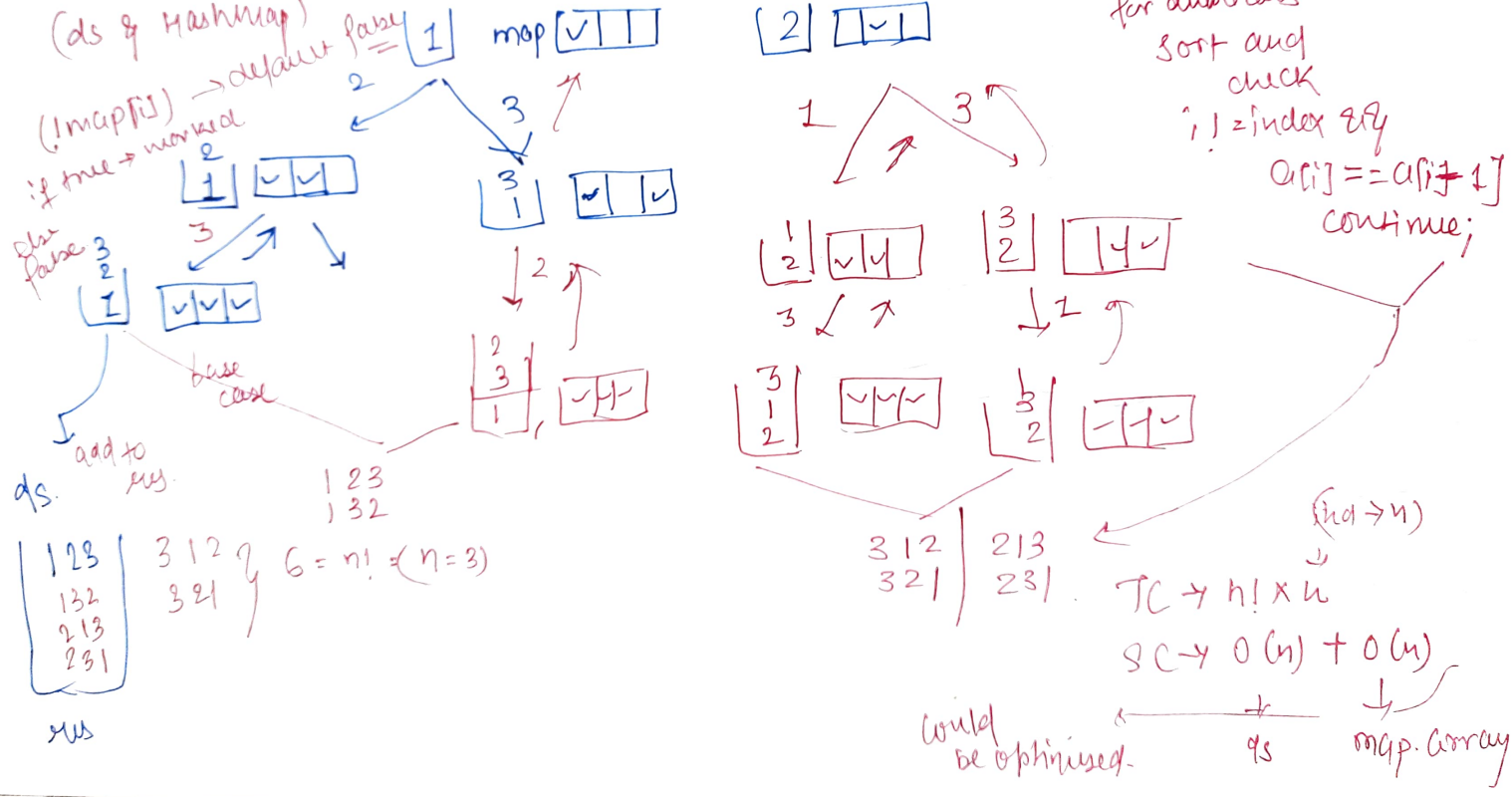
Recursion Tree for permutations

distinct integers.

while backtracking remove from (ds & hashmap)

(!map[i]) → default false = not marked

if true → marked



f(index, arr)

i → (ind → n-1) {

swap(a[index], a[i])

f(index+1, arr)

every number at particular index

if(index == n)
(base case)

TC → n! x (n)
+
perms

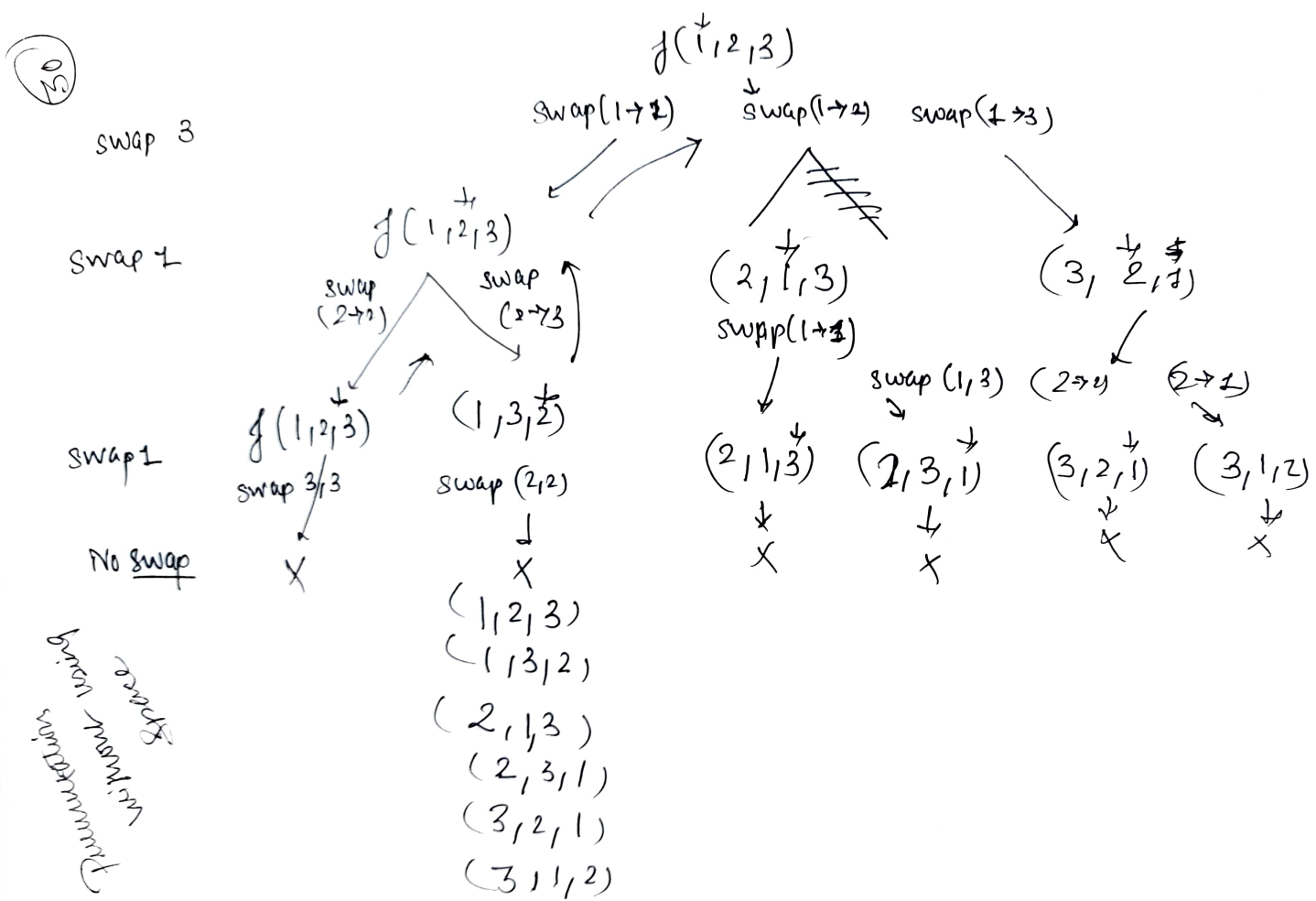
SC → O(1)

LEAST

LIST

O(N) → res

O(N) → recursive stack.

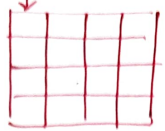

f(1,2,3)
swap(1→1)
swap(1→2)
swap(1→3)
swap 3
swap 2
swap 1
No swap
f(1,2,3)
swap (2→2)
swap (2→3)
(2,1,3)
(3,2,1)
swap(1→1)
swap (1,3)
(2→2)
(2→1)
f(1,2,3)
(1,3,2)
(2,1,3)
(2,3,1)
(3,2,1)
(3,1,2)
swap 3/3
swap (2,2)
X
(1,2,3)
(1,3,2)
(2,1,3)
(2,3,1)
(3,2,1)
(3,1,2)
Permutation using swapping

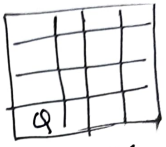
(31) N Queens

Place N queens in N x N chessboard

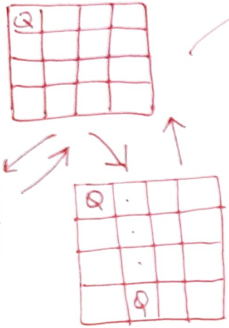
Such that No Queens Attack each other.

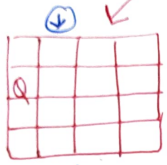
And

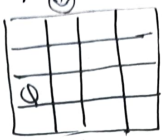


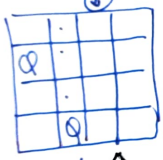


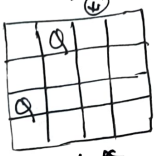
X This wont give any soln,

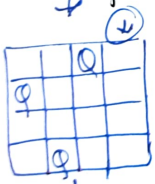


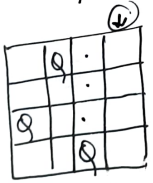


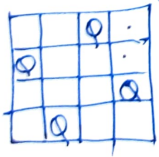




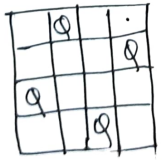




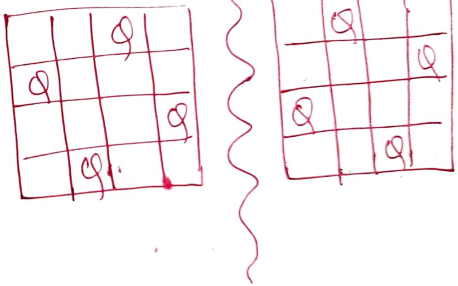




SOLUTION



SOLUTION



Mirror Soln for (N=4)

pseudo code for N queen.

```
f (col)
    if safe
    a[row][col] = 'Q';
    else
        = '.';
    f(col+1)
        a[row][col] = ' '; // Backtracking
```

```
void solve (col, ds, ans, int n)
    // base case
    when (col == n) {
        ans.add (new AL(ds));
        return;
    }
    // placing Queens  for (int i=0; i<
for(int row=0; row<n; row++){
    if (issafe (row, col, board, n)) {  → different
        board[row][col] = 'Q';
        solve (col+1, ds, ans, n);
        board[row][col] = '.';
    }
}

List<List<Integer>> res = new AL<>();
List<Integer> ds = new AL<>();
solve (0, board, ans, N)
    index
```