① Rabin Karp :-

* Way to calc hash ⇒ hash = 0;

"abcd" → 26·(26(26×0 + "a") + "b") + "c"

⇒ 26^2"a" + 26"b" + "c"

** hash = ((26 × hash) + str[i]) % q

i ⇒ [0 - (n-1)]

str[i] ↓ int value

q → a prime number to prevent overflow

* calculate next hash:

most_significant_hash_value = 26 ^ (pattern.size() - 1);

pattern = "xyz", m-s-h-v = 26 ^ (3-1) = 26^2

↳ calculate this in the begining of the code.

---

② Transform String :- Transform A ⇒ B by placing any of A's chars infront of B. Find min nums of operations

logic :- i = n-1, j = m-1 (where n = $size_A$, m = $size_B$)

check if both A and B have same num and freq of chars.

After that, while (i ≥ 0, and j ≥ 0) {

if A_i ! B_j then reduce i and increase ans ← [while (i ≥ 0 & A[i] != B[j]) { i--; res++; } → else dec both.

if (i ≥ 0) { i--; j--; }

---

③ Min swap for Bracket Balancing:

Balanced bracket ⇒ S_1 [S_2] S_1, S_2 can be empty

form 1 ⇒ [] form 2 = [[]]

→ S_1 and S_2 both empty

→ S_1 empty and S_2 = []

Using these 2 forms any form can be made.

example ⇒ [][] S_1 = [], S_2 = empty

[][][] → S_2 = empty

↳ valid → valid

valid + valid = valid

Logic: Test case ⇒ "]]]][][][" (indices 0 1 2 3 4 5 6 7 8 9)

a) calculate the position all the "[".

position ⇒ |4|6|8|9| | | ; set count = 0 ; p = 0, pointer for position arr.

loop i=0 i=n-1

b) Traverse the array, if str[i] = "[" count++;

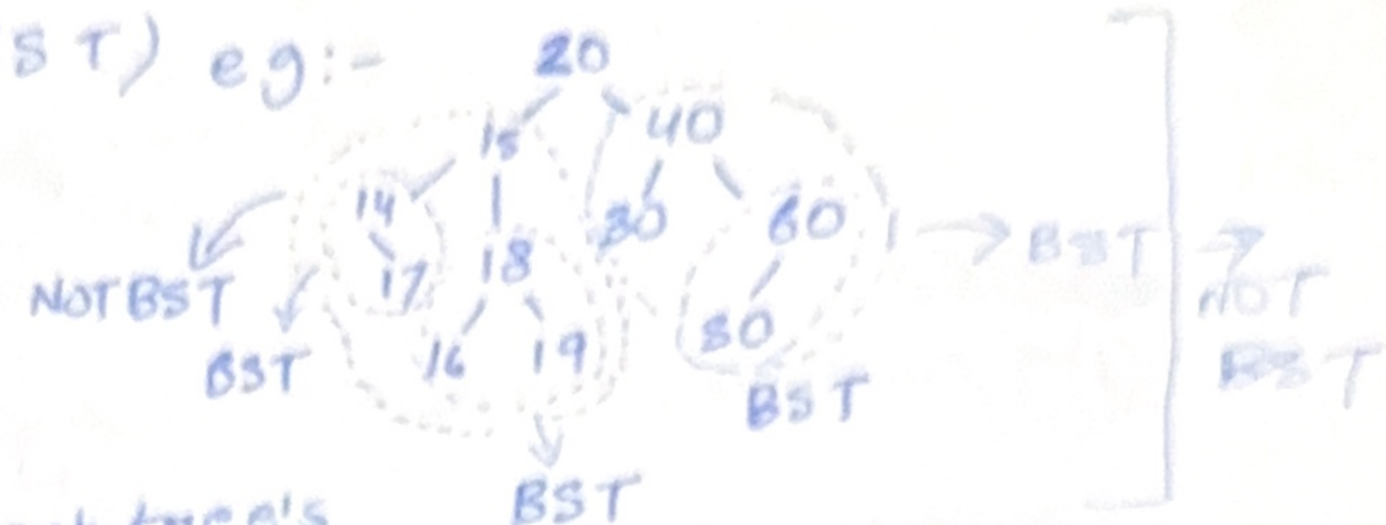
if str[i] = "]" count --;

c) if the count < 0 swap (str[i], str[pos[p]]).

and ans += (pos[p] - i) and count = 1.

** also if str[i] = "]" then count ++ and p ++.

④ Largest BST :- find the max nodes in a valid BST (if a BST) eg:-



o/p ⇒ 4

Brute force:
check if a node is BST. Find length.

Intution:- For BST left subtree's largest node < node.data and right subtree's mn element > node.data.

① Create a class/array sol(mx, mn, count), mx: max element of that subtree
mn: min ele of that subtree.
count: max node of BST in that subtree.

② if (node == null) sol(MIN, MAX, 0)

③ if (lst.max < node.data < rst.min)
sol(max(lst.max, rst.max, node.data), min(..), 1+l.count +r.count),
with min

④ else sol(MAX, MIN, max(l, l.count, r.count));

---

⑤ Arrange elcment Alternatly :-

Array is sorted. [1 2 3 4 5 6] o/p [6 1 5 2 3 4]

Brute force (i=0, J=n-1) (int flg=0)

This approach uses O(n) space.

```
while (k < n)
  if (flg is even) {
    new_arr[k] = arr[j]
    j--, k++ }
  else {new_arr[k] = arr[i]
    i++, k++ }
```

with O(1) space.

store 2 element in one address.

a[i] = a[i] * MAX + a[j] ⇒ insert x

a[i] ⇒ x / MAX

a[j] ⇒ x % MAX

in this case MAX ⇒ a[n-1]+1 → as array is sorted. so a[n-1] is max element

Above:

```
if (flg is even)
  a[k] = a[k] * max + (a[j] % max); j--; k++
else a[k] = a[k] * max + (a[i] % max); i++; k++;
```

same approach :- k=0, i=0, j=n-1 max = a[n-1]+1

for(k<n, k++)

k is even { a[k] = (aj % mx) * mx + (ak % mx)
j--

k is old { a[k] = (ai % mx) * mx + (ak % mx)
i++

for(i=0, i<n, i++)

a[i] /= mx;

---

⑥ Kth smallest element in row/column wise sorted array.

a) create a min heap. b) store the first row in the heap as (arr[r][c], r, c)

c) on every itration, pop the smallest element in the heap. and ~~decrease~~ insert element at (r, c+1).

d) Do this for K itrs. next element to be popped will be ans

---

⑦ Minimum Number of Platform:

Arr : 900 940 950 1100 1500 1800
Dep : 910 1200 1120 1130 1400 200

M1 Sort array base on arrival time. If lst.get(i-1).dep < lst.get(i).arr, then a current train can come on the same platform. If not train cannot come on this platform.

Problem :- If above condition is false we are creating new platform without checking if placing this train is possible of previous platform.

To resolve this we will use PriorityQueue. which will store departure time of train on every platform in asc order.

If a train cannot be placed at pq.peek() then it cannot be placed on any other platform. If it can be, we do pq.pop() and pq.add(lst.get(i).dep); ⇒ O(n log n), O(n)

M2 (without space) Merge both arr[] and dep[] arrays. And sort them in asc order. (keep ARR, DEP tag maintained).

900_A 910_D 940_A 950_A 1100_A 1120_D 1130_D .....

For every arrival inc platform count and every departure reduce platform count. (keep track of max platform)

For $(940_A \ldots 1100_A)$ we have 3 platforms. How are we sure that 3 platforms are needed. This is because if 940_A departs at 942_D then we would have something like $(940_A\ 942_D \ldots 1100_A)$ and on encountering $942_D \Rightarrow$ platform--;

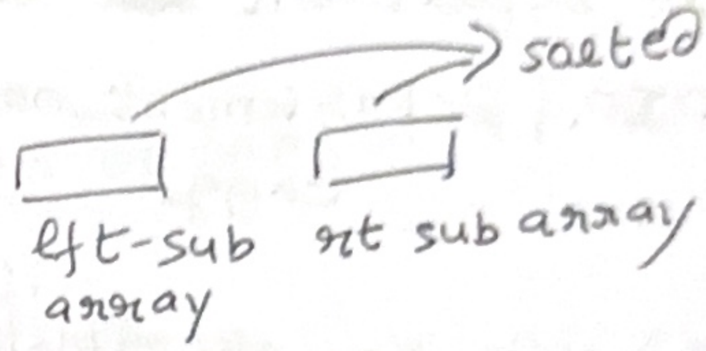
---

* Above Question has 2 variation. (a) How many $\underset{min}{p}$ platform required to perform all the tasks.
(b) How many max trains can come on the same platform.

(a) $\Rightarrow$ above question (b) $\Rightarrow$ Max Activity / Activity selection.

(b); here we sort by dep/end in asc order. If last.dep < curr.arr trains++.

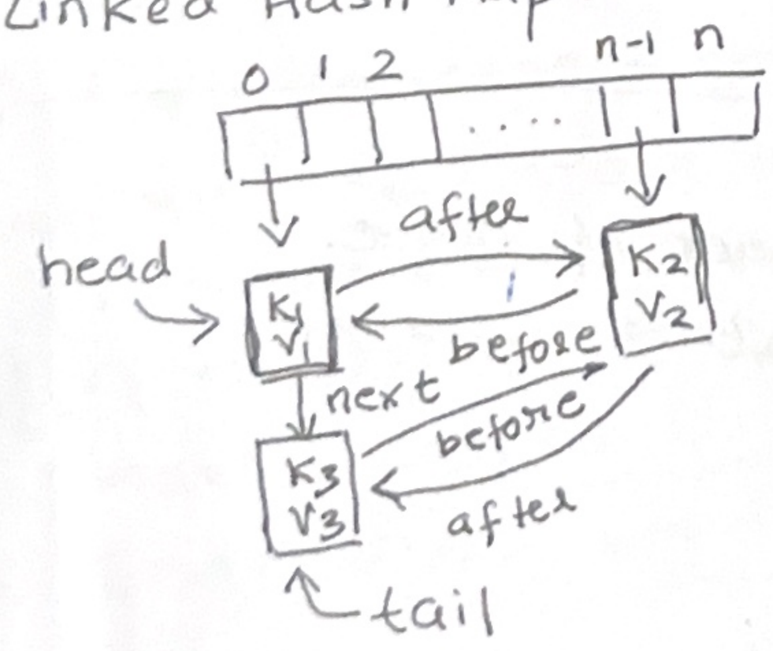
* Count inversions :- perform merge sort. if any element in rt[] is smaller than any element in lft[] then it is smaller than all the next elements in lft[].




```
if (rt[j] < lft[i])
    invCount += (lft.size - i);
```

* in collection framework, add(..), offer(..) are almost same. just that offer returns true. In case of remove(), if no element is present then throws error, poll() in case of no element is present doesnot throw error.

* Linked Hash Map :-



Entry {
key; value; nextPointerInLL;
after (node coming after the curr node);
before (node coming befor curr node)

* Tree Map: uses Red-Black tree.

1NF.- No column should have multiple values.

id	Location	Name
1	X	A, B, C
2	Y	D, E, F
3	Z	G, H, I

This table is not in 1NF.

Normalized form is

id	Loc	Name
1	X	A
1	X	B
1	X	C
⋮	⋮	⋮

2NF: All non-key attesributes should depend on primary key.

EID	employee name	department	Manager
1	A	Engg	B
2	C	Engg	B

→ will show up again and again.

Here department and [redacted] are not dependent on EID.

DID	department_name
1	Engg

EID	E_Name	DID	Manager
1	A	1	B

3NF: Non-key attributes should not be dependent on other non-key attribute.

EID	ENAME	state	city
1	A	KAR	BLR

EID	Ename	LOCID
1	A	2

LOCID	S	C
2	KAR	BLR

Max product subarray

```
for(int x : arr) { if (x < 0) swap(max, min);
                   max= max(max, max * x);
                   min = min(min, min * x);
                   ans= max(ans, max); }
```

where
max = MIN
min = MAX

Max element in all the windows of size k

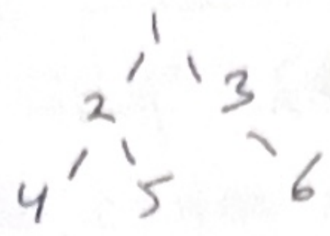
Use Deque. 2 criteria to pop element from back of Deque.

a) If it is less than the window indexes. start = 3, end = 3+K
   last element's idx = 2.

b) if front is smaller than current element.
   while (dq.peek() < ai) dq.removeLast();

ans = for every window return dq.peek() as an ans.

Preorder Traversal using iteration (without recursion):




```
solve (Node node){
    curr = node;
    Stack stk = ....;
    stk.add(curr);
    while ( curr || stk. is not empty ){
        while (curr) stk.push(curr); curr = curr.left;
        curr = s.pop();
        curr = curr.right; }
```

Reverse stack using recursion:-

* Need to use 2 recursive functions.

1) Recursive method to add given element in the bottom of the array.

```
fun( int x){ if (stk.is empty) stk.add(x);
             else { int a = stk.pop(); fun(x); stk.add(a); }}
```

2) another to pop the elements of the stack and call fun in rev order.

```
fun2( ) { int x = stk.pop(); fun2( ); fun(x); }
```

Minimum subarray length to sort so that complete array is sorted:-

① Take 2 pointer, starting from $i=1, j=n-2$

② i will traverse right and j will traverse left.

③ for i, keep track of maximum element, for j keep track of minimum element.

④ After ③ if, $a_i < MAX$, mark end index as i. $a_j > MIN$
mark begin as j.

⑤ Return end - beg + 1

```
for (i=1, i<n; i++){
    max = max(max, ai);
    min = min(min, an-1-i);
    if (ai < max) end = i;
    if (an-1-i > min) beg = n-1-i;
}
ans = end - beg + 1;
```

1 2 5 4 3 6
end
i
4 < 5
2 > min
4 > 3

Reaarrange char so that no adjacent chars are togethee:

① Use priority Queue to store pair (char, freq);
② Use a comparator which sorts based on freq.
③ Add all the elements (pairs) in PQ.
④ while (PQ.size() != 0) { curr = PQ.peek(); PQ.pop(); ← imp
Use this curr to populate the resultant string.

~~⑤ store this curr in the res check if previous~~
⑤ Check if previous pair value if not (-1). if no then store it in the PQ else move forward.
⑥ Do this untill PQ is empty.

Minimum height difference (reduce and increase by K)

given an array, reduce any element by K or inc any element by K. Return the minimum difference b/w max and min values.

① sort the array Arrays.sort(arr);
② for every (i), increase all the element from (0 ⇔ i-1) by K and decrease all the element from (i ⇔ n-1) by K.
Basically partitioning the the array into 2 subarrays

0 .... i-1	i n-1
→ inc by K no effect on max min	→ dec by K no effect on max min

smallest element from both arrays a[0], a[i]
greatest element from both the arrays a[i-1], a[n-1]

③ On every iteration min(a[0], a[i]) = mn max(a[i-1], a[n-1]) = mx
④ ans = min(ans, mx-mn);

```
Array.sort(arr);
for (i=0 ... i++) { ans = min(ans, max(a[i-1], a[n-1])
                                   - min(a[0], a[i])); }

return ans;
```

Kadanes algo on circular array :- Max subarray in a circular arr

```
for (int i=0; i<n; i++) {  → sum1 += arr[i], sum2 += arr[i];
    total_sum += arr[i];
    max_sum = max(max_sum, sum1);
    if(sum1 < 0) sum1 = 0;
    min_sum = min(min_sum, sum2);
    if(sum2 > 0) sum2 = 0; }
return (total_sum == min_sum) ? max_sum : (total_sum
                                           - min_sum);
```

all elements are negative
[-1, -2, -3 ... -n]

** intution : get min sum subarray using kadanes algo. subtract it from total sum to get the max circular array sum.

Gas station Problem :-

gas = [1, 2, 3, 4, 5]
cost = [3, 4, 5, 1, 2]

```
for(int i=0; i<n; i++){
    bal += gas[i] - cost[i];
    if(bal < 0){
        def += bal;
        start = i+1;
        bal = 0; } }
return bal + def >= 0 ? start : -1;
```

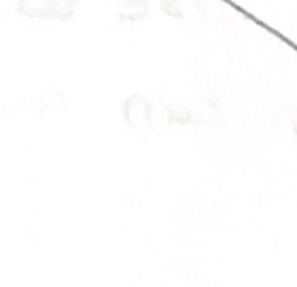
[-1]

initialize before for loop

2-0 >= 2

```
for(int i=0; i<n; i++)
int curr = 0, prev = 0;
    curr += (gi - ci);
    if (curr < 0){
        prev += curr;
        curr = 0;
        ans = i+1; }
return (prev + curr >= 0) ?
                ans : -1;
```

explanation :-



a circular array

going in clockwise direction from [a → e → d → c → b]
(a-e) -ve, (e-d) -ve and so on till (c-b).
now (b-a) is a positive integer. our task can complete if we can traverse (a-b) with what we have at (b-a) and we have already saved what we need for (a-b) traversal in "prev" var and (b-a) in "curr" var, so if (curr+prev) >= 0 (as it is neg then we are good

DSA Note

* Problem like n-sum (2-sum, 3-sum) where we need 2 elements from a sorted array to make a specific sum, it these problms we keep 2 pointers i and j. i=0, j=n-1 and inc i if expected sum is greater and dec j if expected sum is higher.

[1 2 3 4 5 6] K=(sum)=10

i j j

Where as in ~~problem~~ problems where we need a sum K and we need to make by consecutive elements of the array, in this case first thing have to do is to make cum sum of the array (making it sorted, like the above problem) once this is done and the arr is sorted the above solution won't work. The approach here is diff.

Take 2 pointers i=0, j=0, now increment j till $a_j - a_i \geq sum$ if equal to sum (i,j) is the answer. else if (sum >) then reduce window size by incrementing i, now if the sum is lower then increment j. Keep doing this until expected sum is reached.

but

* The 2 problem are little similar by the solution is different.

---

① kadanes algo: in a array of integers find max sum.

iterate over elements and ~~keep add~~ for each element keep adding them into var "sum", and then do max_ = max(max_, sum) to get max sum. if the sum anytime becomes negative make sum as zero

```
for each (arr) {  sum += arr[i];
                  max_ = max(max_, sum);
                  if (sum < 0) sum = 0;
}
ret max_;
```

② Circular Kadane : find max element in a list of integer.
circular

* Better way to implement Kadanes

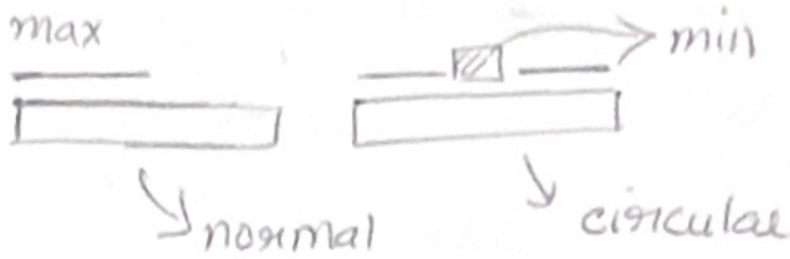
find max:

① — curr = max(curr + a_i, a_i)
max_ = max(max_, curr)

find min:

curr = min(curr + ai, ai); — (11)
min_ = min(min_, curr),

for circular Kadane,




```
circularKadane(){
    for(int i=0; i<n, i++){ normal = ①; circular = (11); total += ai;}
    return max(normal, total-circular);
}
```

③ Buy and sell stock I : find maximum profit by buying and selling stock any number of times.

* find minima element to buy, find maxima element to sell. Repeat this till end of array.

```
i=0; while(i<n-1){ while( ai >= ai+1) i++;
                    buy = i;
                    while(ai <= ai+1) i++;
                    sell = i;
                    //do something will buy and sell
                    }
```

④ Buy and sell stock II - max one transaction allowed

* same as ③ but this time rathar doing transaction any number of times we can do just one transaction.

* for any number at index i we need to find greatest numbers at index j (where j > i) as we need to sell on coming days)
example if we need to find max element at j it would be max(a_j ... a_n) = mx, now curr_ans = (mx − a_i), ans = max(ans, curr)

curr_greatest = MIN; ans = MIN;

for (...) { curr_greatest = max(curr_greatest, ai), —ⓐ

ans = max(ans, curr_greatest - ai); —ⓑ

}

ret ans;

example, [7, 1, 5, 3, 6, 4]

[7, 6, 6, 6, 6, 4] —ⓐ ← this is what ⓐ will do without using extra space

↓ ↓ ↓ ↓ ↓ ↓

0 5 1 3 0 0

=

* another method of doing the previous que via dp.

• whenever u buy profit = -buy + ... and whenever you sell

profit = +sell + ... → 1 ⇒ buy, 0 ⇒ sell

recursion ④

```
f(index, buyORsell) {
  if (buyORsell == 1) { ret max(-a_index + f(index+1, 0),
                                f(index+1, 1)) }   buy → not buy
  else { ret max(+a_index + f(index+1, 1),   → sell
                 f(index+1, 0)) }                not sell
}
```

memoization ⑤

```
f(idx, buyORsell, dp) {
  if(dp[idx][buyORsell] != -1) ret dp[idx][bORS];
  if(buyORsell == 1) {...}
  else {...}
}
```

same as above, also store in dp.

tabulation ⑥

```
dp[n][1] = dp[n][0] = 0; // buying or sell on "n" is not possible.
for(idn: (n-1, 0)) {
  for(buy: (0,1)) {
    if(buy) { ret max(-a_idx + dp[idx+1][0], dp[idx+1][1]); }
    else { ret max(a_idx + dp[idx+1][1], dp[idx+1][0]); }
  }
}
```

⑤ Buy and sell stock III / IV : can buy and sell only k times.
→ k will be generic.
→ k will be given as 2/3, some small value

same dp code as previous.

```
fun(idx, bos, k){
    if(idx == n || k==0) ret 0; // txns are over, reached const limit.
    if(dp[idx][bos][k] != -1) ret dp[idx][bos][k];
    if(bos){ dp[idx][1][k] = max(-a_idx + fun(idx+1, 0, k),
                                 fun(idx+1, 1, k));
    else{ dp[idx][0][k] = max(a_idx + fun(idx+1, 1, k-1),
                              fun(idx+1, 0, k))
                                  ↳ if selling txn will be reduced
    ret dp[idx][bos][k];
}
```

⑥ Buy and sell stock V : will cooldown, cannot buy immideatly after you sell example, you sold on idx = 7
you cannot buy on idx=8,

* same code example as ④ ⑧.

```
fun(idx, bos){
    if(bos==1){... } // same
    else{ ret max(a_i + f(idx+2, 1) , f(idx+1, 0)); }
}
```

↳ as we sold the stock, we will have to skip one index.

⑦ Buy and sell stock VI : fee will apply everytime you sell a stock.

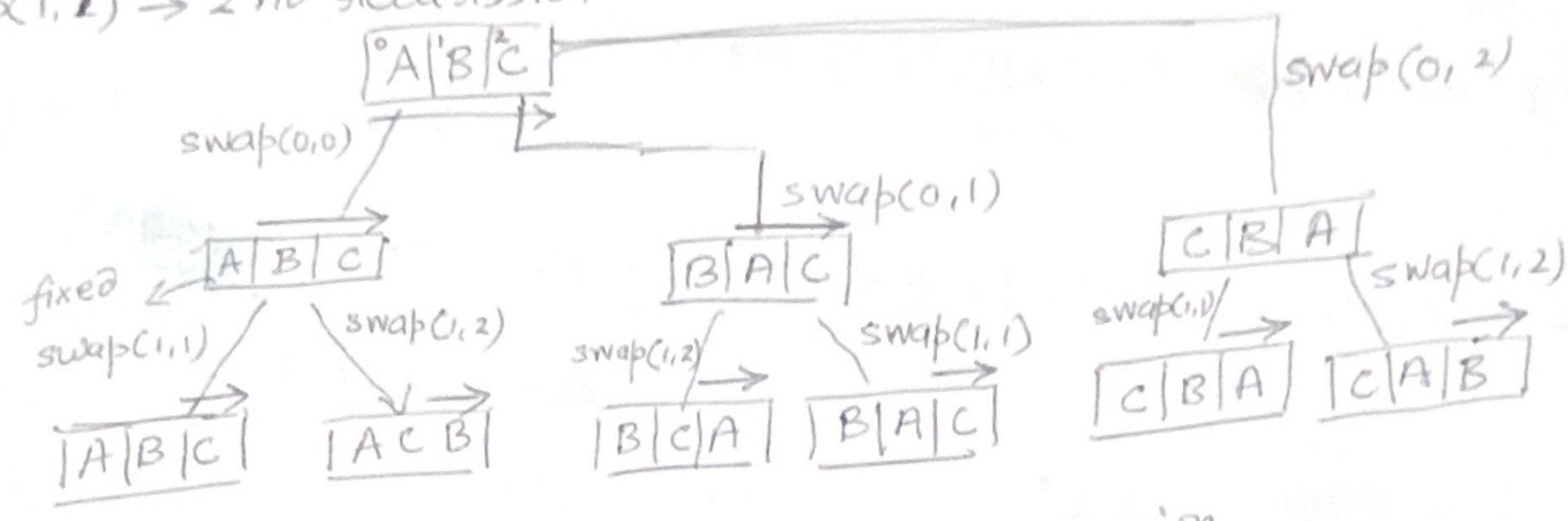
* same example as 4R

```
fun(idx, bos){
    if(bos){ ... } // same
    else{ ret max(a_idx - fee + f(idx+1, 1), f(idx+1, 0)); }
}
```

↳ a fee is applied when we sell.

⑧ All permutation of a string : abc ⇒ {abc, acb, bac, cab, cba}

* sort the string.
* start with "i" at 0. Inside function start a for loop.
  keep "i" fixed and for all "j" from (i, n) start swapping
  character swap(0,0) → 1st recursion, ~~swap(0,1)~~ →
  swap(1,1) → 2nd recursion and so on.



Ri ⇒ ith recursion
Li ⇒ ith loop

```
sort(str);
fun(str, i, container){
    if(i==n) container add(str);
    for(int j=i, j≤n, j++){
        swap(str_i, str_j); // updating str
        fun(str, i+1, contained);
        swap(str_i, str_j); // swaping elements back
    }
}
```

highest

⑨ *** Egg dropping puzzel : minimum moves required to find floor from which it will not break.

```
dp[n+1][k+1]; // n floors, k eggs
dp[i][1] = i // if there is 1 egg only all the floors need to be tested
dp[1][i] = 1 // if only one floor is present, only one egg is required.
```

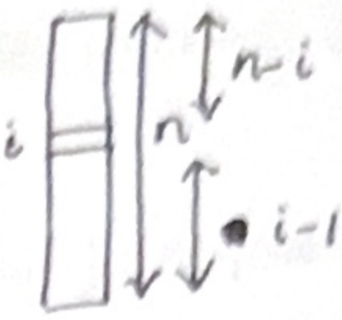
$$f(n,e) = \min_{x=0}^{n-1} \left(\max\left(f(n-x, e), f(x-1, e-1) \right) + 1 \right)$$

egg survived, egg broken

```
for(i:(2, n)){
    for(j:(2,K)){ int temp = MAX;
        for(x:(1, i)){
            int egg breaks = dp[x-1][j-1];
            int egg survives = dp[i-x][j];
            temp = min(temp, max(egg breaks, egg survive)+1);
        }
        dp[i][j] = temp;
    }
}
ret dp[n][k];
```

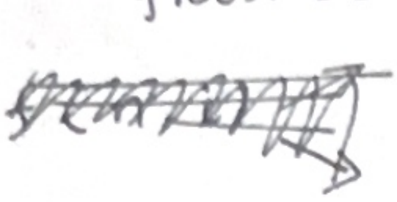
intution behind (9):

if we have "n" floor building there can 2 cases, if we drop an egg from some ith floor.



(a) → egg will break and we will have to check on floor below the current floors, i.e. (i-1) floors.

(b) → egg will not break confirming that egg wont break on any floor below it. Leaving us will (n-i) floors.



f(i, e) → f(i-1, e-1) : breaks
→ f(n-i, e) : doesn't breaks } again the same problems.

becomes new "n" becomes new "e"

if we have n floors and e eggs

we can start with 1 floor and 1 egg and go up to n floors and e eggs.

~~for(n: 1~~ for(i: 1→n){ for(j: 1→e){ ... } } ⇒ help us fill the dp table.

try at every floor ↘ try will all the eggs

* basic approach (recursive approach):

~~int~~ f(n, e){
if(n==0 || n==1) return n;
if(e==1) return n; // need to try all the floors
for(i: (1→n)){ // trying all floors one by one
int temp = ~~int~~ max(f(i-1, e-1)+1, f(n-i, e));
ans = min(ans, temp);
}
return ans; }

---

* if you reach a number "n" by muliplying it by 2 or dividing it by 2 then it take log n time.

for example ⇒ (1 → 512) ⇒ 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 [9 iter] start end

⇒ log 512 = 9 ⇒ log(n)

same goes with (512 → 1) ⇒ 512, 256, 128 .... 1 [9] ⇒ log(n).

example 2 ⇒ for(i → (1→n)){ for(j: (2, n, j=j*2)){ ... } }

O(n log n)

* if we reach any number "n" by multiplying or dividing by K then complexity is $(\log_K n)$.

⑩ Combination Sum: find lists of element from a list of element where sum of element of all the list is equal to a target sum.

example: [2,3,6,7], t = 7 res = [[2,2,3],[7]]

* Not a DP problem but idea is similar, take/not take, include/exclude
* Take ith element reduce t and move, don't ith element and move.

~~f(list, i){~~ f(list, i, t){

if (t==0) add to res. // add sublist to main ans
if (i==n and t>0) return;
if (t<0) return; // t goes negative
f(list, i, t-ai); // take, also add element to sublist
f(list, i+1, t); // dont take

⑪ Combination Sum ii: elements can be use at most once, unique combinations only.

same as ⑨, + f(list, i+1, t-ai);
f(list, i+1, t);

if(t==0) add sublist to a set

in both these case we will move to the next element as elements can be use only once

⑫ Combination sum iii: use numbers [1...9] to make sum "n" while using only k numbers.

example: n=7, k=3 [[1,2,4],

* start from a current element =1, if you want to use it subtract this element from the desired sum "n". and add it to this subans list.

```
f(curr, k, n, ans, subans){
    if(k==0){
        if(n==0) ans.add(subans);
        return;
    }
    if(curr > 9) return; // this is before the previous
                         // if to run this case, [n=45, k=9]
    subans.add(curr);
    if(n >= curr) solve(curr+1, k-1, n-curr, ans, subans);
    subans.remove(subans.size()-1);
    solve(curr+1, k, n, ans, subans);
}
```

* If you want traverse back in a tree (a directed graph), example here you want to go from 3 to 1, you will most probably want to create a directed graph out of this tree.

something like this, [1 → [2,3], 2 → [1], 3 → [1]], this can be done like this

```
f(parent, node){
  adj(parent → value, node → value);
  adj(node → value, parent → value);
  f(node, node → left); f(node, node → right);
}
```

* When solving matrix path problems, rather than writing condition like this f(i+1, j), f(i+1, j+1) .... f(i, j-1) create 2 arrays / list like this,

dx = {1, 1, -1, -1, 1, -1, 0, 0}; dy = {1, -1, 1, -1, 0, 0, 1, -1}

and write a for loop to get all the directions.

* create a character array from a number, example, 1234567 →

['1', '2', '3', '4', '5', '6', '7'] = char[] charArr.

⇒ `char[] charArr = ("" + num).toCharArray();`

* When working DP problems like stair climb or coin change where stairs or coins can be used multiple time in same solution for loop is good, for example, f(n){

```
for(int i=0; i<m; i++){
  ans += f(n-ai);
}
return ans;
}
```

This will give non unique solutions like, 3 = 2+1, and 1+2

If we want unique solutions like 3 = (2+1 OR 1+2) in that case one variable called start should also be included in the method f(...)

```
f(n, start){
  for(int i=start; i<m; i++){
    ans += f(n-ai, i+1);
  }
  return ans;
}
```

// Need to try this solution but it looks fine.

need to work on a for loop solution.

or we can also use include/exclude technique.

```
f(n, i){ → // base condition
  int ans = 0;
  ans = f(n-ai, i+1) + f(n, i+1);
  return ans;
}
```

for unique solutions like 3 = (1+2 OR 2+1)