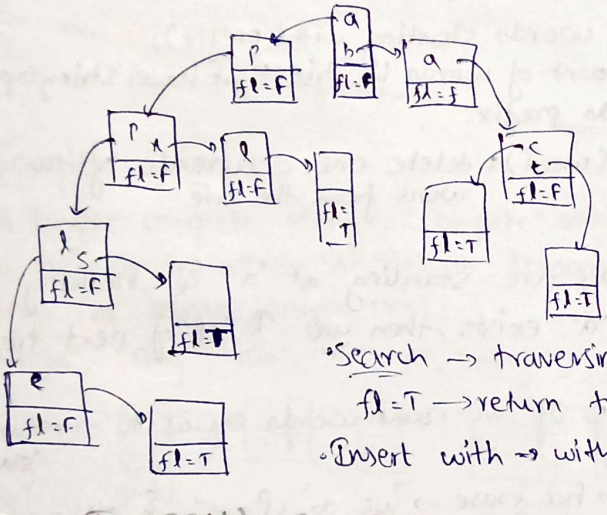
TRIE {DSA}

+) Implement Trie (Prefix Tree)

A 'trie' is a tree D.S used to efficiently store & retrieve keys in a dataset of strings. (Application → autocomplete, spellchecker)

- Trie() → initializes the trie object
- void insert(String word) → inserts string 'word' into trie
- bool search(String word) → returns 'true' if string 'word' is in 'trie'
- bool startsWith(String prefix) → returns 'true' if prev. inserted string 'word' that has the prefix 'prefix' & else 'false'.

Ans) ex: apple, apps, apxl, bac, bat → we make a 'trie' array [26] & 'bool' fl.



- If ele. not present, then add to trie & create a new trie with fl=false. Repeat
- If end of word reached, make 'fl' as true.

- Search → traversing above trie & if we got fl=T → return true, else false.
- Insert with → with we are able to traverse → true.

Code: → $T_c = O(N)$, $S_c = O(N)$

```
struct Node{
    Node* links[26];
    bool flag = false;
    bool containsKey(char ch){
        return (links[ch-'a']!=NULL);
    }
    void put(char ch, Node* node){
        links[ch-'a'] = node;
    }
    Node* get(char ch){
        return links[ch-'a'];
    }
    void setEnd(){flag=true;}
    bool isEnd(){return flag;}
};
```

⊛

```
Node* root; → initially every trie will have a root
Trie(){
    root = new Node();
}
void insert(string word){ → Tc = O(N)
    Node* node = root;
    for(int i=0; i<word.size(); i++){
        if(!node->containsKey(word[i])){
            node->put(word[i], new Node(1));
        }//moves to reference trie
        node = node->get(word[i]);
    }
    node->setEnd();
}
```

⊕

```
bool search(string word){ // T_C = O(N)
    Node* node = root;
    for(int i=0; i<word.size(); i++){
        if(!node->containsKey(word[i])){
            return false;
        }
        node = node->get(word[i]);
    }
    return node->isEnd();
}
```

```
bool startsWith(string prefix){
    Node* node = root;                    // T_C = O(N)
    for(int i=0; i<prefix.size(); i++){
        if(!node->containsKey(prefix[i])){
            return false;
        }
        node = node->get(prefix[i]);
    }
    return true;
}
```

+) Implement Trie II

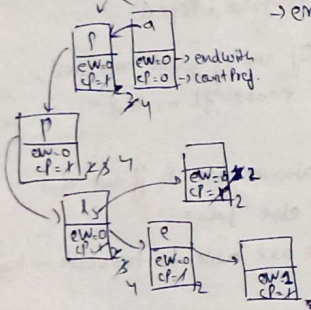
Implement trie DS : ->insert(word)

->count words equal to(word);
return the count of occurences of the string word in the trie.

->count words starting with(prefix);
count of words in trie that have string 'prefix' as prefix.

->erase(word): delete one occurence of the string word from the trie

Ans:) ex: apple
apple (increases cP=2)
appl (cP:2→3)
apps



- We are standing at 'a' & checking if 'a' exists, then we ↑ 'cP' of next trie by '1'
- If we count words equal to -> return 'ew'
- for erase -> we do 'cP--' & 'ew--'.

Code: -> $S_C = O(N)$, $T_C = O(N)$

```
struct Node{
    Node* links[26];
    int cntEndWith=0;
    int cntPrefix = 0;
    bool containsKey(char ch){
        return (links[ch-'a']!=NULL);
    }
    Node* get(char ch){ return links[ch-'a']; }
    void put(char ch, Node* node){ links[ch-'a']=node; }
    void increaseEnd(){ cntEndWith++; }
    void increasePref(){ cntPrefix++; }
    void deleteEnd(){ cntEndWith--; }
    void reducePref(){ cntPrefix--; }
    int getEnd(){ return cntEndWith; }
    int getPrefix(){ return cntPrefix; }
};
```

```
Node* root;
Trie(){
    root = new Node();
}
void insert(string &word){
    Node* node = root;
    for(int i=0; i<word.size(); i++){
        if(!node->containsKey(word[i])){
            node->put(word[i], new Node());
        }
        node = node->get(word[i]);
        node->increasePref();
    }
    node->increaseEnd();
}
```

```
int countWordsEqualTo(string &word){
    Node* node = root;
    for(int i=0; i<word.size(); i++){
        if(node->containsKey(word[i])){
            node = node->get(word[i]);
        }
        else return 0;
    }
    node->getEnd();
}
```

```
int countWordsStartingWith(string &word){
    Node* node = root;
    for(int i=0; i<word.size(); i++){
        if(node->containsKey(word[i])){
            node = node->get(word[i]);
        }
        else return 0;
    }
    return node->getPrefix();
}
```

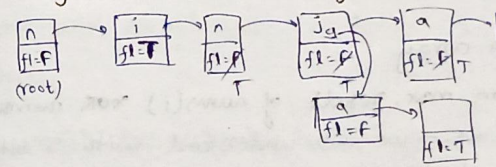
```
Void erase(string &word){
    Node* node = root;
    for(int i=0; i<word.size(); i++){
        if(node->containsKey(word[i])){
            node = node->get(word[i]);
            node->reducePref();
        }
        else return;
    }
    node->deleteEnd();
}
```

Q) Complete String :-

Find longest complete string. "complete" string -> if every prefix of string is also present in array 'A'. Return lexographically smallest one if multiple strings of same length exist.

ex: ["ninja", "ninj", "nin", "ni", "n", "ninga"] → op: ninja

A) n
ninja
ninj
ni
nin
ninga



As we traverse from root -> if 'T' -> add to answer string.

Code: -> $T_C = O(N * K)$
('n' words) (word length)

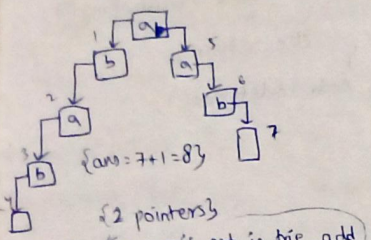
```
class Trie{
    bool checkAllPrefixExists(string word){
        Node* node = root;
        bool flag = true;
        for(int i=0; i<word.size(); i++){
            if(node->containsKey(word[i])){
                node = node->get(word[i]);
                if(node->isEnd()==false) return false;
            }
            else return false;
        }
        return true;
    };
```

```
string completeString(int n, vector<string>& a){
    Trie* obj = new Trie();
    for(auto word : a) obj->insert(word);
    string longest = "";
    for(auto word : a){
        if(obj->checkAllPrefixExists(word)){
            if(word.size() > longest.size()){
                longest = word;
            }
            else if(word.size()==longest.size()
                and word<longest){
                longest = word;
            }
        }
    }
    if(longest=="") return "None";
    return longest;
}
```

Q) Count distinct substrings

Given string 's', return no. of distinct substrings (including empty one).
Implement using trie

A) ex: abab



{2 pointers}

- starting at 0 → if not in trie, add → move to next index
- starting at 1 → if not in trie, add → move to next index

$T_c = O(N^2)$

S_c = Can't measure, as 'tries' reuses the previous space
So it might be → $26^{(a)} \times 26^{(b)} \times 26^{(a)} \times \ldots$
(Not possible)

Code:

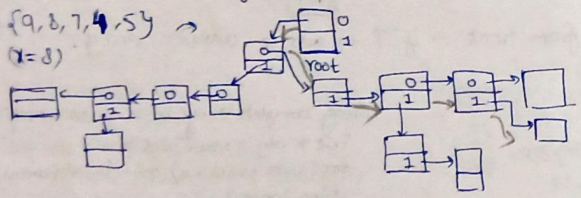
```
struct Node{
    Node* links[26];
    bool containsKey(char ch){
        return links[ch-'a'];
    }
    void put(char ch, Node* node){
        links[ch-'a'] = node;
    }
    Node* get(char ch){ return links[ch-'a']; }
};
int countDistinctSubstrings(string& s){
    int cnt=0;
    Node* root = new Node();
    for(int i=0; i<s.size(); i++){
        Node* node = root;
        for(int j=i; j<s.size(); j++){
            if(!node->containsKey(s[j])){
                cnt++;
                node->put(s[j], new Node());
            }
            node = node->get(s[j]);
        }
    }
    return cnt+1;
}
```

Q) Max. XOR of 2 no.'s in array

Given vector 'nums', return max. result of nums[i] XOR nums[j], $0 \le i \le j < n$

A) No. → 32 bits, but for ex. → we will understand with 5 bits

ex: {9, 8, 7, 4, 5} →



(x = 8)

- We insert binary form of no. into trie. (all no.'s of array)
- Take 'x' & find the max no. from array where (no. ^ x) ↑↑.

{XOR → 1^0 or 0^1 = 1 & 0^0 or 1^1 = 0}

x = 8 → 0 1 0 0 0

check 0 there, if no, insert & move next

0	0	1	1	1
0	1	1	1	1

(1 is there, so inserting 0)

- So insert 'arr 1' in trie & now take each ele. from vector & xor with trie → getMax.

arr[0] ^ (Trie) (inserting) (doing xor)
arr[1] ^ (Trie)
arr[2] ^ (Trie)

$T_c = O(N \times 32) + O(N \times 32)$

(32 bits in binary rep.)

S_c = Can't say specifically as there will be lot of overlaps but for now it might be $O(N \times 32)$

Code:

```
struct Node{
    Node* links[2];
    bool containsKey(int bit){
        return (links[bit]!=NULL);
    }
    Node* get(int bit){
        return links[bit];
    }
    void put(int bit, Node* node){
        links[bit] = node;
    }
};
```

```
class Trie{
    private:
        Node* root;
    public:
        Trie(){
            root = new Node();
        }
        void insert(int num){
            Node* node = root;
            for(int i=31; i>=0; i--){
                int bit = (num>>i) & 1;
                if(!node->containsKey(bit)){
                    node->put(bit, new Node());
                }
                node = node->get(bit);
            }
        }
        int getMax(int num){
            Node* node = root;
            int maxNum = 0;
            for(int i=31; i>=0; i--){
                int bit = (num>>i) & 1;
                if(node->containsKey(1-bit)){
                    maxNum = maxNum | (1<<i);
                    node = node->get(1-bit);
                }
                else node = node->get(bit);
            }
            return maxNum;
        }
};
```

(2)

(on i^{th} index, checking if bit set or not.)

```
int findMaximumXOR(vector<int>& nums){
    Trie trie;
    for(auto it: nums) trie.insert(it);
    int maxi = 0;
    for(auto it: nums){
        maxi = max(maxi, trie.getMax(it));
    }
    return maxi;
}
```

(go to next, no need to do anything)

Q) Max. XOR with an Element From Array

Given queries[i] = [xi, mi]. Ans. to i^{th} query is max. bitwise XOR value of 'xi' & any ele. of nums that does not exceed mi.
Return 'answer' array, having answer to i^{th} query.

ex: nums = [0,1,2,3,4], q = [[3,1],[1,3],[5,6]] → ans: [3, 3, 7]
(0^3) (1^2) (5^2)

A) ex: arr → {1,3,2,5,4} → sort → {1,2,3,4,5}

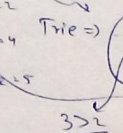
q → {3,1}, 0 ; {5,2}, 1 ; {2,5}, 2 ; {3,1}, 3 → sort →

3,1,3 → 1<=1 → insert 1
5,2,1 → tree size = 2
3,4,0 → tree size = 4
2,5,2 → tree size = 5

(add indexes)

* Ques. similar to prev. prob.

Trie =>



2 > 1 → so stop.

Now do "getMax(3)" ↓ (written previously)

3 > 2 → so stop → getMax(5)

Code:

(2)

```
vector<int> maximumXor(vector<int>&nums, vector<vector<int>>&queries){
    sort(nums.begin(), nums.end());
    vector<pair<int,pair<int,int>>> oQ;   // means "offline queries"
    int q = queries.size();
    for(int i=0; i<q; i++) oQ.push_back({queries[i][1], {queries[i][0], i}});
    Sort(oQ.begin(), oQ.end());   // sorts based on 1st
    vector<int> ans(q,0);
    int idx=0, n=nums.size();
    Trie trie;
    for(int i=0; i<q; i++){
        int ai = oQ[i].first, xi = oQ[i].second.first, qIdx = oQ[i].second.second;
        while(idx<n and nums[idx]<=ai){
            trie.insert(nums[idx]);
            idx++;
        }
        if(idx==0) ans[qIdx] = -1;
        else ans[qIdx] = trie.getMax(xi);
    }
    return ans;
}
```

$T_c = O(32*N + Q\log Q + 32*Q)$

- $32*N$ → (traversing ip array)
- $Q\log Q$ → (sorting)
- $32*Q$ → each no. = 32 bits & 'q' queries, so we traverse trie to find max XOR value.

$S_c = O(32*N + Q)$

- $32*N$ → (of trie)
- Q → (of oQ)

DP → {Enhanced Recursion}

- 'DP' applied when ques. has either of 2 things: 1) Choice → Recursion
  2) Optimal (like min, max, largest)

Parent Problems of DP: → Based on these only, other problems are defined.

1) 0-1 Knapsack (6) → {means 6 variations}
2) Unbounded Knapsack (5)
3) Fibonacci (7)
4) LCS (15)
5) LIS (10)
6) Kadane's Algorithm (6)
7) Matrix Chain Multiplication (1)
8) DP on Trees (4)
9) DP on Grid (14)
10) Others (5)

I) 0-1 Knapsack Problem

1) Subset Sum
2) Equal Sum Partition
3) Count of Subset Sum
4) Minimum subset sum difference
5) Target Sum
6) Number of Subset with given difference

Knapsack
- → Fractional → solved via greedy
- → 0/1 → multiple occurences not allowed → It's either we choose (1) or not choose (0)
- → Unbounded → multiple occurences are allowed

DP: Recursive Soln. → Memoization (DP)

ip:
wt[]: 1 3 4 5
val[]: 1 4 5 7
W = 7

```
int Knapsack(int wt[], int val[], int W, int n){
    Base Case
    Choice Diagram
}
```

Item 1 (W1)
(W1<=W) (W1>W)

think of smallest valid input

here → `if(n==0 || W==0) return 0;`

Recursive form: Fn(ip) → Fn(smaller ip) → ip must become smaller

Recursion: → (after writing recursion → 2 lines change → Memoized)

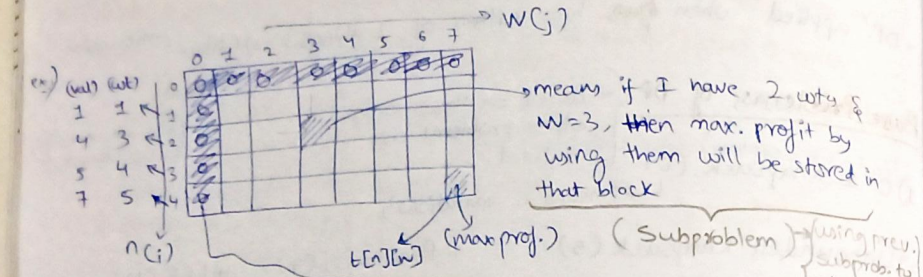
(base case) `if(t[n][W]!=-1) return t[n][W];` (not including so moving to rest of arr.)

`if(wt[n-1]<=W) return max(val[n-1] + Knap(wt, val, W-wt[n-1], n-1), Knap(wt, val, W, n-1));`

t[n][W] = Knap(wt, val, W, n-1);

`int t[n+1][W+1]` (pencil)

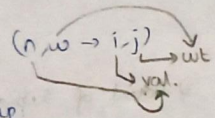
Memoize → recursion + , Bottom-Up → (similar to memoize)



(Subproblem) → (using prev. subprob. to solve curr. subprob.)

(Recursive base case) == (Bottom-Up initialization) → i.e. ⇒ if(n==0||w==0) return 0;

Now 'n', 'w' in memoization ⟷ 'i', 'j' in bottom-up



for(i=0 → n+1)
  for(j=0 → w+1)
    if(i==0||j==0) t[i][j]=0

In bottom-up

```
if(wt[n-1]<=w)
    t[n][w] = max(val[n-1] + t[n-1][w-wt[n-1]], t[n-1][w])
else
    t[n][w] = t[n-1][w]
```

Algo.: (Choice Diagram) → (changed to iterative way)

```
for(int i=1; i<n+1; i++)
  for(int j=1; j<w+1; j++)          // base case
    if(wt[i-1]<=j)
        t[i][j] = max(val[i-1] + t[i-1][j-wt[i-1]], t[i-1][j]);
    else
        t[i][j] = t[i-1][j];        // not including, so 'j' i.e. 'W' remains same
return t[n][w];
```

Pattern: (to identify other 6 variations) If there's max weight & we have choice between items i.e. either Yes/No → then it's knapsack problem.

Here you can see we have '1 item' → basically made of 2 arrays & a maxWt. given.

Sometimes '1' array also can be given & we will have choice, so consider 'wt[]' array always (analogy wise).

1) Subset Sum :

(FLOW)

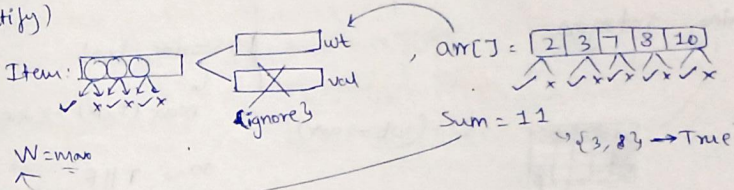
1) Problem Statement : determine if there's a subset present in array whose sum = k.

ex. {2, 3, 7, 8, 10}, k = 11 } → o/p : true

2) Similarity - (identify)

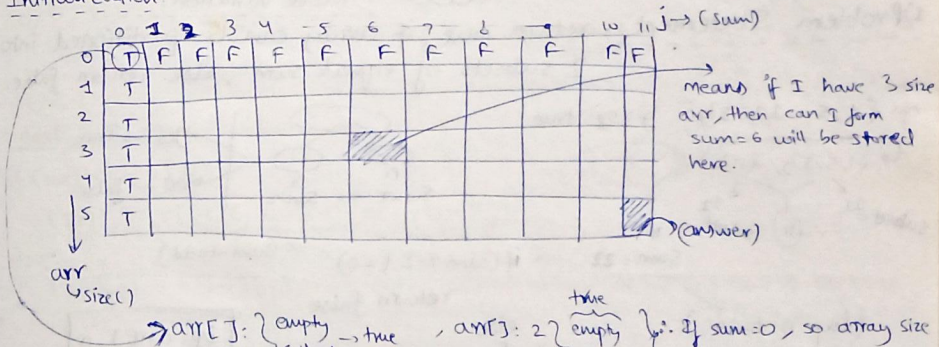
KnapSack: Item: wt, val, (ignore)

3) Code Variation → initialization, code



- (In knapsack, we had a choice to pick an item or not) = (subset sum also, we have a choice to pick an element or not)
- KnapSack → we needed maxProfit = Subset Sum → we need arr. sum = K

Initialization:



arr[]: } empty subset → true
Sum = 0 }

arr[]: 2 } empty subset } ∴ If sum=0, so array size doesn't matter as empty subset is always the answer. (that's why 1st col = 'T')
Sum = 0 }

arr[]: } no ele. in arr. } 'F'
Sum = 1 } so sum can't be '1'

t[n+1][W+1] → t[n+1][sum+1] ,

KnapSack ~ Subset Sum

for(i=0 to n+1
  for(j=0 to sum+1
    if(i==0) t[i][j] = F
    if(j==0) t[i][j] = T

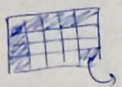
∴ wt[] → arr[]
W → sum
n,w → i,j

Knapsack

```
if(wt[i-1]<=j)
    t[i][j] = max(val[i-1]+t[i-1][j-wt[i-1]], t[i-1][j])
else
    t[i][j] = t[i-1][j]
```

→ Storing integers

(wt → arr)



↳ return t[n][w]

⇓

t[n][sum]

Subset Sum

```
if(arr[i-1]<=j)
    t[i][j] = t[i-1][j-arr[i-1]] or t[i-1][j];
else
    t[i][j] = t[i-1][j]
```

→ Storing bool

↳ Max (T, F) → doesn't make sense

so ↳ T || F (OR)

2) Equal Sum Partition :

FLOW → 1) Problem Statement
2) Subset Sum & similarity
3) Odd/Even significance
4) Code Variation

1) Problem Statement: return true if array can be partitioned into 2 subsets of equal sum, else return false.

ex: {1, 5, 11, 5} → true

{1,5,5}, {11}

subset = 11 (11) subset = 11

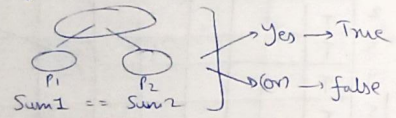
Sum = 22 → if (sum % 2 != 0) → (Sum → odd)

return false.

(loop '0' to 'n')

else
{
}

i/p: arr[], n



Now we need to find subset = "11" → if we found '11', then automatically other will be '11'

Algo:
```
int sum=0;
for(int i=0;i<n;i++) sum = sum+arr[i];
if(sum%2!=0) return false;
else if(sum%2==0) return subsetSum(arr, sum/2);
```
→ T/F

(Now we similarly apply prev. que. logic)

Knapsack → Subset Problem → Equal Sum

3) Count of Subset Sum

(means '0' not included) ← (consists only true no.)

FLOW → Problem Statement: count total no. of subsets in array whose Sum = k.

ex: [2, 3, 5, 6, 8, 10], k = 10

op: 3
↳ {2,8}
{2,3,5}
{10}

→ Similar to Subset Sum

→ Code Variations → Initialization / Code

→ Return type → boolean → int

In subset sum → we used to return 'T'/'F' (1/0)

But here → (count) → (>0)

subset → t[n+1][sum+1] ⇐ knapSack t[n+1][W+1] → Standard

(wt ← arr, w ← sum)

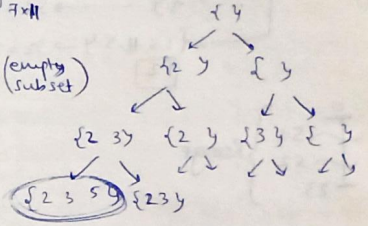
sum → (j)

Size of array (i)	0	1	2	3	4	5	6	7	8	9	10
0	T1	F0	F0	F0	F0	F0	F0	F0	F0	F0	F0
1	T1										
2	T1										1
3	T1										
4	T1										
5	T1										
6	T1										

7 × 11

return t[n][sum]

{?} → T → >0 ; If sum = 0 → (empty subset)
→ F → 0



Subset Sum Algo.:
```
if(arr[i-1] <= j){
    t[i][j] = t[i-1][j-arr[i-1]] or t[i-1][j];
              (include)           (exclude)
else
    t[i][j] = t[i-1][j];
```
(+) → change to '+' → Now it becomes this que. algo.

arr[]: 2 3 5 6 8 10
sum: 10

Subset → {2,3,5} + {10} + {2,8} → if we found '1', there is a chance we can find it later again, so we add here.

As you can see → including → 2,3,5 we got & that only if we exclude few ele.'s → we will get (2,8) as one more answer.

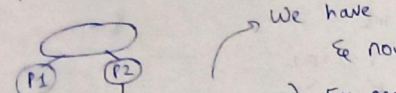
So that's one more reason, why we add.

'if'(arr[i-1]<=j) } → same condition as 'subset sum'

4) Minimum Subset Sum Difference

FLOW → Problem Stmt: Divide array into 2 sets such that the absolute difference btw their sums is min. Find the min. diff.

↳ Solve using previous concepts



→ We have done this que. 'equal sum partition' & now we have a modification

S1 - S2 = 0 ⇒ S1 = S2 → [we need to minimize] → S1 - S2 = minimum. ↳ abs(S1 - S2)

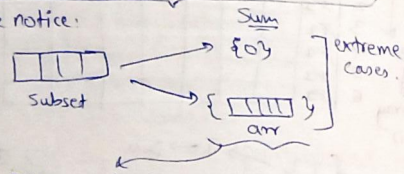
ex: arr: 1, 6, 11, 5 ∴ op: 1

{1,6} - {11,5} ⇒ 16 - 7 = 9 (min.?)

{1+6+5} - {11} ⇒ 12 - 11 = 1 ✓

Now I'm clueless how we will find S1, S2 (S1: sum of subset 1; S2: sum of subset 2)

If we notice:



Either subset sum will be '0' or sum of full 'arr' array. If not these 2, then surely the subset sum will lie in this range.

ex: 1 6 11 5

P1 → {} → S1 = 0

P2 → {1,6,11,5} → S2 = 23

S1, S2 } Range

-23

ex: 1 2 7

0 1 2 3 4 5 6 7 8 9 10

(These 3 can't be subset sum) (7+2) (7+1)

S1/S2 can't be 4,5,6 as you can't make any subset whose sum = (4,5,6)

S1/S2 = { 0, 1, 2, 3, 7, 8, 9, 10 }

If you see, if '0' there → 10-0 = '10' also there
if '1' there → 10-1 = '9' also there
if '2' there → 10-2 = '8' also there
if '3' there → 10-3 = '7' also there

∴ We can break it at (range/2), but we shouldn't take all values.

ex: '4' & '5' in this 1 2 7 can't be formed.

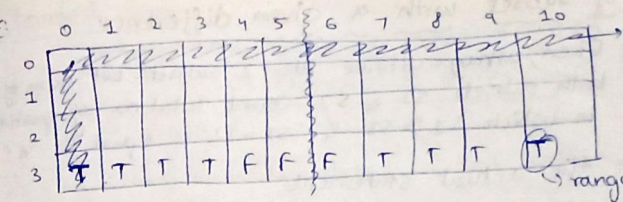
∴ If a subset is 'S1', then other subset is 'range - S1'

∴ S1 - S2 ⇒ Range - 2S1 } minimize

For this, we can apply our 'subset sum' logic.

(Because if 1 S1 is in first half, then S2 will automatically be in 2nd half)

ex:



↳ subsetSum(int arr[], int range) → we will get this table

Now we can take another vector & store the last row till half. Now traverse this vector from end & whichever value is 'true' → return that.

∴ We just used → 1) Subset Sum 2) Vector

Code:

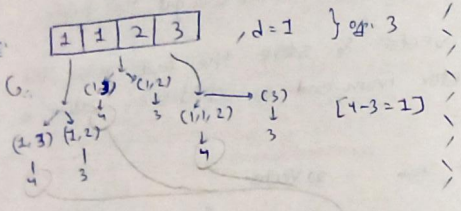
```
int minDifference(vector<int>& arr, int n){
    int sum = 0;
    for(auto i : arr) sum += i;
    vector<vector<bool>> t(n+1, vector<bool>(sum+1, false));
    for(int i=0; i<=n; i++){
        for(int j=0; j<=sum; j++){
            if(i==0) t[i][j] = false;
            if(j==0) t[i][j] = true;
        }
    }
    for(int i=1; i<n+1; i++){
        for(int j=1; j<sum+1; j++){
            if(arr[i-1]<=j) t[i][j] = t[i-1][j-arr[i-1]] or t[i-1][j];
            else t[i][j] = t[i-1][j];
        }
    }
    vector<bool> lastRow(sum+1);
    for(int j=0; j<sum+1; j++) lastRow[j] = t[n][j];

    int halfSum = sum/2;
    int subsetSum = 0;
    for(int j=halfSum; j>=0; j--){
        if(lastRow[j]){
            subsetSum = j;
            break;
        }
    }
    return sum - 2* subsetSum;
}
```

5) Count the no. of subset with a given difference

Problem Stmt: Given, array → divide into 2 subsets. Let sum of both subsets S1 & S2. Count total no. of partitions in which S1 > S2 & S1 − S2 is equal to 'd'

Flow:
- try to reduce the actual statement
- solve it using already solved problem

ex: [1 1 2 3], d = 1 } eg. 3



[4 − 3 = 1]

If you notice,

sum(S1) + sum(S2) = sum(arr)

sum(S1) − sum(S2) = d

$$sum(S1) = \frac{sum(arr) + d}{2}$$

[1 1 2 3], diff. = d

S1 + S2 = sum(arr)

(matches above sum)

→ here → $S1 = \frac{(3+2+1+1)+1}{2} = \frac{8}{2} = 4$

Now problem reduced to "Count of Subset Sum", which we have already solved.

But, here '0's can also be in array. → so small change in initialization & base case

Base Cases:
- if((sum+d)%2 != 0) return 0;
- Initialize whole 't' array with zero & t[0][0] = 1.

- We used to initialise 1st col. as '1', assuming there's only 1 way to make subset sum = 0 i.e. null subset. But this fails if we have 0's as element of array. If we have single 0 present in array, then the subsets will be '{}, {0}' whose sum will be 0. Hence there can be more than 1 ways to make sum = 0.

- Algo.: (Same as "count of subset sum" but 'j' starts with '0')

```
for(i = 1 → n+1)          → (d+sum)/2)
  for(j = 0 → ans+1)
    if(arr[i-1] <= j)  t[i][j] = t[i-1][j-arr[i-1]] + t[i-1][j];
    else               t[i][j] = t[i-1][j];
```

Ans → return t[n][ans];

6) Target Sum

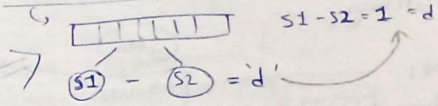
Problem Stmt: Given an array → assign '+'/'−' infront of elements of array such that their sum = k. Return no. of diff. expressions we can build, which evaluates to 'k'

Flow:
- solving it & reducing it
- solve it using already solved problem

ex: [1 1 2 3], sum = 1 } eg. 3

- 1 + 1 + 2 − 3 = 1
- −1 + 1 − 2 + 3 = 1
- 1 − 1 − 2 + 3 = 1

Lets revise prev. que.



S1 − S2 = d

Target Sum

[+ − − + | 1 1 2 3], sum = 1

(+1+3) − (1+2) = 'k'

∴ In this que., d == K (target sum) & rest everything is same!

{ Target Sum → count of subset with given diff. ⇓
(arr. sum) ⟷ (arr. diff) }

Extra base case → if (sum < abs(target)) return 0; (no subset can be formed)

target can be −ve also, ex: nums = [1,0,0], target = −200

So finally,

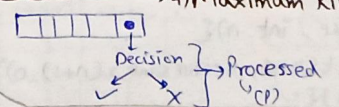
→ if (sum < abs(target) or ((sum + target)%2 != 0) return 0;

//same as prev. ques →code

II) Unbounded Knapsack

1) Rod Cutting
2) Coin change I
3) Coin change II
4) Maximum Ribbon Cut

In 0-1 Knapsack :



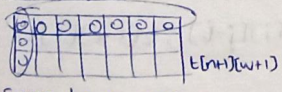
Decision → Processed (P)

, but in unbounded we can have multiple occurences of same ele.

0/1: → P, x → P

Unb.: → P x 3, x → P (can be used again)

0/1 knapsack

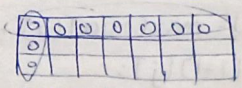


t[n+1][w+1]

- Same base case
- if(wt[i-1] <= j)

```
t[i][j] = max(val[i-1] + t[i-1][j-wt[i-1]], t[i-1][j]);
else
t[i][j] = t[i-1][j];
```

Unbounded knapsack



- Same base case
- if(wt[i-1] <= j)

```
t[i][j] = max(val[i-1] + t[i][j-wt[i-1]], t[i-1][j]);
else
t[i][j] = t[i-1][j];
```

Minor change

1) Rod Cutting

Flow → Prob. Stmt.: Given 'length' & 'price' array, & length rod. Find max. price of rod, if rod can be in any length.

ex:

len[]:	1	2	3	4	5	6	7	8
price[]:	1	5	8	9	10	17	17	20

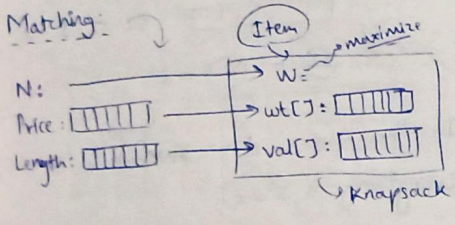
- Matching
- How to identify → 0-1 / Unbounded
- Code Variation (if any)

N = 8 (length of rod)

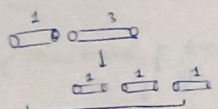
5 + 17 = 22 → we need to Maximize this price.

2 | 6 — can be broken to '2' again, so we can say that it's unbounded knapsack problem

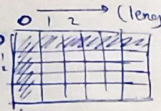
Matching:



Unbounded ↕ Rod Cutting

ex: 

∴ Unbounded

Base Case: 

- if price = 0 → any length cut → it's price = 0
- if length of rod = 0 → any price → 0, as no rod left

t[size+1][n+1] (from unbounded knapsack) (generally same)

∴ Algo:

if (wt[i-1] <= j) → length[i-1], price[i-1], length[i-1]

t[i][j] = max(val[i-1] + t[i][j-wt[i-1]], t[i-1][j])

[might differ if we can cut rod only in particular length ie. 2, 4, 6, 8, —]

else

t[i][j] = t[i-1][j];

Code:

```
int cutRod(vector<int>& price, int n){
    vector<vector<int>> t(n+1, vector<int>(n+1, 0));
    for(int i=1; i<n+1; i++){
      for(int j=1; j<n+1; j++){
        if(i<=j) t[i][j]= max(price[i-1] + t[i][j-i], t[i-1][j];
        else t[i][j]= t[i-1][j];
      }
    }
    return t[n][n];
}
```

2) Coin Change I - max no. of ways

Flow → Problem Stmt.: Given 'coins' array & 'sum', return no. of comb. of coins that make upto 'sum'. We have ∞ no. of each kind of coin

- Recall Subset Sum
- Why unbounded
- Code Variation

ex: coins: [1|2|3], sum = 5 } ans = 5

- 2+3 = 5
- 1+2+2 = 5
- 1+1+3 = 5
- 1+1+1+1+1 = 5
- 1+1+1+2 = 5

(1 reused → so unbounded knapsack)

Matching:

coins[] → wt[] , {If only 1 array, so we discard val[] array}

sum → W (knapsack)

Subset Sum

```
if(arr[i-1]<=j)
    t[i][j] = t[i-1][j] || t[i-1][j-arr[i-1]]
else
    t[i][j] = t[i-1][j]
```

→ then we did 'count of subset' → '+' → change to '+'

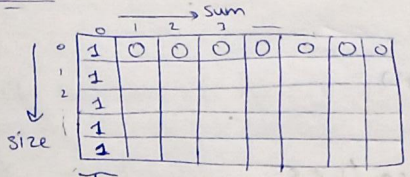
choice diagram

if we got 1 answer, we can get more ans. ahead, so we 'add'

∴ Here, in counting of coins → sum = 'sum'

```
if(coins[i-1] <= j)
    t[i][j] = t[i][j-coins[i-1]] + t[i-1][j];
else
    t[i][j] = t[i-1][j];
```

Initialization: → vector<vector<int>> t(n+1, vector<int>(amount+1, 0));



- if arr[]: sum = 0 } 1 1
- if arr[]: sum = 1 } Not possible → so '0'

if sum = 0 → ans = NULL subset, independent of how many no. of coins we have.

ex: arr[]: [1|2], sum = 0 } 1 1

(t[n+1][W+1] → t[n+1][sum+1])
knapsack → coin-change

3) Coin Change II – min. no. of coins

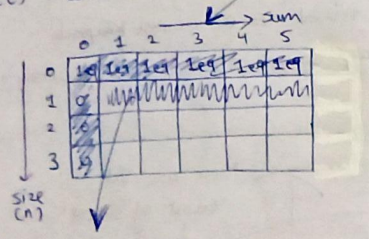
Problem Stmt.: Given 'coins' array & an 'amount', return the fewest no. of coins that you need to make up that amount

Flow:
- Initialization
- Code Variation

ex: [1|2|3], sum = 5 } op: 2

2 ← 2+3 = 5
3 ← 1+2+2 = 5
4 ← 1+1+1+2 = 5
3 ← 1+1+3 = 5
5 ← 1+1+1+1+1 = 5

t[n+1][w+1] ⇒ t[n+1][sum+1]
wt[] → coins[]



- If size = 0 → then getting sum = 0, 1, .. 5 is not possible → so we declare as 'INT_MAX'
- If sum = 0 → then no need to take any coin → so size >= 1 → at sum = 0 → value = 0

- 2nd row initialisation
  - If sum = 3 & coins[] = {7} → then it's not possible to get sum of '3'
  - So in 2nd row, we check if (j % coins[0] == 0) → t[i][j] = j/coins[0] (j > 0) else t[i][j] = INT_MAX;

Now algo. same as before concepts (unbounded knapsack)

(as we are counting no. of coins so adding '1')

if(wt[i-1] <= j) — coins[i-1]

$$t[i][j] = \min(\text{val}[i-1] + t[i][j-wt[i-1]],\ t[i-1][j])$$

else

$$t[i][j] = t[i-1][j]$$

& obvio. → for(i=2; i<n+1; i++)
for(j=1; j<amount+1; j++)

III) Longest Common Subsequence / LCS

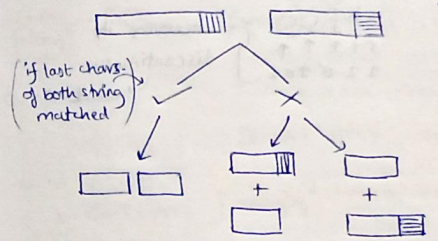
1) Longest Common Substring
2) Print LCS
3) Shortest Common Supersequence
4) Print SCS
5) Min. no. of insertion & deletion : a → b
6) Longest repeating subsequence
7) Length of largest subsequence of 'a' which is a substring in 'b'
8) Subsequence Pattern Matching
9) Count how many times 'a' appear as subsequence in 'b'

10) Longest Palindromic Subsequence
11) Longest Palindromic Substring
12) Count of palindromic substring
13) Min. no. of deletions in a string to make it a palindrome
14) Min. no. of insertions in a string to make it a palindrome.

ex: x: a b c d g h
y: a b e d f h r } op: abdh → subsequence means there can be gap btw ele.s. It be non-continuous

'abd' → not a substring → is always continuous

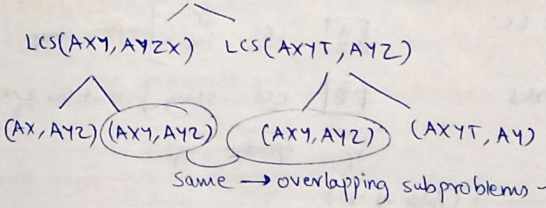
In recursion: 1) Base Case → (consider smallest ip) → if n==0 or m==0 (length of strings given) → no matching → so op = 0
2) Choice Diagram
3) Always go to smaller ip




```
int LCS(string x, string y, int m, int n)
if(m==0 or n==0) return 0;
if(t[m][n] != -1) return t[m][n]
if(x[m-1] == y[n-1])
    return 1 + LCS(x,y,m-1,n-1);
    t[m][n] =
return max(LCS(x,y,m,n-1),
    t[m][n] =          LCS(x,y,m-1,n));
```

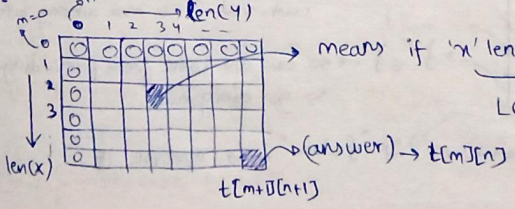
Need to memoize (see pencil)

ex: LCS("AXYT", "AYZX")



We see only the length of strings are getting changed i.e 'n' & 'm'. So we create a 2D array 't' of size (m+1)×(n+1)

- Generally top-down soln. not good → as it can cause stack overflow if the recursion call stack is too large



means if 'x' len = 2 & 'y' len = 3 → (answer) → t[m][n]

LCS in this → it's size will be stored.

if(x[m-1] == y[n-1])
return t[m][n] = 1 + LCS(x,y,m-1,n-1)

else
return t[m][n] = max(LCS(x,y,m-1,n), LCS(x,y,m,n-1))

→ if(x[i-1] == y[j-1])
t[i][j] = 1 + t[i-1][j-1]

→ else
t[i][j] = max(t[i][j-1], t[i-1][j])

obvio. → for(i=1; i<m+1; i++)
for(j=1; j<n+1; j++)

Subsequence

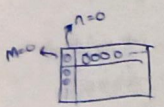
1) Longest Common Substring

Prob. Stmt.: Print size of longest common substring

Flow
- How diff. from LCS
- Code Variation → Initialization, Main Code

ex: a: a b c d e
b: a b f c e
→ ab, c, e → largest 'ab'

a b c d e
a b f c e
1 2 0 1 0 → whenever it's discontinuous → count = 0



Algo:

```
if(a[i-1] == b[j-1])
    t[i][j] = 1 + t[i-1][j-1]
else
    t[i][j] = 0
```

2) Printing LCS btw 2 strings

Prob. Stmt.: Print LCS

Flow
- Identify
- Learn how LCS works
- Code

LCS — o/p: int → prev. done

LCS — string → matches 2/3rd of prev. prob.

i/p — operation — o/p

(Table) → (LCS)

ex:

		0	a	b	c	d	a	f
	0	0	0	0	0	0	0	0
a	0	0	1	1	1	1	1	1
c	0	0	1	1	2	2	2	2
b	0	0	1	2	2	2	2	2
c	0	0	1	2	2	3	3	3
f	0	0	1	2	3	3	3	4

(index of 'b')
(index of 'a')

t[i] != a' → so our ans came from max of top & it's prev. So go in that direction from where you got curr. val. i.e. max among both.

(i-1, j)

f == f' → so our ans. came from prev. diag, so go diagonal

(i, j-1)

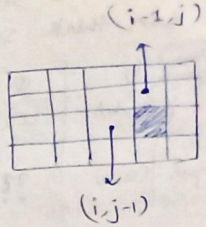
(start from here) o/p: "fcba" → reverse it.

If we reach '0' → means one string became exhausted, so stop loop

∴ if equal → i--, j--
not equal → max(i-1, j), (i, j-1)

Algo:

```
a, b, m, n → b.size()
           → a.size()

int i = m, j = n; string s = "";
while (i>0 and j>0){
    if(a[i-1] == b[j-1]) {
        s.push_back (a[i-1]);
        i--; j--;
    }
    else{
        if( t[i][j-1] > t[i-1][j]) j--;
        else i--;
    }
}
reverse (s.begin(), s.end());
return s;
```



3) Shortest Common Supersequence

Prob. Stmt.: Given 2 strings, merge them is such a way that both subsequences are present in merged string & order of subsequence must not change.

Flow → Code Variation

ex: a: "geek", b: "eke" → o/p: geeke

ex: a: AGGTAB, b: GXTXAYB } one way of merging → AGGTGXABTXAYB

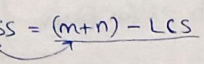
(Supersequence) (sequence of chars. of 'a' & 'b' not changed, so it's a superseq.)

Now we need it's shortest

If we notice, what are the common terms?

A G G T A B
G X T X A Y B
→ LCS → GTAB → write only once

ex: worst Supersequence → merge both as it is:
AGGTABGXTXAYB
AGGTAB | GXTXAYB − GTAB
(GTAB) (GTAB)
(LCS repeating twice, so we subtract it once)

∴ worst case: GTAB GTAB


(Idea)

length of SS = (m+n) − LCS } In this case → m=6, n=7, LCS=4 → AGGXTXAYB → len = 7+6−4 = 9

∴ return m+n − LCS(a,b,m,n);

4) Min. no. of insertion & deletion to convert string 'a' to string 'b'

Prob. Stmt.: ↑

Flow → Relate to LCS → Code variation

ex: a: heap, b: pea } a → b : pHeap → pea ∴ Insert = 1, Del. = 2

At starting, we mentioned, DP applied when
1) Choice
2) Optimal
(here we need min. so we can say we need to apply DP)

Here we have 2 strings → 'a' & 'b' & we need optimal operations
so we can think something related to LCS

Matching Algo	i/p	Q	o/p
LCS :	(a, b)	LCS	int
This que :	(a, b)	??	int

} 2/3rd matching ✓

If we notice : p̶h̶eap̶ → pea
(new member) "ea" → ea
LCS = ea

∴ a → b ; a → LCS → b ; LCS : heap, pea > ea } → heap → pea
2 deletion → ea → 1 insertion

∴ Total no. of del.'s = a.size() − LCS
Total no. of insert.'s = b.size() − LCS

5) Longest Palindromic Subsequence

Flow
- → Prob. Stmt. ✓
- → Is it LCS?
- → Variation
- → Thinking of LCS

ex: agbcba → abcba ✓, bcb ✓, b ✓
Now we need largest palindrome

Note: LCS → child → Palindromic String ques.

Matching Algo:

	i/p	Q	o/p
LCS :	a: b:	LCS	int
This Q :	a:	LP sub	int

(i/p) (question) (op)

In LCS, we have 2 strings. In LPS, we have only 1 string. To get 2nd string, maybe we can write it from 'a'.
Like: → a → a
b → derive(a)
(Hidden) (redundant)

If you notice,
a → a
b → reverse(a)] Now applying LCS
ex: a → agbcba
b → abcbga } ∴ LCS = length: 5

∴ LPS(a) ≡ LCS(a, reverse(a))
∴ return LPS(agbcba) = return LCS(agbcba, abcbga)

string b = reverse(a.begin(), a.end());

6) Min. no. of deletion in a string to make it palindrome

Flow
- → Prob. Stmt. :
- → Code Variation

ex: s: x p o x o
(no. of deletion) → minimize
← palindrome

ex: agbcba — ag̶bcba (1) → minimize → abcba
x̶g̶bcbx̶ (3) → bcb
x̶x̶x̶cx̶x̶ (5) → c
→ Palindrome

∴ we can see;
↑ Length of LPS ∝ 1 / no. of deletions ↓

LPS(s) = LCS(s, reverse(s))

∴ Min. no. of deletions = s.length() − LPS

7) Print shortest common supersequence

Flow
- → Prob. Stmt.
- → Code derivation using LCS

merge(ss) → worst case − LCS
s1, s2 } → s1 + s2 − LCS

ex: → LCS table

	0	a	b	c	d	a	f
0	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1
c	0	1	1	2	2	2	2
b	0	1	2	2	2	2	2
c	0	1	2	3	3 ←	(3)	3
f	0	1	2	3	3	3	(4)

abcd(a)(f)
acb(c)(f)
scs ← caf

In LCS, we need common one's only. But in SCS, we need all strings AND we must have LCS only once in it, to get the shortest.
∴ common → include once
else → rest of the string } → acbcaf = scs

Also in base → in LCS, if 1 string became empty → stop,
but here: ex: (ac, "") → "" / → ac] → shortest common superseq. must have all chars.
→ scs = "ac"

∴ Algo:

```
int i = m, j = n ; string s = "";
while(i > 0 and j > 0){
    if(a[i-1] == b[j-1]){
        s.push_back(a[i-1]);
        i--; j--;
    }
    else{
        if(t[i][j-1] > t[i-1][j]){
            s.push_back(b[j-1]);
            j--;
        }
        else{
            s.push_back(a[i-1]);
            i--;
        }
    }
}
I ←
while(i>0){
    s.push_back(a[i-1]);
    i--;
}
while(j>0){
    s.push_back(b[j-1]);
    j--;
}
reverse(s.begin(), s.end());
return s;
```

8) Longest repeating subsequence

Flow → Prob. stmt. → ex: "AABEBCDD"

AABEBCDD → ABD (ans: 3)

subsequence → may be discontinuous; order must be same; length = 3

S: A A B E B C D D (0 1 2 3 4 5 6 7)

A A B E B C D D

→ Now we want that if 'A' has already been used, then we can't use that same 'A' again & now we have to find another 'A'

(this will have some other index val.)

If we find LCS of same string with itself

Also, AABEBCDD → AABBDD → once → ABD

Explanation: A = (0, 1), A = (0, 1) → (0,1) → one pair

B = (2, 4), B = (2, 4) (i, j) → (i == j) → ignore

E = 3, E = 3 → (i == j), if ele. present only once, then we can't find longest repeating subseq. as it can be used only '1' time, there may be others which can used more time.

Here only 1 index of 'E', so we can't map it with anything else → Hence ignored as (i == j)

∴ LCS + (i == j)

∴ Algo.:

```
if(a[i-1] == b[j-1] and i != j)
    t[i][j] = 1 + t[i-1][j-1];
else
    t[i][j] = max(t[i][j-1], t[i-1][j]);
```

		A(1)	A(2)	B(3)	E(4)	B(5)	C(6)	D(7)	D(8)
	0	0	0	0	0	0	0	0	0
A-1	0	0	1	1	1	1	1	1	1
A-2	0	1	1	1	1	1	1	1	1
B-3	0	1	1	1	1	2	2	2	2
E-4	0	1	1	1	1	2	2	2	2
B-5	0	1	1	2	2	2	2	2	2
C-6	0	1	1	2	2	2	2	2	2
D-7	0	1	1	2	2	2	2	2	3
D-8	0	1	1	2	2	2	2	3	3

9) Sequence Pattern Matching

Flow → Prob. stmt.: Check if 'a' is a subsequence of 'b' or not? → op: true/false

→ Using LCS

ex: a = "AXY", b = "ADXCPY" → true

a = "AXY" → len = 3

b = "ADXCPY"

LCS = "AXY" → (3)

∴ Comparing LCS len. to min len. of given 2 strings

LCS → 0 to min(m, n)

If (t[n][m] == a.length()) (from LCS Table)
    return true;
else return false;

Matching Algo.:

	ip	Q	op
LCS	a, b	Subseq.	int
This Q.	a, b	"	T/F
	ip	Q	op

(2/3rd Matching)

• Space Optimization if only last (row, col) → block is required:

⇒ $S_c = O(mn) \rightarrow O(m)$

```
int m = s.size(), n = r.size();
vector<int> prev(n+1, 0), curr(n+1, 0);
for(int i = 1; i < m+1; i++){
    for(int j = 1; j < n+1; j++){
        if(s[i-1] == r[j-1]){
            curr[j] = 1 + prev[j-1];
        }
        else{
            curr[j] = max(prev[j], curr[j-1]);
        }
    }
    prev = curr; → Copy current row to previous row for the next iteration
}
```

//For above "Sequence Pattern Matching" question;

```
if(curr[n] != m) return false;
else return true;
```

10) Min. no. of (insertion / deletion) in a string to make it a palindrome

Flow → Prob. Stmt.:

Recall: (6) → {prev. page}

Code Variation:

ex: s = "aebcbda"

a de bc b eda → (2) insertions

xyz ade bcbe dazyx → can make any no. of insertions, but we need min.

Matching Algo.

	ip	Q	op
this q.	a	min no.	int
(6)th que.	a	min no.	int

Completely Matching → LCS ✓, LPS ✓

eg: S = "aebcbda"

↓ LPS

↓ a x b c b x a

→ In insertion → either need to add 'e' & 'd' on both sides → so count = 2

→ In deletion → need to delete on both sides, so count = 2

∴ Note: No. of insertion == No. of deletion == (S.length() − LPS)

IV) Matrix Chain Multiplication

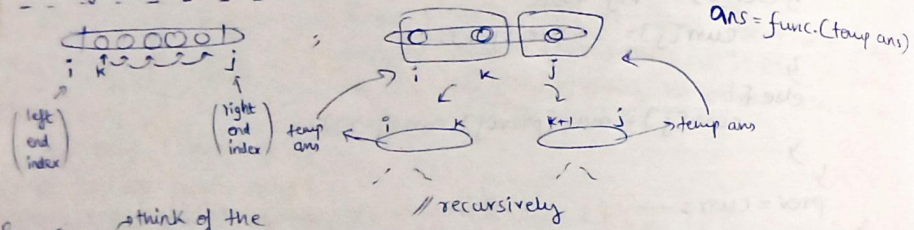
1) MCM 2) Printing MCM

3) Evaluate expression to True/Boolean Parenthesization

4) Min/Max. value of an expression 5) Palindrome Partitioning

6) Scramble String 7) Egg Dropping Problem

Identification Format



Base Case: → think of the smallest valid i/p

↳ think of 1st invalid i/p

Form:

```
int solve(int arr[], int i, int j){
    if(i > j) return 0;        // might differ
    for(int k = i; k < j; k++){
        tempans = solve(arr, i, k) + solve(arr, k+1, j);
    }
    ans = function(tempans);   // maybe → max/min
    return ans;
}
```

→ there will be many variations upcoming, but this is the raw syntax

→ Like MCM format

1) MCM {Matrix Chain Multiplication}

Flow:

Prob. Stmt.: i^{th} matrix = arr[i-1] * arr[i] (given), find min. cost of multiplying all matrices

Code using the format

ex: arr[] = {40 (0), 20 (1), 30 (2), 10 (3), 30 (4)}

size 'n'

∴ n-1 matrices → A_i = arr[i-1] * arr[i]

$A_1 = 40\times20$, $A_2 = 20\times30$, $A_3 = 30\times10$, $A_4 = 10\times30$

(A1(A2A3))A4 ~ C1

((A1A2)(A3A4)) ~ C2

(A1(A2(A3A4))) ~ C3

→ need min. cost

ex: A → 10×30, B → 30×5, C → 5×60

ACB()

(AB)C = (10×30, 30×5), 5×60 = 10×5, 5×60

C1 = 10×30×5 + 5×10×60 = 4500 (tempans)

(10×30, (30×5, 5×60))

C2 = 10×30×60 + 30×5×60 = 27000 (tempans)

→ min. = ans.

Identify:

↳ we need to break the array → 'k'

↳ Need to get ans from tempans

Base Case: if → i = j → means size = 1 → no matrix can be formed.

(left end, right end; i > j)

```
int solve(int arr[], int i, int j){
    if(i >= j) return 0;
```

1) Find 'i' & 'j'

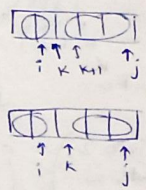
2) Find correct base condition

3) Move k → i to j

i = 1, j = n-1 (if i = 0 → matrix not possible)

2 schemes:

- K = i, K = j−1 → for 'i' to 'k', for 'k+1' to 'j'
- k = i+1, K = j → for 'i' to 'k−1', for 'k' to 'j'



both are same

```
for(int k = i; k <= j-1; k++)
{
    solve(arr, i, k)
  + solve(arr, k+1, j)  + cost
}
(tempans)
```

ex: 40 (0), 20 (1), 30 (2), 10 (3), 30 (4)

(i) (k) (k+1) (j)

for(i to k) for(k+1 to j)

40×20 20×30 30×10 10×30

40×20×30 = C1; 40×30 30×30; 30×10×30 = C2

C3 { 40×30×30

arr[i-1] arr[k] arr[j] } → cost

```
tempans = solve(arr, i, k) + solve(arr, k+1, j)
        + arr[i-1] * arr[k] * arr[j]

if(tempans < ans){
    ans = tempans
}
return ans;
```

∴ Steps:

1) Find 'i' & 'j'
2) Find Base cond.
3) Find k loop scheme
4) Calculate ans from tempans

applying DP as there can be over-lapping subprob.

here 'i' & 'j' in func. are changing, so we create a 2D DP matrix. → initialized with '−1'.

Algo:

```
int t[1001][1001]   // as per constraints in que.
                    // global
int solve(int arr[], int i, int j){
    if(i>=j) return 0;
    if(t[i][j]!=-1) return t[i][j];
    int mn = INT_MAX;
    for(int k=i; k<j; k++){
        int tempans = solve(arr,i,k) + solve(arr,k+1,j)
                      + arr[i-1]*arr[k]*arr[j];
        if(tempans<mn) mn=tempans;
    }
    return t[i][j]=mn;
}
```

```
int main(){
    memset(t, -1, sizeof(t));
    return solve(arr, 1, size-1);
}
```

→ used to initialize 't' with value '-1' & size is full 't'.

2) Palindrome Partitioning

Prob. Stmt: Partition 's' such that every substring is a palindrome.
→ return min. cuts needed for palin. parti. of 's'

Flow
- Follow format:
- further optimize

ex: n i t i n k
- n | i t i n k → 4 cuts
- n i t i | n k → 2 cuts
- minimize

n i t i n k
- (i to k) | (k+1 to j)
- c_1 | c_3 | c_2
- c_1 + c_2 → tempans
- for breaking into 2, we need 1 cut, so $c_3 = 1$

```
if(isPalindrome(s,i,j)==true)
    return 0;
```

```
{ if(i>=j)
    return 0;
  ...
  "a" }
```
↳ both are palindrome

Steps:
1) Find 'i' & 'j'
2) Find correct base cond.
3) Find k loop scheme
4) Calc. 'ans' from tempans

K scheme: i to k
K+1 to j

∴ Base code:

```
int solve(string s, int i, int j)        // 0 ... s.size()-1
    if(i>=j) return 0;
    if(isPalindrome(s,i,j)) return 0;
    int mn = INT_MAX;
    for(int k=i; k<j; k++){
        int temp = 1 + solve(s,i,k) + solve(s,k+1,j);
        if(temp<mn) mn=temp;
    }
    return mn;
}
```

2 variables changing i.e. size of string → i, j, so we create a 2D DP.

```
if(t[i][j] != -1) return t[i][j];
```

! but we need further optimization, as left code is giving TLE,

Opt. code:

```
bool isPalindrome(string &s, int i, int j, vector<vector<int>>& isPal){
    if(i>=j) return true;
    if(isPal[i][j]!=-1) return isPal[i][j];
    if(s[i]==s[j]){
        if(isPalindrome(s,i+1,j-1,isPal)){
            return isPal[i][j]=true;
        }
    }
    return isPal[i][j]=false;
}
int solve(string &s, int i, int j, vector<vector<int>>& t, vector<vector<int>>& isPal){
    if(i>=j) return 0;
    if(isPalindrome(s,i,j,isPal)) return 0;
    if(t[i][j]!=-1) return t[i][j];
    int mn = INT_MAX;
    for(int k=i; k<j; k++){
        if(isPalindrome(s,i,k,isPal)){
            int temp = 1 + solve(s,k+1,j,t,isPal);
            mn = min(mn,temp);
        }
    }
    return t[i][j]=mn;
}
int minCut(string s){                          // Sc = O(n²) ; Tc = O(n³)
    int n=s.size();
    vector<vector<int>> t(n, vector<int>(n,-1));   vector<vector<int>> isPal(n,
    return solve(s,0,n-1,t,isPal);                                  vector<int>(n,-1));
}
```

t[i][j] → min. no. of cuts req. to partition the substring 's[i:j]' into palindromic substrs.

isPal[i][j] → is a boolean val that indicates whether the substr 's[i:j]' is a palindrome.

(if 2 'solve' func. calls written → TLE)

3) Evaluate expression to True / Boolean Parenthesization

Prob. Stmt.: Given expression (boolean) 's' with symbols: T → true, F → false, & → bool AND, | → bool OR, ^ → bool XOR

Count no. of ways, we can parenthesize the expression, so that it's val = true always

Flow
- MCM format

ex: T^F&T → ∴ ans: 2 ways
- (T^F)&T → T&T = true
- T^(F&T) → T^F = true

ex: T|F&T^F
- k
- We can see, we need to divide it at operators ~ kinda MCM

XOR →
- T^F = T
- F^T = T
- T^T = F
- F^F = F

4 steps
1) Find 'i' & 'j' ; i=0, j=s.size-1
2) Base cond.
3) 'k' loop
4) Ans ← func.(temp Ans)

So we pass int solve(string s , int i, int j)

BUT, DO YOU THINK IT'S ENOUGH ? Think...

→ (expr XOR expr) ⟶ ans = lT * rF + rT * lF

(left) → T / F (right) → T / F

(leftTrue) (rightFalse) (rightTrue) (leftFalse)

∴ We need to calculate whether the expr is true/false → it's count.

∴ In func. → int solve(string s, int i, int j, boolean isTrue) → T / F

Now base cond. → if (i>j) return false;

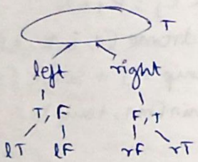
if (i==j) → means a single char. left (it won't be an operator)

```
if(isTrue == true) return s[i]=='T'
else return s[i]=='F'
```

k-loop:

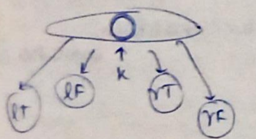
→ ex: T or F and (T) xor F

(i) k=i+1 k+=2 (k+1) k=j-1 (j)




```
∴ int ans=0;
for(int k=i+1; k<=j-1; k=k+2){
    int lT = solve(s, i, k-1, T) , lF = solve(s, i, k-1, F);
    int rT = solve(s, k+1, j, T) , rF = solve(s, k+1, j, F);
}
```

→ temp ans



→ ((_) XOR (_)) left: lT lF right: rT rF → ans += lT*rF + rT*lF

```
∴ if(s[k]=='&'){
    if(isTrue == true) ans += lT*rT;
    else ans+= (lF*rT) + (lT*rF) + (lF*rF);
}
else if (s[k] == '|'){
    if(isTrue== true) ans+= (lT*rT) + (lF*rT) + (lT*rF)
    else ans+= (lF*rF);
}
else if( s[k] == '^'){
    if (isTrue == true) ans+ =(lT*rF) + (lF*rT);
    else ans+= (lF*rF) + (lT*rT);
}
return ans;
```

T&T = T, rest all bool exp = F

→ If any '1' = T, then 'OR' = true

} written inside 'for' loop

Now memoizing at each step, to avoid TLE:

↳ Since here 3 var.'s are changing → so we create 3D Dp

(dimension depends on no. of variables that changes in code.)

Code: (as per constraints in que.) → T/F

```
int t[201][201][2];
int solve(string s, int i, int j, bool isTrue){
    if(i>j) return 0;
    if(t[i][j][isTrue] != -1) return t[i][j][isTrue];
    if(i==j){
        if(isTrue==true) return s[i]=='T';
        else return s[i]=='F';
    }
    int ans = 0;
    for(int k = i+1; k<=j-1 ;k=k+2){
        if(t[i][k-1][1]==(-1)) t[i][k-1][1] = solve(s, i, k-1, true);
        if(t[i][k-1][0]==(-1)) t[i][k-1][0] = solve(s, i, k-1, false);
        if(t[k+1][j][1]==(-1)) t[k+1][j][1]= solve(s, k+1, j, true);
        if(t[k+1][j][0]==(-1)) t[k+1][j][0]= solve(s, k+1, j, false);
        if(s[k]=='&'){
            if(isTrue==true) ans+=(t[k+1][j][1] * t[i][k-1][1]);
            else ans+=(t[k+1][j][1] * t[i][k-1][0] + t[k+1][j][0] * t[i][k-1][1]
                      + t[k+1][j][0] * t[i][k-1][0]);
        }
        else if (s[k]=='|'){
            if (isTrue ==true) ans+=(t[k+1][j][1] * t[i][k-1][0] + t[k+1][j][0] * t[i][k-1][1]
                      + t[k+1][j][1] * t[i][k-1][1]);
            else ans+=(t[k+1][j][0] * t[i][k-1][0]);
        }
        else if(s[k]=='^'){
            if (isTrue ==true) ans+=(t[k+1][j][1] * t[i][k-1][0] + t[k+1][j][0] * t[i][k-1][1]);
            else ans+=(t[k+1][j][1] * t[i][k-1][1]
                      + t[k+1][j][0] * t[i][k-1][0]);
        }
    }
    return ans;
}

int countWays(int n, string s){
    memset(t, -1, sizeof(t));
    return solve(s, 0, n-1, true);
}
```

t[i][j][1] → denotes no. of ways to parenthesize the substring 's[i...j]' so that it evaluates to 'true'.

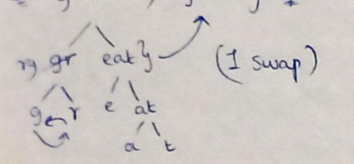
t[i][j][0] → denotes no. of ways to parenthesize the substring 's[i...j]' so that it evaluates to 'false'.

4) Scrambled String

Flow → Prob. Stmt. : I have 2 strings 'a' & 'b' ; if 'b' is scramble string of 'a' → return true, else return false
→ How to identify
→ How to approach
→ How to breakdown
→ Base Case
→ Code.

To scramble the string, choose any non-leaf node & swap it's 2 children.

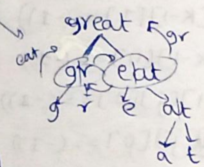
eg: great , rgeat } → true ; great →(MCM) / { great ... great } X → not a scramble string



(1 swap)

∴ We can swap from i=1 to i=n-1

eg: great, great } → scramble ✓ , great, eatgr } scramble

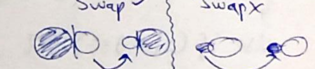


∴ Scramble ✓

swap ✓ great eatgr , great eatgr

swap X great great

∴ In general:



Scramble → i=1 to n-1
swap ✓ : a→first : b→last
swap X : first : b→first

∴ Case - I : Swap ✓

if(solve(a.substr(0,i) , b.substr(n-i,i))
and
solve(a.substr(i,n-i), b.substr(0, n-i))) → (both must be true)

∴ Case - II : Swap X

if(solve(a.substr(0,i), b.substr(0,i))
and
solve(a.substr(i,n-i), b.substr(i,n-i)))

Also
→ if (a.len != b.len)
→ return F.
→ if(a.compare(b)==0)
→ return T
(zero swap)

if → both empty → return T

Either of Case1 or Case2 must be true to prove String scramble.

∴ Basic recursive code.

```
bool solve(string a, string b){
    if (a.compare(b)==0) return T;
    if (a.size()<1) return F,
    int n = a.size(); bool flag = false;
    for(int i = 1; i<=n-1; i++)
        if(case1 || case2)
            flag = true; break;
    return flag;
```

Now memoizing it; → We can use map / 3D matrix → t[i][j][k]

(unordered_map<string,bool> mp) ← (a + " " + b) → will store whether the key is scrambable or not

will store whether substr of len 'k' starting at 'i' in str. 'a' is a scrambled version of the substr of len 'k' starting at 'j' in str. 'b'

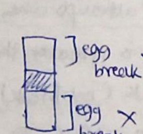
little complex → so using map

Code:

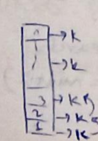
```
unordered_map<string,bool> mp;
int solve(string a, string b){
    if(a.compare(b) == 0) return 1;
    if(a.size() < 1) return 0;
    int flag=0, n=a.size();
    string key = a + " " + b;
    if(mp.find(key) != mp.end()) return mp[key];
    for(int i = 1; i<=n-1; i++){
        if(((solve(a.substr(0,i), b.substr(n-i,i)) == 1 and
            (solve(a.substr(i,n-i), b.substr(0,n-i)) == 1)) or
           ((solve(a.substr(0,i), b.substr(0,i)) == 1) and
            (solve(a.substr(i,n-i), b.substr(i,n-i)) == 1))){
            flag = 1;
            break;
        }
    }
    return mp[key] = flag;
}
```

5) Egg Dropping Problem

Flow → Prob. Stmt : 'e' eggs & 'f' floors. We can drop egg from any floor, if it doesn't break → we can reuse our egg. An egg can also be remain unbroken from a floor. We have to determine min. no. of moves to find the value of 'f' → where 0<=f (egg will not break) & >f (threshold floor) (egg will break)
→ MCM Pattern
→ Break down
→ Recursive Code



∴ We need: in worst case, we need to use best technique to minimize the no. of attempts to find out threshold floor.

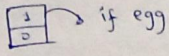


MCM format
1) Find 'i' & 'j'
2) BC
3) K loop
4) Ans.

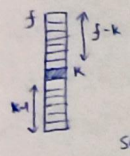
∴ for (k = 1 ; k <= f ; k++)

Base Case: → If '0' egg → not possible
↳ If '1' egg → at worst, we need to go through all floors to find ↳ (maybe last floor is threshold floor)

→ If f = 0 → return '0' } → return 'f'
↳ If f = 1 → return '1' ↳



→ if egg drop → break ✓ → ∴ $f_T = 0$ } ∴ only 1 attempt needed.
→ not break → ∴ $f_T = 1$ }



∴ solve(e,f)

break ✓ → solve(e-1, k-1)

break ✗ → solve(e, f-k)

ex: 10 floor; egg → dropped from 7th floor ← didn't break

need to check next 3 floors, we need to use 'k' loop again.

Here k=1 = 8th floor
k=2 = 9th floor
k=3 = 10th floor

So to answer the initial ques., we need to check min. no. of attempts in next 3 floors & that's why sending (f-k) & k range is (1, f-k).

∴ Base Code:

```
int solve(int e, int f){
    if (f==0 || f==1) return f;
    if (e==1) return f;
    int mn = INT_MAX;
    for(k=1; k<=f; k++){
        int temp = 1 + max(solve(e-1,k-1), solve(e,f-k));
        mn = min(mn, temp);
    }
    return mn;
}
```

(worst case; min. no. of attempts)

* We don't have any facts that from a particular floor, dropping egg → may break or not, similarly for every floor → egg may/may not break.

In memoization: (pencil)

Further optimization:

```
if (t[e-1][k-1] != -1) lo = t[e-1][k-1];
else {
    lo = solve(e-1, k-1);
    t[e-1][k-1] = lo;
}
if (t[e][f-k] != -1) hi = t[e][f-k];
else {
    hi = solve(e, f-k);
    t[e][f-k] = hi;
}
int temp = 1 + max(lo, hi);
```

{full ques: 'coding' file}

Final Opt.: (using BS)

reduction cases: (see next page)

↳ if worst-case no. of attempts when egg breaks (breaked) is greater than when it doesn't break (notBreak), we move the search range to the lower half (hi = mid - 1)

Otherwise, we move the search range to upper half (lo = mid + 1)

Code:

```
vector<vector<int>> t;
int solve(int e, int f){
    if (f<=1) return f;
    if(e==1) return f;
    if(t[e][f]!=-1) return t[e][f];
    int mn=INT_MAX, lo=1, hi=f;
    while(lo<=hi){
        int mid = (lo+hi)/2;
        int breaked, notBreak;
        if(t[e-1][mid-1]!=t1)) breaked = t[e-1][mid-1];
        else{
            breaked = solve(e-1, mid-1);
            t[e-1][mid-1] = breaked;
        }
        if(t[e][f-mid]!=-1) notBreak = t[e][f-mid];
        else{
            notBreak = solve(e, f-mid);
            t[e][f-mid] = notBreak;
        }
        int temp = 1 + max(breaked, notBreak);
        mn = min(mn, temp);
        if(breaked > notBreak) hi = mid-1;
        else lo = mid+1;
    }
    return t[e][f] = mn;
}

int superEggDrop(int e, int f){
    t.resize(e+1, vector<int>(f+1, -1));
    return solve(e,f);
}
```

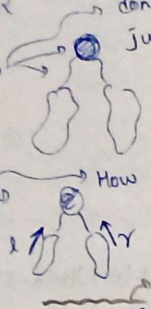
V) DP on Trees

1) General Syntax & how DP can be applied to trees (identification)
2) Diameter of a Binary Tree
3) Max. path sum from any node to any
4) Max. path sum from leaf to leaf
5) Diameter of N-ary tree.

Format

```
int func (←Vp→){
    Base Case
    Hypothesis
    Induction
}
```

→ don't think from where ans. came, but just pass the value.



→ How value must be passed to the node.

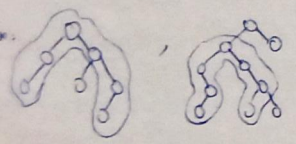
```
⇒ int solve(Node* root, int& res){
    if(root==NULL) return 0;        → base case
    int l = solve(root->left, res);
    int r = solve(root->right, res);   } → Hypothesis → Never changes
    int temp = calculate temp. ans
    int ans = max(temp, relation);     } → Induction
    res = max(temp, ans);
    return temp;
}
```

```
int func(root){
    int res = INT_MIN;
    solve(root, res);
    return res;
}
```

1) Diameter of Binary Tree

Flow → Prob. Stmt. : No. of nodes along longest path btw 2 leaf nodes.
- Syntax discussion :
- Code Variation :

ex:



1 + max(l,r) → temp
(if going via root node)

1 + l + r (if not going via root node)
↓
relation

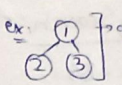
∴ Code:

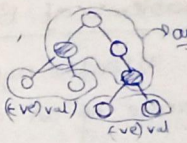
```
int solve(Node* root, int& res){
    // Base Case
    // Hypothesis
    int temp = max(l,r)+1;
    int ans = max(temp, 1+l+r);
    res = max(ans, res);
    return temp;
}
```

Induction → temp, ans, res

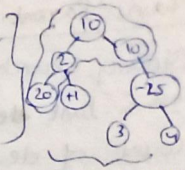
2) Max. Path Sum from any Node to any:

Flow → Prob. Stmt. ↗ , also Node can have (-)ve values also!
- Format Discussion
- Code Variation

ex:  = 6

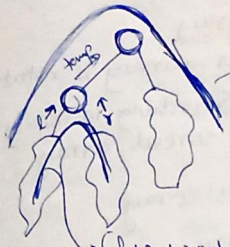


any →



ans = 20+2+10+10 = 42

(Max Sum needed)



temp → (max(l,r) + root->val, root->val)
- max
- Maybe child are (-)ve, so it's better not to include if parent = (+)ve

(l+r+root->val)
↓
('ans' is not via root node)

Induction → temp, ans, res

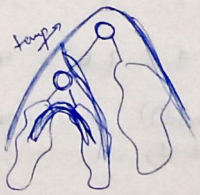
∴ Code:

```
int solve(Node* root, int& res){
    // Base Case
    // Hypothesis
    int temp = max(max(l,r)+root->val, root->val);
    int ans = max(temp, l+r+root->val);
    res = max(ans, res);
    return temp;
}
```

```
int main()
{}
int func(root)
```

3) Max. Path Sum from leaf to leaf:

Flow → Prob. Stmt. ↗ , also Node can have (-)ve values also!
- Format Discussion
- Code Variation



temp = max(root->val, max(l,r) + root->val)

If node → leaf node
↳ max(root->val, temp)

Code:

```
int solve(...){
    //Base Case
    //Hypothesis
    int temp = max(l,r)+root->val;   → calc. maxpathSum through curr. Node to one of its children
    int ans = max(temp, l+r+root->val);   → calc. maxpathSum through the curr. Node including both the children
    if(root->left and root->right) res = max(res, ans);
    return temp;
}
```

(ensures path sum is btw 2 leaf node)

(if single subtree → then discarded)

```
int main(){
    int res = INT_MIN;
    solve(root, res);
    return (res==INT_MIN)? -1 : res;
}
```

BINARY SEARCH

Main / Parent Problems of Binary Search: → whenever 'sorted' given in ques., pause & think how do we apply binary search? It's a hint

1) Binary Search
2) Order agnostic BS
3) 1st & last occurence of an ele. in sorted array
4) Count of ele. in sorted array
5) No. of times → array is rotated
6) Find an ele. in rotated sorted array
7) Searching in nearly sorted array
8) Floor / Ceil of an element
9) Next Letter
10) Index of first '1' in ∞ sorted array
11) Find the position of an ele. in ∞ sorted array
12) Min. difference ele. in a sorted array
13) Bitonic array max. element
14) Search in a Bitonic array
15) Search in row-wise + col-wise sorted matrix
16) Find ele. in sorted array that appears only once
17) Allocate min. no. of pages

I) Binary Search

Sorted: 1 2 3 4 5 6 7 8 9 (indices 0 1 2 3 4 5 6 7 8)

(is must for BS)

start, mid, end

ele = 4

int mid = start + (end - start)/2 ← to avoid overflow issues

```
Code:
int start = 0, end = size - 1
while(start <= end){
    int mid = (start + end)/2;
    if(ele = arr[mid]) return mid;
    else if(ele < arr[mid]) end = mid - 1;
    else start = mid + 1;
}
return -1;
```

On reverse sorted array: 9 8 7 6 5 4 3 2

Changes → if(ele < arr[mid]) start = mid + 1
else end = mid - 1;

II) Order Agnostic BS

Here order won't be given, so → if size == 1 → check ele == arr[0]
(array will be sorted)

else → arr[0] < arr[1] → so ascending
arr[0] > arr[1] → so descending
} Now apply BS accordingly

III) 1st & last occurence of an ele. in sorted array

ex: 2, 4, 10, 10, 10, 18, 20 ; ele. = 10

Prob. = search 10 + ind. (1st occurence / last occurence) → (BS)

1st occurence:

```
int start = 0, end = size - 1, res = 0
while(start <= end){
    int mid = start + (end - start)/2;
    if(ele == v[mid]){
        res = mid;
        end = mid - 1;
    }
    else if(ele < v[mid]) end = mid - 1;
    else start = mid + 1;
}
return res;
```

ele. might be present before curr. index, so ↓ end

for last occurence:

```
res = mid
start = mid + 1;
```

ele. might be present after curr. index, so ↑ start

IV) Count of ele. in sorted array

ex: 2, 4, 10, 10, 10, 18, 20 ; ele = 10

(count = 3)

∴ int first = BS (first occ. of ele.)
int last = BS (last occ. of ele.)
return (last - first + 1);

Whenever 'sorted' array is mentioned, try to think of binary search.

V) No. of times a sorted array is rotated

ex: arr[] = 11, 12, 15, 18, 2, 5, 6, 8 (0 1 2 3 4 5 6 7), initially → 2, 5, 6, 8, 11, 12, 15, 18

∴ If you notice → (no. of times arr. is rotated) = (index of min. ele.) → if clockwise rotation

ex: 18 2 5 6 8 11 12 15 (s, prev, mid, next, e)

; min. ele → less than it's prev. ele.
→ less than it's next ele.

18, 2, 5, 6, 8 — unsorted
6, 8, 11, 12, 15 — sorted

(next) → (mid + 1) % N
(prev) → (mid + N - 1) % N
(to avoid out of bound)

{min. will always lie in unsorted part} ; if(a[mid] <= a[next] and a[mid] <= a[prev]) return mid;

Note: No. of times array rotated depends upon the direction we are considering.

Algo:
if(a[mid] <= a[end]) end = mid - 1
(right half sorted, so ele. in left half)
, else if(a[mid] >= a[start]) start = mid + 1
(left half sorted, so ele. in right half)

If rotated anti-clockwise → no. of times = N − (min. Index)
(Index of min. value)

VI) Find an element in rotated sorted array

ex: 11, 12, 15, 18, 2, 5, 6, 8 → to search in this

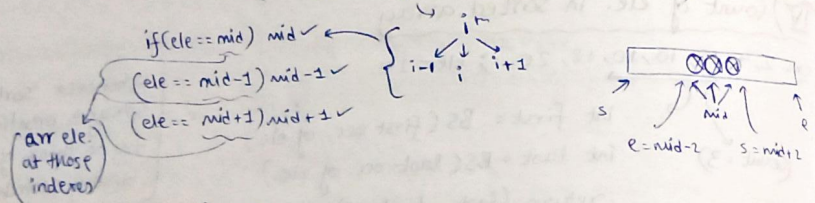
No. of rotation = index of min. ele. → BS(arr, start, end)

we will get → Now do: BS(arr, 0, index−1)
BS(arr, index, size−1)

VII) Searching in nearly sorted array

(elements may have been moved to a neighbouring position.)

ex: 5, 10, 30, 20, 40



Algo.:

```
while(start <= end){          // (N-1)
    int mid = (start+end)/2;
    if(a[mid] == target) return mid;
    if(mid>low and a[mid-1]==target) return mid-1;
    if(mid<end and a[mid+1]==target) return mid+1;
    if(a[mid]> target) end = mid-2;   // target can't be at mid+1, so skip
    else start = mid+2;               // target can't be at mid-1, so skip
}
return -1;
```

VIII) Floor of an ele. in a sorted array

ex: 7.8 → 8 → ceil, 7 → floor ; 7 → ceil, floor

∴ Floor of a no.(N) = greatest ele. smaller than N

ex: 1, 2, 3, 4, 8, 10, 10, 12, 19 ; x = 5 → ans = 4

```
if(a[mid] < ele){
    res = a[mid];          // potential answer
    start = mid+1;
}
else if(a[mid] > ele){
    end = mid-1;
}
if(a[mid] == key)
    return arr[mid];
```

IX) Ceil of an ele. in a sorted array

Ceil of a no.(N) = smallest ele. greater than N

```
if(a[mid] == ele) return a[mid]
else if(a[mid] < key) start = mid+1;
else {
    res = a[mid];          // potential answer
    end = mid-1;
}
```

X) Next Letter

Prob. Stmt: Given array of alphabets and a key, find ceil of key. (if key found, then return it's next greater char)

ex: [a, c, f, h], key = 'f' → ans = 'h'

∴ 'f' found, but we need it's 'Next Letter'

∴ Algo.:

```
if(a[mid] == key) start = mid+1;
else if(a[mid] < key) low = mid+1;      // key is greater
else if(a[mid] > key){                  // key is smaller
    res = mid;
    high = mid-1;                       // potential ans.
}
return '#';
```

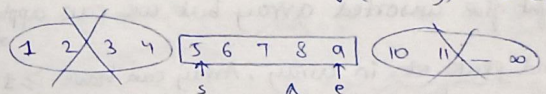
XI) Find position of an ele. in an ∞ sorted array

ex: 0 0 0 0 ···· ∞ ; Ex: key = 7 ; 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ···· ∞

(start) (end)??

Let → s, e, e, e, e

If (e < key) → e = e*2 & moving 's' to prev 'e'.

∴ 


```
∴ int low = 0, high = 1;
while(key > a[high]){
    low = high;
    high = high*2;
}
BS(arr, low, high);          // applying BS
```

XII) Find index of 1st '1' in a ∞ binary sorted array

ex: 0 0 0 0 0 0 1 1 1 1 ... ∞
0 1 2 3 4 5 6 7 8 9
s ... e ... e
(5)

∴ Algo:

→ // Find range (same as prev. que)

→ if (a[mid] == key){
    res = mid;
    end = mid - 1;
}

↳ this que. ≡ (1st occurrence key + ∞ sorted array - key)

XIII) Min. difference ele. in a sorted array

↓

ex: 4, 6, 10 , key = 7 ; We are given arr. & key, we need to find that ele. which on subtraction with key will give min. value.

since sorted array, we can find min. of 'floor' & 'ceil' val. → Need to apply BS → twice

(or) similar approach;

∞ 1, 3, 8, 10, 15 , key = 7 → Apply BS
(hi, lo) → after while loop ends → 'hi', 'lo' point to nearest neighbours

So: abs(arr[lo] - key)
abs(arr[hi] - key) → min.

XIV) Peak Element

Here, 'BS' on answer concept is used

↳ que. might give unsorted array but we can apply BS on our answers

Flow → Prob Stmt.: Find peak ele. in array. Array can have ≥ 1 peaks.
↳ greater than it's neighbours
↳ Criteria: → left to right
→ mid to ans
↳ Code

only 1 neighbour

ex: 5, 10, 20, 15 ; 10, 20, 30, 40 & 40, 30, 20, 10
mid-1, mid, mid+1 ; mid-1 ; mid+1

a[ele] > a[ele+1]
&
a[ele] > a[ele+1]

; 2, 5, 10, 20, 15
s, left, mid, right, e

We move pointers in direction where ele. at mid is smaller. In this case, we move 's' to right side.

∴ Algo:

```
while(s<=e){                                   (arr.size() - 1)
    int mid = s + (s-e)/2;
    if( mid>0 and mid<(size-1)){
        if( a[mid] > a[mid-1] and a[mid] > a[mid+1]) return mid;
        else if(a[mid-1] > a[mid]) e = mid-1;
        else s=mid+1;
    }
    else if(mid==0){
        if(a[0]>a[1]) return 0;
        else return 1;
    }
    else if(mid==(size-1)){
        if(a[size-1] > a[size-2]) return size-1;
        else return size-2;
    }
}
```

XV) Find max. ele. in bitonic array

ex: a[] = 1, 3, 8, 12, 4, 2

↳ array which is monotonically ↑ at first & then monotonically ↓.
↳ This means that array can't have same ele.s i.e. a[i] != a[i+1]

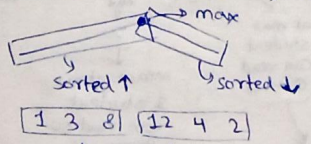
(this que. is similar to find 'peak' element.) → But here peak can occur only once in this que, else there can be many peaks in prev. que.

∴ (Max. ele. in bitonic arr.) = (finding peak ele.)

{∴ Concept BS can be applied on any that is monot. ↑ or monot. ↓}

XVI) Search an ele. in bitonic array

ex: a[]: 1, 3, 8, 12, 4, 2 ; key = 4



⇒ ∴ BS on both

(find peak ele. → prev. que.)

∴ BS(arr, 0, idx-1, asc);
BS(arr, idx, size-1, desc);

Any '1' will give ans.; if both returns '-1', the ele. not present → so ans. = (-1)

Also; Rotated Sorted Array ≠ Bitonic Array

↳ On rotation, this may never give a sorted array.

XVII) Search in row-wise & col-wise sorted array

Ex: 10 20 30 40 → consider 'mid' → i=0, j=m-1

a[][]: 15 25 35 46 ↓x
27 29 37 48 ↓x
32 33 39 50 (n×m)

Boundary condn.s → i>=0 & i<n & j>=0 & j<m

else if (target > mid) ⇒ move to next row

Key = 29 ↳ if target < mid ⇒ move to prev. col

∴ Algo.: int i=0, j=m-1
while(i>=0 and i<n and j>=0 and j<m){
if (a[i][j] == key) return i,j; → depends on que. on how we want to return. (ex: pair, vector)
else if (arr[i][j] > key) j--;
else if (arr[i][j] < key) i++;
}
return -1;

XVIII) Allocate min. no. of pages

This que. is similar to "The Chocolate Problem" → Binary Search Problems-3 → Book 1

Que: arr[i] = no. of pages in i^th book, 'm' studen

Allocate books such a way that;
1) Each student gets atleast 1 book
2) Each book should be allocated to 1 student only
3) Book allocation should be in contiguous manner.

Allocate books to 'm' students such that max. no. of pages assigned to a student is min.

ex: 10 20 30 40 ; m=2
10 30 60 (sum=100)
90 70 40
90 70 60 → we need min. of max. no. of pages
60 ✓

Now:
0 40 Mid 100
Mid = 50
means atmax 1 student can read 50 pages only
at max student can read 0 books
at max a student can read all books.

but if we want a student to read only 1 book, then we will give him max. no. of pages book & rest all other books to someone else. If we give min. no. of pages books, it might give us wrong ans as we need min. of max.

S1 = 10+20+~~30~~
S2 = 30+~~40~~
S3 = 40
we have 2 students but getting 3 students so we need to allocate more pages to a book.
↳ ∴ start = mid+1

50 56 62 75 100
If books allocated = student to receive → decrease end
if books allocated > student to receive → increase start

∴ Algo. int start = max in arr.
int end = sum of all arr. ele.'s
int res = -1;
while (start <= end){
int mid = (start+end)/2;
if (isValid(arr,mid,m) == true){
res = mid;
end = mid-1;
}
else start = mid+1;
}

; bool isValid(arr, mid, m){
int student = 1, sum = 0;
for(int i=0; i<n; i++){
sum = sum + arr[i];
if(sum > mid){
student++;
sum = arr[i];
}
if(student > k)
return false;
}
return true;
}

//Base Case → if size of array < total Students → return -1, as not every student will get a book.

Tips For Coding Rounds

- Give some time to read que.s; → HackerRank → we need to complete only func.
  HackerEarth → we need to complete func. + i/p + o/p.
- Proper Time Management
  ↳ ex: 3 ques. (score), time = 90min
  Easy → 50 → x
  Med. → 100 → 2x
  Dif. → 150 → 3x / 6x
  6x = 90 ⇒ x = 90/6 = 15 min. on each que.
  ∴ E = 15 min, M = 30 min, D = 45 min } ± 2-4 min.

- Approach any coding prob.s:
  → Read question
  → See constraints → Calculation → T_c ; if n ≤ 10^9 → we can take 'int'
  ↓ approach (T2B)
  → By seeing constraints, we can calc. T_c & guess the soln.

Constraints	T_c
$n > 10^8$	O(log n) or O(1) — binary search
$n \leq 10^8$	O(n) ~ linear search
$n \leq 10^6$	O(n log n) ~ merge sort (map)
$n \leq 10^4$	$O(n^2)$

Constraints	T_c
$n \leq 500$	$O(n^3)$
$n \leq 100$	$O(n^4)$
$n \leq 25$	$O(2^n)$ ~ recursion
$n \leq 12$	O(n!) ~ permutations

- How to debug code:
  - → Checking code o/p using 'Custom ip button' & through 'dry run'.
  - → Using of 'print' statements to cross verify our expected & received values.
  - → stub (write down manually the expected value in place of the func. call)
    - ex: `x = func(-){ }` → instead of func. → just write : 'x = 11' (expected)
      - now check if ans. is coming?
        - If yes → error in func. of 'x'
  - → If segmentation fault, 'comment' some part of code } do this for all parts to find the region where segmentation error is coming.

- Types of errors:
  - → Syntax error
  - → Logical error
  - → Runtime error } caused by invalid operations
    - ex: dividing by '0'.
  - → Compilation error
  - → Type error
  - → Resource error } (exhaustion of system resources like memory)
    - (occurs when a program tries to access a memory location that it is not allowed to access.) ← (Segmentation fault)

FIND ALL MISSING & DUPLICATE NO.'s FROM 1 TO 'N' ARRAY

Que: Given array → ele. = '1' to 'N' & there is a missing ele. which is replaced by a duplicate ele. Find both.

Soln.: ex: 1,4,3,4,5 → Using a map → ele. & freq.

↓

$T_c = O(N)$, $S_c = O(N)$

Using Math → (2 var.s → 2 eqn.s)

- sum (1 to N) − sum(arr) = b − e → C1 (b: missing, e: duplicate)
- sum of squares (1 to N) − sum of sq.(arr) = $b^2 - e^2 = (b-e)(b+e)$ → C2

∴ I have C1, C2, b, e → ∴ we can get 'b', 'e' → Here → e = 4, b = 2

Using Math → $T_c = O(n)$, $S_c = O(1)$

Best Approach: Using "Swap Sort"

(Prev. approach good if only 1M & 1D. If >1, we may get confused.)

Swap Sort works well if $\frac{1M}{1D} >=$ 1

(Drawback: swap sort can't be used if 'vector' is const., i.e. can't be modified)

ex: 2, 3, 2, 5, 1 → convert → $c^1_0\ c^2_1\ c^3_2\ c^2_3\ c^5_4$ (arr[i] = i+1)

Ideal: $c^1_0\ c^2_1\ c^3_2\ c^4_3\ c^5_4$

arr[i] != i+1 → ∴ Missing = i+1; Duplicate = arr[i]

(in that case, math approach is better)

ex: 3, 2, 1, 5, 1 (indices 0 1 2 3 4) ; arr[i] → arr[arr[i]-1] } ∴ if (arr[i] != arr[arr[i]-1]) swap

Note: Always use size to calculate here ; (largest no. could be missing, so (largest no. + 1 won't be size always)

ex: 2 3 1 8 2 3 5 1 (Multiple duplicates & missing)

(i) 3 2 1 8 2 3 5 1 (swap)

i: 1 2 3 8 2 3 5 1 ⇒ 1 2 3 1 2 3 5 8 (i5)

duplicate → so move 'i'

$c^1_0\ c^2_1\ c^3_2\ c^1_3\ c^5_4\ c^3_5\ c^2_6\ c^8_7$; 2 3 1 5 1 → Algo. (Swap Sort) → 1 2 3 1 5

Algo.:

```
int i = 0;
while (i < size){
    if (arr[i] != arr[arr[i]-1])
        swap (arr[i], arr[arr[i]-1])
    else
        i++
}
```

```
for(int i=0 ; i<size ; i++)
    if(arr[i] != i+1){
        i+1 → Missing
        arr[i] → duplicate
    }
}
```

$T_c = O(N)$, $S_c = O(1)$; drawback: If array is 'const' i.e. can't be modified, then better to use 'math' approach as this approach changes array ordering.

HEAP

Identification: 1) K + 2) Smallest / Largest

- K + smallest = max heap
- K + largest = min heap

Max Heap: `priority_queue<int> maxh;`

Min Heap: `priority_queue<int, vector<int>, greater<int>> minh;`

If we need to add a pair in heap → use : `pair<int, int>`

to avoid rewriting, `#define ppi pair<int,int>` → Use "ppi"

I) K^th Smallest Element

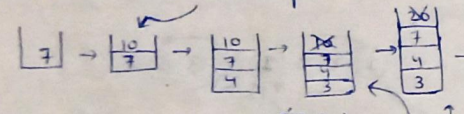
Flow → Prob. Stmt.: Find (K) in array
- Sorting, Identification
- Code
- Other method

ex: arr[]: 7 10 4 3 20 15, K = 3

↓ In pq

3	4	7	10	15	20

K^th Smallest → so "max Heap"



size > K → so pop top ele.

top → K^th Smallest = (n−K−1)^th largest

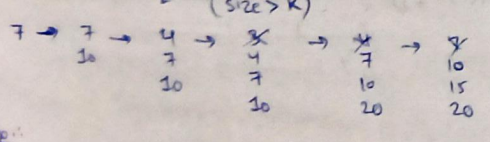
Algo.:

```
for(int i=0;i<size;i++){
    maxh.push_back(arr[i]);
    if(maxh.size() > k) maxh.pop();
}
return maxh.top();
```

II) Return K largest elements in array

K largest → so 'min heap' ; ex: arr[]: 7 10 4 3 20 15, K = 3

(size > K)



∴ Add all ele. in min. heap. If min heap size > K, pop the 'top' ele.

Algo.:

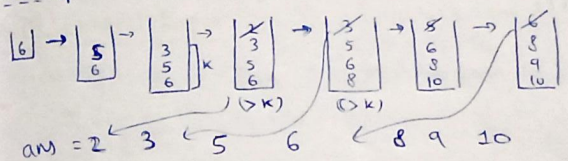
```
while(minh.size()>0){
    cout << minh.top() << " ";
    minh.pop();
}
```

III) Sort a K-Sorted Array

ex: (6, 5, 3, 2, 8, 10, 9) & K = 3 → means curr. ele. (as '0', it's left & right side till K i.e. { ← K ele. → K }, in between that range, the correct ele. will be present which should be at index of curr. ele.)

- We can use sorting on whole array → $T_c = O(n \log n)$
- Using min heap → $T_c = O(n \log K)$ → ex: 6, 5, 3, 2, 8, 10, 9 (K size)

Min Heap: → (sorted internally)



ans = 2 3 5 6 8 9 10

∴ Add ele. in min. heap, if minh.size() > K, then pop that ele. & add to 'ans' vector.

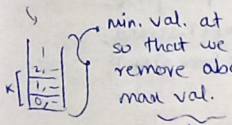
IV) K Closest Numbers

ex: arr[]: 5 6 7 8 9, K = 3, x = 7 → Find 'K' closest no.'s to 'x'

5	6	7	8	9
− 7	7	7	7	7
2	1	0	1	2
↓	↓	↓		
(6)	(7)	(8)		

} abs(a[i] − x)

Now Closest



min. val. at bottom so that we can remove above max val. → ∴ Max Heap

(if we sort this diff.)

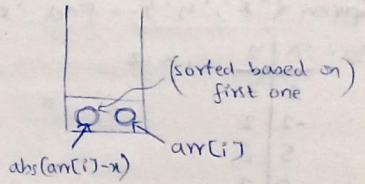
diff: 0 1 1 2 2 → so we need to push a pair in our heap

ele: 7 6 8 5 9

∴ Adding ele. based on 'diff' ✓

Algo.:

```
priority_queue<pair<int,int>>p;
for(int i=0; i<size; i++){
    p.push({abs(arr[i]-x), arr[i]});
    if(p.size()>K) p.pop();
}
while(p.size()>0){
    cout<<p.top().second<<" ";
    p.pop();
}
```



V) Top K Frequent No.'s

arr[]: 1,1,1,3,2,2,4, K = 2

freq
1 → 3 ✓
3 → 1
2 → 2 ✓
4 → 1

* Largest, Greatest, Top → (Min. Heap)

* Smallest, Lowest, Closest → (Max. Heap)

Using → unordered_map → to get ele. + freq. ⇒ for(auto x : arr) mp[x]++;

1 | 1 | 1 | 3 | 2 | 2 | 4 ⇒

1	3
3	1
2	2
4	1

↑ arr[i] ↖ Freq. → Adding to min. heap based on freq.

Algo:

// priority_queue <int, vector<int>, greater<int>> pq (pair<int,int>)

for(auto i=mp.begin(); i!=mp.end(); i++){
  pq.push({i->second, i->first});
  if(pq.size()>k) pq.pop();
}
while(pq.size()>0){
  pq.top()
  pq.top().second
}

VI) Frequency Sort

ex: a[]: 1,1,1,3,2,2,4 } → Sort based on freq. } → op: 1 1 1 2 2 3 4

mp: 1→3, 3→1, 2→2, 4→1 → Adding in max heap based on freq → in / end →

while(maxh.size()>0){
  int freq = maxh.top().first;
  int ele = maxh.top().second;
  for(int i=1; i<=freq; i++){
    cout<<ele;
    maxh.pop();
  }
}

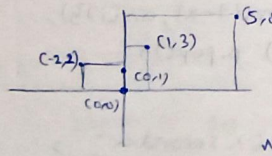
VII) K-Closest Points to Origin

Given {'x','y'} → find 'k' closest points to {0,0}

ex:

x	y
1	3
-2	2
5	8
0	1

, k=2 ≡



distance = $\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$

min $\{\sqrt{x^2+y^2}\}$

.. We store, distances & co-ordinates in our max heap

(not necessary to find sqrt., we can simply add value of x^2+y^2)

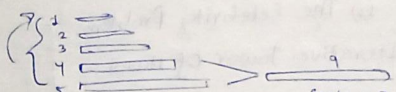
(furthest points will get eliminated)

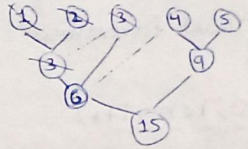
Algo: arr[n][2]; maxh → priority_queue<pair<int, pair<int,int>>> maxh;

for(int i=0; i<n; i++){
  maxh.push({arr[i][0]*arr[i][0] + arr[i][1]*arr[i][1], {arr[i][0], arr[i][1]}});
  if(maxh.size()>k) maxh.pop();
}
while(maxh.size()>0){
  pair<int,int> p = maxh.top().second;
  cout<<p.first<<" "<<p.second;
  maxh.pop();
}

VIII) Connect Ropes

ex: arr[]: 1 2 3 4 5 } → Find min. cost to join them





Cost: 3+6+9+15 = 33 (min)

∴ To get min. → we need smallest 2 val. → add & again put in the DS

(We use min. heap)

base case: (if only 1/0 ele. is in 'pq' → break)

Algo:

while(minh.size()>=2){
  int first = minh.top();
  minh.pop();
  int second = minh.top();
  minh.pop();
  cost += (first+second);
  minh.push(first+second);
}
return cost.

IX) Sum of elements

Given arr. & k1th & k2th → find sum of ele. btw k1 & k2.

(smallest)

ex: arr[] = 1,3,12,5,15,11 ; k1=3, k2=6

1, 3, 5, 11, 12, 15

3rd, 6th

(11+12 = 23)

∴ Algo:

first = ← KthSmallest(arr, k1)
second = ← KthSmallest(arr, k2)
sum = 0;
for(i=0; i<n; i++){
  if((a[i]>k1 and a[i]<k2) or (a[i]<k1 and a[i]>k2)){
    sum += a[i];
  }
}

(return sum) ←

STACK

Main/Parent Problems of Stack:

1) Nearest greater to left
2) Nearest greater to right (or) next largest element
3) Nearest smaller to left (or) nearest smaller element
4) Nearest smaller to right
5) Stock Span Problem
6) Max. Area of Histogram

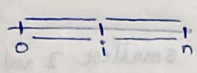
→ 1 concept

7) Max. Area of rectangle in binary matrix
8) Rain Water Trapping
9) Implementing a min. stack
10) Implement Stack using heap
11) The Celebrity Problem
12) Longest Valid Parenthesis
13) Iterative Tower Of Hanoi

→ Concept!!

→ `for(int i=0; i<n; i++)`

for (j→0 to i ; j++ / j→i to 0 ; j-- / j→i to n ; j++ / j→n to i ; j--) ⇝ `j = func(i)` (second loop) → O(n²)



; 'j' loop if dependent on 'i'
⇓
Stack ✓

→ Read Prob. & check if it's any variation.

I) Next Greater to right/Next Largest Element

ex: 1 3 2 4
op: 3 | 4 | 4 | -1

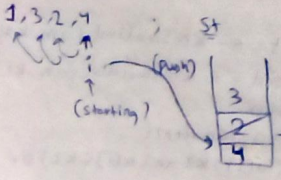
; brute force: `for(int i=0; i<n-1; i++)` `for(int j=i+1; j<n; j++)` → O(n²)

[We are going L→R in 2nd loop, so we go R→L in stack]

∴ Stack ✓ (since 'j' depends on 'i')

Cond.'s:
- stack empty → -1
- st.top > a[i] → st.top()
- st.top <= a[i] → pop → stack empty / st.top() > than a[i]

ex: 1,3,2,4 ; st



→ 'pop' until 'top' ele. is (not) less than curr. ele.

, ans → -1 | 4 | 4 | 3
↳ 'reverse' it

Algo.:
```
vector<int> v, stack<int> s;
for(int i=size-1; i>=0; i--){ ——— (1)
    if(s.size()==0) v.push_back(-1) (3)
    else if(s.size()>0 and s.top()>arr[i]) v.push_back(s.top());
    else if(s.size()>0 and s.top()<=arr[i]){ (4)
        while(s.size()>0 and s.top()<=arr[i]) s.pop();
        if(s.size==0) v.push_back(-1);
        else v.push_back(s.top());
    }
    s.push(arr[i]);
}
reverse(v.begin(), v.end()); ——— (2)
return v;
```

II) Nearest Greatest to left

ex: 1, 3, 2, 4 ; bf → `for(int i=0; i<n; i++)` `for(int j=i-1; j>=0; j--)`
op: -1 | -1 | 3 | -1

'j' depends on 'i' ⇒ ∴ Stack ✓

[We are going R→L in 2nd loop, so going L→R in stack]

ex: 1, 3, 2, 4 ; st ; ans → -1 | -1 | 3 | -1
(i) (start)

→ pop until curr ele is not smaller than 'top' ele.

∴ Changes in above code
- (1) = `for(int i=0; i<size; i++)`
- (2) = remove that line → no need to reverse our array

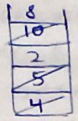
} See code with these changes

III) Nearest Smaller to left

ex: 4, 5, 2, 10, 8 ; bf → `for(int i=0; i<n; i++)` `for(int j=i-1; j>0; j--)`
op: -1, 4, -1, 2, 2

'j' depends on 'i' ⇒ ∴ stack ✓

[We are going R→L in 2nd loop, so we go L→R in stack] ⇒ ∴ ex: 4, 5, 2, 10, 8 → st:



→ ans → -1 | 4 | -1 | 2 | 2

∴ Changes in prev. code
- (1): `for(int i=0; i<size; i++)`
- (2): remove that line
- (3): Change '>' to '<'
- (4): Change '<=' to '>='

} See code with these changes

∴ Changes → 1) greater → less than
4) reverse ✗
3) Left traverse

IV) Next Smaller to right

ex: 4,5,2,10,8

o/p: 2,2,-1,8,-1

bf → `for(int i=0; i<n-1; i++)` `for(int j=i+1; j<n; j++)` — 'j' depends on 'i' ⇒ ∴ Stack

[we are going L→R in 2nd loop, so we go R→L in stack] → Changes $T_C = O(N^2)$ to $T_C = O(N)$

ex: 4,5,2,10,8 → st: [4, 5, 2, 10, 8] → ans: [-1, 2, -1, 2, 2] → (reverse)

∴ Changes in prev. code,
↳ (3) = Change '>' to '<'
(4) = Change '<=' to '>='
} see code with these changes

V) Stock Span Problem

Q: Given 'N' daily prices for a stock, calc. span of stock's price.
(max. no. of consecutive day (ft curr. day) & going backward for which stock price was <= price of that day)

ex: 100, 80, 60, 70, 60, 75, 85
o/p: 1 1 1 2 1 4 6
→ based on prev. 4 concepts, we see that we need to apply 'Next greater to left' concept.

∴ Consecutive smaller/equal to before it = next greater to left

In above ex: 100, 80, 60, 70, 60, 75, 85 (idx 0 1 2 3 4 5 6) → ans. for '75' = 5 - 1 = 4

∴ ans = (index - NGL - i)

∴ i: 100 80 60 70 60 75 85 (0 1 2 3 4 5 6)
v: -1 0 1 1 3 1 0 → storing 'NGL' ele. index.
o/p: 1 1 1 2 1 4 6 → (i - index) → ans ✓

∴ We create a pair (ele, (NGL, index))

∴ Code for NGL:

```
vector<int> v; stack<int> s;                    // stack<pair<int,int>> s;
for(int i=0; i<size; i++){
  if(s.size()==0) v.push_back(-1);
  else if(s.size()>0 and s.top()>arr[i]){
      v.push_back(s.top());                     // (s.top().first)
  }
  else if(s.size()>0 and s.top()<=arr[i]){      // (s.top().second)
    while(s.size>0 and s.top()<=arr[i]) s.pop;
    if(s.size()==0) v.push_back(-1)
    else v.push_back(s.top());                  // s.top().second
  }
  s.push(arr[i]);                               // {arr[i], i}
}
return v;
```

```
for(int i=0; i<v.size(); i++)  v[i] = i - v[i];
return v;
```

∴ (NGL variation Stock Span)

VI) Maximum Area Histogram

Q: Given heights → width = '1' → find area of rectangle → max.

ex: arr[] = {6,2,5,4,5,1,6} → [6, 2, 5, 4, 5, 1, 6] (idx -1 0 1 2 3 4 5 6 7)

NSE | PSE
NGE | PGE
} All 4 are imp. concepts

building can be expanded into building only when other building >= curr. building

5x2 = 10 → max.
4x3 = 12
for NSE (pseudo index)

∴ We need NSE & PSE of a building:

arr: 6 2 5 4 5 1 6
right: 1 5 3 5 5 7 7 (NSE) → (considering 7th idx height as '0' → so writing)
left: -1 -1 1 1 3 -1 5 (PSE)
(-1)th index

∴ width = right[i] - left[i] - 1

∴ area[i] = arr[i] * width[i]

Area: 6 10 5 12 5 7 6 ✓

∴ Algo.:

```
//NSR → same code as before
vector<int> right(n);
• Instead of push_back → do right[i] = ..
• Change 'if' cond. of
    ↳ if(s.empty()) right[i] = n;
}
```

```
//NSL
vector<int> left(n);
• Instead of push_back, do left[i] = ..
• Change 'if' cond. of
    ↳ if(s.empty()) left[i] = -1;
```

```
int ans=0
for(int i=0; i<n; i++) ans = max(ans, arr[i]*(right[i]-left[i]-1));
return ans;
```

VII) Max. Area rectangle in binary matrix

ex:
0 1 1 0
1 1 1 1
1 1 1 1
1 1 0 0

H1 (Histogram)
H2
H3
H4

we need max. area rectangle. } similar to prev. ques.

ans = max(MAH(H1), MAH(H2), MAH(H3), MAH(H4))

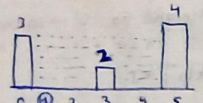
H1 → 0 1 1 0 → MAH , H2 → 1 2 2 1 → MAH , H3 → 2 3 3 2

H4 → 2 4 0 0 → MAH

∴ Algo.: {Matrix = n×m}

```
vector<int> v;
for(int j=0; j<m; j++) v.push_back(arr[0][j]);
int mx = MAH(v);
for(int i=1; i<n; i++){
    for(int j=0; j<m; j++){
        if(arr[i][j]==0) v[j]=0;
        else v[j] = v[j]+arr[i][j];
    }
    mx = max(mx, MAH(v));
}
```

VIII) Rain Water Trapping

ex: 

we need to find total amount of rain that can be stored.

Notice, total unit of water = Σ(water above each building)

0+3+3+1+3+0 = 10

To find height above that block → 3 | 0 | 0 2 0 4

3 = max in left, max in right = 4, min. = 3

∴ water[i] = min(maxL, maxR) − arr[i]

∴ ex: 3 0 0 2 0 4

mxl:	3	3	3	3	3	4
mxr:	4	4	4	4	4	4
min:	3	3	3	3	3	4
−	3	0	0	2	0	4

water[i] = 0+3+3+1+3+0 = 10

∴ Algo.:

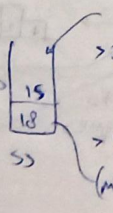
```
int mxl[size], mxr[size]
mxl[0] = arr[0]
for(int i=1; i<size; i++)
    mxl[i] = max(mxl[i-1], arr[i])
mxr[size-1] = arr[size-1]
for(int i=size-2; i>=0; i--)
    mxr[i] = max(mxr[i+1], arr[i])
int water[size]; int sum=0;
for(int i=0; i<n; i++){
    water[i] = min(mxl[i], mxr[i]) - arr[i];
    sum += water[i];
}
return sum;
```

IX) Min. Ele. in Stack With Extra Space

We need to write : push, pop, getMin

We use another stack

ex: 18 19 29 15 16



> If 'ss' = empty → insert ele. which was inserted in 'st'

> If curr ele. (st.top()) is '>' than ss.top() → don't add ele. in 'ss'

> else → add ele. to 'ss'

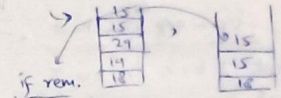
(min. ele.)

> While popping, if st.top() = ss.top() → remove both ele.; else remove only from 'st'

Algo.:

```
stack<int> s, ss;
void push(int a){
    s.push(a);
    if(ss.size()==0 or ss.top()>=a)
        ss.push(a);
    return;
}
int getMin(){
    if(ss.size()==0)
        return -1
    return ss.top();
}
```

ex: 18 19 29 15 15



(min = 15 still, so we need to add ele. → when: ss.top() == a)

```
int pop(){
    if(s.size()==0) return -1;
    int ans = s.top();
    s.pop();
    if(ss.top()==ans) ss.pop();
    return ans;
}
```

X) Min. ele. in Stack with O(1) Space → (Check Max ele. in Stack concept also)

Here we need to implement → push, pop, top, getMin

```
int getMin(){
    if(s.size()==0) return -1;
    return minEle;
}
```

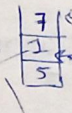
We will maintain this 'variable'

Now: → we modify → we will push (2*x − minEle) in our stack

push: (>= minEle) (<)

(6,5) → 5 | 6 — minele. = 5

ex: 5, 3, 7 →



(if ele. > minEle, push as it is in stack)

minEle = 5 → 3

(3 < 5 → so push (2*3 − 5) & (update minEle)

not actual ele.! but it acts as a checkpoint, that new min. ele. is inserted

if popping that ele. → (st.top() < minEle)

(new) Minele = 2*minEle − st.top()
= 2×3 − 1 = 5
(old minEle)

Algo.:

```
void push(int x){
    if(s.size()==0){
        s.push(x);
        minEle = x;
    }else{
        if(x>=minEle) s.push(x);
        else if(x<minEle){
            s.push(2*x - minEle);
            minEle = x;
        }
    }
}
```

```
void pop(){
    if(s.size()==0)
        return -1;
    else
        if(s.top()>=minEle)
            s.pop;
        else if(s.top()<minEle){
            minEle = 2*minEle - s.top();
            s.pop();
        }
}
```

```
int top(){
    if(s.size()==0)
        return -1;
    else{
        if(s.top()>=minEle)
            return s.top();
        else
            return minEle;
    }
}
```