Steps to create a cpp file & push it to remote repo

1. Create the cpp file.
2. Save it.
3. g++ helloworld.cpp -o helloworld
4. ./helloworld
   helloworld (output)
5. git init
6. git add helloworld.cpp git ignore or git add.
7. git commit -m "First Commit"
8. git remote add origin https://github.com/Adarshpat04/helloworld.git
   Copy paste the github repo link (private/public)
9. git push -u origin main ~~master~~
   After that, simply use git push.

DSA (C++) (DSA-notes)

Arrays :-

(1) Swap Alternate :-

Going into alternate indexes of array & swapping it using swap, otherwise using

```
temp = arr[i]
arr[i] = arr[i+1]
arr[i+1] = temp
```

(2) Find Unique element :-

XOR all elements
Since, $0 \wedge a = a$
$a \wedge a = 0$

Eg :- $2 \wedge 3 \wedge 1 \wedge 2 \wedge 3 \wedge 2 \wedge 2$
$= 0 \wedge 0 \wedge 0 \wedge 1$
$= 1.$

(3) Unique Occurences :-

arr[ ] = {1, 1, 2, 2, 2, 3, 3, 3, 3, 4}
(1 → 2, 2 → 3, 3 → 4)

2, 3 & 4 are unique counts
So, output :- true

Sort the array, using while loop & nested for loop iterating & comparing one element to rest & counting it. Then, incrementing while loop to i != count. And also breaking the for loop if if condition is not satisfied.

Making one more array & comparing one to rest to find unique occurences. Return

(4) Find Duplicate :-

arr[ ] = {1, 2, 3, 4, 4, 5, 6}
4 is the duplicate element.

1, 2, 3, 4, (4), 5, 6 {XOR all elements}
1, 2, 3, 4, 5, 6 {XOR the above with 1 to (N-1) (here 6)}
size of array

⑤ Find all Duplicates :-

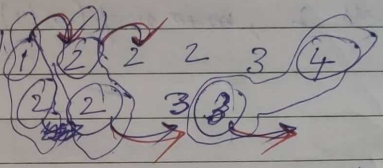
use a different approach

Uses concept of negatition by which the positive number means not visited else negative number means visited.

nums[abs(nums[i]) − 1)]

⑥ Find array Intersection :-

arr1[] = {1, 2, 3}
arr2[] = {1, 2}
Output :- 1, 2



arr[i] < arr[j] ⇒ i++
arr[i] == arr[j] ⇒ i++, j++
arr[i] > arr[j] ⇒ j++

} Two-pointer approach since, arrays are arranged in ~~descending~~ order

⑦ Pair Sum :-

1	2	3	4	5

S = 5

So, output :- sort { (1) 4, (2) 3 }

Using one loop iterating & using other nested comparing one with all other iteration if equal ~~sort~~ find min & max & pushing it ~~to~~ 2D array & finally ~~th~~ sorting the answer.

⑧ Sort 01 :-

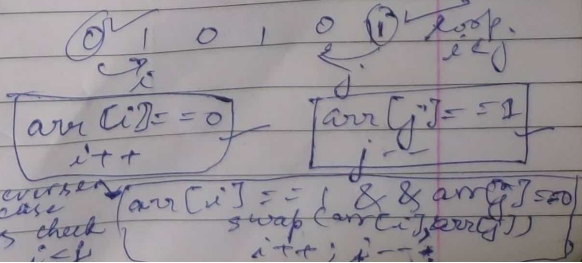
Check

0	1	0	1

Output :-

0	0	0	1	1	1

Two pointer approach



⑨ Sort 012 :- Dutch National Flag Algorithm

Output 0 0 1 1 2 2

l, m, h

1 0 2 1 0 2

low = 0 = mid , high = n-1

if 0 is copied to l

Now, l m goes front, l remains

if (mid] = = 0) swap (l, m), l++, m++

if 2, m is copied to h

remains since, m <= h.

(mid = = 1) m++

(mid = = 2) swap (m, h); h--

swap (l(0), m(1))

if again 1, m goes forward & l remains :

swap (m(2), h(n-1))

if it was 2, swap (m(1), h(0)) ← if l remains h--.

Basic Idea of Time Complexity & Space Complexity :-

Time Complexity :-

Slow → High processing → Algorithm A → Fast machine → o/p → Fast Algo

Fast → Slow processing → Algorithm B → Slow Machine → o/p → Slow Algo

Time complexity

- Amt. of time taken by an algo. to run.
- It is a function of length of the input.

Why Time complexity ?

- For making better programs.
- Comparison of Algo.

Big O notation — Upper bound (Worst case)

Theta θ — Avg. case

Omega Ω — Lower bound

Highest

$O(N!)$
$O(2^n)$
$O(N^3)$
$O(N^2)$
$O(N \log N)$
$O(N)$
$O(\log N)$
$O(1)$

Complexity

Least

Eg :- $N^2 + \log N \rightarrow O(N^2)$.

$8n^3 + 2n^2 + 5$ or $n^3/300 \rightarrow O(N^3)$.

$(n+4)/4 \rightarrow O(N)$.

M	T	W	T	F	S	S

Page No.:
Date:
YOUVA

Nested loops

Eg:- { (0 → N)
{ (0 → M) } } ⟶ O(N*M)

Seperate loops

Eg:- { 0 → N } → O(N + M)
{ 0 → M }

Eg:- Q.) int a = 0;

```
for(int i = 0; i < N; i++){
    for(int j = N; j > i; j--){ a = a+i+j; }}
```

for(0 → N){ ⟶ N
{ for (N → i) } } ⟶ N → O(N²) | i := 2 → N
N → 0 → 0 (worst case) | N-2 or O(N)

Stuck in TLE!!, see this :-

Cons
1 < n < 10⁶
1 < n < 1000

≤ [10...11]	O(n!), O(n⁶)
< [15...18]	O(2ⁿ * n²)
< 100	O(n⁴)
< 400	O(n³)
< 2000	O(n² * log n)
< 10⁴	O(n²) ✓
< 10⁶	O(n log n) ✓
< 10⁸	O(n), O(log n) ✓

(rows up to O(n² * log n) → X TLE)

* **Space Complexity** :- To solve the problems (Small input problem) → memory space needed not the inputs S.C.)

int a, b, arr[5] since, all are fixed → O(1)

vector → O(N) → if vector of length n else const.

Note:- If for (0 → N){
func()
} → Complexity will not be O(N) it depends on the function.
See the function.

Binary ~~tree~~ Search :-

mid = (Starting index + Ending ind.) / 2

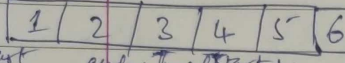
(1) ~~Binary~~ Search :-

Linear Search :- Using for loop doing 1000 comparisons.

Binary Search :- 1000 500 250 ... 10 comparisons.

N $N/2$ ... $N/2^k$ → length = 1

Target : 3



$\therefore N/2^k = 1 \Rightarrow k = \log N$.

Time complexity

~~mid~~ Applicable only if array is ~~the~~ sorted.

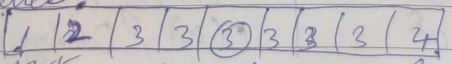
fancy way of writing mid = (start + end) / 2 is

mid = start + (end − start) / 2 to overcome

integer overflow ($2^{31} + 2^{31}$ ✗)

(2) First & last Position of an element in a ~~sorted~~ array :-

Applying the same binary search algo. but if the element is found which will be the mid then end−1 to find the first occurence & start + 1 to find the last occurence.



~~two~~ seperate loops (1st & last)

end = ~~end~~ − 1

For First occurence

start = start + 1

For ~~record~~ last occurence

(3) Total no. of occurences :-

Target :- 3

1 2 3 3 3 3 4

Output :- 5

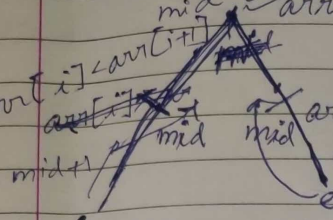
Find the first & last position by the above logic & Total no. of occ.

= (last index − first ind.) + 1

(4) Peak Index in Mountain array :-

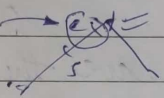
[1, 5, 10, 2]

mid ⇒ arr[i-1] < arr[i] > arr[i+1]



Why mid = e in second case?

Because if e goes less → it goes below the target.



And the loop ~~should~~ stop condition :- s < e not s <= e because if so then, again it will check if conditionals like arr[i] > arr[i+1], which we don't want. When we get the element it is the index.

(5) Find Pivot Index in an array :-

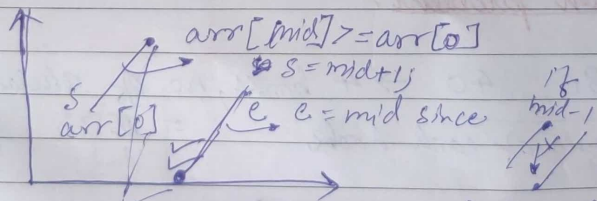
I-Approach

For a pivot the before elements should be greater than the pivot.

4, 5, 6, 7, 0, 1, 2

See the code for explaination.

II-Approach



loop condition :- s < e

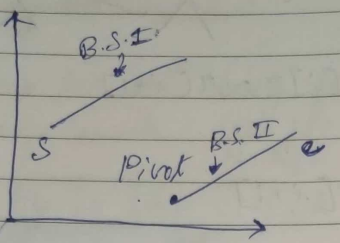
0, 1, 2, 3, 4, 5, 6, 7 → Sorted array

4, 5, 6, 7, 0, 1, 2 → Rotated Sorted array

return s or e

Note :- Pivot will be least element in the array

⑥ Search an element in rotated sorted array:-



⑦ Square root of x:-

Concept:-
Search space → minimize

0 —— 8 ↑

Eg:- $\sqrt{8}$ will have the output between 1 to 8. So, finding the mid element if equals OK otherwise, squaring the mid and checking whether is less than n (here, 8) if so, it can be the answer & hence doing the same for right part else if square is more than n then, checking the square at left part. For more precision, just adding 0.1 or 0.01 or so on to 8 and repeat the same procedure of square < n using nested for loop.

⑧ Book Allocation problem:-

10 20 30 40 → 4 books, no. of students = 2

no. of pages of each book

- Every book should be alloated.
- Every student should get atleast one book.
- such that the maximum number of pages assigned to a student is minimum.
- Contiguous manner 10 20 30 40
  I II

Eg:- 10 20 30 40 n = 4, m = 2

Case-I:- 10 | 20 30 40 — I 10, II 90 → max = 90

Case-II:- 10 20 | 30 40 — I 30, II 70 → max = 70

Case-III:- 10 20 30 | 40 — I 60, II 40 → max = 60

} min = 60

Search space → minimize concept.

0 ——— 10+20+30+40 = 100
s — mid = 50 — e

~~50~~ → No possible soln. → right part
0 ——— mid ——— 100
s = mid+1

Binary search — isPossible soln. (mid)

10 20 | 30 | 40 ✗
I → [10+20] + 30 ✗
II → [30] + 40 ✗
[1] → ✓ (n = 2)

51 ——— [75] ——— 100 → Possible soln. → left part
s — mid — e
e = mid−1

10 20 30 | 40
I → [10+20+30] + 40 ✗ ✓
II → 40

⑨ Painter's partition problem:-

array of boards lengths (one board / one unit time unit time)

| 5 | 5 | 5 | 5 | k = ② → painters

0 — [10] — 20

I → 5+5 store ans.
II → 5+5 move forward

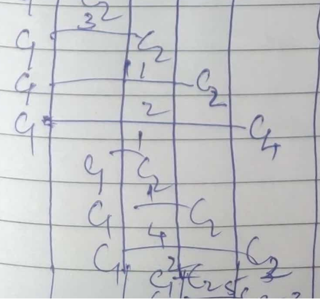
• Allocate in contiguous
• Q.) partition the boards such that time gets minimum

Same as book allocation.

⑩ Agressive cows problem:-

no. of cows

arr[ ] = {4, 2, 1, 3, 6}



max 5

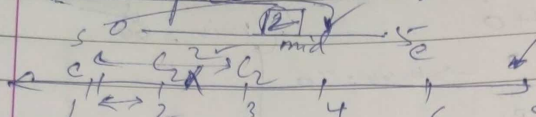
Assign the cows to the stalls, such that the minimum dist. b/w any two of them is as large as possible.

Eg:- [1 2 3] ✗ [4] ↓ not possible [5 6 7 8 - —]

When to apply Search space:- Minimize concept.

4 2 1 3 6

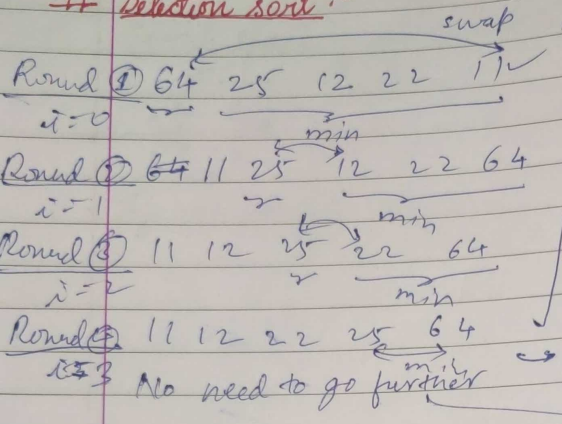
Search space :- min = 0, max = max - min = 5



If not possible → left part

end = mid - 1

Selection sort :-



- Space Complexity :- O(1) since only constants are used. Note :- Vector is used but as an input (parameter) to function.

- Time complexity :-

```
for(0 ⟶ <(n-1))
    for(i+1 ⟶ <n)
```

a	b c d e	4	(n-1)
a	b c d e	3	(n-2)
		⋮	⋮
		1	(n-2)

$1+2+3+4+\cdots+(n-2)+(n-1)$

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

- Best case T.C :- Already sorted case 1 2 3 4 5 but, still it will apply same logic ; O(n²).
- Worst case T.C :- 5 4 3 2 1 ⟶ O(n²).
- Use case :- array / vector / list size is small.

* Stable or Unstable :-

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

For Eg, consider array of pairs where the first element of the pair is the value & the second element is the index.

[(4,0), (3,1), (3,2), (5,3)]

A stable sort will preserve the order of the two (3,1) & (3,2) pairs because they have equal keys. So, the stable sorted output will be:-

[(3,1), (3,2), (4,0), (5,3)]

An unstable sort may not preserve this order, so the output could be:-

[(3,2), (3,1), (4,0), (5,3)]

⇒ Selection sort algorithm is not stable because it selects the minimum element from the unsorted part & swaps it with the first element in the unsorted part, which can change the relative order of elements with equal keys.

Bubble sort :- (Round ith) ith largest → right placed

1st Round → 1st largest element of array placed

Round-1
10 1 7 6 14 9
1 10 7 6 14 9 (swap)
1 7 10 6 14 9 (swap)
1 7 6 10 14 9 (swap)
1 7 6 10 14 9 (ignore)
1 7 6 10 14 9 (swap)
1 7 6 10 9 14 (ignore, sorted)

Round-2
1 7 6 10 9 14 (ignore)
1 6 7 10 9 14 (swap)
1 6 7 10 9 14 (ignore)
1 6 7 9 10 14 (swap)
1 6 7 9 10 14 (sorted)

Round-3
1 6 7 9 10 14

Round-4
1 6 7 9 10 14

Round-5
1 6 7 9 10 14

Round-6
1 6 7 9 10 14

a < b → ignore
a > b → swap

For n = length of array 6 so, No. of rounds = n - 1 = 5
= n-2 n-1

1 0 1 7 6 14 9 → j
sorted
0 → (n-1)
0 → (n-2) cause 14 is already sorted

- T.C. :-
1 → (n-1) comparisons
2 → (n-2)
.
.
(n-1)th → 1
T.C. = 1+2+3+ ... +(n-2)+(n-1)
= n(n-1)/2 = O(n²)

- S.C. :- O(1)

Optimize :-

- Best case → already sorted → (1, 2, 3, 4, 5, 6) → O(n)
a<b, b<c, c<d, d<e
a<b<c<d<e → sorted
↳ every time swapping not done

- Worst case :- ~~Reverse~~ ~~5,4,3,2,1~~ sorted → $O(n^2)$.

- Stability :- Stable

In Bubble sort, when two equal elements are encountered, the algorithm doesn't swap them, thus preserving their relative order. Therefore, Bubble sort is a stable sorting algorithm.

* In-place sorting :- (using same array as reference)

In-place sorting refers to sorting algorithms that rearrange data directly within the dataset, using only a constant amount of extra space. This is typically done by swapping data elements.

Examples of in-place sorting algorithms include quicksort, heapsort, insertion sort, selection sort, & bubble sort.
It's important to note that in-place doesn't necessarily mean the algorithm has a small space complexity. Eg :- quicksort → $O(n)$ → Space complex

Insertion sort:-

first element already sorted

i = 1

Round 1 :- 10 | 1 7 4 8 2 11

store & do right shift

but no space so right shift

j+1

Round 2 :- 1 10 | 7 4 8 2 11 if (arr[j] > arr[i]/temp)

shift = arr[j+1] = arr[j]

else break

j = 0 to i-1

Round 3 :- 1 7 10 | 4 8 2 11

⋮

Round 6 :- 1 2 4 7 8 10 11

- Adaptable
- Stable
- ~~Applied~~ Suitable for small & partially sorted arrays.
- 1st → 1
  2 → 2
  ⋮
  (n-1)

  } $O(n^2)$ ← T.C

- T.C. :- B.C. = $O(n)$ W.C. = $O(n^2)$.

  already sorted

  1 | 2 | 3 | 4 ... 4

  1st 1

  II → 1 (n-1) → $O(n)$

- S.C. = $O(1)$.

Standard ~~template~~ library (STL) :-

(1) Array :-

(Static)

array < int , 4 > a = { 1, 2, 3, 4 };

- Size :- a.size() ← ~~Empty~~ Size of array
- Empty :- a.empty() ← Check if empty or not (returns bool value)
- Front :- a.front() ← Front element (returns)
- Back :- a.back() ← Back element (returns)
- Element at Index :- a.at(2)

(2) Vector :-

(dynamic)

vector < int > v;

- v.capacity(); Output :- 0 -: Capacity (gets double everytime)
- v.push_back(1); -: Adding element
  v.capacity(); Output :- 1.
  ~~v.capacity()~~ v.push_back(2); v.capacity(); Output :- 2
  v.push_back(3); v.capacity(); Output :- 4.

Everytime capacity gets double as ~~xxx~~ elements are getting pushed into array.

- v.size(); Output :- 3 -: Size of array
- Index :- v.at(2);
- Front :- v.front();
- Back :- v.back();
- Pop_back :- v.pop_back(); ← Remove last element
- Size :- v.size();
- Clear :- v.clear(); ← Clears the array (size = 0)

vector < int > a(5, 1) → initialize all elements by 1

size of array

means 0 to 4 index

③ Deque :-

deque<int> d;

- Push_back :- d.push_back(1);
- Push_front :- d.push_front(2);
- Pop_front :- d.pop_front() ← removes front element
- Index :- d.at(1);
- Front :- d.front();
- Back :- d.back();
- Empty :- d.empty();
- Size :- d.size();
- Erase :- d.erase(d.begin(), d.begin()+1);
  ↳ we have to specify the range / maybe in between

After erasing also deque will contain the same memory as before.

④ List :-

Derived from doubly linked list (uses two different pointers front & back)
Direct access like .at() is not possible, it should be done by traversing the list.

since, we have to iterate O(n) T.C.

list<int> l;

- Push_back :- l.push_back(1)
- Push_front :- l.push_front(2)

} All O(1) T.C.

- Erase :- l.erase(l.begin()).
- Size :- l.size().

list<int> n(5, 100);
size of array ↳ initializes all element to 100.

dlly, begin, end, empty, back, front

⑤ Stack:-

(first in last out)

jo last meh jayega woh first meh nikalega

Stack <string> s;

- Push:- s.push("1");
  s.push("2");
- Top:- s.top(); Output:- 2
- Pop:- s.pop(); Output:- 1
- Size:- s.size();
- Empty:- s.empty();

⑥ Queue:-

(first in first out)

jo pehle gaya woh pehle nikalega

queue <string> q;

- Push:- q.push("1");
  q.push("2");
- front:- q.front();
- pop:- q.pop();
- front:- q.front(); Output:- 2.
- size:- q.size();

⑦ Priority Queue:-

(default priority:- max heap) (if you want min heap then convert)

While taking out, if taken from max heap, element will get the max heap, if taken from min heap, you will get the min element only.

min heap

priority_queue <int> pq; — max heap (max)

priority_queue <int, vector<int>, greater<int>> mini;

- **Push :-**

```
maxi.push(1);
maxi.push(3);
maxi.push(2);
maxi.push(0);
```

```
mini.push(5);
mini.push(1);
mini.push(0);
mini.push(4);
```

- **Printing the output :-**

```
int n = maxi.size();
for(int i=0; i<n; i++){
cout<<maxi.top()<<" ";
maxi.pop();
}
```

Done so because everytime size gets changed

print max element

removes that particular element to print the next max element.

Output :- 3 2 1 0

```
int m = mini.size();
for(int i=0; i<m; i++){
cout<<mini.top()<<" ";
mini.pop();
}
```

printing with element

Output :- 0 1 4 5

cause all element are popped before

- **Empty :-** `mini.empty();` Output : 1

⑧ **Set :-**

Uses BST

If you put 5 five times it will store it only onetime. Thus, it stores every element only ones. All elements are unique in it. Either put the element or delete, no modification. Elements gets printed in sorted order.

Unordered set : Set is slower than it, will printing it comes in a trend not in a sorted way.

```
set<int> s;
```

Note:- T.C:- erase, find, insert → O(log n), others → O(1)

Binary Search:-

- Insert:- → T.C:- O(log n)

s.insert(5); s.insert(5); s.insert(5);
s.insert(1); s.insert(6); s.insert(0);
Output:- Using for-each loop,
0 1 5 6 → printed only ones
sorted

- Erase:-

set<int>::iterator it = s.begin(); it++;
s.erase(it); Output:- 0 5 6 (Printed using for-each loop)

- Count:- s.count(5); ← Checks whether 5 is there or not. Output:- 0.
- Find:- set<int>::iterator itr = s.find(5);

for(auto it = itr; it != s.end(); it++) {cout << *it << " ";}
may be used in JavaScript

Output:- 5 6 ← Prints the reference & after it all the remaining elements.

⑨ Map:-

Stores in key, value pairs. All keys are unique.
bijective relation (key one points to one value & but value can be same).
key = value ✓ key = value ✗
key key value value

map<int, string> m;

- Initialization:-

m[1] = "Hi"
m[5] = "Hello"
(or)
m.insert({2, "World"});

key → value

- Printing:-

for(auto i:m) {
cout << i.first << " " << i.second << endl;
}

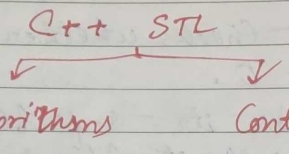
Output:- 1 Hi
2 World
5 Hello
sorted

- Count:- m.count(13); ← find - 13.

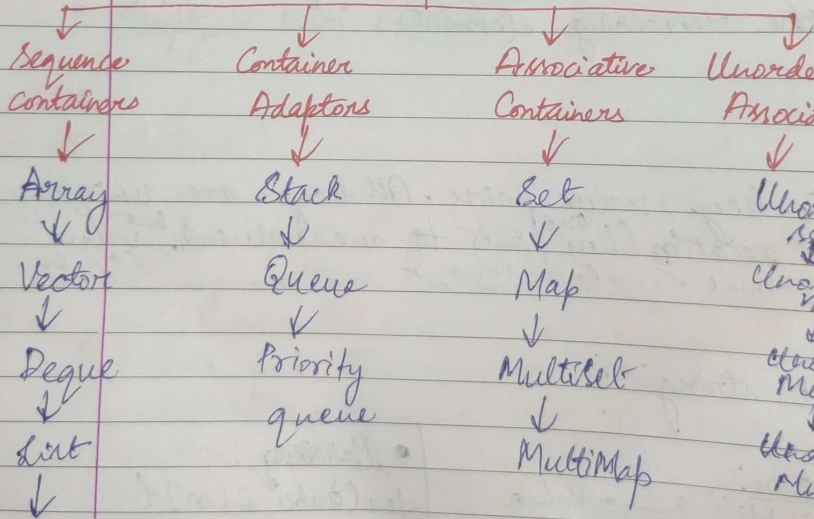
- Erase :- m.erase(13) ← Erases 13 key value pair.
- Find :- m.find(5) ← key-value pair from 5 & after search

returns the iterator

Sorted for map & for unordered → not sorted
Sorted map (Red Black tree or balanced tree)
(S0, log O(log n) for insert... & others O(1)
In case of, unordered map - O(1) because of Hash-table implementation.



① Containers



And we have covered containers till know.

V.begin() → first element
V.end() → element past the last element
iterator is similiar to pointer (*it)

② Algorithms :- → import library (#include <algorithm>)

1) Binary search :-

vector<int> v = {1, 3, 6, 7};
cout << binary_search (v.begin(), v.end(), 5);
Output :- 0 (False) → means 5 is not there in the array.

2) Lower bound :- (first element not less than)

lower_bound (v.begin(), v.end(), 6) - v.begin();
first iterator is used
Output :- 2 → the element position
gives the index

3) Upper bound :- (first element greater than)

upper_bound (v.begin(), v.end(), 6) - v.begin();
Output :- 3 → the next element's position

4) Max :-

max (a, b);

5) Min :-

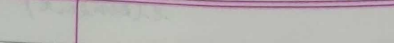
min (a, b);

6) Swap :-

swap (a, b); ← swapping two numbers

7) Reverse :- reverse (abcd.begin(), abcd.end());

8) Rotate :-



rotate (v.begin(), v.begin()+1, v.end());

Output :- 3 6 7

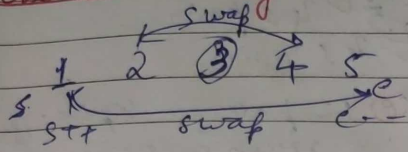
9) Sort :-

sort (v.begin(), v.end());

Note :- Based on intro sort :-
Combinations of three algorithms quick sort, heap sort & insertion sort.

More questions on arrays :-

(1) Reverse an array :-



Output :- 5 4 3 2 1

(2) Merged sorted array :-

sorted ~~also~~ individually

arr1[] = {1, 3, 5, 7, 9}; arr2[] = {2, 4, 6}

merge it

arr3 = {1, 2, 3, 4, 5, 6, 7, 9} → sorted ~~it~~ together

i: 1 3 5 7 9
j: 2 4 6
remaining elements

Check i < j so, arr3[k] = arr1[i]; i++
else if j < i so, arr3[k] = arr2[j]; j++
k++

Copy arr1 if arr1 has more elements, else copy arr2 to arr3

(3) Move zeroes :-

swap, to same index
ignore
non-zero++
arr[] = {1, 2, 0, 4, 3, 0, 5, 0}; arr[i] == 0 → ignore; arr[i] != 0 → swap
swap
not zero
zero

Output :- 1, 2, 4, 3, 5, 0, 0, 0 → all zeroes moved to end
same order, not sorted

(4) Rotate :-

arr[] = {1, 2, 3, 4, 5, 6}; k = 3

Output :- 4 5 6 1 2 3

Eg :- 4 % 10 = 4; 15 % 10 = 5; arr[i % 10] = {0 - 9};

∴ arr[i % n] = {0 to n-1}.

1 2 3 4 5 6 ① ② ③

(i+k) % n ↖ 2 ... i ... n-1 n n+1 n+2

(n+2) % n = 2.

temp[(i+k) % n] = arr[i]

⑤ Sorted & Rotated :-

Ist → 1, 2, 3, 4, 5 } 1 pair

II^nd → 3, 4, 5, 1, 2 } 1 pair

III^rd → 3, 5, 7, 1, 6 } 2 pair

IV^th → 1, 1, 1

count <= 1;
return count == 0 (or) 1

See the code for explaination

⑥ Sum of arrays :-

I :- 1 2 3 4 (val1 + carry) ; 6 → val1 + val2 + carry

1 2 4 0

II :- 1 2 3 4 (val2 + carry) ; 6

1 2 4 0

III :- sum = carry 9 9 9

9 9

carry

Eg :- 9

9

8

18 / 10 = 1

18 % 10 = 8

carry = sum/10 ; sum = sum % 10;

Char arrays & Strings :-

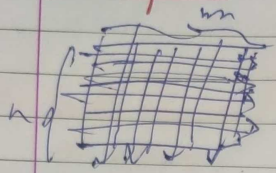
S.length() → used for string data type.
strlen(s) → used for char array.

1. Length, reverse, check word palindrome in a string
2. Reverse chars of words in a string
3. Reverse order of words in a string
4. ~~Reverse~~ Remove all occ. of a substring
5. Replace spaces
6. Permutation in string
7. Sentence palindrome
8. Remove all adjacent duplicates in a string.
9. String compression
10. Get max. occ. of a char

See the code for explanation

2D-arrays :-

① Wave print :-



T.C. :- $O(m \times n)$ See code for explaination

② Spiral Print :-

T.C. :- $O(m \times n)$ See code

③ Binary search in 2D array :-

T.C. :- O(log mxn) See code

④ Search in a 2D array :-

Note :- • Each row is sorted.
• Each col is sorted.

Basic Maths :-

① Sieve of Erathosthenes :-

n = 40 → T.C.2. → O/P :- 12

~~1~~ (2) (3) ~~4~~ (5) ~~6~~ (7) ~~8~~ ~~9~~ ~~10~~
(11) ~~12~~ (13) ~~14~~ 15 ~~16~~ (17) ~~18~~ (19) ~~20~~
21 ~~22~~ 23 ~~24~~ 25 ~~26~~ 27 ~~28~~ 29 ~~30~~
31 ~~32~~ 33 ~~34~~ 35 ~~36~~ 37 ~~38~~ 39 ~~40~~

① Mark every number as a prime number

② Mark non-prime for tables / multiples.

Complexity :- $(n/2 + n/3 + n/5 + n/7 + n/11 + \ldots)$
$= n(1/2 + 1/3 + 1/5 + 1/7 + 1/11 + \ldots)$
⇓
H.P. → primes no.
⇓
$O(n * \log(\log n))$ → T.C.

See code

② Fast Exponentation :-

a^b → $(a^{b/2})^2$ → if b is even
→ $(a^{b/2})^2 \times a$ → if b is odd

1024 / 11

Usual → O(n) & This → O(log n) ← → T.C.

Note :- a % n → [0 → n−1] answer (incl. incl.)

Q → Print answer modulo $10^9 + 7$ → else fat jayega

$(a+b) \% m \Rightarrow a \% m + b \% m$.
$(a * b) \% m \Rightarrow a \% m * b \% m$.

Eg :- Factorial of no. 212! % m

See code

③ Catalan number :-

$$C_n = \frac{(2n)!}{(n+1)!\,n!}$$

Pointers :-

Size of int = 4 | Size of pointer = 8

int *p; → bad practice pointing some garbage address ⇒ int *p = 0;

int i = 5
int *ptr = &i ✓

int *p = 0 → Segmentation fault
p = &i ✓
int *p = &num

p+1 → next address +4 (int)

Copying an pointer ⇒ int *q = p → as it is *q gives value & q give address

arr[10] | *arr, arr[0] → value of zeroth index

arr[0], arr, &arr[0] → First location address

*arr+1 → value +1

*(arr+1) → next index value

arr[i] = *(arr+i) or i[arr] = *(i+arr)

p & &p → pointer address gives different addresses

arr[5] = {1, 2, 3, 4, 5} cout << arr → address
char [6] = "abcde" cout << ch → abcde
char *c = ch[0] cout << c → abcde
char temp = 'z'; char *p = &temp; cout << p → z ?? prints till null character if is not fetched
char ch[6] = "abcdef" ✓ copy
char *C = "abcdef" * pointing to first value

int getSum (int arr[], int n) (or) *arr
size of (arr) → 8 means pointer gets passed

arr++ ✗ p++ ✓
∴ arr = arr+1 ? ?

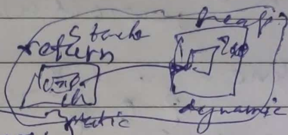
Reference variable, static & Dynamic memory :-

int & j = i ; ~~reference~~ int & X func () { bad practice — local variable

int func (int & j) — garbage value — memory gets deleted }

Array :- [int arr[n];] → runtime — Bad practice — should be done in compile time (cin >> n) → stack — static memory / dynamic → heap (new / request)

[new char;] → address return



[char * ch = new char;] → 4 heap, stack — 12 bytes

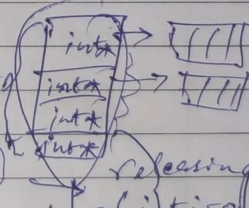
Static :-
while (true) { static }
use - free

dynamic :-
while (true) { dynamic }
use - unfree
delete [] arr; free using delete

DMA in 2D array :-

[int ** arr = new int * [n];] → n

for ( ) {
arr[i] = new int [n];
}



releasing memory
for (int i = 0; i < row; i++) {
delete [] arr[i]
}
delete [] arr

Macro, global variables & inline func. & def. args:-

~~Macro:-~~ #include <iostream>

Macro:- → preprocessor directive

#define PI 3.14 → A piece of code in a program that is replaced by value of macro.

global:- int score = 15

& score++; ✗ bad practice

now it is used

inline:- inline ... (){

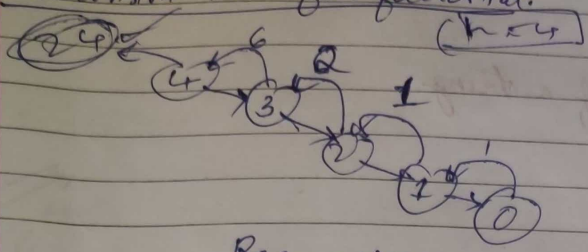
}

// 2 lines before compilation → int main(){

default args:- funct(int i=0, ...){

Recursion :-

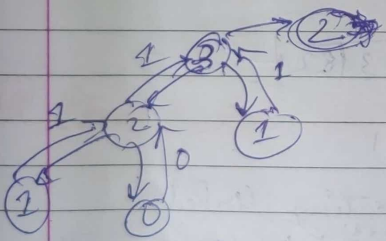
Recursion tree for factorial :-



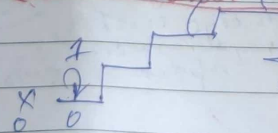
Recursion

printf(n) Tail	printf(n) Head
5	1 base case
4 base case	2 R.R. } mandatory
3 processing (opt)	3
2 R.R.	4 processing (opt) → optional
1	5

1. Factorial
2. Power of two
3. Reach Home
4. Fibonacci :-



5. Count distinct way to climb stairs :-



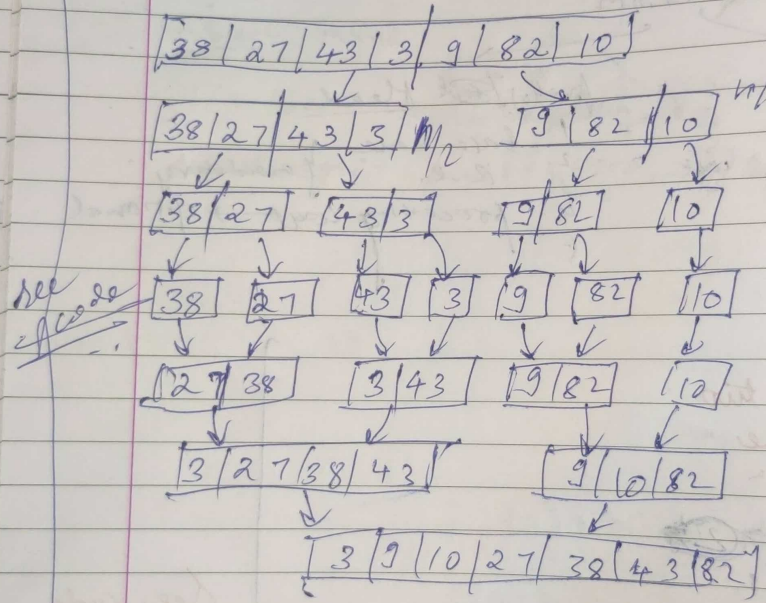
Condi
1 step } 2 ways
2 steps

6. SayDigit
7. isSorted
8. GetSum

See code

~~for explaina~~
for explaination

9. Linear Search
10. Binary Search
11. Power
12. Reverse the chars of a string
13. Check palindrome
14. Bubble sort
15. Merge sort :-



S.C. :- O(n), T.C. : O(nlogn)

Merge sort is useful for sorting linked lists in O(nlogn) time. [Worst: O(n log n), Avg/Best]

16. Quick sort :- → S.C. : O(1) ; T.C. :- O(nlogn)

Quick is in-place requires less space, merge sort requires more space

No, quick sorts is not stable

17. Inversion count :-

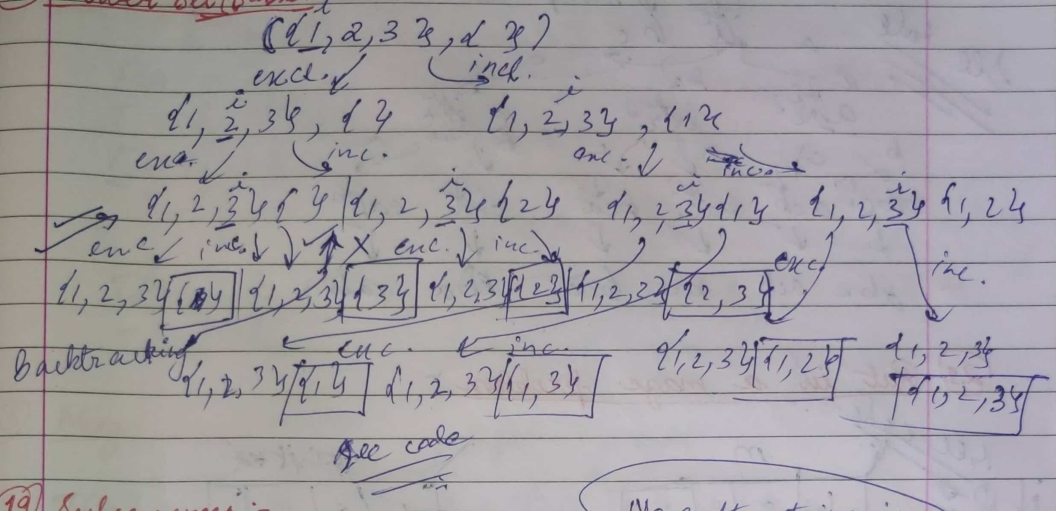
arr[i] (5,1) (5,2) ... for {5,4,3,2,1}

arr[j] count of pairs of inversions

for {1,2,3,4,5} ∴ inversion count = 0

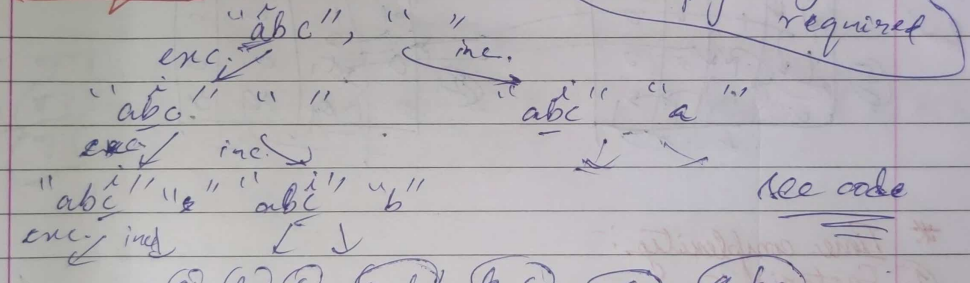
condition not satisfied

(18) Power set/subsets :-

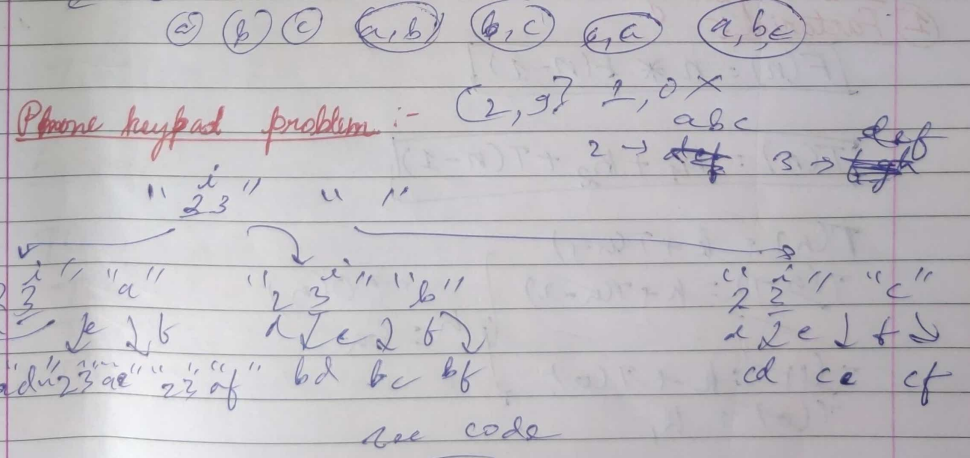


(19) Subsequences :-

No empty string is required

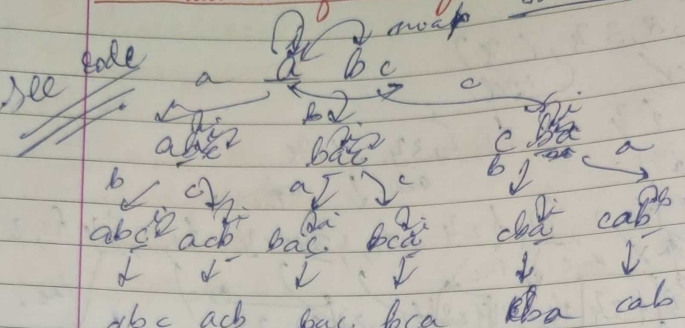


(19) Phone keypad problem :-

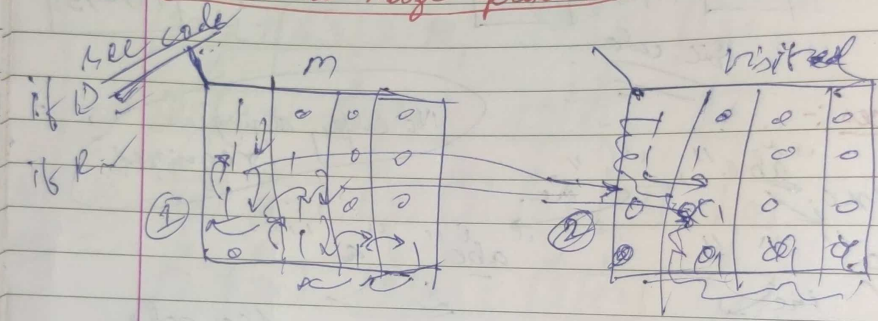


(20) Permutations of string :-

Recursion tree :-



(21) Rat in a maze problem :-



* Time complexity :-

(1) Factorial :-

$$F(n) = n * f(n-1)$$

$$T(n) = K_1 + K_2 + T(n-1)$$

$$T(n) = k + T(k-1)$$
$$T(k-1) = k + T(k-2)$$
$$\vdots$$
$$T(1) = k + T(0)$$
$$T(0) = K_1$$

$k = \text{constant} \rightarrow O(n)$

(2) Binary Search :-

$n \to n/2 \to n/2^2 \to n/2^k$

$F(n) = xyz + f(n/2) \Rightarrow T(n) = k_1 + k_2 + T(n/2)$

$T = k + T(n/2)$

$T(n/2) = k + T(n/4)$

$T_2(n) = k + T(1)$

$T(1) = k$

$n \to n/2 \to n/2^2 \to n/2^3 \to n/2^k$

$\frac{n}{2^a} = 1 \Rightarrow a = \log n$

$T(n) = k * \log n$

$= \log n$

(3) Merge sort :-

left, right

$T(n) = k_1 + k_2 + T(n/2) + T(n/2) + k_3 n + k_4 n$

$= k + 2T(n/2) + nk$

$T(n) = 2T(n/2) + nk$

$T(n/2) = 2T(n/4) + n/2\, k$ $= 4T(n/4) + nk$

$T(n/4) = 2T(n/8) + n/4\, k$ $= 8T(n/8) + nk$

$\log n$ times

$T(1) = k$

$T(n) = a * nk = a \cdot n$

$\log n = O(n \log n)$

(4) Fibonacci series :-

$T(n) = k_1 + k_2 + T(n-1) + T(n-2)$

$T(n) = k + T(n-1) + T(n-2)$

$T(n-1) = k + T(n-2) + T(n-3)$

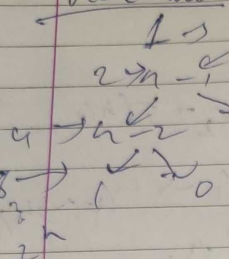
$T(n-2) = k + T(n-3) + T(n-4)$

n Ω

$T(1) = k_1$

$T(0) = k_1$

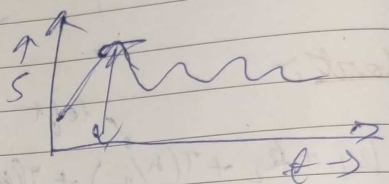
Rec. tree :-



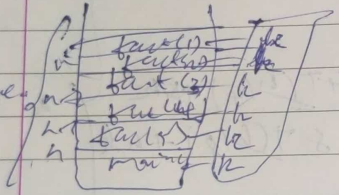
T.C. = k * (2^{n+1} - 1)
= O(2^n).

* Space Complexity :-

max. space req. at any instant



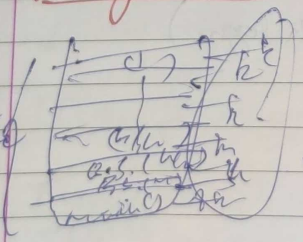
① Factorial :-



S.C. → k * n
→ O(n)

② Binary Search :-

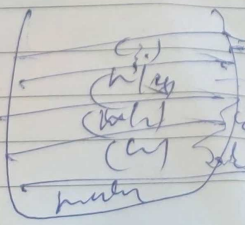
space required



S.C. → k log n
= O(log n)

~~S.C. → k * log n~~
~~= O(log n)~~

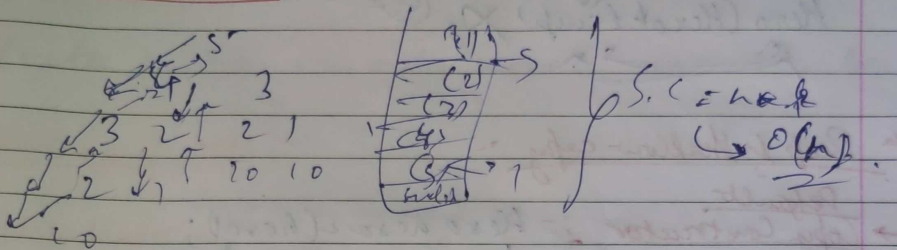
③ Merge sort :-



T.C. = k log n + n
n = 2^k
log n = k

∴ S.C. → O(n)

(4) Fibonacci series :-



OOPS :-

Hero (Hero& temp) ✗ (&) ✓

* Deep & Shallow copy :-

→ Default Copy Constructor :- Hero hero2(hero1);
Babbar → hero1.name[0] = 'G';
Output :- For both Hero1 & Hero2 → Gabbar

copy constructor

Note :- To not do so go to copy constructor & Hero(Hero& temp) → make a new ch and copy to temp.

* Destructors :-

↓

To de-allocate memory
↳ name → class name
↳ no return type
↳ no i/p parameter

~Hero() { cout << "Destructor called" }
int main {

Static → Hero a; → automatically destructor gets called Output :- "Des___"

Dynamic → Hero *b = new Hero()
& manually call
delete b;
Output :- "Des___"
"Des___"

* Static :-

it belongs to object ~~belong~~ but belongs to class

it belong

Time to complete depend on Hero ✗ Enemy ✗

int Hero :: time to complete = 5;

Static function :-

Static functions can only access static members.

Pillars of OOPS :-

① Encapsulation :- → Data hiding
private :- → full encapsulation

② Inheritance :-

* ⓐ Single ⓑ Multi-level ⓒ Multiple ⓓ Hierarchical
ⓔ Hybrid

Access modifiers :-

public → everywhere
protected → same class, derived class
private → same class

public	public	public
public	private	private
public	protected	protected
protected	public	protected
protected	private	private
protected	protected	protected
private	public	NA
private	private	NA
private	protected	NA

Inheritance ambiguity :-

class A | class B | class C : A, B | obj C → C → func() ;
∴ obj A.func() ; obj B.func() ;

③ Polymorphism :- same func. diff parameters

① Function Overloading } Compile time
② Operator Overloading

↓

Eg: rettype operator + (diff obj) { }
void operator () () { }

① Method overriding % Runtime polymorphism
② Inheritance
③ same prototype
④ rank

④ Abstraction :-
Implementation hiding

Linked list:-

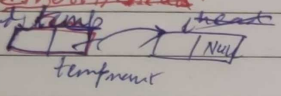
Collection of nodes → [Data | next node address] → [D2 | Null]

Dynamic D.S. (grow / shrink)

Runtime create linked list

① Singly LL:-

ⓐ Insert At head:-

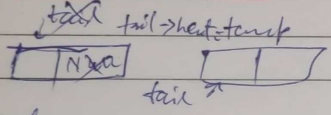


ⓑ Print LL:-

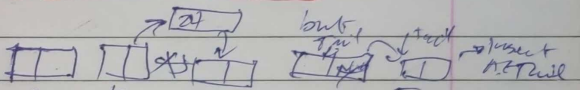
temp ? Null

temp = temp -> next (agla temp)

ⓒ Insert At Tail:-

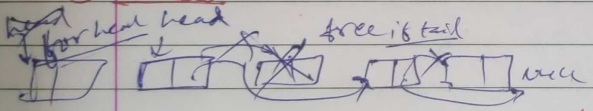


ⓓ Insert At Position:-

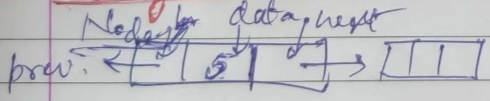


code

ⓔ Delete:-



② Doubly linked list:-



③ Circular linked list:-