1. K^{th} largest Element in an Array without sorting

nums = 3, 2, 1, 5, 6, 4 k=2

0 1 2 3 4 5

out 5

sort ⇒ 6 5 4 3 2 1

without sorting we need How? Use Heaps → max? min?

max heap = 2 = min heap

	max	min
3	3 2 1 (n)	3 2 1 (n')
2	5 6 4	5 6 4
1		
5	(12) 14	(1) (1)
6	14 (6) (5)✓	
4		

nums = 3, 2, 3, 1, 2, 4, 5, 5, 6 k=2

0 1 2 3 4 5 6 7 8

out = 4

sort ⇒ 6 5 5 4 3 3 2 2 1

2	
3 ✓	✗ 5 ✓ 6 ✓ 4
5	
6 5	→ 3
5	5 → 5
5	

so select min heap of size k

pop out top ele if it is less than ith ele & push ith ele

2. K^{th} Smallest Element in a sorted Matrix.

n × n matrix memory < $O(n^2)$

0	1	5	9
1	10	11	13
2	12	(13)	15
	0	1	2

k=8 3×3 = 9

out = 13 8/3 = 2 = row 8%3 = 2

this row any — available

2+1=17 col value

row = 2

col = 1

1	2	3
4	5	6
7	8	9

%3 = 1 2 0 = 1 2 0 = 1 2 0 } col

```
for(i=0→n-1){
  for(j=0→n-1){
    cnt++
    if(cnt==k)
      return
  }
}
```

O(n²) O(1) O(n)?

```
if(k<n)
  return matrix[0][k-1]
```

8/3 = 2

	0	1	2
0	00	01	02
1	10	11	12
2	20	21	22

1 2 3 → 1 2 0

1 1 2 → 1 2 0

2 2 3 → 1 2 0

	0	1	2
0	1	5	9
1	10	11	13
2	12	13	15

k^{th} smallest ele

max_heap — Iterate over every element & store k elements

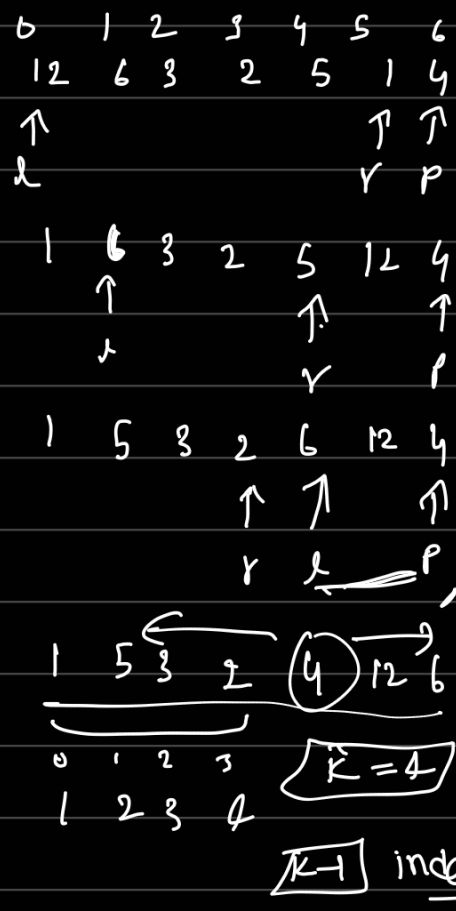
0	1	2	3	4	5	6
12	6	3	2	5	1	4

↑

```
quicksort(arr, l, r){
    p = partition(arr, l, r);
    quicksort(arr, l, p-1);
    quicksort(arr, p+1, r);
}

partition(arr, l, r){
    pivot = arr[r];
    while(l<r){
        if(arr[l]>pivot && pivot<arr[r]){
            swap(arr[l], arr[r]);
            l++; r--;
        }
        while(arr[l]<pivot){
            l++;
        }
        while(arr[r]>pivot){
            r--;
        }
    }
    swap(arr[l], arr[p]);
    return l;
}
```

l<p r>p →



3. Nearly sorted - inplace

almost k away from its target position

Inp → 6 5 3 2 8 10 9 k=3

2 3 5 6 8 9 10

O(k) ⇒ PQ ⇒ max heap

PQ ⇒ O(nlogk)

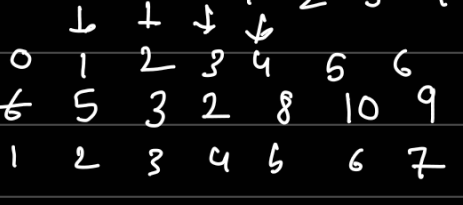
SC ⇒ O(k)

k=2

1 4 5 2 3 6 7 8 9 10

1 2 3 4 5 6 7 8 9 10

k=3 k+1



min heap

6 5 3 2 8

```
index=0;
while(i<n){
    if(min_heap.size() < k+1){
        min_heap.push(arr[i]); i++;
    }
    else{
        arr[index] = min_heap.top(); min_heap.pop()
        min_heap.push(arr[i]); index++; i++;
    }
}
```

```
while(!minheap.empty()){
    arr[index] = minheap.top();
    min_heap.pop();
}
```

4. Merge K sorted LISTS.

K linked lists → each linked list is sorted in ascending order

merge all LL into 1 sorted LL & return it.

lists = [[1,4,5], [1,3,4], [2,6]]

1→4→5
1→3→4
2→6

k=3

min_heap

1 4 5
1 3 4
2 6

1 LL

out = [1, 1, 2, 3, 4, 4, 5, 6]

used custom comparator in min heap

```
struct compare {
    bool operator()( ListNode * l1, ListNode * l2)
        return l1->val > l2->val;
    }
};
priority_queue< ListNode*, vector< ListNode *>, Compare > min_heap;
```

```
auto compare = [](ListNode * l1, ListNode * l2)
        return l1->val > l2->val;
    };
priority_queue < ListNode *, vector < ListNode *>, decltype(compare)>
                                        min_heap(compare);
```

---

5. Replace elements by its rank in the array

inp: 20 15 26 2 98 6

out: 4 3 5 1 6 2

2 6 15 20 26 98

1 2 3 4 5 6

2 loops

```
for( i=0→n-1){
    set={}                          O(n^2)
    for( j=0→n-1){                  O(n)
        if( arr[j] < arr[i] ){
            set.insert( arr[j]);
        }
    }
    cout set.size(); // rank for that i
}
```

arr: 40 10 20 30 (1 1 1)

4 1 2 3

heap

sarr: 10 20 30 40

1 2 3 4

dummy arr copy O(n)+O(n)

O(nlogn) sort dummy arr map{e:R}

+

or

O(n)

↑

O(n) Iterate over sorted dummy & store each rank of ele

Now Iterate over original arr & print out each ele rank

pair arr

arr	20	15	26	2	98	6
	0	1	2	3	4	5

arr Rank

	20	15	26	2	98	6
	0	1	2	3	4	5
	4	3	5	1	6	2

O(n)

pair arr sort ⇒

ele	2	6	15	20	26	98
ind	3	5	1	0	2	4

} or min_heap

O(nlogk)

pair arr

100	100	100
0	1	2
1		

+ O(nlogk)

rank = 1;

Assign 1st min from heap & maintain prev value to compare in loop prev = (100, 0)

Iterate over min_heap {

if (curr.f != prev.f) arr[i.s] = ++rank

else arr[i.second] = rank; prev = curr;

} O(n)

6. Task Scheduler

vector<char> &tasks, int n

tasks = [A, A, A, B, B, B] n = 2 → n gap b/w same task cpu can be idle

output = 8

A → B → idle → A → B → idle → A → B

1 2 3 4 5 6 7 8

tasks = [A, C, A, B, D, B] n = 1

6

A → B → C → D → A → B

1 2 3 4 5 6

⟶ A C A B D B n = 1

A → B → A → B → (→D

[A, A, A, B, B, B] n = 3

A → B → idle → idle → A → B

1 2 3 4 5 6

→ idle → idle → A → B 10

7 8 9 10

A: ✗ ✗ 0

B: ✗ ✗ 0

C: 1 → 0

D: 1 → 0

A → B →

A: ✗ 2

B: ✗ 2

time = 0, 1

A B

pq ((3,0) (3,0))

q ((2,0)

AAA BBB n = 2

3A A B _ A B _ A B 8

3B

n = 0

3A

3B A AA BBB

n = 2

3A

3B

1C

1D

1E A B _ C AB _ D AB _ E

result = # of tasks + idle time

3A A _ _ A _ _ A

1B

1C

n = 2

max freq = 3

max count = 1

blank seq len = n - (max count - 1)

NO. of blank seq = max - 1 = 2 seq

(001 down but we can

fill other values also there

3A
3B
1C
1D
n=2

A B _ _ A B _ _ A B   → ① → ① → 2 − (2−1) = 2−1 = ①

① ②

max = 3
max count = 2
2−1 = ①

blank seq * blank seq len = total max blanks
= 2 × 1 = 2 ⇒ empty slots

If we know how many non-max tasks we have we can fill the idle time

available tasks = # of task − (max * max count)

3A
3B
1C
n=2

A B C A B _ A B
1 2 3 4 5 6 7 8

max = 3
max count = 2

idle time = empty slots − available tasks

Total tasks = 7   max task count = 3*2 = 6

available tasks (non-max) = 7−6 = 1

result = # of task + idle time

blank seq = max − 1 = 3−1 = 2
blank seq len = n − (max count − 1)
= 2 − (2−1)
= 2−1 = 1

empty slots = 2*1 = 2
available tasks = 1
Idle time = 2−1 = 1

Result = # of tasks + idle time
= 7 + 1 = 8

2A
13B
n=2

max 13
maxc 1
2 − (0)
= 2

blank seq = max − 1 = 12
→ blank seq len = 2

12*2 = 24 empty

13*1 = max slots
# of tasks = 15 − 13 = 2
non max tasks
= 24 − 2 = 22 idle time

result = 15 + 22 = 37

B A _ B A _ B _ _ B _ B _ _ B _ _ B _ _ B B B B B B

6 3 3 6 6 6 6 1
6

6 6 36+1 = 37

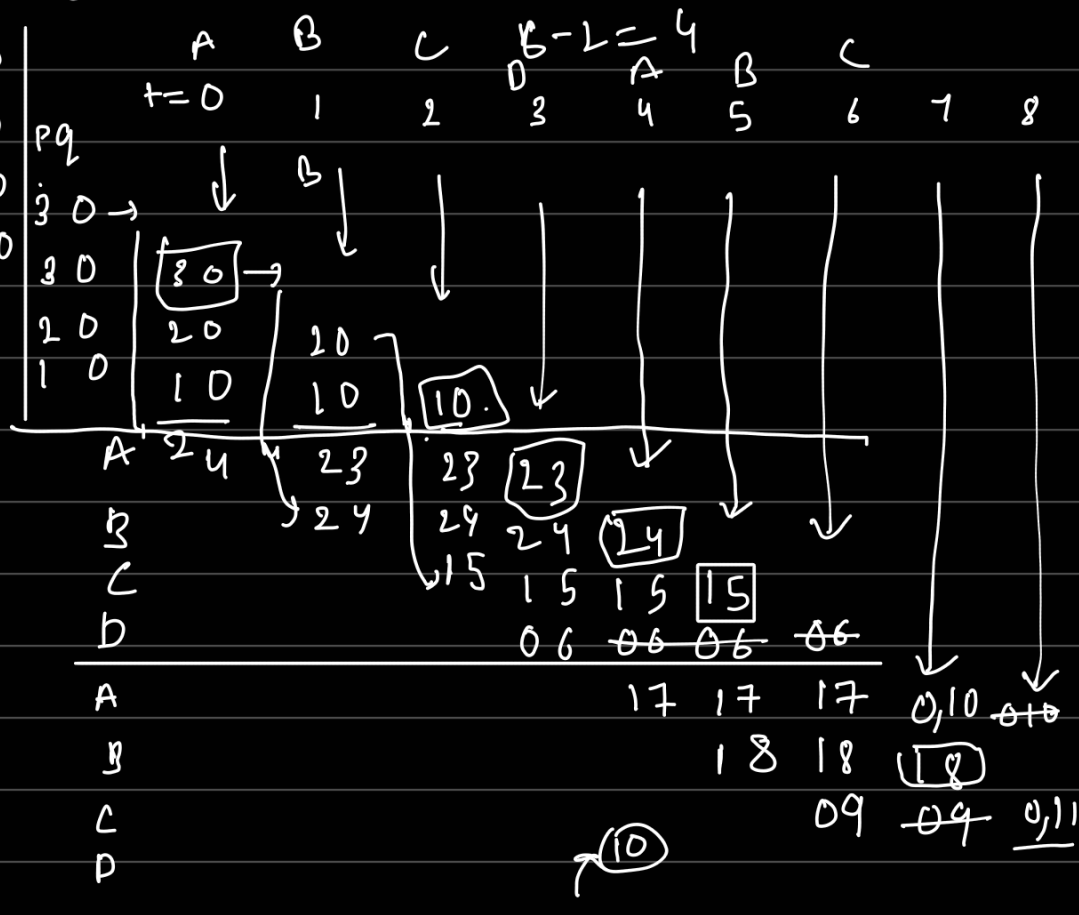
2A
2B
2C
1D
n=4

A B C A B D

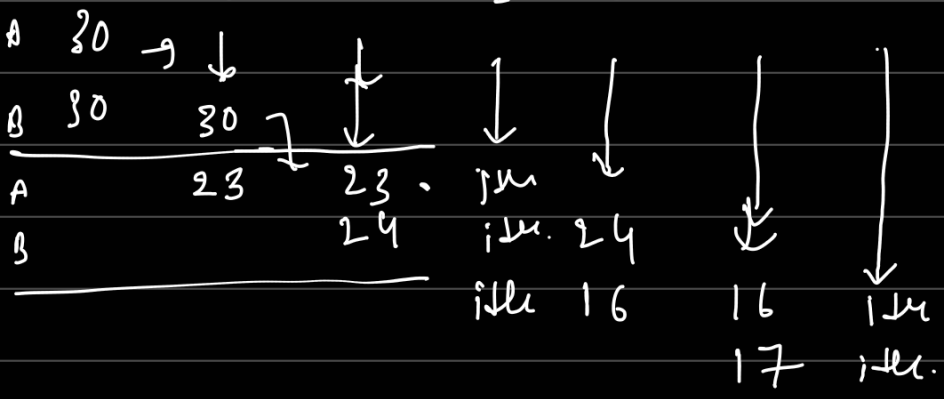
man = 2 man slots = 4
mancart = 2 blameseq = 1
blockseq len = 1 − (2+1)
nonmax seq = 6 − 4 = 2 1 − 1 = 0
idle seq = 0 2 empty slots = 1 * 0 = 0
= 2

task	freq	time
3A	A	3 : 0
3B	B	3 : 0
2C	C	2 : 0
1D	D	1 : 0

n=3
n+1
start



t = 0 1 2 3 4 5 6 7 8 9 10
A B C D A B C A B



		0	1	2	3	4	5 (5-2=3)	6	7	8	9	10
4A												
1B	40											
1C	10	10										
1G	10	10	10									
	10	10	10	10								
1D	10	10	10	10	10							
n=1		32	32	32	32	32						
			~~02~~	~~04~~	~~05~~	~~06~~	27	idle (27)	19	010		

n=2 1 2 3 4 5 6 7 8
A B A C A G A D ✓

1 2 3 4 5 6 7 8 9 10
A B C A D G A _ _ A

	cycle	PQ	store	taskcount
4A	~~3~~ ~~2~~ ~~1~~	~~4~~ ~~1~~ ~~1~~ 1 1	3 [ ]	3
1B		3 0 0 →	[ ]	
1C				
1G	~~3~~ ~~2~~ ~~1~~	~~3~~ ~~1~~ ~~1~~	2	3
1D		2 0 0 →	[ ]	
	~~3~~ 2	~~2~~ [ ] ↓	1	1

cycle != 0 cycle = 2 = idle 2

~~3~~ 2	~~1~~	[ ]	1
	0 [ ]	[ ]	10

empty

7. Hand of Straights - consecutive cards - so it should be consecutive cards for sure (+1)

hand = 1, 2, 3, 6, 2, 3, 4, 7, 8
groupsize = 3 out : true
rearranged = (1,2,3) (2,3,4), [6,7,8]

hand = 1, 2, 3, 4, 5
groupsize = 4 out = false
can't be rearranged
size < gs
[-1] < curr

1 2 2 3 3 4 6 7 8 min_heap

vector< vector<int>> [(1)]

1 : 1 → map<int, int> ⇒ gives output → first
2 : 2
3 : 2
4 : 1
6 : 1
7 : 1
8 : 1

map.begin() → first gives smaller ele by default

1	2	3
2	3	4
6	7	8

remove smallest ele & check for its consecutive elements

g=3

i+0 (curr) i+1 i+2

```
if (map(curr+i) <= 0){
    m.erase(curr+i)
}
```

while (m.empty()) ⇒ O(n)
m.begin()->first ⇒ O(logn)
for ⇒ O(k)

O(nlogn + n.k)

n=3

map{}

1 : x0 → 1+0 1+1 1+2 ⇒ 1 2 3 ⇒ x0 x1 x2
2 : x1 → 2+0 2+1 2+2 ⇒ 2 3 4 ⇒ x0 x0 x0
3 : x0
4 : x0
5 : x1 → 5+0 5+1 5+2 ⇒ 5 6 7 ⇒ x0 x0 x0
7 : x0
8 : x0

if any of these are 0 before -- we return false

if map is empty return True else false

O(nlogn + n.k)

O(n.(logn + k))

Hashmap + min heap ↑ map Neetcode

1 2 2 3 3 4 6 7 8

→ -

→ (1 2 3) 1
(2 3 4) 1
1:1
2:2 3:2 (6 7 8)
4:1 6:1 7:1 8:1

openGrps = ∅ x x
= x0

2-1 = 1 need extra group
2-2 = 0 no need extra grp
H = 0 ⇒

↳ How many grps can share this # depends on freq minfreq = grps if more grps already present curr freq < openg return false

groupQ = [x x 0 0
↓ group size
3 ⇒ | now close a group

cog | 2 | | | | |
gQ | 1 0 0 0 0 0

cardcount ⇒ freq (map) ⇒ Frequency cnt
currentOpenGroups ⇒ (int) ⇒ track of # of open grps
groupStartQueue ⇒ (queue) ⇒ track of no. of new grps starting with each card value

[1, 2, 3, 2, 3, 4] n=3

freq = { 1:1, 2:2, 3:2, 4:1, 6:1, 7:1, 8:1 }

O(nlogn + n)

3 queue<int> gsq = [1, 1, 0 | [1] [1, 1] [x 1 0] x00 x01 x10 x00
2 last card = x x x 3 | x1 2 3 4 6 7 8
1 curr card = x x 3 | 1 2 3 4 6 7 8
4 currentOpenGrps = ∅ x 2 | ∅1 2 x1 x0 x1 x1 x0

1 2 2 3 3 4 6 7 8

3 = len of NewGrp

		NewGrp	Active CurrGrp
→	1:1	1	1
→	2:2	(2-1)=1	2
	3:2	[2-2]=0	2 2-1=1
	4:1	[1-1]=0	1
	6:1	[1-1]=0	1
	7:1	[1-1]=0	1
	8:1	[1-1]=0	1

$O(n \log n + n)$

$O(n)$

Most optimal way ⇒ NO sorting / getting smaller elements first ($\log n$)

create freq map

Iterate over hand arr:

Each card ⇒ find starting card by decrementing -- it until no value is found in map

⇒ once start card is found | form a consecutive seq of group size

↳ if any card in blw not present ⇒ return false

↳ if each card is present ⇒ decrement by 1 freq

1 2 2 3 3 4 6 7 8 n=3

↑

card = 1

1:~~1~~ 0 Start card = ~~1~~ 0 0<=1

2:~~2~~ 1 1 1<=1

3:~~2~~ 1

4:1 [7]:1 0, n=3 1+3=4 <4

6:1 nextcard = 1 2 3

7:1 ~~1~~ ~~2~~ ~~2~~

8:1 0 1 1

8. Design Twitter - code all the functions

```
Twitter(){
}

void postTweet(int userId, int tweetId)

vector<int> getNewsFeed(int userId)

void follow(int followerId, int followeeId)

void unfollow(int followerId, int followeeId)
```

// need list of users

Vector<int> userIds = {}; // each user has many tweets Ids

↳ create users Objects to store list of users. or lets implement simply

unordered_map<int, vector<int>> users;

{ userId: [Tw1, Tw2. .....] (stack)
userId : [Tw1, Tw2. ...] }

{ 1: [5 6 7 8 9],
2: [10 11 12 5 9] }

unordered_map<int, set<int>> followers, followee;

Twitter() { }

1 → 3 5 7 8
2 → 5 7 8
3 → 1 2 8
8 → 3

```
void postTweet(int userId, int tweetId){
    // create new tweet (tweetId) for user (userId)
    this.users[userId].pushback( tweetId);
}

vector<int> getNewsFeed(int userId) {
    vector<int> feed; int n = this.users[userId].size();
    for(int i=n; i>n-10; i--){
        feed.push_back(this.users[userId][i]);
    }
    return feed;
}
```

NOt that Simple

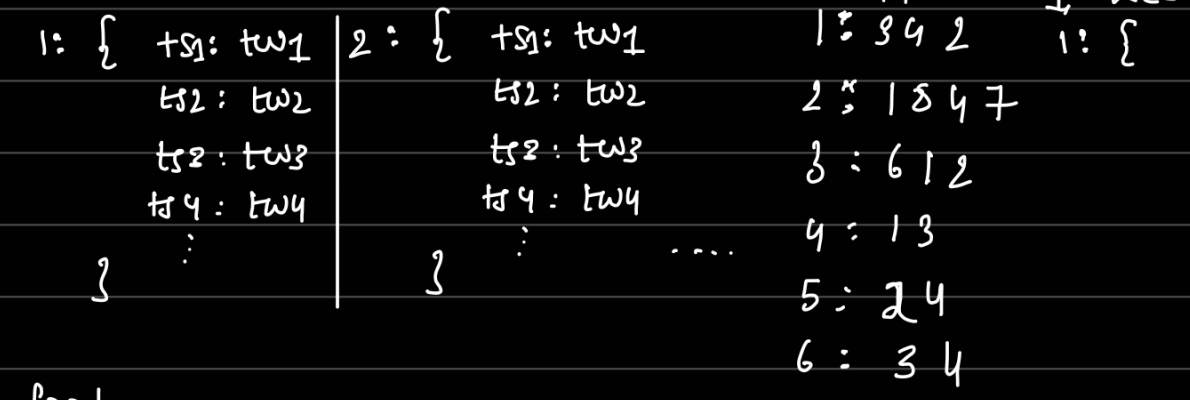
posted by user himself or the users he is following

{ followerId: { followeeId } }

```
void follow(int followerId, int followeeId){
```

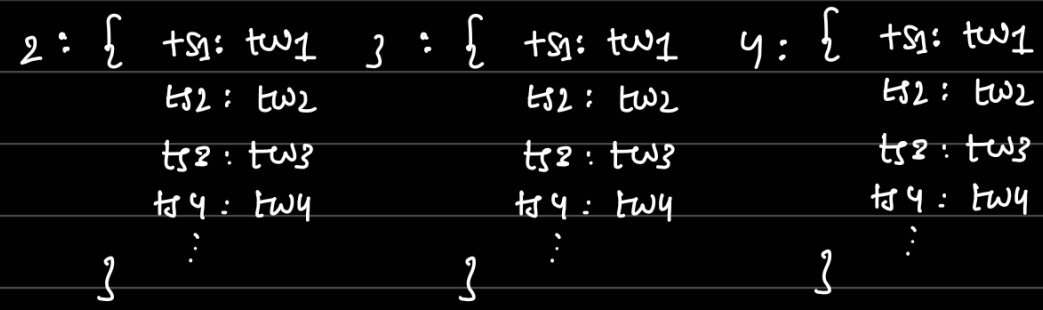
followees : { 1: { 3 5 7 8 }
2: { 5 7 8 }
3: { 1 2 8 }
8: { 3 }
}

postTweet

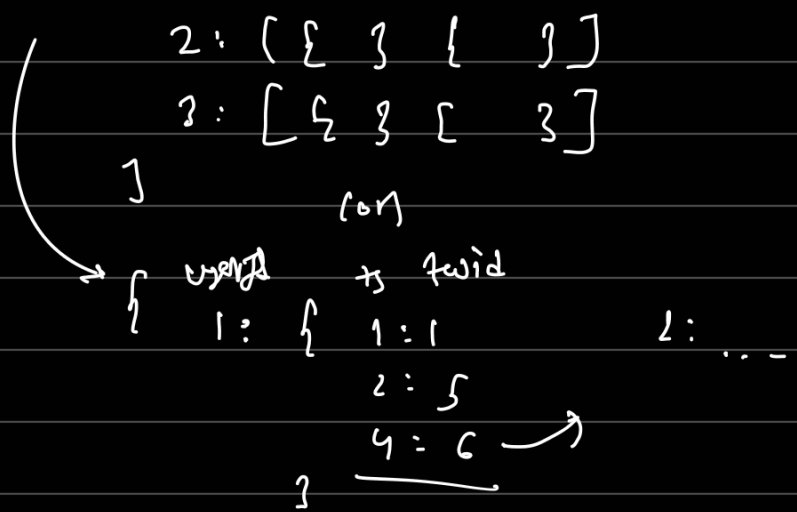


feed

user 1 : 3 4 2 feed [ts1 : tw1 ts2 : tw2 .. - (10)]



user_tweets = { 1 : [{ 1, 1 3 } , { 2, 5 } , { 4, 6 }]



unordered_map < int, map<int, int, greater <int>> > user_tweets; { _ : { : } }

unordered_map < int, unordered_set <int>> user_followeeids; { _ : { , } }

9. Minimum cost of ropes

arr = 4 3 2 6 ⇒ rope length ⇒ connect all ropes

⇒ cost to connect 2 ropes is sum of their length

4 3 2 — 6

5 2+3 ⇒ 4 5 6

9 4+5 ⇒ 9 6

15 9+6 ⇒ 15

29

2 3 4 6

5 4 6

9 6

15

4 2 7 6 9

2 4 6 7 9

6 6 7 9

12 7 9

19 ~~7~~

~~28~~

6+12+19+28

65

7 9 12

16 12

use min heap

push all ele & get top 2 sum & push them back again return it

$\frac{4}{2}$ $\frac{3}{4}$ $\frac{2}{5}$ $\frac{1}{8}$

10. Kth Largest Element in a Stream

K = 3 [4 5 8 2] 3 5 10 9 4

4 5 8 2

store in min heap (k values - larger)

4 → min val out of top K

5

8

11. Maximum sum combination

return maximum K valid sum combinations from all possible sum combinations

N = 2 K = 2 A = 3,2 B = 1,4

N = 4 K = 3 A = 1 4 2 3 B = 2 5 1 6

4 3 2 6 5 2

h1 4 3 2 1 h2 6 5 2 1

Greedy

2 for loops & storing sum in arr

sorting for first K ele

K = 3

heap (10, 9, 9, 8, 5, 7, 4

6+4 = 10

$m_1+c_2 = 4+5=9 \Rightarrow 3+2=5 \Rightarrow 2+1=3$

$c_1+m_2 = 6+3=9 \Rightarrow 5+2=7 \Rightarrow 2+1=3$

$c_1+c_2 = 3+5=8 \Rightarrow 2+2=4 \Rightarrow 1+1=2$

$m_1 = c_1$

$m_2 = c_2$

c++; c2++;

K = 4

6 + 10 = 16

6 + 9 = 15

6 + 8 = 14

6 + 7 = 13

10+1 = 4

9+1 = 10

8+1 = 9

7+1 = 8

m1 4
6 1 1 1
10 9 8 7
m2 2

6+10 = 16
6+9 = 15
10+1 = 11
1+9 = 10

(16, 15, 11, 10)

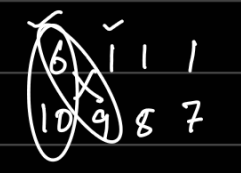
0 1 2 3
1 1 1 6
7 8 9 10

(3,3)
(2,3)
(3,2)

2 2
3 1

(16)
11
15
14
10

ans = 16, 15



p 1 2 3
1 1 1 6
7 8 9 10
0 1 2 3

sort A
sort B
} custom comparator a>b;

✓ priority_queue < pair <int, pair<int,int>>> max_heap;

✓ i=0 j=0 max_heap.push ({A[i]+B[j], {i,j}});

✓ set < pair<int,int>> visited;

while (k>0) {

curr = max_heap.top();

push to ans

check for Not visited

max_heap push i+1, j
i, j+1
i+1, j+1
} mark them visited

k--;

}

return ans;

12. Find Median from Data stream

arr = 2, 3, 4 ⇒ out = 3

arr = 2, 3 ⇒ (2+3)/2 = 2.5

sort input data stream

even ⇒ take 2 middle values mean

Odd ⇒ middle value

even ⇒ median = mean of 2 middle value

Odd ⇒ middle value

8/2 = 4-1 = 3

odd 1
7/2 = 3 = mIndex

Even ⇒ (/2)+1 ⇒ mIndex 1

next top is mIndex 2

return mean of it

max_heap.size() ⇒ odd ⇒ mIndex

1 ↩ 1 pop ↗ times & return top value

TLE ↗ ↙

1 4 2 3 — 1 4 2 3

1 2.5 2 2.5

(1+4)/2 = 2.5

2 == 1

max heap	min heap
1	4
2	3
(2)	(3)

10 40 20 1 5 39 8 7 2 3 — (5)

10	40	(5)
20	1	
5	39	
8	7	
2	3	
(20)	(1)	

10 => even (21)/2 = 10.5 ✓

1 2 3 5 | 7 8 | 10 20 39 40

(7+8)/2 = 7.5

	max	min	diff (2)	median
10	10 (1)	- 0	1 ✓	10
40	10, ~~40~~ (x1) ⟶	40 (x1)	½ 0 ✓	(10+40)/2 = 25
20	10, 20 (2)	< 40 (1)	1 ✓	20
1	1, 10, ~~20~~ (x2)	20, 40 (x2)	½ 0 ✓	(10+20)/2 = 15
5	1, 5, 10 (3)	20, 40 (2)	1 ✓	10
39	1, 5, 10, ~~39~~ (x3)	20, 39, 40 (x3)	½ 0 ✓	(10+20)/2 = 15
8	1, 5, 8, 10 (4)	20, 39, 40 (3)	1 ✓	10
7	1, 5, 7, 8, ~~10~~ (x4)	10, 20, 39, 40 (x4)	½ 0 ✓	(8+10)/2 = 9
2	1, 2, 5, 7, 8 (5)	10, 20, 39, 40 (4)	1 ✓	8
3	1, 2, 3, 5, 7, ~~8~~ (x5)	8, 10, 20, 39, 40 (x5)	½ 0 ✓	(7+8)/2 = 7.5

addNum ⇒ log n ⇒ Heap push pop

findMedian() ⇒ O(1)

1021	1021 (1)	- 0	1
9540	1021, ~~9540~~ (2)	9540 (0)	
7788	1021, ~~7788~~ (2)	9540 (1)	1 ✓
9278	1021, 7788, ~~9278~~	9278, 9540 (0)	0 ✓
6541	1021, 6541, 7788 (3)	9278, 9540 (2)	1
224	224, 1021, 6541, (3)	7788, 9278, 9540 (x3)	½ 0 ✓
9344	224, 1021, 6541, 9344 (4)	7788, 9278, 9540 (3)	1 ✓
5105			

224, 1021, 6541 ← 7788, 9278, 9344, 9540

224, 1021, 6541, 7788 9278, 9344, 9540

13. Top K Freq Elements return k most freq elements
& return in any order

nums = 1, 1, 1, 2, 2, 3 k=2

1:3 2:2 3:1 out ⇒ [1,2] Algo TC should be better than O(nlogn)

nums = 1 k=1

1:1 out ⇒ [1]

unordered_map ⇒ umap ⇒ Iterate over every ele & cut freq
if k add it to the ans

1, 2 k=2 out ⇒ [1,2]

1:1, 2:1

min_heap TC ⇒ O(n + mlogn)
SC ⇒ O(m+k)

(1,3) (2,2) (3,1) ⇒ (1,3)
1
(3,1) < (2,2) < ? ← insert
✓

1. find freq map
2. use custom compare for min_heap pairs [return P1.second > P2.second]
& store k elements every time
↳ if size == k
store only larger element in the heap
so min_heap returns min_freq ele to compare
4. append the k ele to vector ans & return.