July 20, 2025

STL APPLICATION IDEA

#1 Prices are given $P_1, P_2, P_3 ---- P_n$. There are Q queries with Budget $B_1, B_2 --- B_n$. Find out the maximum amount of items that can be bought for a given B_i and output the cost associated with it.

Example:

5 items with following prices:

N → 5, 4, 2, 1, 6, 3

Q queries with budget b_i

Q → 2, 5, 3, 10, 7

- $B_1 = 2$

Max ele that can be bought

Either P: 1 | 2

Ans = 1 (since we will try to pick the lowest budget.

- $B_2 = 5$

#items = (1, 2) = 1+2 = 3 cost

- $B_3 = 3$

#items = (1, 2) = 3 cost

- $B_4 = 10$

(1, 2, 3, 4) = 10

- $B_5 = 7$

(1, 2, 3) = 6 (lowest hence Ans)

or

(1, 2, 4)

→ Pick items greedily with lowest prices first

$P_i \leq B_i$

1. Sort all the elements in P_i
2. Maintain a prefix sum, this will gives us an array of sum of cost.
3. Use upper bound for each B_i

ans = UB(prefix, prefix+n, B_i) − prefix.

Cost = Prefix[Ans] $\leftarrow$ son

Time complexity : O(nlogn + qlogn)

#2 There is a running stream of numbers that is operations like add() and remove() are supported at any point. Maintain this structure to output mean.

Example:
Queries:
→ Add 1
[1], m=1
→ Add 2
[1,2], m=1.5
→ Add 3
[1,2,3] m=2
→ Return mean
⇒ 3

```
class mean {                          → Support add()
    int sum = 0;
    int cnt = 0;
    int add(int x) {
        sum += x;
        cnt++;
    }
    double mean() {                   → if (cnt != 0) to avoid 0/0
    { return (double) sum/cnt; }
```

→ Support remove()

The mean function will remain same, Some edges cases to think about: what if the query tells us to remove a number x which doesn't exist.

→ Support calculation of Mode().

Mode: number that occurs maximum no. of times

Use of frequency mapping.

→ Add 1
{(1,1)}
max=1, mode=1
→ Add 2
{(1,1)(2,1)}
max=1, mode=1
→ Add 2
{(1,1)(2,2)}
max=2, mode=2

→ if (incoming_element_freq > max) (e)
update the mode to e
max = e_frequency

→ Remove 2
We have to remove 1 instance of 2.
How to restore mode value to 1?
Use of reverse mapping.
map<int, set<int>> mp;

- Add 1
→ (1) → 1 ← num
freq

- Add 2
1 → 1, 2

Query
↓
mode return *(mp.rbegin() → second.begin())

- Add 2
1 → 1
2 → 2
- Remove 2
1 → (1), 2
- Mode
⇒ 1

add/remove O(log n)

#3 LRU Cache https://leetcode.com/problems/lru-cache/description/

Design a data structure that follows the constraints of a **Least Recently Used (LRU) cache**.

Implement the `LRUCache` class:

- `LRUCache(int capacity)` Initialize the LRU cache with **positive** size `capacity`.
- `int get(int key)` Return the value of the `key` if the key exists, otherwise return `-1`.
- `void put(int key, int value)` Update the value of the `key` if the `key` exists. Otherwise, add the `key-value` pair to the cache. If the number of keys exceeds the `capacity` from this operation, **evict** the least recently used key.

The functions `get` and `put` must each run in `O(1)` average time complexity.

Example:
Capacity = 3 (cannot maintain more than this)
{ } ← initial empty

→ Put(1, 2)
{1:2}
→ Put(2, 5)
{1:2, 2:5}
→ Put(3, 6)
{1:2, 2:5, 3:6}

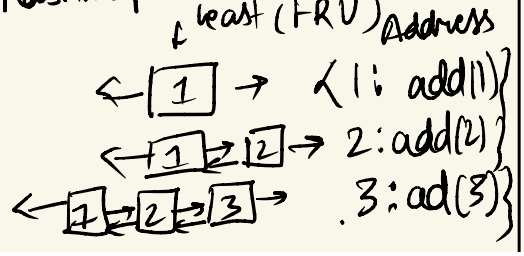
→ Get(1)
⇒ 2
Once this query is done on 1 then 1:2 becomes most recently used then 2:5 will become LRU {2:5, 3:6, 1:2}

Continuing the above example,
→ Put(4, 5)
But our cache is full as it already has reached its capacity of 3.
So the LRU element will be removed so, 2:5 will be removed first
{3:6
1:2
4:5}

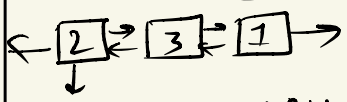
We will use doubly linked list to maintain this structure

1. Put 1:2
2. Put 2:3
3. Put 3:1
4. get 1
5. put 4:5

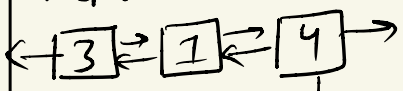
Maintain a hashmap and make a node for DLL.
{1: 2
2: 3
3:1



→ get 1
Return 2
1:2 becomes MRU
Unlink 1 and add to last ← MRU



This becomes LRU
→ Put(4, 5)
Capacity full, Remove LRU



LRU
{1:2
3:1
4:5}

MRU
Address Hashmap
{1: add(1),
3: add(3),
4: Add(4)}

Time complexity
get() O(1)
↓
map lookup
put() O(1)
worst case
→ Linked List removal
→ Re doing pointers
→ if capacity is filled for put()

AlgoZenith

- *When adding a new **key-value** pair, insert it as a **new node** at the **head** of the **doubly linked list.** This ensures that the **newly** added **key-value** pair is marked as the **most recently used.***
- *If the key is already **present** in the **cache** , get the **corresponding node** in the doubly linked list using **hashmap** , **update** its value and move it to the head of the list, and **update** its **position** in the **hashmap** also. This operation ensures that the accessed **key-value** pair is considered the **most recently used.***
- *The priority of nodes in the **doubly linked list** is based on their **distance** from the **head** . **Key-value** pairs closer to the head are **more recently used** and thus have **higher priority** . Also, key-value pairs closer to the **tail** are considered **less recently used** and have **lower priority.***
- *When the **cache** reaches its **maximum capacity** and a **new key-value pair** needs to be added, **remove** the node from **hashmap** as well as from the **tail** in the **doubly linked list** . **Tail** node represents the **least recently** used **key-value** pair and is **removed** to make **space** for the **new entry.***