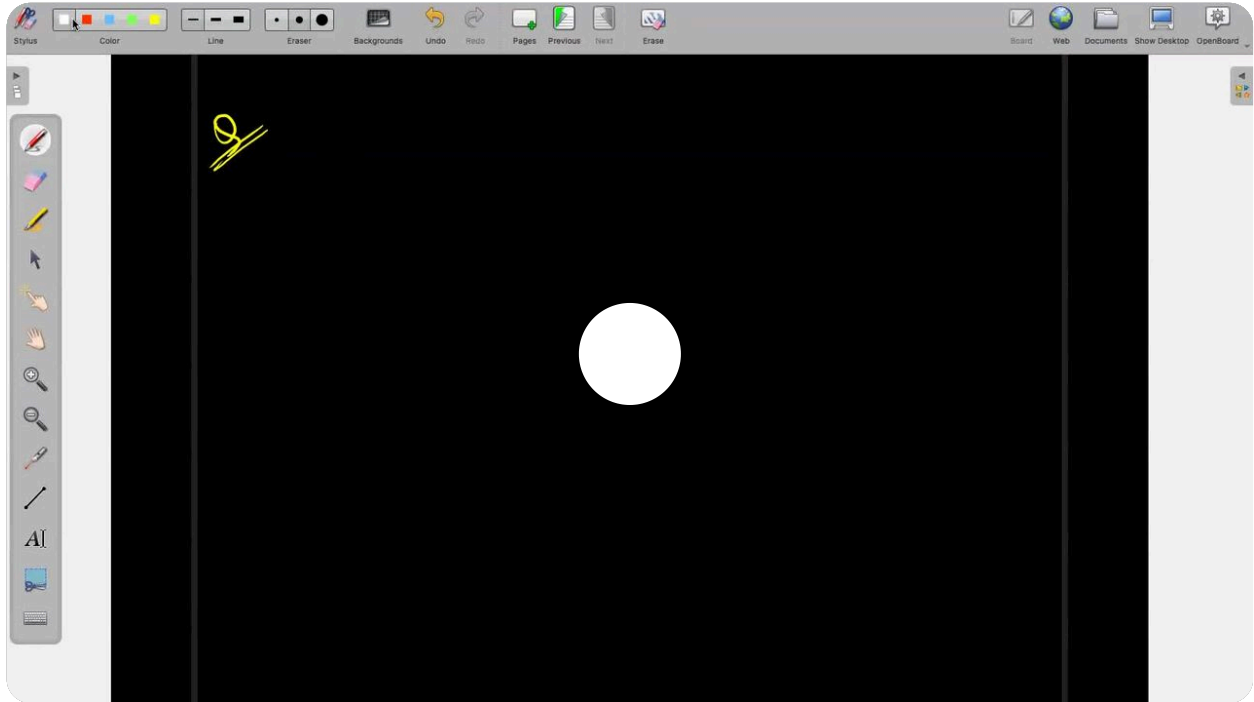




# STL Application



## STL Applications and Problem Ideas - 1

Learning the Standard Template Library (STL) in C++ is crucial for efficient and robust programming. Let's explore some STL applications and problem ideas to help reinforce your understanding.

**1. Problem: Given a shop with  $N$  items, where each item is indexed by  $i$  and has a price  $P_i$ . The task is to determine the maximum number of items that can be purchased given a budget  $B$ .**

**Algorithm Explanation:**

1. Sort the items based on their prices in non-decreasing order.
2. Initialize a variable `maxItems` to keep track of the maximum number of items that can be bought.
3. Iterate through the sorted items:
  - If the price of the current item is less than or equal to the remaining budget  $B$ :
    - Deduct the price of the current item from the budget  $B$ .
    - Increment the `maxItems` count.
  - If the price of the current item exceeds the remaining budget, break the loop.


- Return the value of `maxItems`, which represents the maximum number of items that can be bought within the budget.

**Code:**

```
#include <bits/stdc++.h>
using namespace std;

int maxItems(vector<int>& prices, int budget) {
    sort(prices.begin(), prices.end());
    int maxItems = 0;
    for (int i = 0; i < prices.size(); ++i) {
        if (prices[i] <= budget) {
            budget -= prices[i];
            maxItems++;
        } else {
            break;
        }
    }
    return maxItems;
}

int main() {
    int N, B;
    cin >> N;
    vector<int> prices(N);
    for (int i = 0; i < N; ++i) {
        cin >> prices[i];
    }
    cin >> B;
    int max_items = maxItems(prices, B);
    cout<< max_items << endl;
    return 0;
}
```

 Copy

**Time Complexity:**

- Sorting the items takes  $O(N \log N)$  time.
- Iterating through the sorted items takes  $O(N)$  time.
- Thus, the overall time complexity of the algorithm is  $O(N \log N)$  due to the sorting step.

---

**2. Problem: Given a shop with  $N$  items, where each item is indexed by  $i$  and has a price  $P_i$ . The task is to determine the maximum number of items that can be purchased. But now you are given  $M$  queries each carry the budget  $B_j$ .**

**Algorithm Explanation:**

- Sort the items based on their prices in non-decreasing order.
- For each query:
  - Initialize a variable `maxItems` to keep track of the maximum number of items that can be bought with the current budget.

- Iterate through the sorted items:
  - If the price of the current item is less than or equal to the current budget:
    - Deduct the price of the current item from the budget.
    - Increment the `maxItems` count.
  - If the price of the current item exceeds the remaining budget, break the loop.
- Print or store the value of `maxItems` for the current query.

### Code:

```
#include <bits/stdc++.h>
using namespace std;

int maxItems(vector<int>& prices, int budget) {
    int maxItems = 0;
    for (int i = 0; i < prices.size(); ++i) {
        if (prices[i] <= budget) {
            budget -= prices[i];
            maxItems++;
        } else {
            break;
        }
    }
    return maxItems;
}

int main() {
    int N, M;
    cin >> N;
    vector<int> prices(N);
    for (int i = 0; i < N; ++i) {
        cin >> prices[i];
    }
    sort(prices.begin(), prices.end());
    cin >> M;
    for (int j = 0; j < M; ++j) {
        int budget;
        cin >> budget;
        int max_items = maxItems(prices, budget);
        cout << budget << ": " << max_items << endl;
    }
    return 0;
}
```

 Copy

### Time Complexity:

- Sorting the items takes  $O(N \log N)$  time.
- For each of the  $M$  queries, iterating through the sorted items takes  $O(N)$  time.
- Thus, the overall time complexity of the algorithm is  $O(M \cdot N + N \log N)$ . Sorting dominates the time complexity, and subsequent queries only involve linear time operations.

But, we can still improve if use `upper_bound()` and prefix sum concept.

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n;
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }
    sort(arr, arr + n);
    for (int i = 1; i < n; i++) {
        arr[i] += arr[i - 1];
    }

    int m;
    cin >> m;
    while (m--) {
        int budget;
        cin >> budget;
        cout << upper_bound(arr, arr + n, budget) - arr << endl;
    }

    return 0;
}
```

 Copy

This code initializes an array `arr` to store the cumulative sum of the sorted prices. Then, for each query, it finds the index of the upper bound of the budget in the `arr` array, which effectively gives the number of items that can be purchased with that budget. This approach provides better efficiency compared to iterating through the array for each query.

**Time Complexity:** The time complexity is dominated by the sorting operation ( $O(n \log n)$ ) and then for each query, finding the upper bound ( $O(\log n)$ ), so overall it is  $O((n + m) \log n)$ .

---

## Algorithmic Design

Algorithmic design involves the systematic creation and optimization of algorithms to solve specific computational problems. In the context of the given problem, the objective is to design an efficient algorithm to handle a series of queries related to student records.

Each query specifies an operation, such as adding a student with a given name and marks, erasing entries for a specific student, erasing all entries for a student, or printing the first entry of marks for a given student.

A well-designed algorithm should consider the efficiency of operations, minimizing time and space complexities. Utilizing appropriate data structures, such as hash tables or associative

containers, can facilitate quick access and modification of student records. The algorithm must gracefully handle cases where entries do not exist for a given student.

Additionally, an optimal solution should strive for simplicity and maintainability, ensuring ease of understanding and maintenance as the algorithm evolves or scales to larger datasets.

---

### Let's directly jump to one example and understand the things better:

Imagine managing a database of dashboard of some firm with the following operations:

**Insert:** Add a new user to the server named x, with some value.

**Sum:** Sum of all the value of user.

**Erase Entries:** Remove particular user.

**Get Max:** Retrieve the max value of the user.

**Get Distinct:** No. of users with distinct value.

### Algorithmic Explanation:

#### 1. Insert Operation:

- When inserting a new user with value x:
  - Increment the current sum by x.
  - Update the map **mp** to keep track of the frequency of each value.

#### 2. Remove Entries Operation:

- When removing a user with value x:
  - Decrement the current sum by x.
  - Decrement the frequency of x in the map **mp**.
  - If the frequency becomes 0, remove the entry for x from the map.

#### 3. Sum Operation:

- Simply return the current sum stored in the variable **cur\_sum**. This operation runs in constant time  $O(1)$ .

#### 4. Get Max Operation:

- Find the maximum value in the map **mp** by accessing the last entry (since it's sorted in ascending order by default).

#### 5. Get Distinct Operation:

- Return the size of the map **mp**, which represents the number of distinct users.

### Code:

```
#include <iostream>
#include <map>

using namespace std;

struct bag {
    int cur_sum = 0;
    map<int, int> mp;

    void insert(int x)    //  $\theta(\log n)$ 
    {
        cur_sum += x;
        mp[x]++;
    }

    void remove(int x)    //  $\theta(\log n)$ 
    {
        cur_sum -= x;
        mp[x]--;
        if (mp[x] == 0) {
            mp.erase(x);
        }
    }

    int sum()             //  $\theta(1)$ 
    {
        return cur_sum;
    }

    int getmax()          //  $\theta(1)$ 
    {
        auto it = mp.end();
        it--;
        return it->first;
    }

    int getdistinct()     //  $\theta(1)$ 
    {
        return mp.size();
    }
};

int main() {
    bag b;
    b.insert(5);
    b.insert(7);
    b.insert(5);
    cout << "Sum: " << b.sum() << endl; // Output: Sum: 17
    cout << "Max: " << b.getmax() << endl; // Output: Max: 7
    cout << "Distinct: " << b.getdistinct() << endl; // Output: Distinct: 2
    b.remove(5);
    cout << "Sum after removing 5: " << b.sum() << endl; // Output: Sum after removing 5: 7
    return 0;
}
```

This implementation efficiently manages user data in a server dashboard, supporting various operations with respective time complexities.

Mark as Read ✓

Write a comment

**B**

***I***

**U**

**</>**

**🔗**

### • Before you post...

I Understand ✕

- **No Cheating or Violation of Honor Code** : Users are usually prohibited from discussing or sharing solutions for active contests. This is to prevent cheating and to maintain the integrity of the platform.
- **Avoid Posting Full Solutions** : Users are encouraged not to post entire solutions to problems, especially for active contests. Instead, it's common to share hints, tips, or discuss general approaches without giving away the entire solution.
- **Be Respectful** : Users are expected to be respectful and considerate of others. Avoid using offensive language, making personal attacks, or engaging in any form of harassment.
- **Stay On Topic** : Discussions should stay relevant to the problem at hand. Off-topic discussions or spamming are generally discouraged.
- **No Advertising or Self-Promotion** : Users should not use the platform for advertising or self-promotion. This includes promoting external services, products, or personal projects.

## All Comments