1_cheet_sheet.cpp

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

// Basics
    int max = INT_MIN;
    int min = INT_MAX;
    // MIN MAX
    std::max(max_profit, current_profit);
    std::min(max_profit, current_profit);

    // MOD OF 10^9 + 7
    const int MOD = 1e9 + 7;
    int num  = 7;
    int cnt  = __builtin_popcount(num); // no of set bits

    long long num = 1234567843232;
    long long cnt = __builtin_popcountll(num);

    // Permutations
    string s = "abc";
    string s = "123";
    string s = "321";
    sort(s.begin(), s.end());

    do {
        cout<<s<<endl;
    } while(next_permutation(s.begin(), s.end()));

    int maxele = *max_element(a, a+n);
    int minele = *min_element(a, a+n);


-----------------------------------------------------------------------------------------
--
// Pairs
    pair<int , int> p = {1, 2};
    cout<<p.first<<" "<<p.second;

    pair<int, pair<int, int>> p = {1, {2, 3}};
    cout<<p.first<<" "<<p.second.first<<" "<<p.second.second;

    pair<int int> p[] = {{1, 2}, {3, 4}, {5, 6}};
    cout<<p[1].second;
-----------------------------------------------------------------------------------------
--
// Array
    int numbers[5]; // declares an array of 5 integers
    int numbers[5] = {1, 2, 3, 4, 5}; // declares and initializes the array
    int numbers[] = {1, 2, 3, 4, 5}; // size is implicitly determined to be 5

    numbers[0] = 10; // sets the first element to 10
    int value = numbers[1]; // gets the second element
```

```cpp

    // Looping through an Array
    for(int i = 0; i < 5; i++) {
        std::cout << numbers[i] << " ";
    }
    // Multidimensional Arrays
    int matrix[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };
-------------------------------------------------------------------------------------------
--
// Vectors
    #include <vector>

    // Declaration & Initialization
    vector<int> v;

    vector<int> v(5);
    vector<int> v(5, 0);
    vector<int> v(5, -1);

    vector<int> v(v1);

    // Operations
    v.push_back(1);
    v.emplace_back(2);

    vector<pair<int, int>> v;
    v.push_back({1, 2});
    v.emplace_back(1, 2);

    v.push_back({1, 2});
    v.pop_back();

    cout<<v[0];
    cout<<v.at(0);
    cout<<v.front();
    cout<<v.back();
    cout<<v.size();

    v.erase(v.begin() + 1);
    v.erase(v.begin() + 1, v.begin() + 3); // [start, end)

    vector<int> v(2, 100);
    v.insert(v.begin(), 300)
    v.insert(v.begin() + 1, 2, 200)
    v.insert(v.begin(), v2.begin(), v2.end());

    v1.swap(v2);
    v.clear();
    cout<<v.empty();

```

```
    // Iterators
    vector<int>::iterator it = v.begin();
    it++
    cout<<*(it);

    it = it + 2;
    cout<<*(it);

    vector<int>::iterator it = v.begin();
    vector<int>::iterator it = v.end();
    vector<int>::iterator it = v.rend();
    vector<int>::iterator it = v.rbegin();

    for(vector<int>::iterator it = v.begin(); it != v.end(); it++){
        cout<<*(it)<<" ";
    }

    for (auto it=v.begin(); it!=v.end(); it++) {
        cout<<*(it)<<" ";
    }

    for(auto it : v){
        cout<<it<<" ";
    }
-------------------------------------------------------------------------------------------
--
// 2D array
    vector<vector<int> > &A

-------------------------------------------------------------------------------------------
--
// Sorting
    // Basic
        // sorting in descending order 54321
        sort(A.begin(), A.end(), greater<int>());
        // sorting in ascending order 12345
        sort(A.begin(), A.end());
        sort(a, a+n);

        sort(a+2, a+4);
        sort(a, a+n, greater<int>);



    // Comparator for Pairs
        bool comp(pair<int, int> &a, pair<int, int> &b){
            if(a.first < b.first){
                return true;
            }else if(a.first == b.first){
                return a.second > b.second;
            }
            return false;
        }

```

```cpp
        pair<int, int> a[] = {{1, 2}, {3, 4}, {5, 6}};
        sort(a, a+n, comp);

    // min max of array
        int arr[] = {10, 20, 5, 40, 50};
        int n = sizeof(arr) / sizeof(arr[0]); // size of the array

    // Finding the minimum element
        int minElement = *min_element(arr, arr + n);

    // Finding the maximum element
        int maxElement = *max_element(arr, arr + n);

-------------------------------------------------------------------------------------------
--
// unordered_map
    // Declare an unordered_map
    std::unordered_map<std::string, int> umap;

    // Inserting key-value pairs into the unordered_map
    umap["apple"] = 1;
    umap["banana"] = 2;
    umap["orange"] = 3;

    // Accessing elements by key
    std::cout << "Apple count: " << umap["apple"] << std::endl;

    // Check if "apple" is in the map
    if (umap.count("apple") > 0) {
        std::cout << "'apple' is present in the map." << std::endl;
    } else {
        std::cout << "'apple' is not present in the map." << std::endl;
    }

    // Checking if a key exists using find()
    if (umap.find("banana") != umap.end()) {
        std::cout << "Banana is in the map" << std::endl;
    } else {
        std::cout << "Banana is not in the map" << std::endl;
    }

    // Iterating through all elements in the unordered_map
    for (const auto& pair : umap) {
        std::cout << "Key: " << pair.first << ", Value: " << pair.second << std::endl;
    }

    // Erasing an element by key
    umap.erase("orange");

    // Size of the unordered_map
    std::cout << "Size of the unordered_map: " << umap.size() << std::endl;

-------------------------------------------------------------------------------------------
--
```

```cpp
// unordered_set
    // Declare an unordered_set of integers
    std::unordered_set<int> uset;

    // Inserting elements into the unordered_set
    uset.insert(10);
    uset.insert(20);
    uset.insert(30);
    uset.insert(10); // Duplicate, won't be added

    // Checking if an element is in the set
    if (uset.find(20) != uset.end()) {
        std::cout << "20 is in the set" << std::endl;
    } else {
        std::cout << "20 is not in the set" << std::endl;
    }

    // Iterating through the unordered_set
    std::cout << "Elements in the set: ";
    for (const auto& elem : uset) {
        std::cout << elem << " ";
    }
    std::cout << std::endl;

    // Removing an element from the set
    uset.erase(20);

    // Checking the size of the unordered_set
    std::cout << "Size of the set: " << uset.size() << std::endl;

------------------------------------------------------------------------------------------
--
// Set
    set<int> s;
    s.insert(1);
    s.emplace(2);

    auto it = s.find(1);
    s. erase(it);
    s.erase(2);

    auto it = s.lower_bound(1);
    auto it = s.upper_bound(1);
------------------------------------------------------------------------------------------
--
// Multiset
    multiset<int> ms;
    ms.insert(1);
    ms.insert(1);
    ms.insert(1);

    ms.erase(1)

    int cnt = ms.count(1);
```

```
260
261     ms.erase(ms.find(1))
262
263     ms.erase(ms.find(1), ms.find(1)+2);
264 ------------------------------------------------------------------------------------------
    --
265 // Map
266     map <int, int> m;
267     map <int, pair<int, int>> m;
268     map <pair<int, int>, int> m;
269
270     m[1] = 3;
271     map.emplace({1 , 3})
272     map.insert({1, 3});
273
274     for(auto it: m){
275         cout<<it.first<<" "<<it.second<<endl;
276     }
277
278     cout<<m[1];
279
280     auto it = m.find(1);
281     cout<<*(it).second;
282 ------------------------------------------------------------------------------------------
    --
283 // Multimap
284 ------------------------------------------------------------------------------------------
    --
285 // Binary Search
286     int left = 0, right = A, result = 0;
287
288     while (left <= right) {
289         int mid = left + (right - left) / 2;
290         long long sum = (long long)mid * (mid + 1) / 2;
291
292         if (sum == A) {
293             return mid;
294         } else if (sum < A) {
295             result = mid; // mid can be a possible answer
296             left = mid + 1;
297         } else {
298             right = mid - 1;
299         }
300     }
301
302     return result;
303 ------------------------------------------------------------------------------------------
    --
304 // Linked List :
305     struct ListNode {
306         int val;
307         ListNode *next;
308         ListNode(int x) : val(x), next(NULL) {}
309     };
```

```
310
311     ListNode* Solution::solve(ListNode* A, int B, int C) {
312
313         ListNode *new_node = new ListNode(B);
314
315         if(A == NULL){
316             return new_node;
317         }
318         if (C == 0){
319             new_node->next = A;
320             A = new_node;
321             return A;
322         }
323
324         ListNode *temp = A;
325         int count = 0;
326         while(temp->next != NULL){
327         count++;
328         if( count  == C-1){
329                 temp = temp->next;
330                 break;
331         }else{
332                 temp = temp->next;
333         }
334         }
335
336         new_node->next = temp->next;
337         temp->next = new_node;
338
339         return A;
340
341     }
342 ------------------------------------------------------------------------------------------
    --
343 // List
344     list<int> ls;
345
346     ls.push_back(1);
347     ls.emplace_back(2);
348
349     ls.push_front(1);
350     ls.emplace_front(2);
351 ------------------------------------------------------------------------------------------
    --
352 // Stacks
353
354     #include <stack>
355
356     // Create a stack of integers
357     std::stack<int> s;
358
359     // Push elements onto the stack
360     s.push(10);
361     s.push(20);
```

```
    s.push(30);

    // Access the top element
    std::cout << "Top element: " << s.top() << std::endl;  // Output: 30

    // Remove the top element (pop operation)
    s.pop();

    std::cout << "Top element after pop: " << s.top() << std::endl;  // Output: 20

    // Check if the stack is empty
    if (s.empty()) {
        std::cout << "Stack is empty!" << std::endl;
    } else {
        std::cout << "Stack is not empty!" << std::endl;
    }

    // Get the size of the stack
    std::cout << "Stack size: " << s.size() << std::endl;  // Output: 2
-------------------------------------------------------------------------------------
--
// Queues
    #include <queue>

    // Create a queue of integers
    std::queue<int> q;

    // Enqueue elements (add elements to the back of the queue)
    q.push(10);
    q.push(20);
    q.push(30);

    // Access the front element
    std::cout << "Front element: " << q.front() << std::endl;  // Output: 10

    // Access the rear (back) element
    std::cout << "Back element: " << q.back() << std::endl;  // Output: 30

    // Dequeue (remove) the front element
    q.pop();

    std::cout << "Front element after pop: " << q.front() << std::endl;  // Output: 20

    // Check if the queue is empty
    if (q.empty()) {
        std::cout << "Queue is empty!" << std::endl;
    } else {
        std::cout << "Queue is not empty!" << std::endl;
    }

    // Get the size of the queue
    std::cout << "Queue size: " << q.size() << std::endl;  // Output: 2

```

```
-----------------------------------------------------------------------------------------------
--
// Deque
    deque<int> dq;
    dq.push_back(1);
    dq.emplace_back(2);
    dq.push_front(1);
    dq.emplace_front(2);

    dq.pop_back();
    dq.pop_front();

    dq.front();
    dq.back();
-----------------------------------------------------------------------------------------------
--
// trees
    void preorder(TreeNode* node, vector<int>& result) {
        if (node == NULL) {
            return;
        }

        // Visit the root node
        result.push_back(node->val);

        // Traverse the left subtree
        preorder(node->left, result);

        // Traverse the right subtree
        preorder(node->right, result);
    }

    vector<int> Solution::preorderTraversal(TreeNode* A) {
        vector<int> result;
        preorder(A, result);
        return result;
    }
-----------------------------------------------------------------------------------------------
--
// Heap / Priority Queue
    // Max Heap
    priority_queue<int> maxHeap;

    // Insert elements into the Max Heap
    maxHeap.push(10);
    maxHeap.push(20);
    maxHeap.push(15);

    while (!maxHeap.empty()) {
        std::cout << maxHeap.top() << " "; // Access the maximum element
        maxHeap.pop(); // Remove the maximum element
    }

    maxHeap.empty();
```

```
465
466     maxHeap.top();
467     maxHeap.pop();
468
469     // Min Heap
470     priority_queue<int, vector<int>, greater<int>> minHeap;
471
472     minHeap.push(10);
473     minHeap.push(30);
474
475     minHeap.empty();
476
477     minHeap.top();
478     minHeap.pop();
479
480     // Array to Heap
481     vector<int> heap = {10, 20, 15};
482
483     // Create a Max Heap
484     make_heap(heap.begin(), heap.end());
485
486     // Add a new element
487     heap.push_back(25);
488     push_heap(heap.begin(), heap.end());
489
490     cout << "Max Heap elements: ";
491     while (!heap.empty()) {
492         pop_heap(heap.begin(), heap.end()); // Moves max element to the end
493         cout << heap.back() << " "; // Access the maximum element
494         heap.pop_back(); // Remove the maximum element
495     }
496
497 ----------------------------------------------------------------------------------------------
    --
498 // Graphs
499
500 ----------------------------------------------------------------------------------------------
    --
```