Competitive
Programming and
Mathematics
Society

Tacking Coding Interviews

With C++

Programming Portfolio

Attendance

Pairing Amazonians

To enforce Amazon's value of 'letting the young learn from their elders', the company pairs the least experienced software developer with the most experienced software developer, the second least experienced developer with the second most experienced developer and so on. The experience of a pair is the average of the experiences of the two individuals. For example, if we have 6 employees with experiences (in years) of 1,9,12,5,8,6, then we'd have the pairs $(1, 12)$, $(5, 9)$, $(6, 8)$ following Amazon's pairing rule.

Pairing Amazonians

To enforce Amazon's value of 'letting the young learn from their elders', the company pairs the least experienced software developer with the most experienced software developer, the second least experienced developer with the second most experienced developer and so on. The experience of a pair is the average of the experiences of the two individuals. For example, if we have 6 employees with experiences (in years) of 1,9,12,5,8,6, then we'd have the pairs $(1, 12)$, $(5, 9)$, $(6, 8)$ following Amazon's pairing rule.

How many *unique* experiences of pairs are there? In our example, the respective experiences of these pairs are $\frac{13}{2}$, 7, 7, so there are two unique experiences ($\frac{13}{2}$ and 7).

Which Data Structure?

Which Data Structure?

Whenever a question asks about any form of ‘uniqueness’, this is a good sign that we can use a . . .

Which Data Structure?

Whenever a question asks about any form of ‘uniqueness’, this is a good sign that we can use a . . . set!

What is a Set?

- A set is a collection of **unique** elements.
- It supports operations like insertion, deletion, unions, intersections, and membership testing.
- In C++, sets are provided in the Standard Template Library (STL)
    - Can be accessed using "#include <set>"

Different Types of Sets

- **set**:
    - Implemented using binary search trees (usually Red-Black trees)
    - Stores elements in order
    - $O(\log n)$ insertion, lookup, deletion

Different Types of Sets

- **set**:
  - Implemented using binary search trees (usually Red-Black trees)
  - Stores elements in order
  - $O(\log n)$ insertion, lookup, deletion
- **unordered_set**:
  - Implemented using hash tables
  - Elements are not stored in order
  - $O(1)$ insertion, lookup, deletion (on average) but in the worst case could be $O(n)$ (during rehashing)

Different Types of Sets

- **set**:
  - Implemented using binary search trees (usually Red-Black trees)
  - Stores elements in order
  - $O(\log n)$ insertion, lookup, deletion
- **unordered_set**:
  - Implemented using hash tables
  - Elements are not stored in order
  - $O(1)$ insertion, lookup, deletion (on average) but in the worst case could be $O(n)$ (during rehashing)
- **multiset**:
  - Can store repeating elements (non-unique) in order
  - Implemented using binary search trees (usually Red-Black trees)
  - $O(\log n)$ insertion, lookup, deletion

Different Types of Sets

- **set**:
  - Implemented using binary search trees (usually Red-Black trees)
  - Stores elements in order
  - $O(\log n)$ insertion, lookup, deletion
- **unordered_set**:
  - Implemented using hash tables
  - Elements are not stored in order
  - $O(1)$ insertion, lookup, deletion (on average) but in the worst case could be $O(n)$ (during rehashing)
- **multiset**:
  - Can store repeating elements (non-unique) in order
  - Implemented using binary search trees (usually Red-Black trees)
  - $O(\log n)$ insertion, lookup, deletion

We also sometimes use union, intersection which just follows the same properties as a set in maths.

Time Complexities of Set Operations

- **Ordered Set / Multiset (‘set / multiset‘)**
  - `Insert`: $O(\log n)$
  - `Erase (single element)`: $O(\log n)$
  - `Erase (multiple elements)`: $O(k \log n)$ for k elements
  - `Find`: $O(\log n)$
  - `Lowerbound/Upperbound`: $O(\log n)$
  - `Count`: $O(\log n)$
  - `Size`: $O(1)$

Time Complexities of Set Operations

- **Ordered Set / Multiset (‘set / multiset‘)**
  - `Insert`: $O(\log n)$
  - `Erase (single element)`: $O(\log n)$
  - `Erase (multiple elements)`: $O(k \log n)$ for k elements
  - `Find`: $O(\log n)$
  - `Lowerbound/Upperbound`: $O(\log n)$
  - `Count`: $O(\log n)$
  - `Size`: $O(1)$
- **Unordered Set (‘unordered_set‘)**
  - `Insert`: Average $O(1)$, Worst $O(n)$
  - `Erase (single element)`: Average $O(1)$, Worst $O(n)$
  - `Find`: Average $O(1)$, Worst $O(n)$
  - `Count`: Average $O(1)$, Worst $O(n)$
  - `Size`: $O(1)$

Using Sets

```
// Initialising a set
set<int> s;

// Inserting values into the set
s.insert(10);
s.insert(20);
s.insert(30);

// Checking the size of the set
cout << "Size: " << s.size() << endl; // Prints 'Size: 3'

// Finding an element in the set
if (s.find(20) != s.end()) {
    cout << "20 is in the set" << endl;
}

// A different type of lookup: lower_bound, upper_bound (like binary search)
cout << *s.lower_bound(10) << endl; // will output 10
cout << *s.upper_bound(10) << endl; // will output 20

// Removing an element from the set
s.erase(10);
```

Using Sets

```
// Iterating through a set
for (auto it = s.begin(); it != s.end(); ++it) {
    cout << *it << " ";
}
cout << endl;

// Another way to iterate using range-based for loop
for (int x : s) {
    cout << x << " ";
}
cout << endl;

// Check if the set is empty
if (s.empty()) {
    cout << "The set is empty" << endl;
}

// Clear all elements in the set
s.clear();
```

It’s your turn!

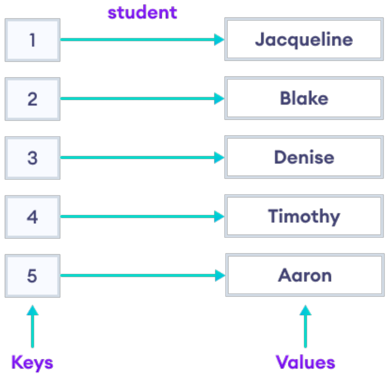
```cpp
#include <bits/stdc++.h>

int uniqueExperiences(vector<int> xp) {
    int numEmployees;
    cin >> numEmployees; // analagous to scanf("%d", &numEmployees) in C

    vector<int> xp; // dynamically sized array to store the experience levels

    // scan in all the experiences (in years) of the Amazon employees
    for (int i = 0; i < numEmployees; i++) {
        int currentXp;
        cin >> currentXp;
        xp.push_back(currentXp);
    }

    // TODO: Return the number of unique experiences!
}
```

A Nice and Easy Solution

```cpp
#include <bits/stdc++.h>

int uniqueExperiences(vector<int> xp) {
    sort(xp.begin(), xp.end());
    set <int> uniquePairs;

    for (int i = 0; i < xp.size() / 2; i++) {
        int currentPairXp = xp[i] + xp[xp.size() - 1 - i];
        uniquePairs.insert(currentPairXp);
    }

    return uniquePairs.size();
}
```

Maps

Maps are conceptually similar to sets, but instead of containing unique elements, maps contain unique "keys" that are mapped to associated "values".



Using Maps

```
// Initialising a map
map<int, string> m;

// Inserting values into the map
m[1] = "Jacqueline";
m.insert(make_pair(2, "Blake"));

// Modifying values
m[2] = "Jake";

// Accessing a value
cout << m[1] << endl; // Prints 'Jacqueline'

// Other types
map<int, int> m2;
m2[1] = 2;
m2[2] = 3;
m2[1]++;  // Now m2[1] == 3
```

Using Maps

```cpp
    // Iterating through a map
    for (auto it = m.begin(); it != student.end(); ++it) {
        cout << it->first << " " << it->second << endl;
    }
    for (auto x : m) {
        cout << x.first << " " << x.second << endl;
    }

    // Check if empty
    m.empty();

    // Get size
    m.size();

    // Remove a key-value pair
    m.erase(1);
```

Ordered Maps

Just like with sets, we have a choice between ordered maps and unordered maps.

Ordered maps will maintain the order of the key-value entries, sorting on the key, while unordered maps do not maintain this order.

This is useful if you want to iterate through your map in order of key. Furthermore, it allows you to search for a certain key in $O(\log n)$ time.

```
// Returns an iterator to the first element not less than the given key
lower = lower_bound(v.begin(), v.end(), 2);

// Returns an iterator to the first element greater than the given key
upper = upper_bound(v.begin(), v.end(), 3);
```

Other Maps

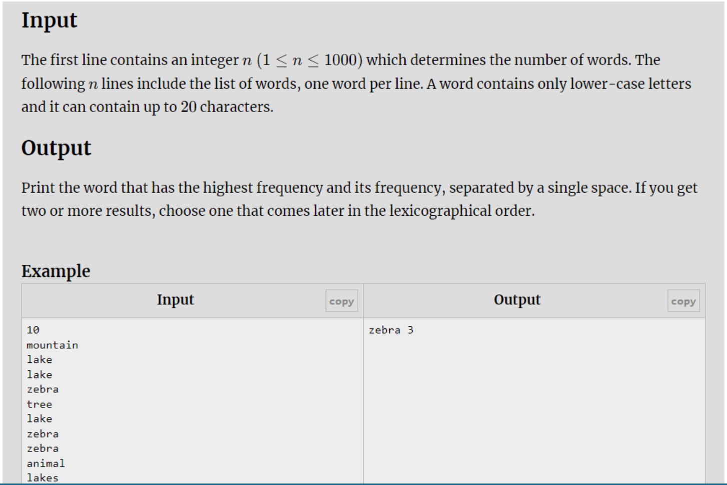
Again, like sets, there are other variations of maps.

- **map**: stores key-value pairs in order, $O(logn)$ insertion, lookup, deletion
- **unordered_map**: does not store key-value pairs in order, $O(1)$ insertion, lookup, deletion (on average)
- **multimap**: like a map but can store multiple copies of same key
- **unordered_multimap**: like a unordered_map but can store multiple copies of same key

So, use unordered_maps if you don't need to worry about maintaining order, otherwise it may be a good idea to use a regular ordered map.

Map Problem

Given a list of words, you are asked to write a program to find the most frequent word and how many times it occurs. (https://vjudge.net/contest/583927problem/F)



Input

The first line contains an integer n $(1 \le n \le 1000)$ which determines the number of words. The following n lines include the list of words, one word per line. A word contains only lower-case letters and it can contain up to 20 characters.

Output

Print the word that has the highest frequency and its frequency, separated by a single space. If you get two or more results, choose one that comes later in the lexicographical order.

Example

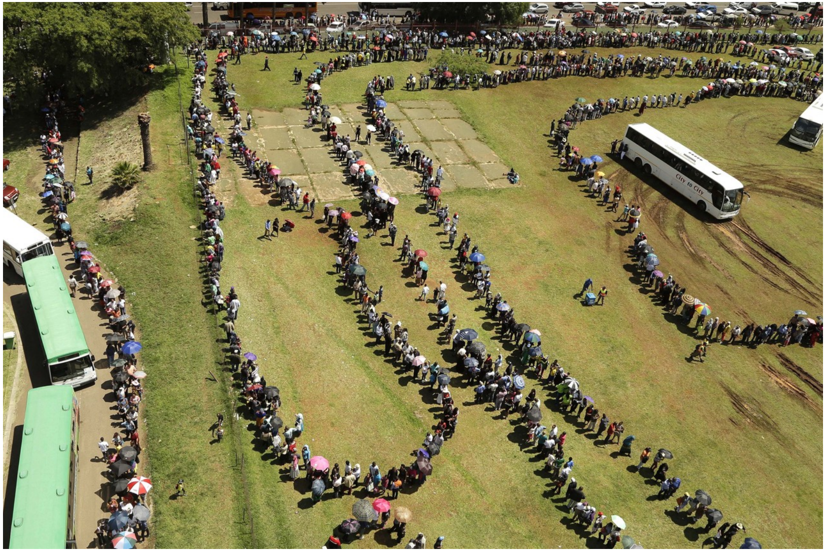
Input	Output
10 mountain lake lake zebra tree lake zebra zebra animal lakes	zebra 3

Problem Set

Map Problem

```
// Sample solution
int n; cin >> n;

unordered_map<string, int> m;
for (int i = 0; i < n; i++) {
    string s; cin >> s;
    m[s]++;
}

string word = "";
int maxFreq = 0;
for (const auto& entry : m) {
    if (entry.second > maxFreq || (entry.second == maxFreq && entry.first > word)) {
        word = entry.first;
        maxFreq = entry.second;
    }
}
cout << word << ' ' << maxFreq << endl;
```

Queue



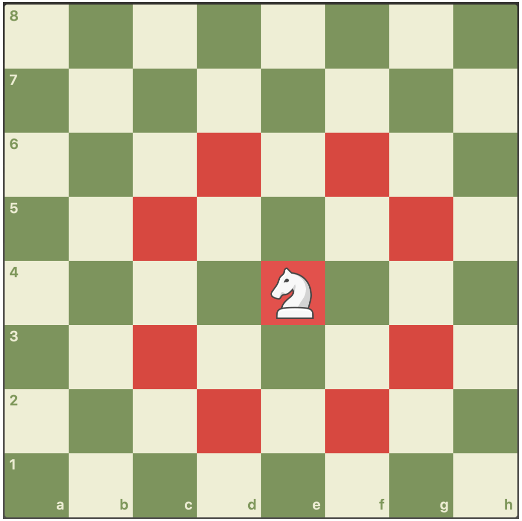
Using Queues

```
// Initialising a queue
queue<pair<int, int>> q;
// Inserting values into the queue O(1)
q.push(1);
q.push(3);
// q = 1 -> 3
// Getting the size of the pq O(1)
cout << q.size() << endl; // 2

// Getting the element at the front of the queue O(1)
cout << q.front() << endl; // 1

// Dequeuing the first item in the queue and storing it O(1)
q.pop()// q = 3
//checks if the queue is empty O(1)
q.empty() //false
```

Knights

Given a square chessboard of N x N size, the position of the Knight and the position of a target are given. Find the minimum steps a Knight will take to reach the target position.

Knights



Knights

- We can think of every move by the knight as having a distance or cost of 1. If we can get to our current cell in n moves then we can get to the surrounding 8 ish cell in n + 1 moves (unless they can already be achieved in less).

Knights

- We can think of every move by the knight as having a distance or cost of 1. If we can get to our current cell in n moves then we can get to the surrounding 8 ish cell in n + 1 moves (unless they can already be achieved in less).
- Since we're trying to look for the minimum number of moves, it would be sensible to consider the unexplored neighbours of cells that require less moves to get to before we explore the ones that take a long time to get to. These cells are near the beginning of the queue.

Knights

- We can think of every move by the knight as having a distance or cost of 1. If we can get to our current cell in n moves then we can get to the surrounding 8 ish cell in n + 1 moves (unless they can already be achieved in less).
- Since we're trying to look for the minimum number of moves, it would be sensible to consider the unexplored neighbours of cells that require less moves to get to before we explore the ones that take a long time to get to. These cells are near the beginning of the queue.
- If we start the queue with the square that takes no moves to get to. As we keep taking neighbours of cells that takes few moves to get to, squares that takes less moves to get to will always be queued before ones that take more moves to get to. They will thus also be considered first.

Knights and Queues

- add the starting position of the knight onto the queue (least moves, that being 0)

Knights and Queues

- add the starting position of the knight onto the queue (least moves, that being 0)
- take the square at the start of the queue, check if it is the target.

Knights and Queues

- add the starting position of the knight onto the queue (least moves, that being 0)
- take the square at the start of the queue, check if it is the target.
- if not, add its unvisited neighbours to the end of the queue, indicating that it takes one more move than the one we for to it from.
- repeat until the queue is empty (when there are no more unvisited squares we can get to)

Knights and Queues

- add the starting position of the knight onto the queue (least moves, that being 0)
- take the square at the start of the queue, check if it is the target.
- if not, add its unvisited neighbours to the end of the queue, indicating that it takes one more move than the one we for to it from.
- repeat until the queue is empty (when there are no more unvisited squares we can get to)
- **notice that at any point in time, the items closer to the beginning of the queue will take less moves to get to than the ones near the back of the queue (they can be equal) as we always process the ones that take less moves to get to.

Implementation

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <queue>

using namespace std;

int N;
int x1, y1;
int x2, y2;

typedef pair<int, int> coordinate;
queue <pair<coordinate, int>> q;
vector <vector <bool>> visited;

// all 8 ways a knight can jump (ordered from left to right)
int dx[8] = {-2, -2, -1, -1, 1, 1, 2, 2};
int dy[8] = {1, - 1, 2, -2, 2, -2, 1, -1};
```

Priority Queue

- Priority queues are used to store elements in a sorted order.

Priority Queue

- Priority queues are used to store elements in a sorted order.
- What makes priority queues useful is that they can facilitate dynamic operations

Priority Queue

- Priority queues are used to store elements in a sorted order.
- What makes priority queues useful is that they can facilitate dynamic operations
- For example: we can add new elements into the priority queue and the priority queue will ensure that it will still be sorted - **push**
- We can also delete the biggest element from the priority queue - **pop**

Priority Queue

- Priority queues are used to store elements in a sorted order.
- What makes priority queues useful is that they can facilitate dynamic operations
- For example: we can add new elements into the priority queue and the priority queue will ensure that it will still be sorted - **push**
- We can also delete the biggest element from the priority queue - **pop**
- However, priority queue only allows you to see the biggest element in the priority queue. To see the second largest element, you have to first perform the **pop** operation, then look at the biggest element in the priority queue which will be the second largest element.

Priority Queue

- Priority queues are used to store elements in a sorted order.
- What makes priority queues useful is that they can facilitate dynamic operations
- For example: we can add new elements into the priority queue and the priority queue will ensure that it will still be sorted - **push**
- We can also delete the biggest element from the priority queue - **pop**
- However, priority queue only allows you to see the biggest element in the priority queue. To see the second largest element, you have to first perform the **pop** operation, then look at the biggest element in the priority queue which will be the second largest element.
- And of course, we have the operation **size** which allows the user to query how many elements are in the priority queue.

Priority Queue

- Under the hood, priority queues are implemented as a heap (if you don’t know what a heap is do not worry). This makes it so that most operations are **logarithmic** time.

Priority Queue

- Under the hood, priority queues are implemented as a heap (if you don’t know what a heap is do not worry). This makes it so that most operations are **logarithmic** time.
- **size()** - $O(1)$

Priority Queue

- Under the hood, priority queues are implemented as a heap (if you don't know what a heap is do not worry). This makes it so that most operations are **logarithmic** time.
- **size()** - $O(1)$
- **push()** - $O(\log n)$

Priority Queue

- Under the hood, priority queues are implemented as a heap (if you don't know what a heap is do not worry). This makes it so that most operations are **logarithmic** time.
- **size()** - $O(1)$
- **push()** - $O(\log n)$
- **pop()** - $O(\log n)$

Priority Queue

- Under the hood, priority queues are implemented as a heap (if you don't know what a heap is do not worry). This makes it so that most operations are **logarithmic** time.
- **size()** - $O(1)$
- **push()** - $O(\log n)$
- **pop()** - $O(\log n)$
- **top()** - $O(1)$

Using Priority Queues

```
// Initialising a priority queue
priority_queue<int> pq;

// Inserting values into the pq
pq.push(1);
pq.push(3);

// Getting the size of the pq
cout << pq.size() << endl; // 2

// Gettting the largest element of the pq
cout << pq.top() << endl; // 3

// Deleting the biggest element from the pq
pq.pop();
```

Priority Queue

- Technically, a set can do every operation that a priority queue can do. However, priority queue tends to be a tiny bit faster (less constant factor) and is generally preferred over a set for algorithms like Dijkstra's algorithm etc.

Question Time

Farmer John has N cows that need to be milked (1 <= N <= 10,000), each of which takes only one unit of time to milk.

Being impatient animals, some cows will refuse to be milked if Farmer John waits too long to milk them. More specifically, cow i produces g_i gallons of milk ($1 \leq g_i \leq 1000$), but only if she is milked before a deadline at time d_i ($1 \leq d_i \leq 10000$). Time starts at $t = 0$, so at most x total cows can be milked prior to a deadline at time $t = x$.

Please help Farmer John determine the maximum amount of milk that he can obtain if he milks the cows optimally.

Priority Queue

- The most greedy approach unfortunately would not work.

Priority Queue

- The most greedy approach unfortunately would not work.
- The approach goes something like this: construct the scheduling starting from $t = 0$, choosing the available cow of largest milk output g_i at each time step.

Priority Queue

- The most greedy approach unfortunately would not work.
- The approach goes something like this: construct the scheduling starting from $t = 0$, choosing the available cow of largest milk output g_i at each time step.
- A case where this would not work would be where we have two cows, C and D, with deadlines at $t = 1$ and $t = 2$, respectively.
- If cow D has a larger output, this rule would cause us to pick D first, but then C becomes unavailable, even though we could have chose both cows by picking C first.

Priority Queue

- Instead, let us try doing the greedy approach starting at $t = 10000$ and working towards the beginning.

Priority Queue

- Instead, let us try doing the greedy approach starting at $t = 10000$ and working towards the beginning.
- Again, the rule will be, at each time step, to choose the best available cow available at that time.

Priority Queue

- Instead, let us try doing the greedy approach starting at $t = 10000$ and working towards the beginning.
- Again, the rule will be, at each time step, to choose the best available cow available at that time.
- The key difference here is that once a cow becomes available (i.e., t decreases below the cow's deadline (d_i) it will always be available thereafter.
- Hence, we can never miss a cow by delaying to take it (unless we haven't taken it before reaching $t = 0$).

Priority Queue

- Now that we have a greedy algorithm planned out, we need to figure out how to implement it.

Priority Queue

- Now that we have a greedy algorithm planned out, we need to figure out how to implement it.
- A naive implementation would take $O(dN)$ time (where d is the maximum deadline); there are d time steps and at each time step you have to determine the best of N cows.

Priority Queue

- Now that we have a greedy algorithm planned out, we need to figure out how to implement it.
- A naive implementation would take $O(dN)$ time (where d is the maximum deadline); there are d time steps and at each time step you have to determine the best of N cows.
- We can improve upon this by using a priority queue. As t decreases, any cow that becomes available gets added to the priority queue.
- When we need to find the best cow, we can just pop a cow off the top of the queue. Using the priority queue gives a solution of $O((d + N) \log N)$ time.