Sets and Maps in C++

Each competitive programming problem has a time limit, and one of the hallmarks of a harder problem is one where a solution exists, but it takes too long to solve some cases. A classic task that comes up as a portion of many problems is determining if some item has previously been seen in the data. A straightforward way of handling this with a vector would be to simply loop through the whole vector looking for the item in question.

```
vector<string> names;
// fill names.

// To search for item…
bool found = false;
for (int i=0; i<names.size(); i++)
    if (names[i].compare(item))
        found = true;
```

While this works, if the vector names is large, the amount of time it would take to search for many different items would add up. For example, if the vector had 10^5 names in it, and you were asked to perform 10^5 searches, then up to $10^5 \times 10^5 = 10^{10}$ simple operations would be necessary, and this is too many for a competitive programming problem with time limits of only a few seconds.

Luckily, C++ has a built in data structure to use called a set, which can greatly speed up the search time for an item. Mathematically, a set is a collection of objects without duplicates. (So if you add 3 into an initially empty set twice, the set will only have one item in it.) Sets allow you to do the following operations:

1. Add an item to them.
2. Delete an item from them.
3. Search for an item in the set.

Each of these operations will only take about $\log_2 n$ steps in a set with n items. Just for reference, if $n = 10^5$, then $\log_2 n$ is roughly equal to 17. Thus, instead of the previous set of actions taking 10^{10} simple steps, they would take roughly $17 \times 10^5 = 1.7$ million number of steps, which is very easily manageable for competitive programming time limits.

In addition, sets in C++ are ordered, thus, in addition to the functionality above, we can also do the following:

4. Loop through the sets in increasing or decreasing order.
5. Get a pointer to the least element greater than or equal to a given element.
6. Get a pointer to the least element strictly greater than a given element.

#4 takes linear time (n steps) while both 5 and 6 take roughly $\log_2 n$ steps.

The set documentation is here:

https://cplusplus.com/reference/set/set/

Let's look at a problem that doesn't require a set, but one that can be more easily solved with a set:

Kattis Problem: Trip Odometer
Here is the link to the problem:

https://open.kattis.com/problems/tripodometer

After carefully reading the problem, we recognize that we have to add up all the input values, and then to generate the output list, subtract out each **unique** value from the input list to create the output list. Since we want these differences in order, we want to start by subtracting out the largest item (thus, we want to loop through the **unique** input values in descending order.

Thus, our plan is as follows:

Create a set of integers, and a total sum variable. Read in each number, adding each into the set (insert), and add each number (including duplicates) into the total sum variable. Then, loop through the set in descending order (using rbegin and rend).

When using the iterators, recall that the pointer is to a memory address. Thus, to obtain the value at that memory address, use the star operator (*). Here is how we would print out the values in a set items in backwards order:

```
for (auto ptr=items.rbegin(); ptr != items.rend(); ptr++)
    cout << (*ptr) << " ";
```

Now, let's look at the full solution:

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

int main() {
    int n, tmp, total = 0;
    cin >> n;
    set<int> items;

    for (int i=0; i<n; i++) {
        cin >> tmp;
        total += tmp;
        items.insert(tmp);
    }

    cout << items.size() << endl;
    for (auto ptr=items.rbegin(); ptr != items.rend(); ptr++)
        cout << total - (*ptr) << " ";
    cout << endl;

    return 0;
}
```

Even though this problem can be solved without a set, the set simplifies the code and makes it quite efficient. One thing to note is that the bounds of this problem limited the individual trip lengths so that instead of a set, a boolean array of size 10001 could be used. But, if we simply changed the bounds to allow for longer trips (say upto length 10^9), then we would no longer be able to create a boolean vector that was long enough. This is where sets are particularly important: in problems or situations where the range of values is too large to use a frequency/boolean vector. Let's take a look at a problem that has a larger range of input values to keep track of:

Kattis Problem: Odd Man Out
Here's a link to the problem:

https://open.kattis.com/problems/oddmanout

In this problem, a list of numbers is given, where several numbers appear exactly twice and one number appears exactly once. The goal is to figure out which number appears exactly one time. The numbers range from 1 to $2^{31} - 1$, so the full range of positive integers that fit in the int data type. Due to the range of the input, a boolean vector can't be made to store the frequency of each value. Instead, we can add each item to a set the first time we see it. When we see it a second time, we can remove the item from the set. When we are done, there should be one item left in the set.

The key detail to finding an element in the set is that the find function returns an iterator to the element, if found, if not, it returns end().

Let's take a look at the solution to the problem:

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

int main() {

    // Get # of cases.
    int nC;
    cin >> nC;

    // Process cases.
    for (int loop=1; loop<=nC; loop++) {

        int n, tmp;
        cin >> n;
        set<int> items;

        // Just keep track of trip sum and each unique length.
        for (int i=0; i<n; i++) {
            cin >> tmp;

            set<int>::iterator ptr = items.find(tmp);

            // Not in set, add it.
            if (ptr == items.end())
                items.insert(tmp);

            // Already in set, so erase.
            else
                items.erase(ptr);
        }

        // Header
        cout << "Case #" << loop << ": ";

        // This should just be one value...
        for (auto ptr=items.begin(); ptr != items.end(); ptr++)
            cout << (*ptr);
        cout << endl;
    }

    return 0;
}
```

Note the syntax for the return type of the find operator.

Other than these two problems, the other two set methods that are used fairly frequently are:

`lower_bound()`

and

`upper_bound()`

These methods are quite similar. Both return a pointer either an element greater than or equal to the input parameter (lower_bound) or an element strictly greater than the input parameter. Here is a short example:

https://cplusplus.com/reference/set/set/lower_bound/

In the example in the link, lower_bound on 30 returns a pointer to the smallest item greater than or equal to 30 in the set (which is 30). Secondly, the upper_bound of 60 is 70 because 70 is the smallest item strictly greater than 60. The erase method erases every item within the range specified by the two pointers itlow and itup.

Maps in C++

While a set tracks unique values, a map takes a set one step further, storing unique values called keys, and allowing a single value to be mapped to each unique key. For example, we could create a map storing the number of pets each person owns:

Amanda → 2
Byron → 0
Claire → 3
Deron → 2

In the map represented above, there are four entries (keys): Amanda, Byron, Claire and Deron. Furthermore, we can look up for each key, it associated value, which for this example, is the number of pets that person owns. Notice that it's okay for two different keys to map to the same value. Both Amanda and Deron have 2 pets. Here is the API listing for maps:

https://cplusplus.com/reference/map/map/

Just like sets, the keys for maps are ordered. The entries of maps are pairs. One nice syntax item is that to obtain the associated value for a key known to be in the map, the [] operators (just like vectors) can be used.

The first problem we'll look at is a very straightforward map problem, which simply creates a dictionary (this is the word used in Python for the equivalent of a map).

Kattis Problem: Babelfish
Here is the link to the problem:

https://open.kattis.com/problems/babelfish

The first part of the input provides a dictionary (set of entries to put into the map), and then in the second part of the input, translations are required. If a key isn't in the dictionary, we must respond, "eh". The bounds are such that there's not enough time for each look up, to loop through the whole word list. This is where the map comes in handy.

The insert function is used to add entries into the map.
The find function determines if a particular key exists in a map.
The [] allow us to access the corrresponding value for a key.

Here's the solution to the problem. Note that some care must be taken with the parsing due to the input format. For brevity, the getTokens function previously shown in the notes isn't displayed.

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

int main() {

    string line;
    map<string,string> dictionary;

    getline(cin, line);

    while (line.compare("")) {

        // Tokenize.
        vector<string> toks = getTokens(line, ' ');

        // Add matching pair in desired order.
        dictionary.insert(pair<string,string>(toks[1], toks[0]));

        // Get next line.
        getline(cin, line);
    }

    // Continue reading until end of file.
    while (getline(cin, line)) {

        map<string,string>::iterator ptr = dictionary.find(line);

        if (ptr == dictionary.end())
            cout << "eh" << endl;
        else
            cout << dictionary[line] << endl;
    }

    return 0;
}
```

Common Use of Maps

Maps can get more complicated than straight mappings like this. The most common use case is usually when we have a set of identifiers (strings), but we would like to relabel them with unique identifiers 0, 1, 2, … In the input, if we have a identifier we've never seen before, we want to assign it to the next available ID. For example, if we had the list of strings:

SARAH
JAMAL
SARAH
SARAH
ROXY
JAMAL

then we would want to assign the identifiers as follows:

SARAH → 0
JAMAL → 1
ROXY → 2

Here is a segment of code that accomplishes this, where we first read in an integer n, the number of names, and then the names follow:

```
int n, id = 0;
cin >> n;
map<string,int> mymap;

for (int i=0; i<n; i++) {

    string tmp;
    cin >> tmp;

    map<string,int>::iterator ptr = mymap.find(tmp);

    if (ptr == mymap.end())
        mymap.insert(pair<string,int>(tmp, id++));
}
```

Basically, in this code segment, after we read in each string, we check to see if it's in the name. If not, we add it mapped to the current id value, followed by incrementing id (which we can do in one statement).

Another common use case with this data would be to map each string to how many times it appears. (For example, imagine that the names lists were all votes for those individuals.) Here is the edit of the code above that does this:

```
int n;
cin >> n;
map<string,int> mymap;

for (int i=0; i<n; i++) {

    string tmp;
    cin >> tmp;

    map<string,int>::iterator ptr = mymap.find(tmp);

    if (ptr == mymap.end())
        mymap.insert(pair<string,int>(tmp, 1));
     else
        mymap[tmp]++;
}
```

So the neat thing here is that once we know a key is in the map, we can almost treat it like a vector, where we index with the key!

Alternatively, we can even map a single value, such as a string, to a vector or something more complicated! Consider the following problem:

Kattis Problem: Grandpa Bernie
Here is the problem:

https://open.kattis.com/problems/grandpabernie

We get a long list of trips (upto 100000). Unlike the previous example where the input is just strings, in addition to the strings (locations), we also get a year the trip occurred. A string occurring more than once indicates that that location was visited more than once. Thus, our desired map might look something like this:

Mexico → 2015, 2020
Italy → 2016, 2018, 2023

Each item needs to be mapped to a vector of integers, representing the years that the country was visited. The rest of the problem is given a country and an integer k, determine the year that Grandpa Bernie took the kth trip to that country. Upto 100,000 of these queries have to be answered.

Thus to solve the problem efficiently, we need to create a map that maps strings to vectors of integers. When we read in a city, we must see if that's in the map. If not, create an entry for it. If it is, then just add the year of that trip. When we finish this, each of the vectors will be unsorted, so we have to go sort each of them. Then, to answer each query, we use the map to find the country and then since the vector is sorted, we can just output the item in index k-1. (Since the input is 1-based.)

It turns out that this approach (at least in the implementation below) gets a Time Limit Exceeded verdict. The likely reason for this is that in this solution, we add new entries into the map, one by one, so each vector (value that is mapped) gets created one by one. This creation as the program is running is slowing down the program. Still, it's good to look at the code for the purpose of learning the appropriate syntax for a more complicated map:

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

int main() {

    int n;
    cin >> n;
    map<string,vector<int>> trips;

    for (int i=0; i<n; i++) {

        string country;
        int year;
        cin >> country >> year;

        map<string,vector<int>>::iterator ptr = trips.find(country);
        if (ptr == trips.end())
            trips.insert(pair<string,vector<int>>(country, {year}));

        // We can just add to the list.
        else
            trips[ptr->first].push_back(year);
    }

    // Sort each vector.
    for (auto ptr=trips.begin(); ptr != trips.end(); ptr++)
        sort(trips[ptr->first].begin(), trips[ptr->first].end());

    int q;
    cin >> q;
    for (int i=0; i<q; i++) {

        // Get query, make 0 based.
        string country;
        int rank;
        cin >> country >> rank;
        rank--;

        // Get appropriate list.
        map<string,vector<int>>::iterator it = trips.find(country);
        vector<int> mylist = it->second;

        // Answer query.
        cout << mylist[rank] << "\n";
    }

    return 0;
}
```

So, let's consider some different ways to speed up this code, keeping in mind that we don't want to dynamically allocate new vectors, one by one. We know that there will not be more than n vectors, so what we could do is create a vector of n vectors at the beginning, like this:

```
vector<vector<int>> years(n);
```

By putting the n in the parameters of the constructor, we tell the compiler to go ahead and create n separate empty vectors. This is faster than dynamically adding those n vectors, one by one, as our program is in the middle of reading in data.

But now, we can not index this vector with strings (countries). Instead, we must reassign each country to a unique id, and then use that id to index the vector of vector years. So, here is a visual of this alternate solution which will speed up our program that got TLE:

Original Data
Mexico 2020
Italy 2018
Italy 2023
Mexixo 2015
Italy 2016

How We Will Store It
Mexico → 0
Italy → 1

years[0] → 2020, 2015
years[1] → 2018, 2023, 2016

After initially storing the data, then we can just sort each non-empty vector:

years[0] → 2015, 2020
years[1] → 2016, 2018, 2023

To answer queries, we get the country, and use the country map to figure out that country's ID (in this case, 0 or 1), and then we use the rank-1 as the index into the appropriate vector in years.

Let's take a look at the accepted solution that uses this strategy on the next page:

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

int main() {

    int n, id=0;
    cin >> n;
    map<string,int> cMap;
    vector<vector<int>> years(n);

    // Read names.
    for (int i=0; i<n; i++) {

        string country;
        int year;
        cin >> country >> year;

        map<string,int>::iterator ptr = cMap.find(country);

        // New country
        if (ptr == cMap.end())
            cMap.insert(pair<string,int>(country,id++));

        // Add this year.
        years[cMap[country]].push_back(year);
    }

    // Sort each vector.
    for (int i=0; i<years.size(); i++)
        if (years[i].size() > 0)
            sort(years[i].begin(), years[i].end());

    // Handle queries.
    int q;
    cin >> q;
    for (int i=0; i<q; i++) {

        // Get query, make 0 based.
        string country;
        int rank;
        cin >> country >> rank;
        rank--;

        // Answer query.
        cout << years[cMap[country]][rank] << "\n";
    }

    return 0;
}
```

One thing to notice is that even though the picture of this storage system seems quite convoluted, the code itself is reasonably succinct.