Sorting in C++

In order to solve many problems, it's very useful to sort data in some sort of order. All primitive types, such as int, long long, double, char, etc. have a pre-defined built in order (that is consistent with how we usually think about these items), from lowest to highest value. Luckily. C++ and most other languages have built in functions to sort primitive data, and the facility to custom sort more complicated data.

Let's first learn how to sort a vector of primitives. In the algorithm library, there is a sort function that takes in the beginning pointer and end pointer of the data to be sorted. So a typical use case looks like this:

```
vector<int> data;
// Fill data with values.
sort(data.begin(), data.end());
```

As previously noted in the vector section, we could identify a consecutive subsequence of a vector to sort, but this is rarely done.

Let's go ahead and look at a Kattis problem where it's necessary to sort the data. In many instances, having the data in sorted order allows for a greedy solution to a problem (one where not all options have to be explored because we can prove the best option can be easily found.)

Kattis Problem: Planting Trees (sorting integers in C++)

The link to the problem is here:

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

The problem is about a set of trees to be planted. Each day, one tree can be planted. Different trees take different amounts of time to grow. Farmer Jon would like to throw a party, but for stylistic reasons, refuses to do so until all the trees have finished growing. The problem is to figure out the fastest amount of time (in days) he can throw his party. For example, if there are four trees to plant, which take 2, 3, 4, and 3 days, respectively to grow, his best option is to plant the tree that takes 4 days to grow on day 1, then on days 2 and 3 plant the trees that take 3 days to grow, followed by planting the tree that takes 2 days to grow on day 4. According to the problem, the party is the day after all the trees have finished growing, so the party can happen on day 7. (The last two trees complete growing on day 6.)

It should be fairly clear from this description that the solution is to sort the input data of number of days each tree takes to grow in reverse order, and then simulate planting the trees in this order only and figure out the longest time any of the tress takes to grow.

We'll present two solutions to this problem, which show us how to

(a) Sort a vector in increasing order.
(b) Sort a vector in decreasing order.

In the first solution, we'll sort our input in increasing order: [2, 3, 3, 4] for the sample previously given. Then, we'll loop through the data backwards, planting the last tree first, and so forth.

Let's look at this solution:

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

int main() {

    vector<int> trees;
    int n, tmp;
    cin >> n;

    // Read trees.
    for (int i=0; i<n; i++) {
        cin >> tmp;
        trees.push_back(tmp);
    }

    // Sort the vector from smallest to largest.
    sort(trees.begin(), trees.end());

    // Go through the vector backwards. j represents what we add
    // to the growing time for when the party could be.
    int res = 0;
    for (int i=n-1,j=2; i>=0; i--,j++)
        res = max(res, trees[i]+j);

    cout << res << endl;

    return 0;
}
```

Also note the use of two loop indexes in the loop towards the end. While the value of j can be calculated from i, often times, it's more simple (requires less mental strain), to just use two indexes moving in the same loop in the manner shown above. Syntactically, just separate both the initialization and increment statements with a comma.

Obviously, it would be more natural to go through this data from largest to smallest. In order to do this, we must add a parameter to the sort function so that it "knows" to sort in reverse order. (Alternatively, we can call the reverse function for vectors…) Here is how to sort a vector in reverse order:

```
sort(data.rbegin(), data.rend());
```

rbegin() refers to a the beginning of the reverse iterator, and rend() refers to the ending of the revrse iterator. With this in mind, here is the alternate solution to Planting Trees that sorts the data in reverse order:

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

int main() {

    vector<int> trees;
    int n, tmp;
    cin >> n;

    // Read trees.
    for (int i=0; i<n; i++) {
        cin >> tmp;
        trees.push_back(tmp);
    }

    // Sort the vector from largest to smallest.
    sort(trees.rbegin(), trees.rend());

    // Go through the vector forwards. i+2 represents the waiting
    // days and completion day for each tree.
    int res = 0;
    for (int i=0; i<n; i++)
        res = max(res, trees[i]+i+2);

    cout << res << endl;

    return 0;
}
```

Sorting Pair Data

Now, we turn to sorting data in pairs. By default, C++ sort sorts pair data by the first item, breaking ties by the second item. This can be extremely handy in problems where the method by which you want to order items is more complicated: say sorting by last name, but breaking ties by first name, for example.

In order to sort pair data, just fill a vector with pairs and just call sort as before:

```
sort(data.begin(), data.end());
```

Now let's consider the following problem Kattis problem Cups, where each cup has two attributes: color and size:

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

In the problem, we must read in some data about cups, both their color and size. One strange twist is that unlike other problems, the data isn't always given in the same order. Sometimes the color comes first, other times a number, representing the size, comes first. Furthermore, if the number comes first, then the number is the diameter of the cup, but if the number comes second, it represents the radius of the cup. We would like to sort these cups from smallest size to largest size. Thus, it makes sense for us to store the cups as pairs with the first item being an int, the second a string (color), since the first item to sort by is size. We will store diameters so we use ints only.

Here is the solution:

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

bool isPosInt(string s);

int main() {

    vector<pair<int,string>> cups;
    int n, tmp;
    cin >> n;

    // Read cups.
    for (int i=0; i<n; i++) {
        string s, t;
        cin >> s >> t;

        // If first is int, that's the diameter.
        if (isPosInt(s))
            cups.emplace_back(stoi(s), t);

        // In this case, second item is radius, so mult by 2.
        else
            cups.emplace_back(2*stoi(t), s);
    }

    // Sort the vector.
    sort(cups.begin(), cups.end());

    // We want to print out the colors...so second...
    int res = 0;
    for (int i=0; i<cups.size(); i++)
        cout << cups[i].second << endl;

    return 0;
}

// Returns true iff s is a positive integer (not checking bounds though),
// without any signs in front.
bool isPosInt(string s) {

    // Go through each character.
    for (int i=0; i<s.size(); i++) {

        // Two obvious cases it fails.
        if (s[i]>'9') return false;
        if (s[i]<'0') return false;

        // Can't start with a 0.
        if (i==0 && s[i] == '0') return false;
    }

    // Okay if we get here.
    return true;
}
```

Note that it was simple enough to write a function that checked if a string was a valid integer or not, so we just went through character by character and applied standard logic, knowing that all of the digits have Ascii values that are consecutive. If you can cleverly form pairs, many, many sorts that are helpful to solve problems can be done in this manner, utilizing how pairs are naturally sorted.

Custom Sorting – use of struct

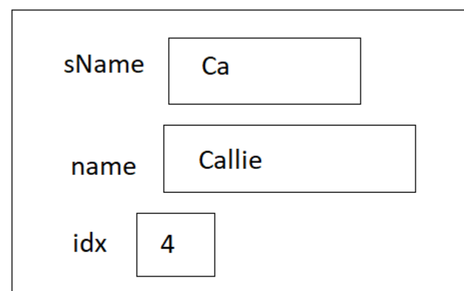
On occasion, though, the data to be sorted might not easily be classified into a pair that can be naturally sorted. For these instances, it's best to create a struct (your own data type) and overload the less than operator. Once you tell C++ what < than means for your struct, then C++'s sort function will be able to sort a vector with the struct you created.

structs are extremely useful in programming and in a longer course, these would be introduced formally on their own with all the details taught carefully over the course of several days. However, when starting competitive programming, one can avoid using structs most of the time, and in many cases, can get away with just using them for custom sorting, so instead of fully explaining structs here, a minimal explanation will be given, enough so that data can be sorted as necessary.

Here is a simple struct definition that will be used in a sample problem:

```
typedef struct person {
    string sName;
    string name;
    int idx;
} person;
```

The typedef is not necessary, but is often used to reduce future typing. Use this word first, followed by the word struct. After that, give a name to your structure (a new type that you are creating). This is followed by an open parenthesis. Inside the struct definition list each field you want to be part of your "object." For this example, each object will have two string fields and an integer field. We can picture a person with the sName of "Ca", name of "Callie", and an integer idx = 4, as follows:



We can create a variable of this type as follows:

```
person myfriend;
```

The typedef allows us to just type “person” instead of “struct person” as the type of the variable myfriend. The memory given is exactly as shown above, with three fields inside the variable with the names shown. To access a field of a struct, use the dot operator (.). If a pointer to a struct is given then the arrow operator is used to access a field (->).

Kattis Problem: Sort of Sorting

We will use the struct previously defined to solve the Kattis Problem: Sort of Sorting. Here is the link:

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

We are asked to sort the input names just by their first two letters. If there is a tie between the first two letters, these are broken by the order in which the names appeared originally in the input. As we can see, sorting by name doesn’t suffice, because if we had:

Callie
Caden
Cammy

sorting just by name would give us:

Caden
Callie
Cammy

but the problem wants us to keep the names in the same order because all three start with “Ca”.

For each name, we’ll create a struct where we store the full name, the first two letters, and the index (order in input) of the name. Once we do this, we can then define a less than operator for the struct which will first break ties in cases with unequal “short names”, and then if both short names are same, will break ties based on the index. Here is how to declare the method for the struct, inside the struct definition:

```
typedef struct person {
    string sName;
    string name;
    int idx;

    inline bool operator < (const person& other) {
        if (sName.compare(other.sName) != 0)
            return sName.compare(other.sName) < 0;
        return idx < other.idx;
    }

} person;
```

The entire function first line should be copied (changing the type to be that of the struct. Inside the function, if we use a field name, this refers to the left-hand struct in the comparison. If we use other.field, then we are referring to the right-hand struct in the comparison.

In this function, you can see if the compare is non-zero, this means the two sNames are different, and if the compare returns an integer less than 0, then the LHS is less than the RHS, so we want to return true. If the sNames are the same, we return true if and only if the LHS index is less than the RHS index. Now, we can look at the full solution:

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

typedef struct person {
    string sName;
    string name;
    int idx;

    inline bool operator < (const person& other) {
        if (sName.compare(other.sName) != 0)
            return sName.compare(other.sName) < 0;
        return idx < other.idx;
    }

} person;

int main() {

    int n;
    cin >> n;
    bool flag = false;
    while (n != 0) {

        // Blank line between cases.
        if (flag) cout << "\n";
        flag = true;

        // Read in people.
        vector<person> items;
        for (int i=0; i<n; i++) {
            string name;
            cin >> name;
            items.emplace_back(person{name.substr(0,2), name, i});
        }

        // Sort structs.
        sort(items.begin(), items.end());

        for (int i=0; i<n; i++)
            cout << items[i].name << endl;

        // Get next case.
        cin >> n;
    }

    return 0;
}
```