C++ [3]

В этой лекции мы посмотрим на то, что C++ нам предоставляет в плане несложных алгоритмов.

Стандартная сортировка

В C++ сортировать можно не только вектора, но и любые последовательные данные, которые можно линейно упорядочить. У нас же примеры применения сортировок будут в основном на векторах.

Чтобы использовать встроенную сортировку, нужно подключить заголовочный файл *algorithm*. По умолчанию, последовательности сортируются от меньшего к большему.

Пусть у нас есть вектор v, можно вызвать функцию *sort*, которая его отсортирует. Её аргументами являются два итератора на начало и конец вектора (либо другого контейнера). Итераторы, простыми словами - умные стрелочки, которые указывают на элементы контейнера.

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

using namespace std;

int main() {
    int n;
    cin >> n;
    vector<int> v(n);
    for (int& x: v) {
        cin >> x;
    }
    sort(v.begin(), v.end()); // посортируй элементы от начала
    ↪  до конца
```

```
    for (int x: v) {
        cout << x << ' ';
    }
    cout << endl;
}
```

Если вы хотите отсортировать какую-то часть вектора, то к итераторам можно прибавлять числа, тем самым изменяя границы отрезка.

```
sort(v.begin() + 2, v.end() - 2); // сортируем отрезок
↪  массива, не включая первые и последние 2 элемента
sort(v.begin(), v.begin() + 5); // первые 5 элементов
```

Компараторы

Иногда нужно сортировать числа не от меньшего к большему, а как-то по-другому. Например, отсортировать по модулю. Для взятия модуля в C++ есть функция $abs()$.

Для таких целей нужно использовать компараторы (compare - сравнить) - функции, возращающие булевые значения: $true$ - когда элементы упорядочены как нужно, $false$ - когда не упорядочены.

После того, как у вас эта функция будет написана, нужно передать её имя третьим аргументом в функцию $sort$. Просто и удобно.

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

using namespace std;

bool cmp(int lhs, int rhs) {
    return abs(lhs) < abs(rhs);
}

int main() {
    int n;
    cin >> n;
    vector<int> v(n);
    for (int& x: v) {
        cin >> x;
```

```
    }
    sort(v.begin(), v.end(), cmp); // третьим аргументом
    for (int x: v) {
        cout << x << ' ';
    }
    cout << endl;
}
```

Сортировка строк

Для стандартных типов определены операторы сравнения. Строки не исключение, их стандартно сравнивают, что называется, лексикографически (относительно таблицы ascii). Это значит, что цифры будут идти до символов, а заглавные латинские символы будут считаться меньшими в сравнении со строчными.

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

using namespace std;

int main() {
    int n;
    cin >> n;
    vector<string> v(n);
    for (string& x: v) {
        cin >> x;
    }
    sort(v.begin(), v.end()); // сортирует почти
    ↪ лексикографически
    for (string x: v) {
        cout << x << ' ';
    }
    cout << endl;
}
```

Если нам вдруг понадобится отсортировать по-человечески, нужно будет снова написать компаратор.

Строки - объекты большие, поэтому их нужно принимать по ссылке. К тому же при сравнении строк сами строки не должны никак меняться, поэтому будем принимать их константными.

Предположим, что в наших строках будут храниться только буквы. Будем приводить их к нижнему регистру с помощью функции *tolower*() из *cctype*.

```
bool cmp_lex(const string& lhs, const string& rhs) {
    int n = min(lhs.size(), rhs.size()); // min лежит в
    ↪ algorithm

    for (int i = 0; i < n; i++) {
        char lc = tolower(lhs[i]); // лежит в cctype
        char rc = tolower(rhs[i]);
        if (lc != rc) { // если не равны, то возвращаем
            return lc < rc; // сравнение этих элементов
        }
    }
    return lhs.size() < rhs.size(); // если одна строка
    ↪ совпадает с частью другой, то другая больше
}
```

Строки, которые равны лексикографически, будут идти в каком-то неизвестном порядке. Если вам известно понятие устойчивости (stable) сортировки, то здесь сортировка не устойчива.

Окей. Ну это типо строки. Строки это клёво, строки это мощно, но иногда всё-таки хочется иметь свои типы. Не всегда получится обойтись типами из стандартной библиотеки.

Структуры в C++

Для того, чтобы определить свой тип, нужно написать *struct* имя_типа {* *что содержит внутри** };.

Давайте для примера создадим тип "точка". Он будет содержать координаты x и y. Создадим вектор этих точек и посмотрим, как обращаться к координатам на коде, который был раньше.

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

using namespace std;

struct point {
    int x;
    int y;
}; // не забывайте ставить точку с запятой

int main() {
    int n;
    cin >> n;
    vector<point> v(n);
    for (int i = 0; i < n; i++) {
        cin >> v[i].x >> v[i].y; // обращение через точку по
        ↪ названию
    }
    sort(v.begin(), v.end());
    for (point vp: v) {
        cout << vp.x << ' ' << vp.y << endl;
    }
}
```

Если вы попробовали скомпилировать программу, написанную выше, вы увидели, что компилятор выдал ошибку. Ошибка связана с тем, что при сортировке наши точки пытаются сравниваться при помощи оператора $<$, но он не знает, как их сравнивать.

Поэтому можно снова написать компаратор, который будет сравнивать сначала по иксам, если они будут равны, то по игрекам. А затем просто передать в *sort*.

```
bool cmp_points(point lhs, point rhs) {
    if (lhs.x != rhs.x) {
        return lhs.x < rhs.x;
    }
    return lhs.y < rhs.y;
}
```

Перегрузка операторов

Вместо того, чтобы писать функцию-компаратор и потом каждый раз её передавать, в рамках C++ можно обойтись несколько более простым приёмом. Этот приём называется перегрузкой оператора и заключается он в том, что вы говорите определённому оператору, как вести себя при взаимодействии с таким-то типом.

В прошлом примере алгоритм сортировки искал как применить оператор меньше и не нашёл. Давайте явно укажем это.

Для этого берём код предыдущего компаратора и изменяем его имя на специальное.

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

using namespace std;

struct point {
    int x;
    int y;
}; // не забывайте ставить точку с запятой

bool operator<(point lhs, point rhs) {
    if (lhs.x != rhs.x) {
        return lhs.x < rhs.y;
    }
    return lhs.y < rhs.y;
}

int main() {
    int n;
```

```
21      cin >> n;
22      vector<point> v(n);
23      for (int i = 0; i < n; i++) {
24          cin >> v[i].x >> v[i].y; // обращение через точку по
            ↪  названию
25      }
26      sort(v.begin(), v.end());
27      for (point vp: v) {
28          cout << vp.x << ' ' << vp.y << endl;
29      }
30  }
```

Также теперь можно использовать этот оператор в каки-то своих выражений. Такое можно проворачивать также на арифметических операторах, только там должно возвращаться значение того типа, который вы хотите получить.

Что важно для алгоритма сортировки: ваше сравнение обязано возвращать $false$ для одинаковых элементов, так как для сортировки используется оператор меньше и для двух одинаковых значений он возвращает $false$. Если это будет нарушено, то программа может иногда внезапно падать.

Сложность алгоритмов

Почему стандартный алгоритм сортировки вам полезен? Если вас учили сортировкам в школе, то это были скорее всего сортировки выбором, пузырьком. Эти сортировки медленные. Если вас учили алгоритму быстрой сортировки, прикольно.

Давайте поговорим, что для нас значит скорость. Мы с вами будем оценивать асимптотическую сложность алгоритмов. Асимптотическая сложность - это то, как сильно растёт время выполнения программы с ростом размеров входных данных.

Мы будем говорить, что время выполнения программы является линейным относительно размера входных данных и записывать это $O(n)$, если, грубо говоря, при увеличении размера входных данных в два раза, время работы программы увеличится примерно в 2 раза. (на самом деле на скорость выполнения в компьютере влияет ещё его загруженность в определённый момент, промахи по кэшу, но об этом узнаете позже).

Что это за алгоритмы? Это, например, простые циклы до n. Очевидно, если n станет в 2 раза больше, то время работы станет в 2 раза больше.

Если вы пишете несколько циклов рядом, то их сложности складываются. Для двух циклов из прошлого примера это будет равно $O(n+n) = O(2n)$, но так не пишут. Действительно, здесь, при увеличении n в 2 раза, время работы всё равно вырастет в 2 раза. Поэтому константы в этом обозначении мы опускаем.

Теперь давайте посмотрим на пример программы, находящей сумму модулей всех попарных разностей элементов:

```cpp
#include <iostream>
#include <vector>

int main() {
    int n;
    std::cin >> n;
    std::vector<int> v(n);
    for (int i = 0; i < n; i++) {
        std::cin >> v[i];
    }
    int sum = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            sum += std::abs(v[i] - v[j]);
        }
    }
    std::cout << sum << std::endl;
}
```

Разберём по частям.

1. Цикл чтения входной последовательности работает за $O(n)$.
2. Два вложенных цикла от 0 до n. На каждую итерацию внешнего цикла будет n итераций внутреннего, в котором выполняются простые операции. Но внешний совершает также n итераций. Значит в целом сложность этого блока кода будет равна $O(n^2)$.
3. Вывод ответа $O(1)$.

Итого $O(n+n^2+1)$. Но какое из этих слагаемых растёт быстрее всего? Если увеличить n в два раза, время на чтение увеличится в два раза, но время работы второй части увеличится в 4 раза. Поэтому можно опустить то, что растёт не так быстро, оно не очень важно. Значит, асимптотика нашего алгоритма равна $O(n^2)$.

Если у вас несколько вложенных циклов как-то безусловно друг в друга вложенных, то общая сложность будет $O(n^k)$, где k - количество вложенных циклов. Если сложность получается $O(nkq + n)$, где k и q как-то зависят от n, но не очень понятно как, то мы также оставляем только $O(nkq)$, так как оно больше $O(n)$.

Частые шаблоны в программах

Какие ещё частые шаблоны появляются в программах?

Вычисление логарифма числа, округлённого вниз, плюс 1.

```cpp
#include <iostream>

using namespace std;

int main() {
    int n;
    cin >> n;
    int result = 0;
    while (n != 0) {
        n /= 2;
        result++;
    }
    cout << result << endl;
}
```

В цикле, который мы записали, при увеличении размера входных данных n в 2 раза, время работы увеличится на одну итерацию. Эквивалентная этому функция - это логарифм. Основание обычно не пишут, так как асимптотически логарифмы равны. Значит, сложность нашего алгоритма $O(log\, n)$. Хорошая, медленно растущая функция, для чисел порядка 2^{64} её значение равно 64.

Зачем нам это всё знать? Это нужно для того, чтобы понимать, сколько наша программа будет работать. У вас для каждой задачи есть ограничение по времени, которое она может потратить на свою работу.

Стандартная эвристика

Стандартная эвристика заключается в том, что за одну секунду вы можете выполнить $C \cdot 10^8$ операций. Вы смотрите на придуманный алгоритм, и оцениваете его сложность. Допустим, у вас получилась сложность $O(n)$, тогда вы можете ожидать, что если n порядка 10^8, то это отработает за одну секунду.

Нужно ещё знать, что разные операции имеют разную стоимость. Например, сложить два числа - дёшево, а вычислить синус - нет. Вычисление синуса состоит из нескольких элементарных операций.

Поэтому нужно делать поправку на сложность операций. Эту поправку нельзя ввести единственным образом, потому что она будет варьироваться от сервера к серверу, в зависимости от производительности серверов. Но если на самых больших данных у вас получается что-то типо 10^8, то с большой вероятностью это заходит.

Часто бывает, если во входных данных $n \approx 10^4$, в большинстве случаев вы можете неявно предполагать, что сложность алгоритма, который нужно написать, равна $O(n^2)$. Например, два вложенных цикла, обход двумерного массива и тому подобные. Так по длине входных данных можно предполагать тип алгоритма, который нужно писать.

Если $n \approx 500$, например, алгоритм возможно будет иметь сложность $O(n^3)$.

Если $n \approx 10^5$, обычно сложность подразумевается равной $O(n \, log \, n)$. Если постараться с такой сложностью может зайти и $n \approx 10^6$.

За сколько работает стандартная сортировка

В C++ сортировка работает $O(n \, log \, n)$. В её реализации вариация на тему быстрой сортировки.

Сложности переборных алгоритмов

Если вы видите маленькие ограничения, скорее всего от вас требуется написать какой-то перебор.

- $O(2^n)$
- $O(n!)$

Бинарный поиск

Сортировка, конечно, весёлая штука. Но зачем она вообще нужна?

Когда вы посортировали свой массив, вы, взяв элемент в центре, можете сказать, что все элементы слева меньше его, а все элементы справа больше его. На этом строится бинарный поиск. Как он работает?

Вам нужно найти какой-то элемент в массиве, пусть это будет элемент key. Сначала сортируем этот массив. Затем находите его центральный элемент $v[mid]$. Если искомый элемент $key > v[mid]$, нужно искать его в отрезке правее от центра. Иначе в отрезке левее центра, включая его. После перехода в эти меньшие отрезки, проделываем там то же самое, пока длина отрезка не станет равной единице. То есть, пока не придём к одному элементу. Если этот элемент равен искомому, то мы нашли ответ. Если нет, то искомый элемент не присутствовал в массиве.

Так как мы каждый раз делим массив на две части, сложность бинпоиска будет $O(log\, n)$, n - длина массива. Как правильно писать бинпоиск:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
int main() {
    int n;
    std::cin >> n;
    std::vector<int> v(n);
    for (int& a: v) std::cin >> a;
    std::sort(v.begin(), v.end());
    int key;
    std::cin >> key;
    int l = 0, r = n;
    while (r - l > 1) {
        int mid = (l + r) / 2;
        if (key >= v[mid]) {
            l = mid;
        }
        else {
            r = mid;
        }
    } // l-й элемент будет искомым ответом
    std::cout << (key == v[l] ? "Yes" : "No");
}
```

Встроенный бинарный поиск

Умение писать бинпоиск самостоятельно важно, но в C++ есть встроенная реализация которая выполняет то же самое, возвращая $true$ или $false$ в зависимости от того, присутствует элемент в массиве или нет.

```
binary_search(v.begin(), v.end(), key);
```

Также важно использовать одну и ту же функцию сравнения в сортировке и бинпоиске, иначе если у вас элементы упорядочены с помощью одного подхода, а ищете вы используя другой, то вы ничего не найдёте.

Если вы хотите, используя функции определить индекс искомого элемента, смотрите в сторону $lower\_bound$ и $upper\_bound$. Смысл у них такой: ищется позиция, в которую можно вставить искомый элемент, не нарушив порядок в отсортированном массиве.

Возвращают они итераторы, вы можете записывать их в переменные, объявленные с помощью auto. Истинный их тип записывается длинно, типо $vector < int >:: iterator$.

```
auto lower = lower_bound(v.begin(), v.end(), key); // первая
↪  позиция искомого
auto upper = upper_bound(v.begin(), v.end(), key); // позиция
↪  за последним значением искомого
```

Чтобы получить из итератора номер позиции, нужно вычесть из него итератор на начало, получим расстояние между итераторами.

```
int lower_id = lower - v.begin(); // индекс левого
int upper_id = upper - v.begin(); // индекс правого
int dist = upper - lower; // расстояние между левым и правым
```

Чтобы вывести элемент, на который указывает итератор, нужно его разыменовать. Для этого есть оператор звёздочка $*$.

```
cout << *lower << endl;
cout << *upper << endl;
```

Заметка про передачу итераторов

Если вы создадите два вектора a и b, а затем напишите:

```
sort(a.begin(), b.end());
```

То ваша программа скомпилируется, не выдав предупреждений...

Интерактивные задачи

В контесте C++ [3] вы встретитесь с новым типом задач, в которых нужно не просто считать сначала данные, а потом с ними что-то сделать и вывести ответ. В интерактивных задачах с вами будет взаимодействовать другая программа, интерактор, которому вы обычно будете задавать какие-то вопросы. На основании его ответов вы будете вычислять, что интерактор хочет получить в итоге в качестве ответа.

Простейшей интерактивной задачей является реализовать бинарный поиск. Интерактор загадал какое-то число, вы пытаетесь его угадать, выводя свои предположения.

Что важно. В интерактивных задачах вы не должны использовать способы ускорения, потому что оно не совсем предсказуемо себя ведёт. Всегда необходимо следовать формату, который вам говорят. Если сказано выводить перевод строки, переводите.

После каждой операции **вывода** вы должны гарантированно выводить, что ваш поток вывода накопил. Для это пишите $cout.flush()$.

Итог

Используйте сортировку, бинарный поиск. Думайте о сложности алгоритма ещё до того, как писать код. Дорешивайте задачи. Это очень важно.

Доп. контесты

Я буду выкладывать какие-то дополнительные контесты тогда, когда после очередного основного наберётся 20+ человек, которые решили все задачи. Это будет уже целесообразно + вам дополнительный стимул всё дорешать.

Если такое количество не набирается, но желание решать есть, можно организовывать "набеги"на Тимус. Как это происходит: собирается группа людей, кому не хватает задач, создаётся набор задач на тимусе и решает их. Кто быстрее всё прорешал, тот выиграл.