Содержание

Data structures and alghorithms

Data structures

Dynamic arrays

Их идея очень проста, вместо статического массива с фиксированной длиной мы используем массив переменной длины. Это удобно тк не всегда удобно работать со статическим массивом, тем более когда его надо обработать. Наиболее популярный динамический массив в C++ это vector:

```
vector<int> v;
v.push_back(3); // [3]
v.push_back(2); // [3, 2]
v.push_back(5); // [3, 2, 4]
```

Вывод чисел аналогичен привычному:

```
cout << v[0] << "\n"; // 3
cout << v[1] << "\n"; // 2
cout << v[2] << "\n"; // 5
```

Также у вектора есть удобный метод size():

```
for (int i=0; i<v.size(); ++i) {
  cout << v[i] << "\n";
}
```

Но не забываем про v.at()

Но последние стандарты C++ позволяют упростить вывод чисел:

```
for (auto x : v) {
  cout << x << "\n";
}
```

Есть и другие интересные методы у vector, быть может потом напишу про них.

Stack

Он предоставляет нам две O(1) операции:

- push(): добавление элемента “наверх”
- pop(): удаление элемента “сверху”

А также одну O(1) функцию доступа:

- top(): возвращение “верхнего” элемента

Таким образом в стеке у нас есть доступ только к элементу сверху(top).

Stack = FILO (first in last out)

```
stack<int> s;
s.push(3); // {3}
s.push(2); // {2, 3}
s.push(5); // {5, 2, 3}
cout << s.top(); // 5
s.pop();
cout << s.top(); // 2
```

Подробнее о стеке можно почитать здесь.

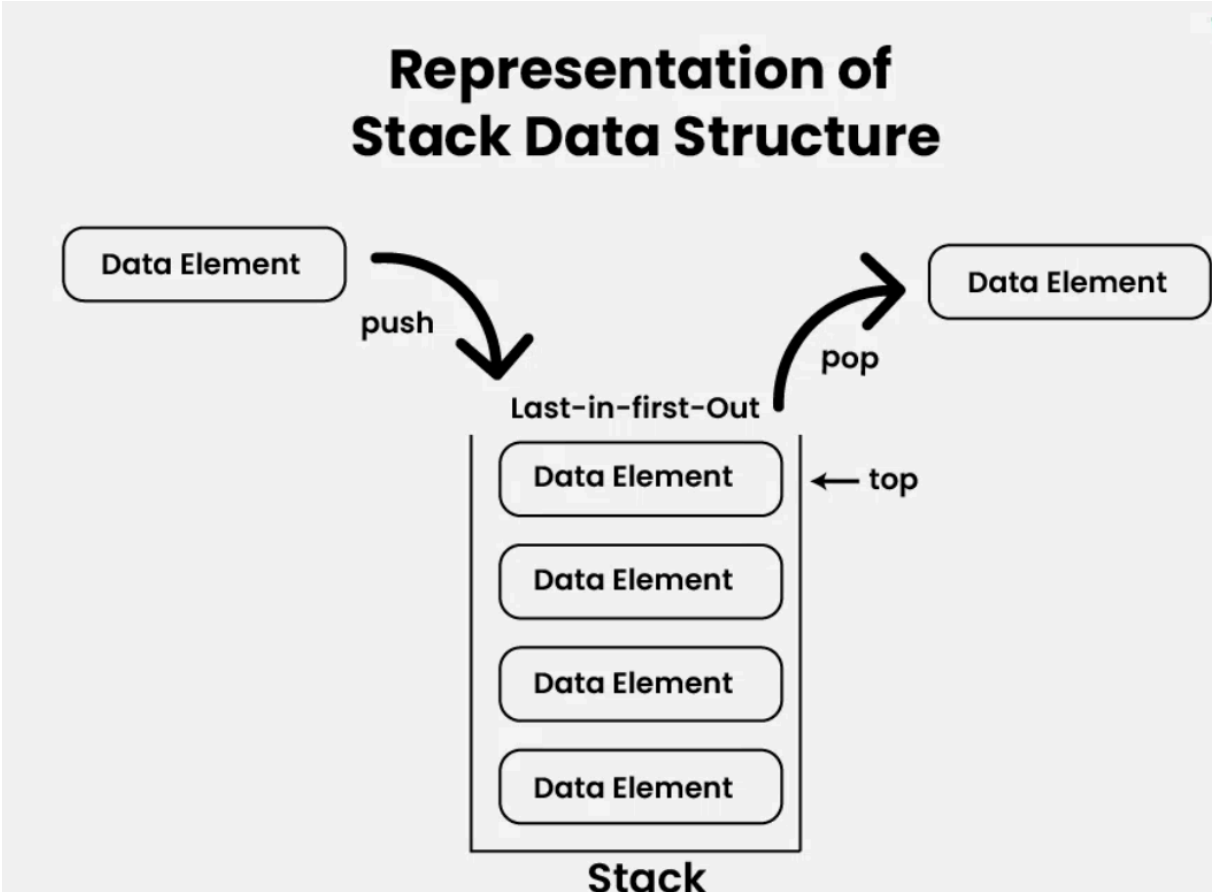

Figure 1: Схема работы стека

Queue

Очередь тоже предоставляет две O(1) операции:

- push(): добавление элемента в “конец” очереди
- pop(): удаление первого элемента из очереди

А также одну O(1) операцию доступа:

- front(): возвращает первый элемент очереди

Таким образом в очереди у нас есть доступ к первому и последнему элементам, а сама queue это FIFO(first in first out).

```
queue<int> q;
q.push(3); // {3}
q.push(2); // {3, 2}
q.push(5); // {3, 2, 5}
cout << q.front(); // 3
q.pop();
cout << q.front() // 2
```

Подробнее про очередь можно почитать здесь.

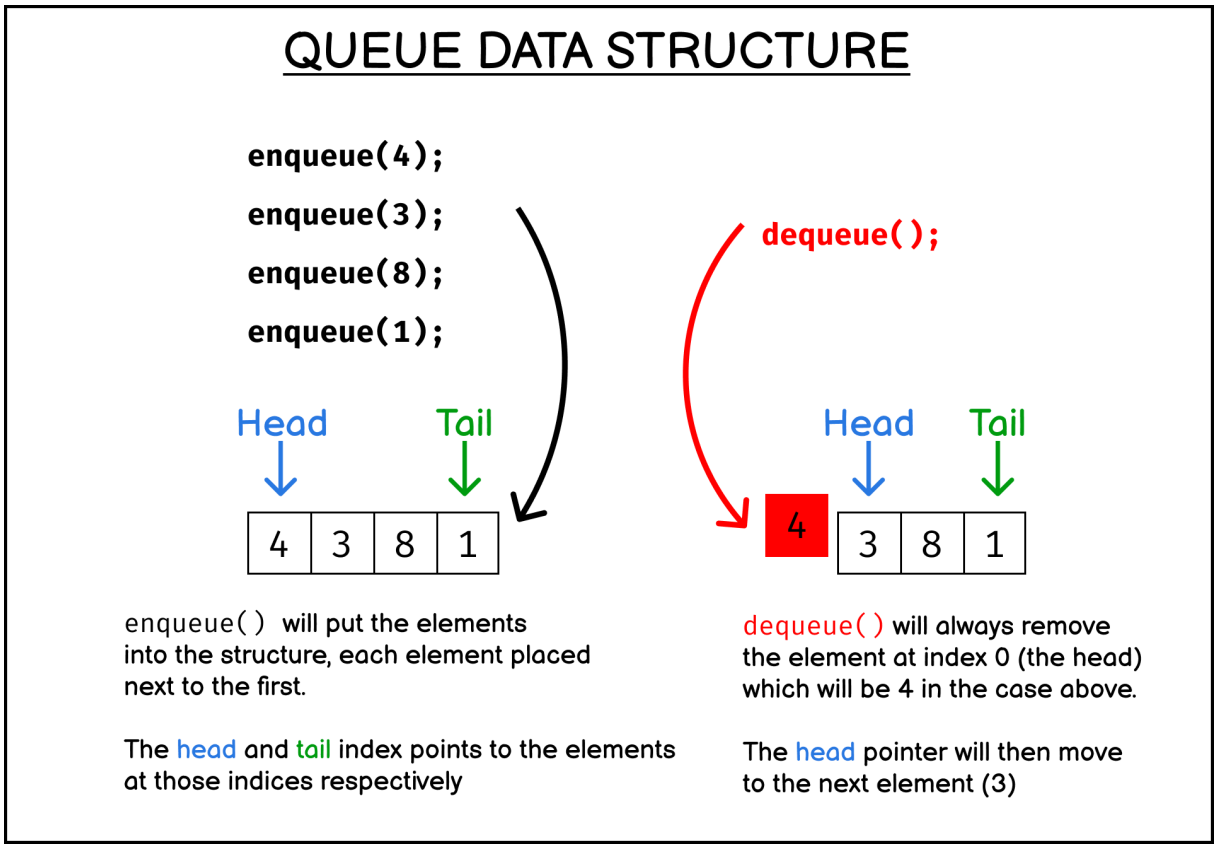

Figure 2: Схема работы очереди (да, тут другие названия, но суть та же)

Deque

это двусторонняя очередь. Она предоставляет нам следующие O(1) методы:

- push_back()
- pop_back()
- push_front()
- pop_front()

А также O(1) операции доступа:

- back()
- front()

```
deque<int> d;
d.push_back(5); // {5}
d.push_back(2); // {5, 2}
d.push_front(3); // {3, 5, 2}
d.pop_back(); // {3, 5}
d.pop_front(); // {5}
```

Внутреннее устройство сложнее чем у вектора, поэтому deque медленнее, но тем не менее O(1) есть. Подробнее о деке можно почитать здесь.

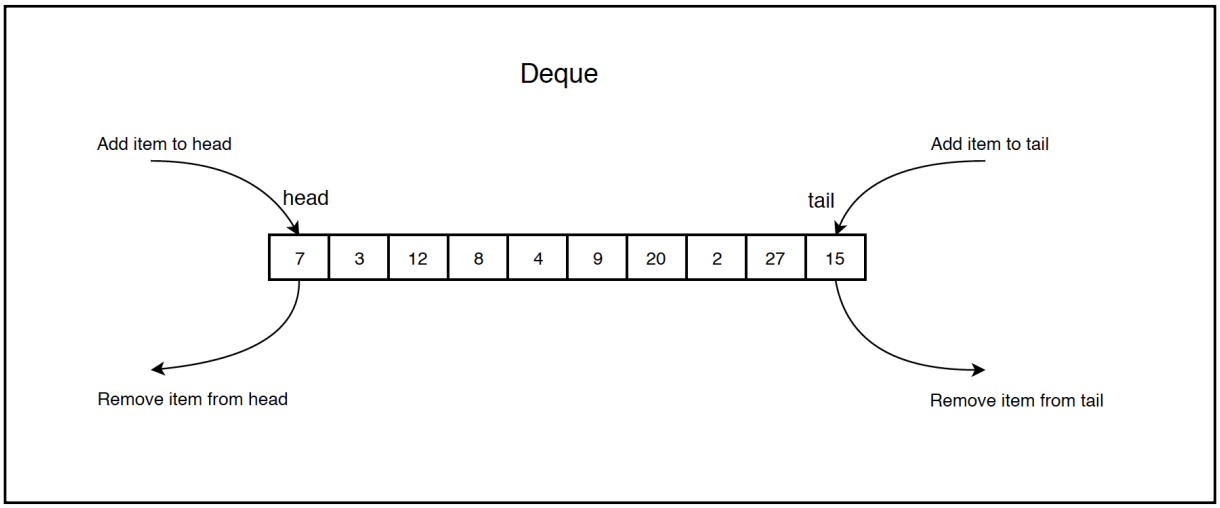

Figure 3: Схема работы дека

Sets

Множеством называется структура данных, в которой хранится набор элементов. Основные операции над множествами - вставка, поиск и удаление. Множества реализованы так, что все эти операции эффективны, что часто позволяет улучшить время работы алгоритмов.

Множества и мультимножества

В стандартной библиотеке C++ имеются две структуры, относящиеся к множествам:

- set основана на сбалансированном двоичном дереве поиска, его операции работают за время O(log n)
- unordered_set основана на хэш-таблице и работает в среднем за O(1). (да, есть крайне малая вероятность выполнения работы алгоритма за O(n))

(Позже мы обсудим, что за деревья и хэш-таблицы такие).

Пример работы из коробки:

```
set<int> s;
s.insert(3);
s.insert(2);
s.insert(5);
cout << s.count(3) << '\n'; // 1
cout << s.count(4) << '\n'; // 0
s.erase(3);
s.insert(4);
cout << s.count(3) << "\n"; // 0
cout << s.count(4) << "\n"; // 1
```

)

Идея множеств в том, что в ней нет дубликатов, только уникальные элементы. Если элемент уже есть во множестве, его невозможно добавить дважды.

```
set<int> s;
s.insert(3);
s.insert(3);
cout << s.count(3) << "\n"; // 1
```

Множества можно использовать как вектор, однако доступ к элементам с помощью оператора [] невозможен. В коде ниже выводится количество элементов, а затем эти элементы перебираются:

```
cout << s.size() << "\n";
for (auto x : s) {
  cout << x << "\n";
}
```

Функция find(x) возвращает итератор, указывающий на элемент со значением x. Если же множество не содержит x, то возвращается итератор end().

```
auto it = s.find(x);
if (it == s.end()) {
  // x не найден
}
```

Упорядоченные и неупорядоченные множества

Как можно догадаться из названия либо множество упорядочено, либо нет, но что же это значит? Если порядок неважен, берем set, иначе unordered_set:

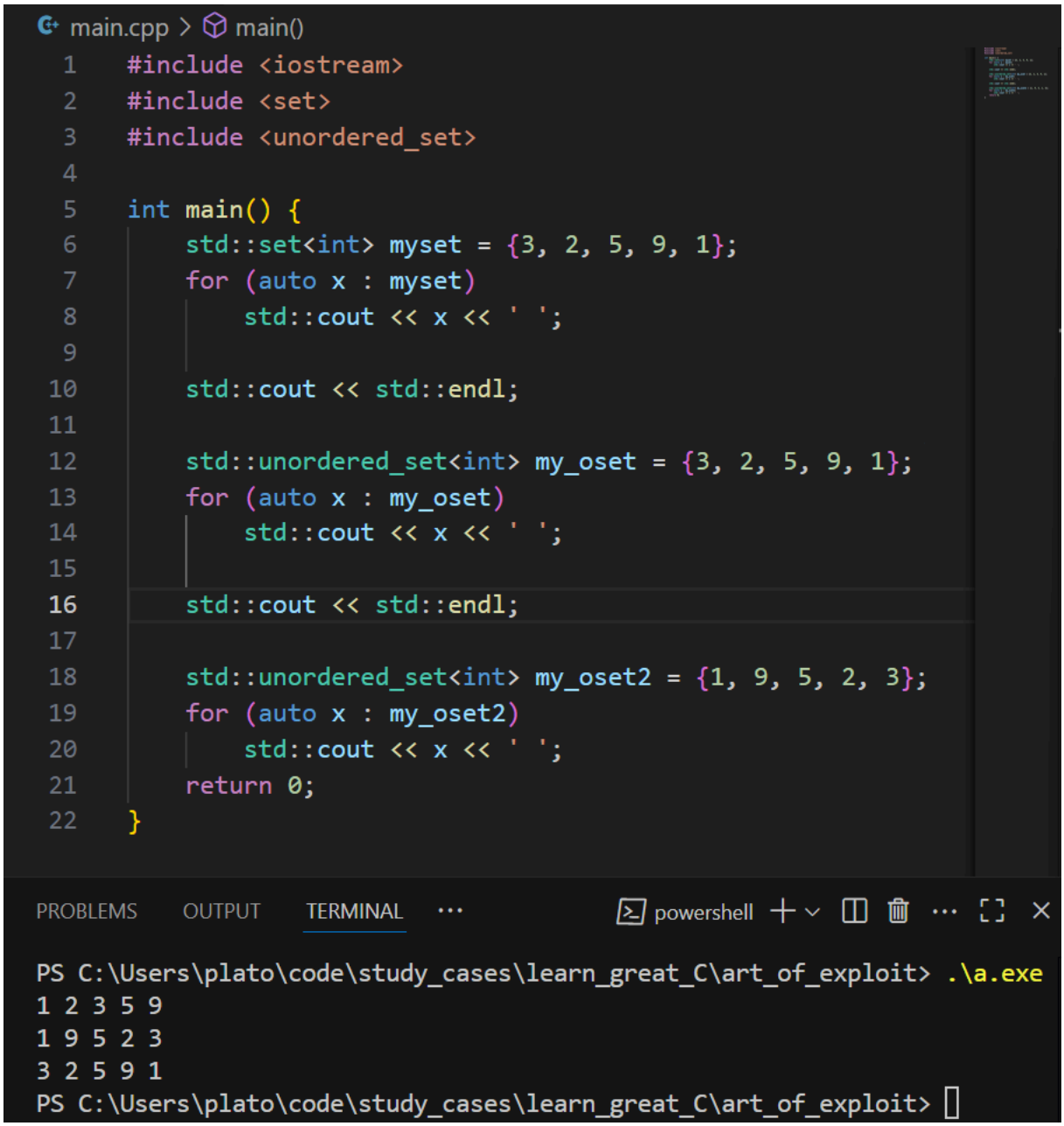

Figure 4: Пример работы set и unordered set

Multisets

Да, есть еще и мультимножества и они тоже либо упорядочены либо нет, отличие от множеств в том, что они допускают повторения.

```
multiset<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 3
```

Функция erase удаляет все копии значения из мультимножества.

```
s.erase(5);
cout << s.count(5) << "\n"; // 0
```

Если требуется удалить только одно значение, то можно поступить так:

s.erase(s.find(5)); cout << s.count(5) << “n”;

Но count и erase работают за O(k), где *k* количество подсчитываемых или удаляемых элементов

Maps

А вот и наш любимчик map(отображения), идея проста, есть ключ, тогда есть и значение(как логин и пароль, или как индекс и значение по данному индексу).

Но в map ключом может быть и строка, и число, и любой другой тип, при этом необязательно последовательный.

Устройство Отображений таково:
1. У нас есть Хэш-функция
2. Передав в Хэш-функцию ключ, мы получаем число, взяв остаток от числа(некоторого n, это либо capacity, либо size) получим номер бакета (ячейки памяти, куда в дальнейшем и будем складывать наши ключи со значениями, если бакет переполнится, то произойдет рехэширование)

В STL имеется две структуры как и в случае со множествами:
- map, основан на сбалансированном двоичном дереве со временем доступа O(log n).
- unordered_map, в основе которого лежит техника хэширования со средним временем доступа к элементам O(1).

Приступим к примерам:

```
map<string, int> m;
m["monkey"] = 4;
m["banana"] = 3;
m["harpsichord"] = 9;
cout << m["banana"] << "\n"; // 3
```

Если в map такого ключа нет, то он создастся и в него запишется заданное значение.

При простом обращении к элементу по несуществующему ключу, помимо создания ключа, будет присвоено значение по умолчанию. Например, в следующем коде в отображение добавляется ключ “aybabtu” со значением 0.

```
map<string, int> m;
cout << m["aybabtu"] << "\n"; // 0
```

)

Функция count проверяет, существует ли ключ в отображении:

```
if (m.count("aybabtu")) {
  // работаем с ключом
}
```

А так можно напечатать все ключи и значения отображения:

```
for (auto x : m) {
  cout << x.first << " " << x.second << "\n";
}
```

Priority Queue

Очередь с приоритетом это мультимножество(или множество), которое поддерживает вставку, а также извлечение и удаление минимального или максимального элемента(в зависимости от очереди). Вставка и удаление занимают O(log n), а извлечение O(1).

Очередь с приоритетом обычно основана на heap structure, а не сбалансированном двоичном дереве, что в разы проще.

По умолчанию, элементы в C++ PQ отсортированы по убыванию, и все же это реально найти и удалить наибольший элемент в очереди. Код ниже это демонстрирует:

```
priority_queue<int> q;
q.push(3);
q.push(5);
q.push(7);
q.push(2);
cout << q.top() << "\n"; // 7
q.pop();
cout << q.top() << "\n"; // 5
q.pop();;
q.push(6);
cout << q.top() << "\n"; // 6
q.pop();
```

Для того, чтобы PQ поддерживал поиск и удаление наименьшего элемента, можно сделать следующее:

```
priority_queue<int, vector<int>, greater<int>> q;
```

Algorithms

Sorting Algorithms

Нечего обсуждать, есть массив - надо отсортировать...

Bubble Sort

Пузырьковая сортировка это самый простой и популярный сортировочный алгоритм, который работает за $O(n^2)$. Логика проста: у нас есть n раундов и на каждом из них алгоритм сравнивает два последовательных элемента, если они в неправильном порядке, то их меняют местами.

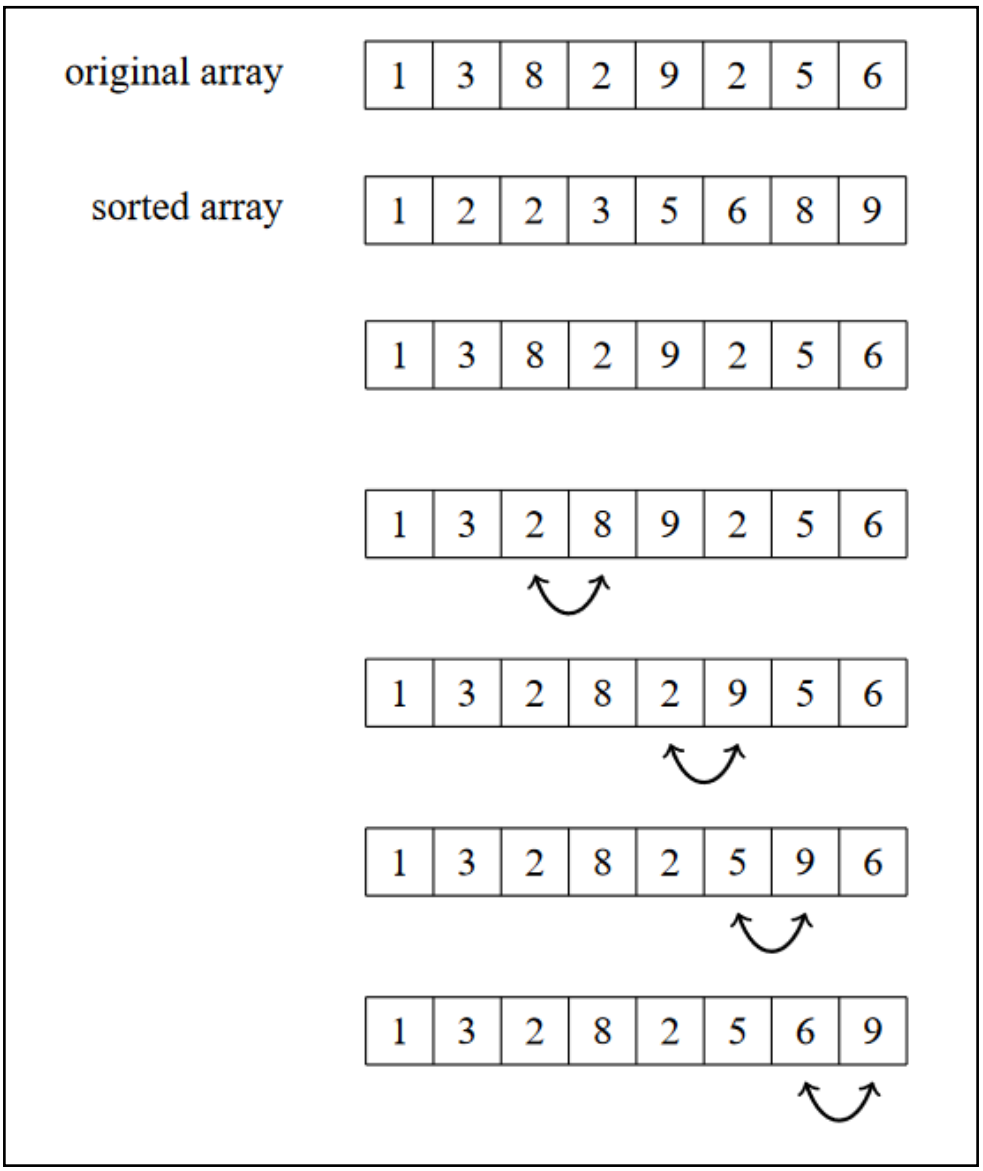

Figure 5: Схема работы Bubble Sort

Реализация

```
for (int i=0; i<n; ++i) {
  for (int j=0; j<n; ++j) {
    // по возрастанию
    if (array[j] > array[j+1]) {
      swap(array[j], array[j+1]);
    }
  }
}
```

Таким образом после первой операции самый большой элемент окажется на правильном месте, в общем говоря, после k раундов, k-наибольших элементов будут на правильных местах. Так что после n операций весь массив будет отсортирован.

Пара слов про анализ сложности

Также для анализа сложности алгоритма сортировки удобно использовать *инверсии*

$$(a, b) : a < b \text{ и } \text{array[a]} > \text{array[b]}$$

То есть такие пары чисел (вообще-то это транспозиции), у которых индексы идут по возрастанию, а сами числа не соответствуют возрастанию.

0	1	2	3	4	5	6	7
1	2	2	6	3	5	9	8

инверсии: (3, 4), (3, 5) и (6, 7)

Число инверсий показывает сколько работы нужно для сортировки массива, то есть массив отсортирован тогда, когда инверсий не осталось. С другой стороны массив чисел по убыванию содержит максимальное количество инверсий:

$$1 + 2 + ... + (n - 1) = \frac{n(n-1)}{2} = O(n^2),$$

Очевидно, что swap от последовательных элементов неправильного порядка удаляет только 1 инверсию, поэтому остальные инверсии остаются, а сложность алгоритма все также $O(n^2)$.

Разберем еще один O(n2) алгоритм - сортировка вставкой.

Алгоритм:
1. проходя по массиву начиная со 2 элемента, берем i-й элемент.
2. ищем позицию для вставки и сдвигаем элементы пока она не будет обнаружена.
3. вставляем элемент в подходящую позицию

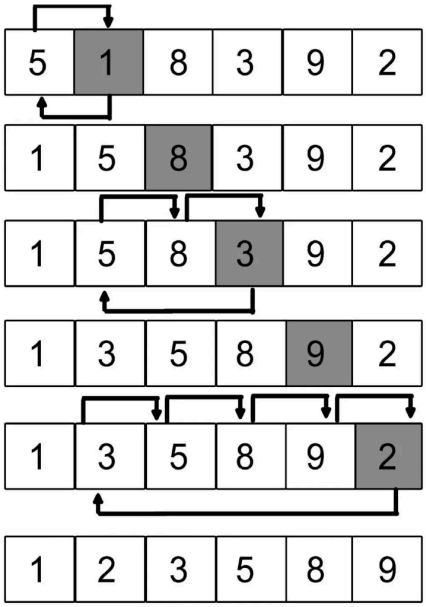

Figure 6: Схема работы Insertion Sort

Реализация:

```
void insertion_sort(int* array, int size) {
  int j, temp;
  for (int i=0; i<size; ++i) {
      temp = array[i];
      j = i-1;
      while(j >= 0 && array[j] > temp) {
          array[j+1] = array[j];
          j--;
      }
      array[j+1] = temp;
  }
}
```

Selection Sort

Сортировка выбором это грубо говоря оптимизированный пузырек. Вместо сравнения 2х соседних элементов по всему массиву, ищется минимальный от i до конечного.

Реализация алгоритма:

1. Запускаем проход по индексам массива
2. Ищем индекс минимального элемента от i до конечного
3. Меняем i-й элемент с минимальным

Таким образом каждый раз поиск минимального элемента будет в уменьшаемом диапазоне.

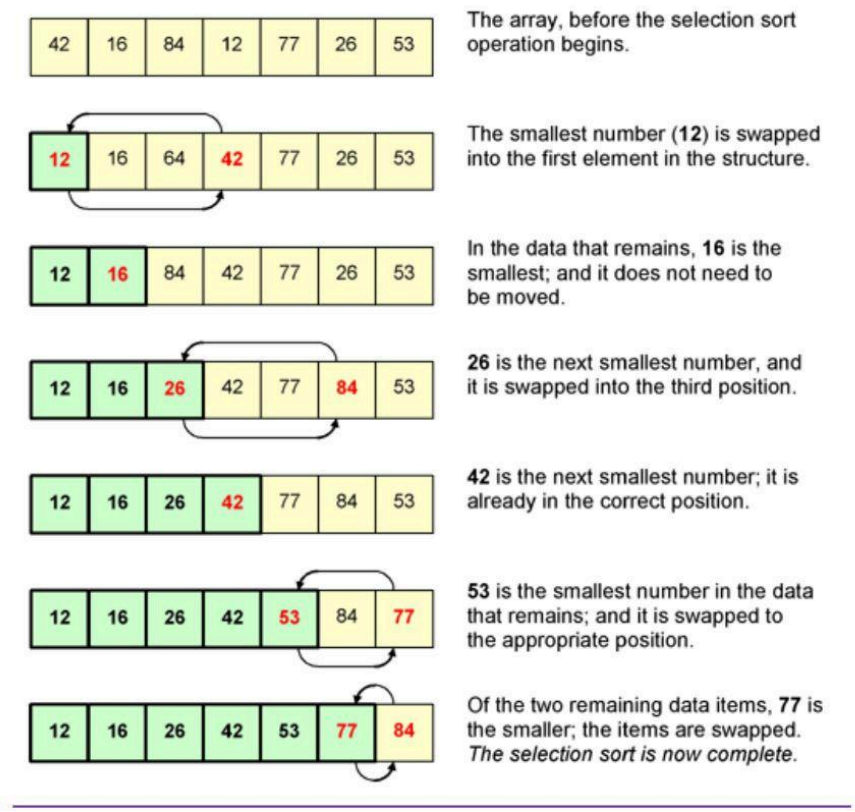

Figure 7: Схема работы Selection Sort

Реализация:

```
void selection_sort(int* array, int size) {
  for (int i=0; i<size; ++i) {
      int min = array[i];
      int ind_min = i;
      for (int j=i+1; j<size; ++j) {
          if (array[j] < min) {
              min = array[j];
              ind_min = j;
          }
      }
      array[ind_min] = array[i];
      array[i] = min;
  }
}
```

Временная сложность: $O(n^2)$

Merge Sort

Наконец-то поговорим об алгоритме из категории “Разделяй и Властвуй”. Для создания эффективного алгоритма сортировки нам недостаточно замены соседних элементов в случае bubble sort, здесь то и появляется merge sort с O(n log n) time.

Вкратце алгоритм разделяет массив на подмассивы сортирует их, а затем склеивает их сортируя вновь.

Merge sort sorts a subarray array[a…b] as follows:

1. if $a == b$, skip, cause a subarray with 1 element is already sorted.
2. Calculate the position of the middle element:

$$k = \left\lfloor \frac{a+b}{2} \right\rfloor$$

3. Recursively sort the subarray array[a…k].
4. Recursively sort the subarray array[k+1…b].
5. Merge the sorted subarrays array[a…k] and array[k+1…b] into sorted subarray [a…b].

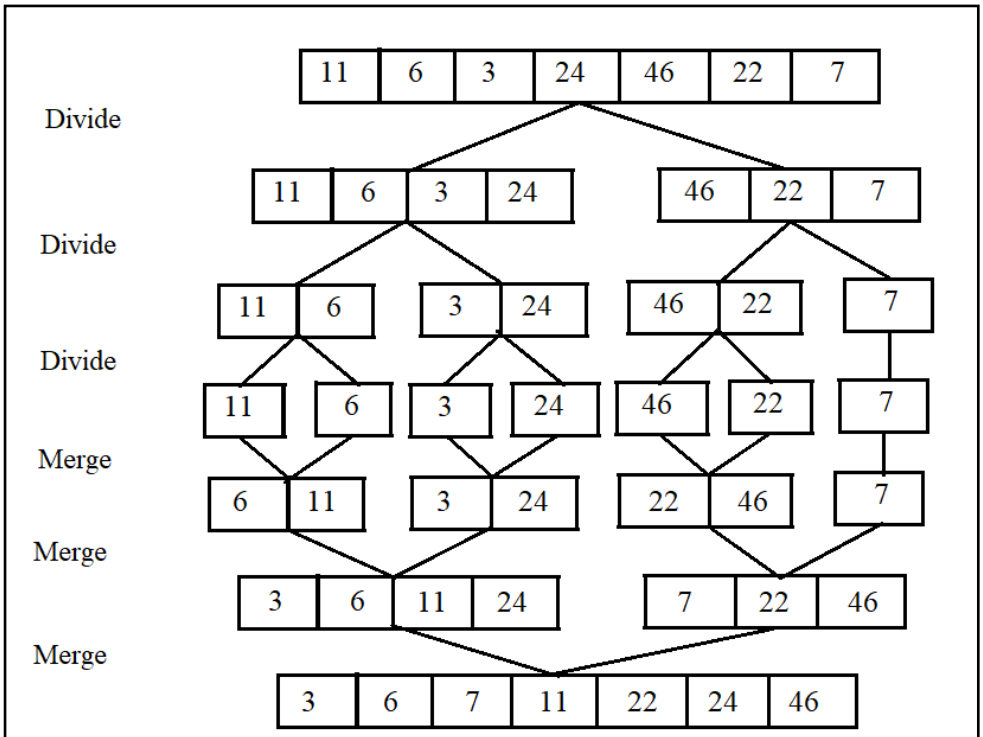

Figure 8: Схема работы Merge Sort

Реализация

```
    // thx too my old friend..
    void merge(int* ar, size_t size) {
    int* res_ar = new int[size];
    int ind_res = 0;
    int middle = size >> 1;
    int left = 0;
    int right = middle;
    // sort array while left and right are in right range
    while (left < middle && right < size) {
        // store the least of subarrays into res
        if (ar[left] <= ar[right])
            res_ar[ind_res++] = ar[left++];
        else
            res_ar[ind_res++] = ar[right++];
    }
    // if numbers remain, then sort and store them too
    while (left < middle)
        res_ar[ind_res++] = ar[left++];
    while (right < size)
        res_ar[ind_res++] = ar[right++];
    // rewrite sorted array into primary
    for (size_t i=0; i<size; ++i) {
        ar[i] = res_ar[i];
    }
    delete [] res_ar;
}
void merge_sort(int* ar, size_t size) {
    if (size <= 1)
        return;
    // calculate middle (w/ bit shift)
    int middle = size >> 1;
    // dividing array into 2 subarrays
    merge_sort(&ar[0], middle);
    merge_sort(&ar[middle], size-middle);
    // merge them
    merge(ar, size);
  }
```

Временная сложность: $O(n \log n) = O(n)O(\log n)$

Since there are O(log n) recursive levels, and processing each level takes a total of O(n) time, the algorithm works in O(n log n) time.

Sorting Lower Bound

Реально ли достичь сортировки быстрее чем $O(n \log n)$? Это невозможно, если сортировка основана на сравнении элементов.

Нижняя граница временной сложности может быть доказана рассмотрением сортировки как процесса, где каждое сравнение двух элементов дает больше информации о содержимом массива:

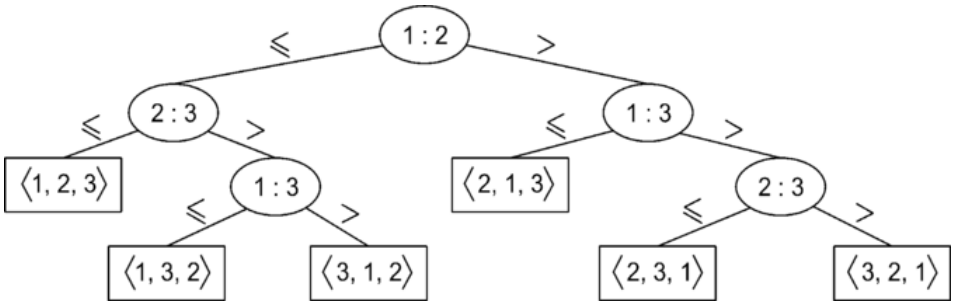

Figure 9: Схема работы сортировок на основе сравнения

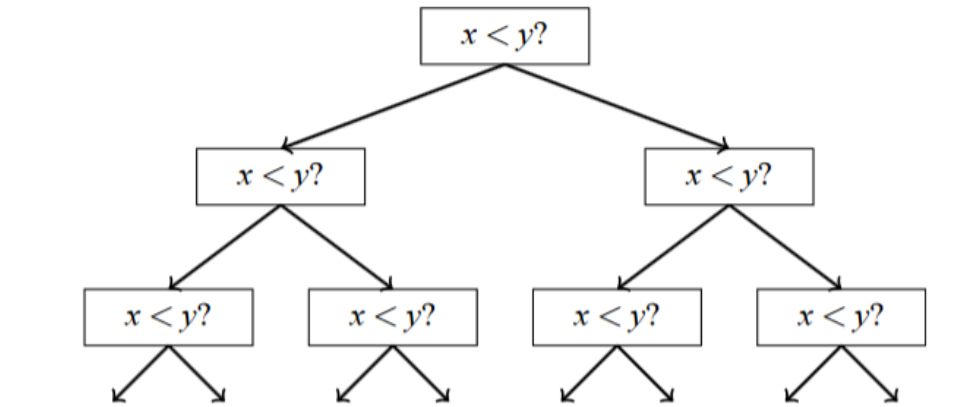

Figure 10: Общая схема работы сортировок на основе сравнения

“$x < y$?” означает сравнение некоторых элементов x и y, верное утверждение отправляет в левую ветку, иначе в правую.

Значит у каждого узла именно 2 потомка(тоже узла). Т.к. на вход мы получаем массив из n элементов, то количество возможных массивов равно кол-ву его различных перестановок - n!, а тогда количество листьев дерева не менее n!

(в противном случае некоторые перестановки были бы не достижимы из корня, а значит, алгоритм неправильно работал бы на некоторых исходных данных).

Т.к двоичное дерево высоты(глубины) h имеет не более 2^h листьев имеем:

$$n! \leq l \leq 2^h,$$

где l число листьев. Тогда получим, что высота дерева не менее:

$$\log_2(n!) = \log_2(1) + \log_2(2) + ... + \log_2(n).$$

Несложно доказать оценку:

$$\log_2(n!) \geq \left(\frac{n}{2}\right) *$$

И наконец получаем:

$$\frac{n}{2} * \log_2\left(\frac{n}{2}\right) = \frac{n}{2}(\log_2(n) - 1) = \Omega(n \log(n))$$

Доп. источник

Counting Sort

Но нижняя оценка временной сложности не подходит для сортировок, не основанных на сравнениях. Сортировка подсчетом основана на подсчете элементов и работает за O(m), где m - диапазон чисел.

Рассмотрим первое приближение алгоритма, когда числа строго неотрицательные.

Логика алгоритма:

1. инициализируется массив с max+1 количеством нулей, где max максимальный элемент массива.
2. затем начинается проход по исходному массиву с подсчетом включений:

counting_array[array[i]] += 1;

3. Затем сортируем исходный массив:

array[index++] = i, повторяя counting_array[array[i]] раз, чтобы учесть повторения.

Реализация:

```
void counting_sort(int* array, size_t size) {
  int max = array[get_ind_max(array, size)];
  int counting_array[max+1] = {0};
  for (int i=0; i<size; ++i) {
      counting_array[array[i]] += 1;
  }
  int index = 0;
  for (int i=0; i < max; ++i) {
      for (int j=0; j<counting_array[i]; ++j) {
          array[index++] = i;
      }
  }
}
```

Но можно улучшить алгоритм, чтобы он учитывал и отрицательные числа:

```
void counting_sort2(int* array, size_t size) {
  int max = array[get_ind_max(array, size)];
  int min = array[get_ind_min(array, size)];
  int counting_array[max-min+1] = {0};
  for (int i=0; i<size; ++i) {
      counting_array[array[i]-min] += 1;
  }
  int index = 0;
  for (int i=min; i < max; ++i) {
      for (int j=0; j<counting_array[i-min]; ++j) {
          array[index++] = i;
      }
  }
}
```

Теоретически можно сделать алгоритм рабочим не только для целых чисел, но для этого нужно использовать hashmap.(но это кал какой-то)