Алгоритмы.

План занятия:  
1. Что такое сортировка и зачем она нужна?   
2. Популярные алгоритмы сортировки   
3. Компаратор  
4. Перегрузка операторов сравнения и метод sort  
5. Метод swap  
6. Что такое поиск и зачем он нужен?   
7. Популярные алгоритмы поиска   
8. Алгоритмическая сложность

Отличная статья, в которой есть красивые анимации с шариками для разных сортировок: <https://tproger.ru/digest/sorting-algorithms-visualized/>

Оглавление

[Введение и польза алгоритмов 1](#_Toc127177013)

[Что такое сортировка и зачем она нужна? 3](#_Toc127177014)

[Популярные алгоритмы сортировки 3](#_Toc127177015)

[Пузырьковая сортировка (Bubble sort) 4](#_Toc127177016)

[Сортировка выбором (Selection sort) 6](#_Toc127177017)

[Сортировка вставками (Insertion sort) 9](#_Toc127177018)

[Быстрая сортировка (Quick sort) 13](#_Toc127177019)

[Сортировка слиянием (Merge sort) 16](#_Toc127177020)

[Сортировка Шелла (Shell sort) 18](#_Toc127177021)

[Сортировка кучей(пирамидальная) (Heap sort) 21](#_Toc127177022)

[Компаратор 26](#_Toc127177023)

[Перегрузка операторов сравнения и метод sort 26](#_Toc127177024)

[Метод swap 28](#_Toc127177025)

[Что такое поиск и зачем он нужен? 29](#_Toc127177026)

[Алгоритмическая сложность 30](#_Toc127177027)

# Введение и польза алгоритмов

**Алгоритм** – это совокупность четко сформулированных инструкций, которые решают точно обозначенную задачу, если им следовать. Здесь решение задачи подразумевает преобразование входных данных в желаемый конечный результат.

Многие умные алгоритмы, такие как растеризация, отсечение, back-face culling и сглаживание, служат для создания потрясающих 3D-миров в реальном времени.   
Алгоритмы позволяют найти кратчайший маршрут к месту назначения в городе, содержащем тысячи улиц.   
Благодаря алгоритмам машинного обучения произошла революция в таких областях, как распознавание изображений и речи.   
Перечисленные алгоритмы – это только вершина айсберга, и изучение этого предмета откроет множество дверей в карьере программиста.

Стандартная библиотека **<algorithms>** содержит большое количество алгоритмов для работы с контейнерами стандартной библиотеки. Доступные инструменты покрывают значительную часть встречающихся алгоритмических задач.

Использование стандартных алгоритмов вместо их самостоятельной реализации является хорошим стилем программирования по следующим причинам:

* Экономия времени. Мы не тратим время на реализацию и отладку алгоритма.
* Гарантия отсутствия ошибок в логике работы алгоритма.

***iota, for\_each и transform***

Большое количество циклов for в коде, который выполняет манипуляции со структурами данных, обычно говорит о недостаточном использовании стандартных алгоритмов. Так, если необходимо применить некоторую функцию ко всем элементам контейнера, то можно рассмотреть использование алгоритма **for\_each.**

*Решим следующую задачу: вывести в стандартный поток квадраты натуральных чисел от 1 до 100. Следующий код показывает, что эта задача может быть решена в четырех строчках кода и без явного использования циклов:*

Способ из модуля 1:

#include <iostream>

using namespace std;

int main()

{

int n, i;

n = 100;

for (i = 1; i < n; i++)

{

cout << i \* i << endl;

}

}

Способ с использованием алгоритмов:

#include <vector>

#include <iostream>

#include <numeric> // iota

#include <algorithm> // for\_each

using namespace std;

int main() {

vector<int> v(100);

iota(v.begin(), v.end(), 1); // v = [1, 2, 3, ..., 100]

for\_each(v.begin(), v.end(), [](int& a) {a = a \* a; }); // v = [1, 4, 9, ..., 10000]

for\_each(v.begin(), v.end(), [](int a) {cout << a << ' '; });

return 0;

}

* Мы воспользовались алгоритмом **iota** из библиотеки **<numeric>**, чтобы проинициализировать массив набором последовательных целых чисел.
* Затем мы два раза использовали алгоритм for\_each: для вычисления квадратов и для вывода значений в стандартный поток.
* Третьим аргументом алгоритм for\_each принимает функцию одного аргумента. Тип аргумента должен соответствовать типу элементов контейнера. Вместо обычной функции бывает удобно передать лямбда-выражение, что мы и сделали оба раза в этом примере. Лямбда-выражение позволяет определить функцию в месте ее использования.   
  Квадратные скобки [] указывают на начало лямбда-выражения; в круглых скобках указываются аргументы выражения; в фигурных скобках содержится тело лямбда-выражения.

Что такое сортировка и зачем она нужна?  
  
**Сортировка** – это задача упорядочивания последовательности чисел в порядке возрастания. Более формально мы можем определить эту проблему в виде входных и выходных данных: входными данными является любая последовательность чисел, а выходными – перестановка входной последовательности, при которой меньшие элементы располагаются перед большими. Алгоритм можно считать правильным, если для каждого из возможных входных данных он дает желаемый результат. Алгоритм должен быть однозначным, то есть инструкции должны быть понятными и иметь только единственное толкование. Алгоритм также не должен выполняться бесконечно и завершаться после конечного числа действий.

# Популярные алгоритмы сортировки

* Пузырьковая сортировка (Bubble sort)
* Сортировка выбором (Selection sort)
* Сортировка вставками (Insertion sort)
* Быстрая сортировка (Quick sort)
* Сортировка слиянием (Merge sort)
* Сортировка Шелла (Shell sort)
* Сортировка кучей(пирамидальная) (Heap sort)

## Пузырьковая сортировка (Bubble sort)

template <class Item>

void bubble(Item a[], int l, int r)

{

for (int i = l; i < r; i++)

for (int j = r; j > i; j--)

compexch(a[j - 1], a[j]);

}

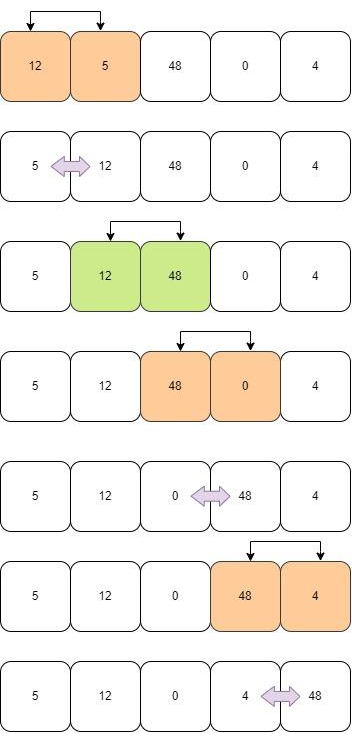
В пузырьковой сортировке каждый элемент сравнивается со следующим. Если два таких элемента не стоят в нужном порядке, то они меняются между собой местами. В конце каждой итерации (далее называем их проходами) наибольший/наименьший элемент ставится в конец списка.

Прежде чем писать код, разберем сортировку визуально на примере массива из пяти элементов. Отсортируем его в порядке возрастания.



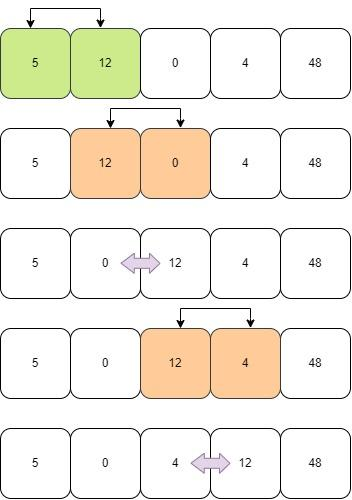
**Проход №1**

Оранжевым отмечаются элементы, которые нужно поменять местами. Зеленые уже стоят в нужном порядке.



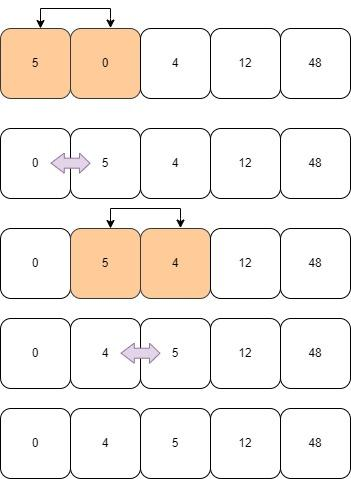
Наибольший элемент — число 48 — оказался в конце списка.

**Проход №2**

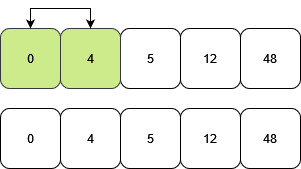


Наибольший элемент уже занимает место в конце массива. Чтобы поставить следующее число по убыванию, можно пройтись лишь до 4-й позиции, а не пятой.

**Проход №3**



**Проход №4**



После четвертого прохода получаем отсортированный массив.

Функция сортировки в качестве параметров будет принимать указатель на массив и его размер. Функцией swap() элементы меняются местами друг с другом:

#include <iostream>

using namespace std;

void bubbleSort(int list[], int listLength)

{

while (listLength--)

{

bool swapped = false;

for (int i = 0; i < listLength; i++)

{

if (list[i] > list[i + 1])

{

swap(list[i], list[i + 1]);

swapped = true;

}

}

if (swapped == false)

break;

}

}

int main()

{

int list[5] = { 3,19,8,0,48 };

cout << "Input array ..." << endl;

for (int i = 0; i < 5; i++)

cout << list[i] << '\t';

cout << endl;

bubbleSort(list, 5);

cout << "Sorted array ..." << endl;

for (int i = 0; i < 5; i++)

cout << list[i] << '\t';

cout << endl;

}

## Сортировка выбором (Selection sort)

Для каждого i от l до r-1 элемент a[i] меняется местами с минимальным элементом из a[i], ..., a[r]. По мере продвижения индекса i слева направо элементы слева от него занимают свои окончательные позиции в массиве (и больше не перемещаются), поэтому, когда i достигнет правого конца, массив будет полностью отсортирован.

template <class Item>

void selection(Item a[], int l, int r)

{

for (int i = l; i < r; i++)

{

int min = i;

for (int j = i + 1; j <= r; j++)

if (a[j] < a[min]) min = j;

exch(a[i], a[min]);

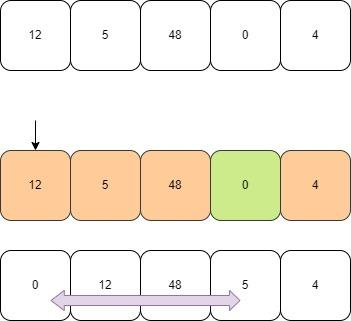
}

}

Ищем наименьшее значение в массиве и ставим его на позицию, откуда начали проход. Потом двигаемся на следующую позицию.

Возьмем тот же массив из пяти элементов и отсортируем его.

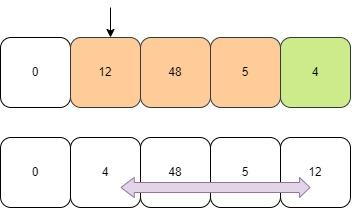
**Проход №1**



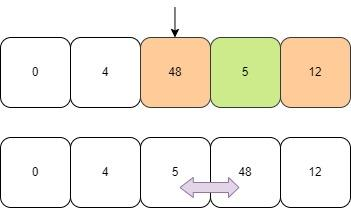
Зеленым отмечается наименьший элемент в подмассиве — он ставится в начало списка.

**Проход №2**

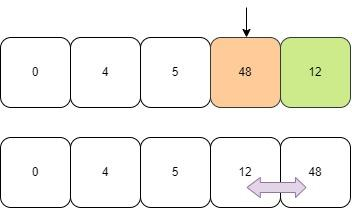
Число 4 — наименьшее в оставшейся части массива. Перемещаем четверку на вторую позицию после числа 0.



**Проход №3**



**Проход №4**



Напишем функцию поиска наименьшего элемента и используем ее в сортировке:

#include <iostream>

using namespace std;

int findSmallestPosition(int list[], int startPosition, int listLength)

{

int smallestPosition = startPosition;

for (int i = startPosition; i < listLength; i++)

{

if (list[i] < list[smallestPosition])

smallestPosition = i;

}

return smallestPosition;

}

void selectionSort(int list[], int listLength)

{

for (int i = 0; i < listLength; i++)

{

int smallestPosition = findSmallestPosition(list, i, listLength);

swap(list[i], list[smallestPosition]);

}

return;

}

int main()

{

int list[5] = { 12, 5, 48, 0, 4 };

cout << "Input array ..." << endl;

for (int i = 0; i < 5; i++)

cout << list[i] << '\t';

cout << endl;

selectionSort(list, 5);

cout << "Sorted array ..." << endl;

for (int i = 0; i < 5; i++)

cout << list[i] << '\t';

cout << endl;

}

## Сортировка вставками (Insertion sort)

В отличие от сортировки выбором, время выполнения сортировки вставками зависит главным образом от исходного порядка ключей во входных данных. Например, если файл большой, а ключи уже упорядочены (или почти упорядочены), то сортировка вставками выполняется быстро, а сортировка выбором — медленно.

template <class Item>

void insertion(Item a[], int l, int r)

{

int i;

for (i = r; i > l; i--) compexch(a[i - 1], a[i]);

for (i = l + 2; i <= r; i++)

{

int j = i; Item v = a[i];

while (v < a[j - 1])

{

a[j] = a[j - 1]; j--;

}

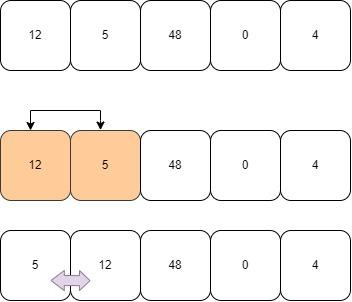
a[j] = v;

}

}

В сортировке вставками начинаем со второго элемента. Проверяем между собой второй элемент с первым и, если надо, меняем местами. Сравниваем следующую пару элементов и проверяем все пары до нее.

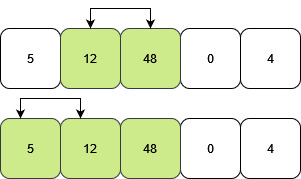
**Проход №1. Начинаем со второй позиции.**



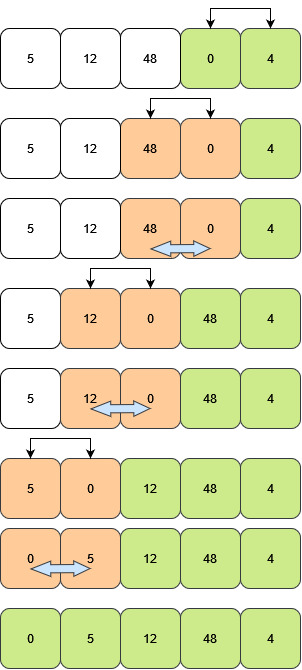
Число 12 больше 5 — элементы меняются местами.

**Проход №2. Начинаем с третьей позиции.**

Проверяем вторую и третью позиции. Затем первую и вторую.

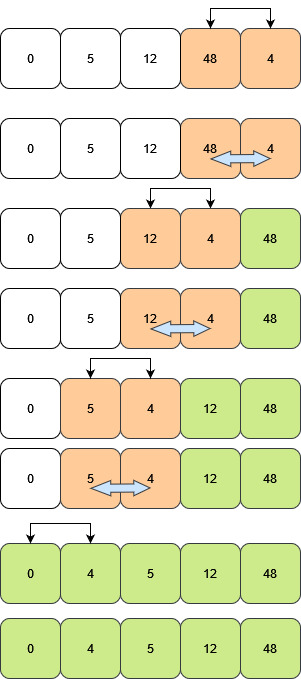


**Проход №3. Начинаем с четвертой позиции.**



Произошло три смены местами.

**Проход №4. Начинаем с последней позиции.**



#include <iostream>

using namespace std;

void insertionSort(int list[], int listLength)

{

for (int i = 1; i < listLength; i++)

{

int j = i - 1;

while (j >= 0 && list[j] > list[j + 1])

{

swap(list[j], list[j + 1]);

cout << "\ndid";

j--;

}

}

}

int main()

{

int list[8] = { 3,19,8,0,48,4,5,12 };

cout << "Input array ...\n";

for (int i = 0; i < 8; i++)

{

cout << list[i] << "\t";

}

insertionSort(list, 8);

cout << "\n\nSorted array ... \n";

for (int i = 0; i < 8; i++)

{

cout << list[i] << "\t";

}

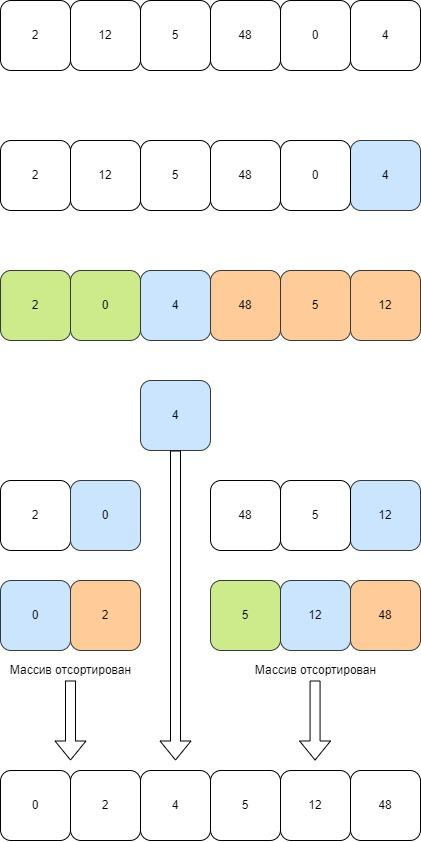
return 0;

}

## Быстрая сортировка (Quick sort)

В основе быстрой сортировки лежит стратегия «разделяй и властвуй». Задача разделяется на более мелкие подзадачи. Подзадачи решаются отдельно, а потом решения объединяют. Точно так же, массив разделяется на подмассивы, которые сортируются и затем сливаются в один.

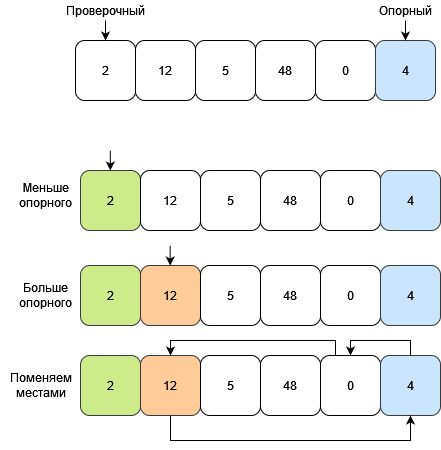
В первую очередь выбираем опорный элемент. Отметим его синим. Все значения больше опорного элемента ставятся после него, остальные — перед.

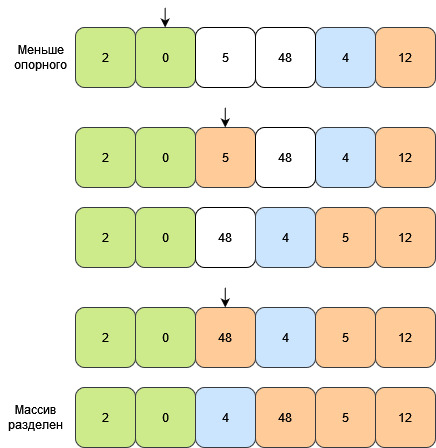


На иллюстрации массив разделяется по опорному элементу. В полученных массивах также выбираем опорный элемент и разделяем по нему.

Опорным может быть любой элемент. Мы выбираем последний в списке.

Чтобы расположить элементы большие — справа от опорного элемента, а меньшие — слева, будем двигаться от начала списка. Если число будет больше опорного, то оно ставится на его место, а сам опорный на место перед ним.





Напишем функцию разделения **partition(),** которая возвращает индекс опорного элемента, и используем ее в сортировке.

#include <iostream>

using namespace std;

int partition(int list[], int start, int pivot)

{

int i = start;

while (i < pivot)

{

if (list[i] > list[pivot] && i == pivot - 1)

{

swap(list[i], list[pivot]);

pivot--;

}

else if (list[i] > list[pivot])

{

swap(list[pivot - 1], list[pivot]);

swap(list[i], list[pivot]);

pivot--;

}

else i++;

}

return pivot;

}

void quickSort(int list[], int start, int end)

{

if (start < end)

{

int pivot = partition(list, start, end);

quickSort(list, start, pivot - 1);

quickSort(list, pivot + 1, end);

}

}

int main()

{

int list[6] = { 2, 12, 5, 48, 0, 4 };

cout << "Input array ...\n";

for (int i = 0; i < 6; i++)

{

cout << list[i] << "\t";

}

quickSort(list, 0, 6);

cout << "\n\nSorted array ... \n";

for (int i = 0; i < 6; i++)

{

cout << list[i] << "\t";

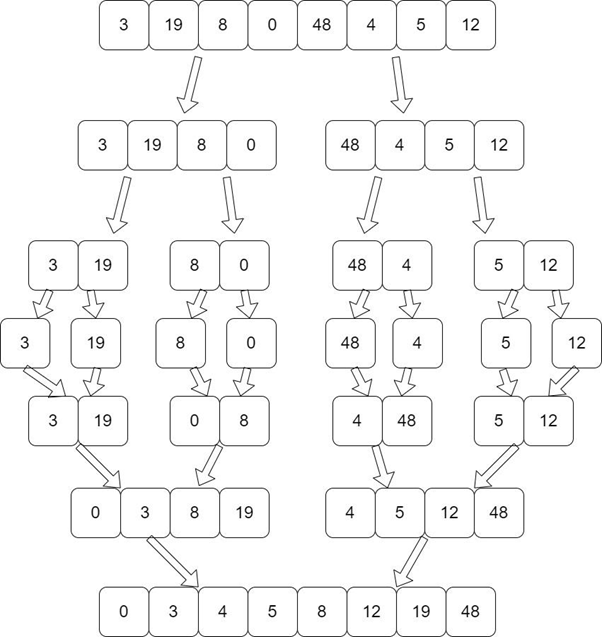
}

return 0;

}

## Сортировка слиянием (Merge sort)

Сортировка слиянием также следует стратегии «разделяй и властвуй». Разделяем исходный массив на два равных подмассива. Повторяем сортировку слиянием для этих двух подмассивов и объединяем обратно.



Цикл деления повторяется, пока не останется по одному элементу в массиве. Затем объединяем, пока не образуем полный список.

Алгоритм сортировки состоит из четырех этапов:

1. Найти середину массива.
2. Сортировать массив от начала до середины.
3. Сортировать массив от середины до конца.
4. Объединить массив.

* Для объединения напишем отдельную функцию merge().
* Алгоритм объединения массивов:
* Циклично проходим по двум массивам..
* В объединяемый ставим тот элемент, что меньше.
* Двигаемся дальше, пока не дойдем до конца обоих массивов.

#include <iostream>

using namespace std;

void merge(int list[], int start, int end, int mid);

void mergeSort(int list[], int start, int end)

{

int mid;

if (start < end) {

mid = (start + end) / 2;

mergeSort(list, start, mid);

mergeSort(list, mid + 1, end);

merge(list, start, end, mid);

}

}

void merge(int list[], int start, int end, int mid)

{

int mergedList[8];

int i, j, k;

i = start;

k = start;

j = mid + 1;

while (i <= mid && j <= end) {

if (list[i] < list[j]) {

mergedList[k] = list[i];

k++;

i++;

}

else {

mergedList[k] = list[j];

k++;

j++;

}

}

while (i <= mid) {

mergedList[k] = list[i];

k++;

i++;

}

while (j <= end) {

mergedList[k] = list[j];

k++;

j++;

}

for (i = start; i < k; i++) {

list[i] = mergedList[i];

}

}

int main()

{

int list[8] = { 3,19,8,0,48,4,5,12 };

cout << "Input array ...\n";

for (int i = 0; i < 8; i++)

{

cout << list[i] << "\t";

}

mergeSort(list, 0, 7);

cout << "\n\nSorted array ... \n";

for (int i = 0; i < 8; i++)

{

cout << list[i] << "\t";

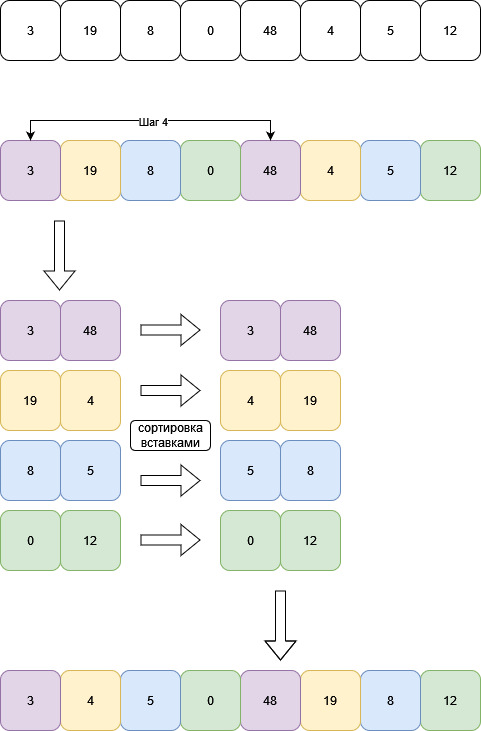
}

}

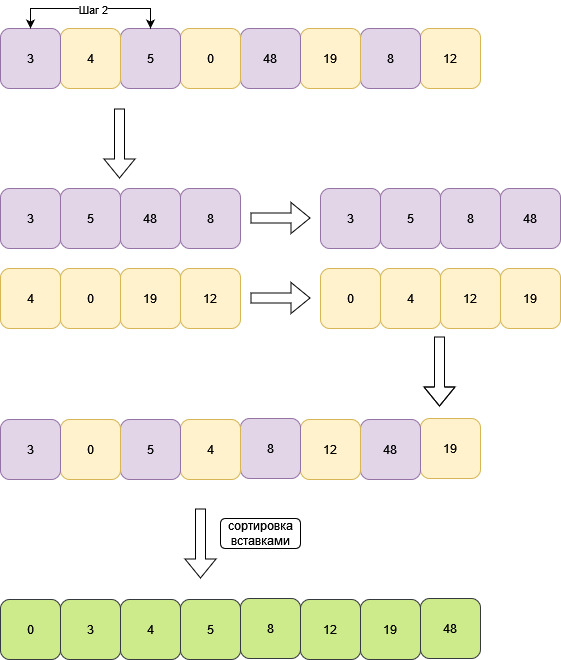
## Сортировка Шелла (Shell sort)

Алгоритм включает в себя сортировку вставками. Исходный массив размером N разбивается на подмассивы с шагомN/2.   
Подмассивы сортируются вставками. Затем вновь разбиваются, но уже с шагом равным N/4.   
Цикл повторяется. Производим целочисленное деление шага на два каждую итерацию. Когда шаг становится равен 1, массив просто сортируется вставками.

У массива размером с 8, первый шаг будет равен 4.



Уменьшаем шаг в два раза. Шаг равен 2.



#include <iostream>

using namespace std;

void shellSort(int list[], int listLength)

{

for (int step = listLength / 2; step > 0; step /= 2)

{

for (int i = step; i < listLength; i += 1)

{

int j = i;

while (j >= step && list[j - step] > list[i])

{

swap(list[j], list[j - step]);

j -= step;

cout << "\ndid";

}

}

}

}

int main()

{

int list[8] = { 3,19,8,0,48,4,5,12 };

cout << "Input array ...\n";

for (int i = 0; i < 8; i++)

{

cout << list[i] << "\t";

}

shellSort(list, 8);

cout << "\n\nSorted array ... \n";

for (int i = 0; i < 8; i++)

{

cout << list[i] << "\t";

}

}

## Сортировка кучей(пирамидальная) (Heap sort)

Исходный массив представляем в виде структуры данных **кучи**. Куча – это один из типов бинарного дерева.

У кучи есть следующие свойства:

* Родительский узел всегда больше дочерних;
* На i-ом слое 2i узлов, начиная с нуля. То есть на нулевом слое 1 узел, на первом – 2 узла, на втором – 4, и т. д. Правило для всех слоев, кроме последнего;
* Слои заполняются слева направо.

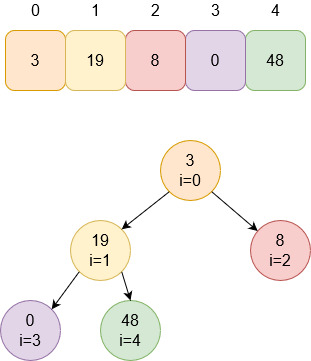
После формирования кучи будем извлекать самый старший узел и ставить на конец массива.

Алгоритм сортировки кучей:

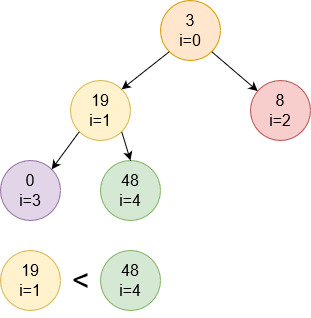
1. Формируем бинарное дерево из массива.
2. Расставляем узлы в дереве так, чтобы получилась куча (метод heapify()).
3. Верхний элемент помещаем в конец массива.
4. Возвращаемся на шаг 2, пока куча не опустеет.

Обращаться к дочерним узлам можно, зная, что дочерние элементы i-го элемента находятся на позициях**2\*i + 1** (левый узел) и **2\*i + 2** (правый узел).

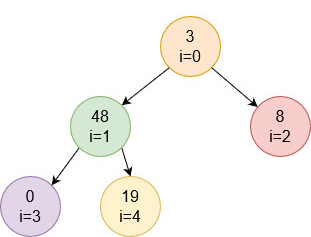
Изначальная куча:



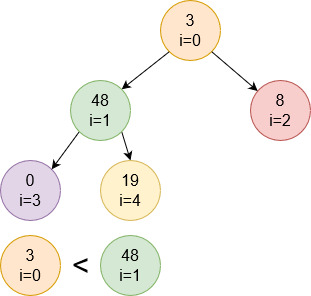
Индекс с нижним левым узлом определим по формуле n/2-1, где n – длина массива. Получается 5/2 – 1 = 2 – 1 = 1. С этого индекса и начинаем операцию heapify(). Сравним дочерние узлы 1-й позиции.



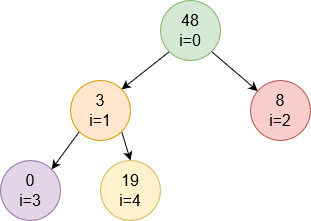
Дочерний узел оказался больше. Меняем местами с родителем.



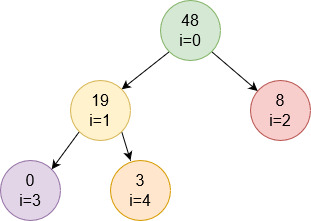
Теперь проверяем родительский узел от позиции 1.



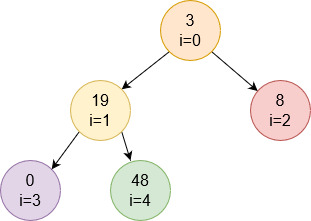
48 больше 3. Меняем местами.



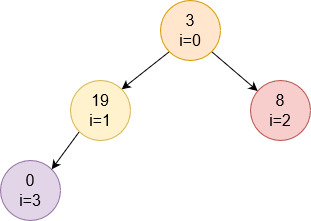
После смены проверяем все дочерние узлы элемента, который опустили. То есть для числа 3 проводим heapify(). Так как 3 меньше 19, меняем местами.



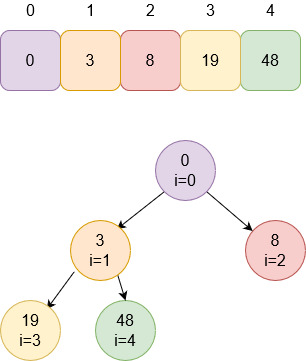
Наибольший элемент оказался наверху кучи. Осталось поставить его в конце массива на позицию 4.



Теперь продолжаем сортировать кучу, но последний элемент игнорируем. Для этого просто будем считать, что длина массива уменьшилась на 1.



Повторяем алгоритм сортировки, пока куча не опустеет, и получаем отсортированный массив.



#include <iostream>

using namespace std;

void heapify(int list[], int listLength, int root)

{

int largest = root;

int l = 2 \* root + 1;

int r = 2 \* root + 2;

if (l < listLength && list[l] > list[largest])

largest = l;

if (r < listLength && list[r] > list[largest])

largest = r;

if (largest != root)

{

swap(list[root], list[largest]);

heapify(list, listLength, largest);

}

}

void heapSort(int list[], int listLength)

{

for (int i = listLength / 2 - 1; i >= 0; i--)

heapify(list, listLength, i);

for (int i = listLength - 1; i >= 0; i--)

{

swap(list[0], list[i]);

heapify(list, i, 0);

}

}

int main()

{

int list[5] = { 3,19,8,0,48 };

cout << "Input array ..." << endl;

for (int i = 0; i < 5; i++)

cout << list[i] << '\t';

cout << endl;

heapSort(list, 5);

cout << "Sorted array" << endl;

for (int i = 0; i < 5; i++)

cout << list[i] << '\t';

cout << endl;

}

# Компаратор

***Что такое компаратор***

**Компаратор** — это функция, которая как бы учит сортировать sort. Так, например можно сортировать по:

* Кратности на 3.
* Четности или нечетности.
* Изменить сторону сортировки на — по убыванию.

**sort передает элементы компаратору, а компаратор проверяет их по вашему алгоритму и передает true или false.**

***Как создать компаратор***

Самого начала создаём функцию, которая и будет компаратором.

bool comp(<первый элемент>, <второй элемент>) {

return <первый элемент> <условный оператор> <второй аргумент>;

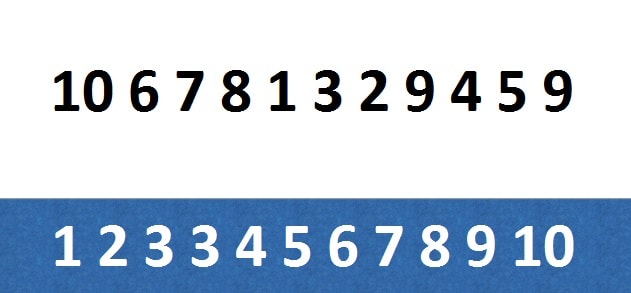
}

* **<первый и второй аргумент>** — здесь нужно быть аккуратным, потому что если мы укажем неправильно аргументы — никакая сортировка не произойдет.
* **return** — эта строчка является основной в этом коде, так как именно она изменяет сортировку.
* В аргументах нужно всего лишь указать имя компаратора.

# Перегрузка операторов сравнения и метод sort

*Что такое функция sort*

Это функция, которая может сортировать указанный контейнер или обычный массив. По умолчанию она сортирует по неубыванию, но это можно изменить путем применения компаратора, об этом поговорим позже.



Принцип работы построен на алгоритме быстрой сортировки (quicksort), так что за быстроту можно не волноваться.

Пример для вектора:

#include <iostream>

#include <vector> // vector

#include <algorithm> // sort

using namespace std;

int main() {

setlocale(0, "");

int n; vector <int> vec;

cout << "Введите количество элементов последовательности: "; cin >> n;

int a;

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

cout << i + 1 << ") "; cin >> a;

vec.push\_back(a);

}

cout << "Вот как выглядит последовательность до: ";

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

cout << vec[i] << " ";

}

sort(vec.begin(), vec.end()); // сортировка

cout << endl << "После сортировки: ";

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

cout << vec[i] << " ";

}

sort(vec.begin() + n / 2, vec.end(), comp);

cout << endl << "А вот еще раз: ";

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

cout << vec[i] << " ";

}

system("pause");

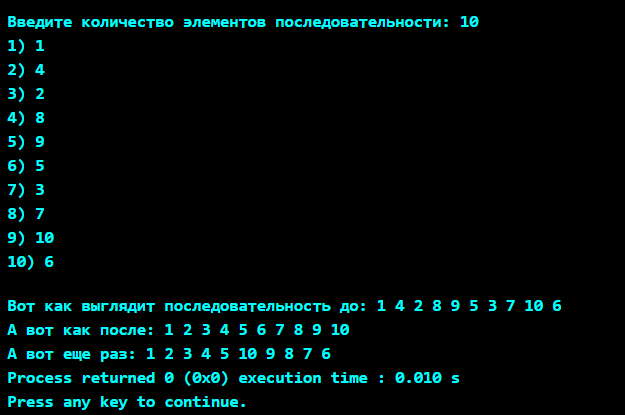
return 0;

}

В строках 14 — 17: добавляем элементы в вектор **vec**.

В строке 25: сортируем последовательность.

В строке 32: нашей стартовой точкой стала **n / 2**, а также мы применили компаратор, из-за которого смогли поменять сторону сортировки (по не возрастанию). **vec.begin() + n / 2** — так прибавлять к итератору можно только для вектора и массива, для других контейнеров нельзя. Подробнее почитайте про итераторы здесь.



# Метод swap

Функция swap() обменивает значения своих аргументов.

#include <iostream>

void swap(int\* a, int\* b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

int main() {

int x1 = 10, x2 = 20;

std::cout << "x1 = " << x1 << ", x2 = " << x2 << std::endl;

swap(&x1, &x2); // Передаём ссылки

std::cout << "x1 = " << x1 << ", x2 = " << x2 << std::endl;

return 0;

}

# Что такое поиск и зачем он нужен?

Алгоритм std::find ищет последовательным перебором первое вхождение элемента, равного заданному. Результатом является итератор, который указывает на найденный элемент. Если же ничего не найдено, то возвращается правый конец переданного полуинтервала.

#include <algorithm>

#include <deque>

#include <iostream>

int main() {

std::deque<int> d = { 3, 14, 15, 92, 6 };

// Такой элемент есть, мы его точно найдём

auto iter1 = std::find(d.begin(), d.end(), 15);

// Итераторы дека можно вычитать, напечатается индекс найденного элемента

std::cout << (iter1 - d.begin()) << "\n";

auto start = d.begin();

// К итераторам дека и вектора можно прибавлять целые числа

auto end = start + 3;

// Полуинтервал [start; end) теперь ограничивает подпоследовательность 3, 14, 15

auto iter2 = std::find(start, end, 19);

if (iter2 == end) {

std::cout << "No such element!\n";

}

else {

std::cout << \*iter2 << "\n";

}

// Напечатает No such element

}

Вариация №2:

template <typename Iter, typename Value>

Iter find(Iter first, Iter last, const Value& value) {

while (first != last) {

if (\*first == value) {

return first;

}

++first;

}

return last;

}

# Алгоритмическая сложность

**Оценка сложности**

Сложность алгоритмов обычно оценивают по времени выполнения или по используемой памяти. В обоих случаях сложность зависит от размеров входных данных: массив из 100 элементов будет обработан быстрее, чем аналогичный из 1000. При этом точное время мало кого интересует: оно зависит от процессора, типа данных, языка программирования и множества других параметров. Важна лишь асимптотическая сложность, т. е. сложность при стремлении размера входных данных к бесконечности.

Допустим, некоторому алгоритму нужно выполнить 4n^3 + 7n условных операций, чтобы обработать n элементов входных данных. При увеличении n на итоговое время работы будет значительно больше влиять возведение n в куб, чем умножение его на 4 или же прибавление 7n. Тогда говорят, что временная сложность этого алгоритма равна О(n3), т. е. зависит от размера входных данных кубически.

Использование заглавной буквы О (или так называемая О-нотация) пришло из математики, где её применяют для сравнения асимптотического поведения функций. Формально O(f(n)) означает, что время работы алгоритма (или объём занимаемой памяти) растёт в зависимости от объёма входных данных не быстрее, чем некоторая константа, умноженная на f(n).

***Примеры***

**O(n) — линейная сложность**

Такой сложностью обладает, например, алгоритм поиска наибольшего элемента в не отсортированном массиве. Нам придётся пройтись по всем n элементам массива, чтобы понять, какой из них максимальный.

**O(log n) — логарифмическая сложность**

Простейший пример — бинарный поиск. Если массив отсортирован, мы можем проверить, есть ли в нём какое-то конкретное значение, методом деления пополам. Проверим средний элемент, если он больше искомого, то отбросим вторую половину массива — там его точно нет. Если же меньше, то наоборот — отбросим начальную половину. И так будем продолжать делить пополам, в итоге проверим log n элементов.

**O(n2) — квадратичная сложность**

Такую сложность имеет, например, алгоритм сортировки вставками. В канонической реализации он представляет из себя два вложенных цикла: один, чтобы проходить по всему массиву, а второй, чтобы находить место очередному элементу в уже отсортированной части. Таким образом, количество операций будет зависеть от размера массива как n \* n, т. е. n2.

Бывают и другие оценки по сложности, но все они основаны на том же принципе.

Также случается, что время работы алгоритма вообще не зависит от размера входных данных. Тогда сложность обозначают как O(1). Например, для определения значения третьего элемента массива не нужно ни запоминать элементы, ни проходить по ним сколько-то раз. Всегда нужно просто дождаться в потоке входных данных третий элемент и это будет результатом, на вычисление которого для любого количества данных нужно одно и то же время.

Аналогично проводят оценку и по памяти, когда это важно. Однако алгоритмы могут использовать значительно больше памяти при увеличении размера входных данных, чем другие, но зато работать быстрее. И наоборот. Это помогает выбирать оптимальные пути решения задач исходя из текущих условий и требований.

