

## Оглавление

Общая часть.....	2
Часть 1 — Алгоритмы сортировки .....	2
Сортировка пузырьком .....	3
Сортировка выбором .....	4
Сортировка простыми вставками.....	4
Сортировка вставками со сторожевым элементом.....	6
Сортировка Шелла.....	6
Задание для алгоритмов сортировки .....	9
Часть 2 — Алгоритмы поиска .....	12
Поиск значения перебором.....	12
Бинарный поиск .....	13
Фибоначчиев поиск .....	14
Интерполяционный поиск .....	15
Задание для алгоритмов поиска .....	16
Индивидуальные задания.....	18
Приложения.....	21
Приложение А. Алгоритмы сортировки.....	21
Приложение Б. Алгоритмы поиска .....	22

# АЛГОРИТМЫ СОРТИРОВКИ И ПОИСКА

**Цель работы:** освоить основные алгоритмы сортировки и поиска данных. Проанализировать быстродействие алгоритмов.

## Общая часть

### Часть 1 — Алгоритмы сортировки

Сортировка — это упорядочивание элементов списков структур данных по какому-либо признаку. Когда речь заходит о числовых массивах, под сортировкой обычно понимается расположение чисел в порядке возрастания или убывания. Если имеются в виду строковые данные, то речь, как правило, идет о сортировке записей в алфавитном порядке. Сортировка относится к важнейшему классу алгоритмов обработки данных и осуществляется большим количеством способов. Все эти способы можно сравнивать между собой, используя различные критерии:

**Временная сложность** (вычислительная сложность) — основной параметр, характеризующий быстродействие алгоритма. Временная сложность обычно оценивается путём подсчёта числа элементарных операций, осуществляемых алгоритмом. Время исполнения одной такой операции при этом берётся константой, то есть асимптотически оценивается как  $O(1)$ . Для сортировки важны худшее, среднее и лучшее поведения алгоритма для массива размера  $n$ . Для типичного алгоритма хорошее поведение — это  $O(n \log n)$  и плохое поведение — это  $O(n^2)$ . Идеальное поведение для сортировки —  $O(n)$ .

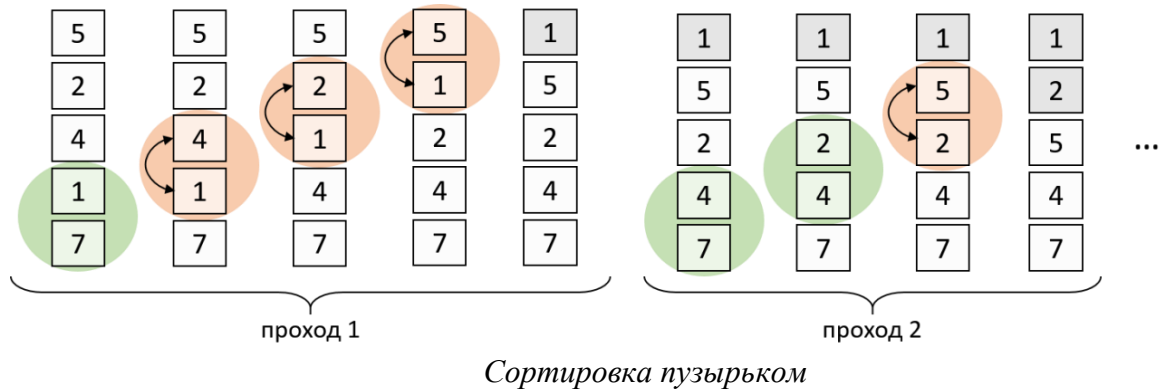
**Память** — ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных. При оценке используемой памяти не учитывается место, которое занимает исходный массив и независимые от входной последовательности затраты, например, на хранение кода программы.

**Устойчивость** — устойчивая сортировка не меняет взаимного расположения равных элементов. Такое свойство может быть очень полезным, если они состоят из нескольких полей, а сортировка происходит по одному из них.

**Естественность поведения** — эффективность метода при обработке уже отсортированных, или частично отсортированных данных. Алгоритм ведёт себя естественно, если его эффективность на таких данных повышается.

## Сортировка пузырьком

Идея метода: шаг сортировки состоит в проходе снизу вверх по массиву. По пути просматриваются пары соседних элементов. Если элементы некоторой пары находятся в неправильном порядке, то меняем их местами.

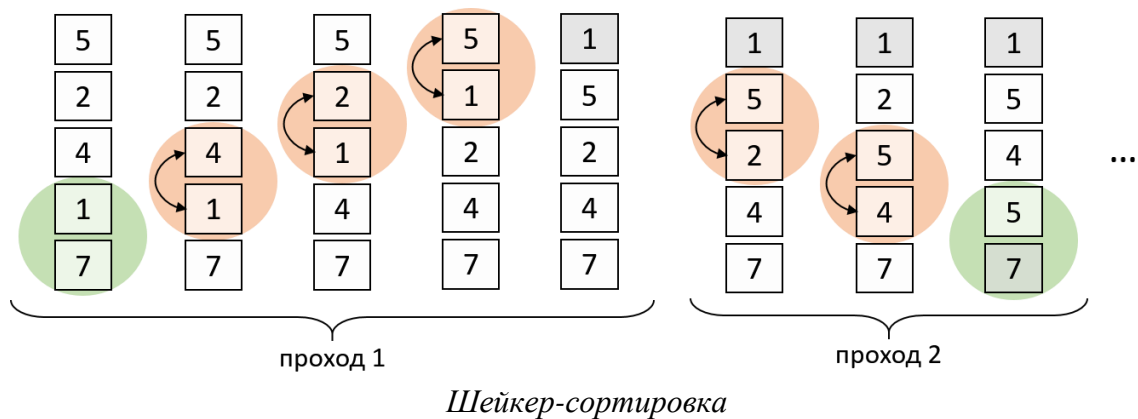


После нулевого прохода по массиву «вверх» оказывается самый «легкий» элемент — отсюда аналогия с пузырьком. Следующий проход делается до второго сверху элемента, таким образом, второй по величине элемент поднимается на правильную позицию.

Делаем проходы по все уменьшающейся нижней части массива до тех пор, пока в ней не останется только один элемент. На этом сортировка заканчивается, так как последовательность упорядочена по возрастанию. Временная сложность алгоритма составляет  $O(n^2)$ .

Качественно улучшение алгоритма можно получить из следующего наблюдения. Хотя легкий пузырек снизу поднимется вверх за один проход, тяжелые пузырьки опускаются с минимальной скоростью: один шаг за итерацию. Так что массив  $a = \{2, 3, 4, 5, 6, 1\}$  будет отсортирован за 1 проход, а сортировка последовательности  $b = \{6, 1, 2, 3, 4, 5\}$  потребует 5 проходов.

Чтобы избежать подобного эффекта, можно менять направление следующих один за другим проходов. Получившийся алгоритм иногда называют «шейкер-сортировкой». Сложность такого алгоритма не уменьшается.



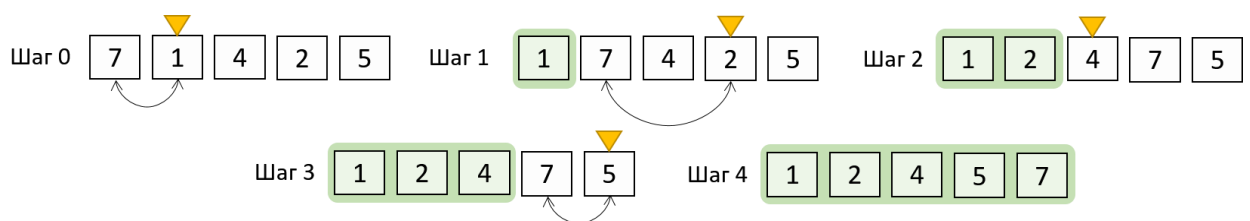
### ***Сортировка выбором***

Идея метода состоит в том, чтобы создавать отсортированную последовательность путем присоединения к ней одного элемента за другим в правильном порядке.

1. В неотсортированном подмассиве ищется локальный максимум (минимум).
2. Найденный максимум (минимум) меняется местами с последним (первым) элементом в подмассиве.
3. Если в массиве остались неотсортированные подмассивы — см. пункт 1.

Будем строить готовую последовательность, начиная с левого конца массива. Алгоритм состоит из  $n$  последовательных шагов, начиная от нулевого и заканчивая  $(n-1)$ -м. На  $i$ -м шаге выбираем наименьший из элементов  $a[i] \dots a[n]$  и меняем его местами с  $a[i]$ .

Вне зависимости от номера текущего шага  $i$ , последовательность  $a[0] \dots a[i]$  является упорядоченной. Таким образом, на  $(n-1)$ -м шаге вся последовательность, кроме  $a[n]$  оказывается отсортированной, а  $a[n]$  стоит на последнем месте по праву: все меньшие элементы уже ушли влево. Сложность такого алгоритма составляет  $O(n^2)$ .



### ***Сортировка простыми вставками***

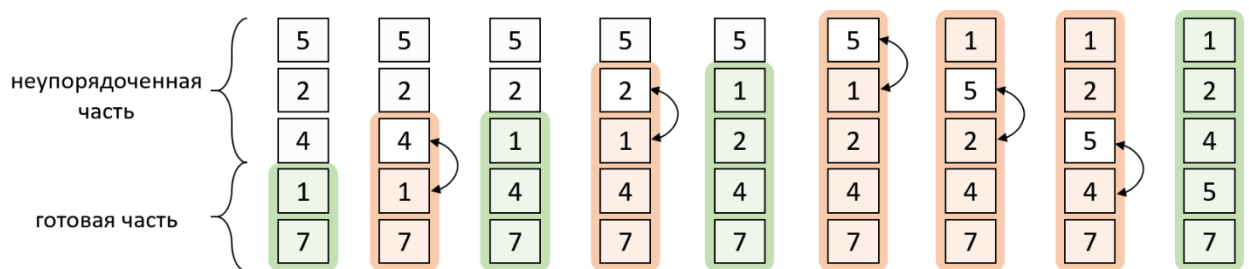
Сортировка простыми вставками в чем-то похожа на вышеизложенные методы. Общая идея всех видов сортировок вставками такова:

1. Перебираются элементы в неотсортированной части массива.
2. Каждый элемент вставляется в отсортированную часть массива на то место, где он должен находиться.

Так же, как и в предыдущем методе, делаются проходы по части массива, и в его начале «вырастает» отсортированная последовательность. Однако в сортировке пузырьком или выбором можно было четко заявить, что на  $i$ -м шаге элементы  $a[0]...a[i]$  стоят на правильных местах и никуда более не переместятся. Здесь же подобное утверждение будет более слабым: последовательность  $a[0]...a[i]$  упорядочена. При этом по ходу алгоритма в нее будут вставляться (см. название метода) все новые элементы.

Будем разбирать алгоритм, рассматривая его действия на  $i$ -м шаге. Как говорилось выше, последовательность к этому моменту разделена на две части: готовую  $a[0]...a[i]$  и неупорядоченную  $a[i+1]...a[n]$ .

На следующем,  $(i+1)$ -м каждом шаге алгоритма берем  $a[i+1]$  и вставляем на нужное место в готовую часть массива. Поиск подходящего места для очередного элемента входной последовательности осуществляется путем последовательных сравнений с элементом, стоящим перед ним. В зависимости от результата сравнения элемент либо остается на текущем месте (вставка завершена), либо они меняются местами и процесс повторяется.



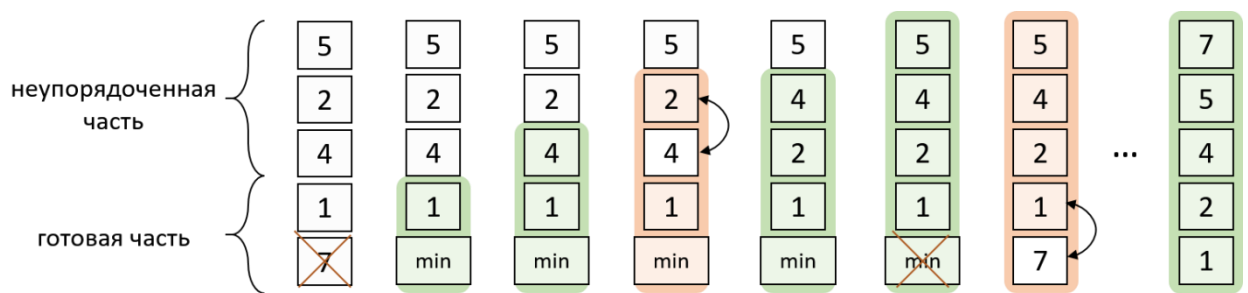
*Сортировка простыми вставками*

В данном алгоритме не задействуется дополнительная память, он является очень устойчивым. Хотя сложность алгоритма составляет  $O(n^2)$ , главное его преимущество, как и большинства сортировок вставками, — очень быстрая обработка почти упорядоченных массивов. Этому способствует основная идея этого алгоритма — перестановка элементов из неотсортированной части массива в готовую. При близком расположении близких по величине данных место вставки обычно находится близко к краю отсортированной части.

### Сортировка вставками со сторожевым элементом

Предыдущий алгоритм можно слегка улучшить. Заметим, что на каждом шаге внутреннего цикла проверяются 2 условия. Можно объединить их в одно, поставив в начало массива специальный сторожевой элемент. Он должен быть заведомо меньше всех остальных элементов массива.

Тогда при  $j=0$  будет заведомо верно  $a[j] \leq a[i]$ . Цикл остановится на нулевом элементе. Таким образом, сортировка будет происходить правильным образом, а во внутреннем цикле станет на одно сравнение меньше. Однако отсортированный массив будет не полон, так как из него исчезло первое число. Для окончания сортировки это число следует вернуть назад, а затем вставить в отсортированную последовательность  $a[1]...a[n]$ .

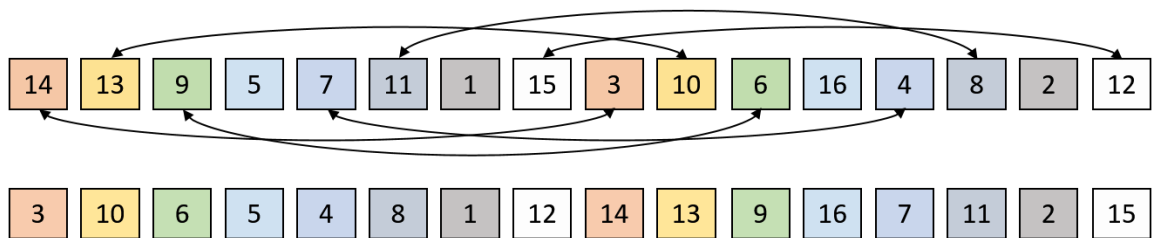


Сортировка вставками со сторожевым элементом

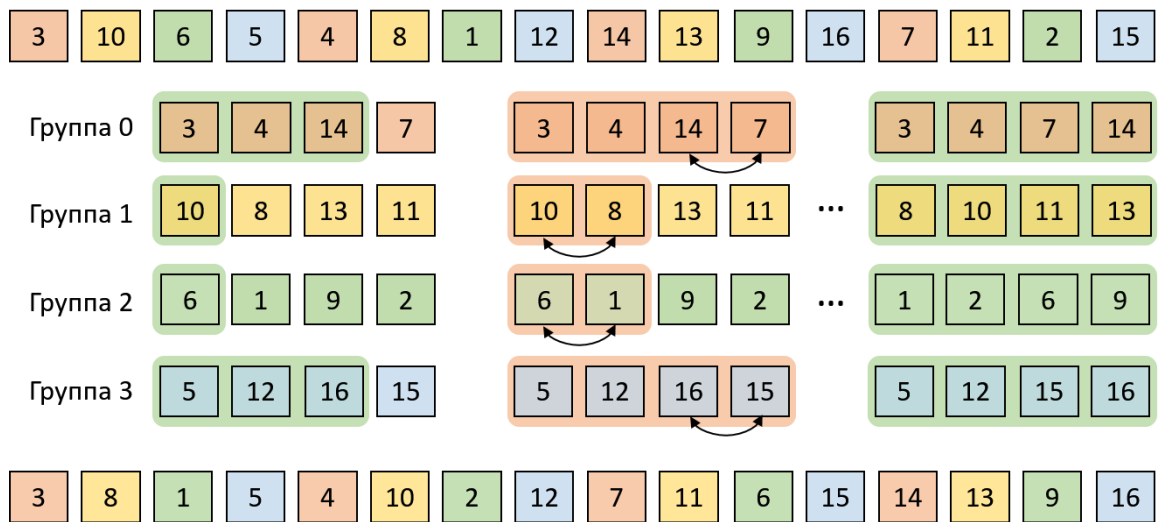
### Сортировка Шелла

Еще одна модификация алгоритма сортировки простыми вставками — сортировка Шелла. Рассмотрим следующий алгоритм сортировки массива  $a[0]..a[15]$ .

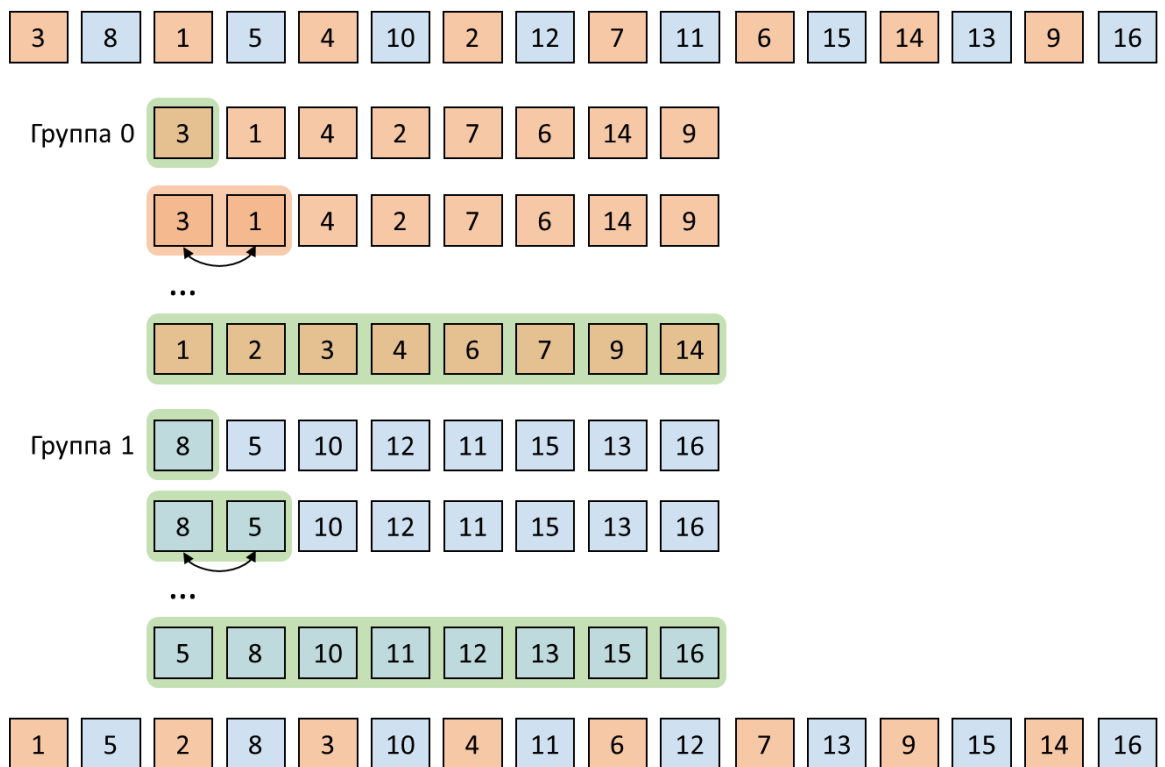
1. Вначале сортируем простыми вставками каждые 8 групп из 2-х элементов ( $a[0]$ ,  $a[8]$ ), ( $a[1]$ ,  $a[9]$ ), ..., ( $a[7]$ ,  $a[15]$ ).



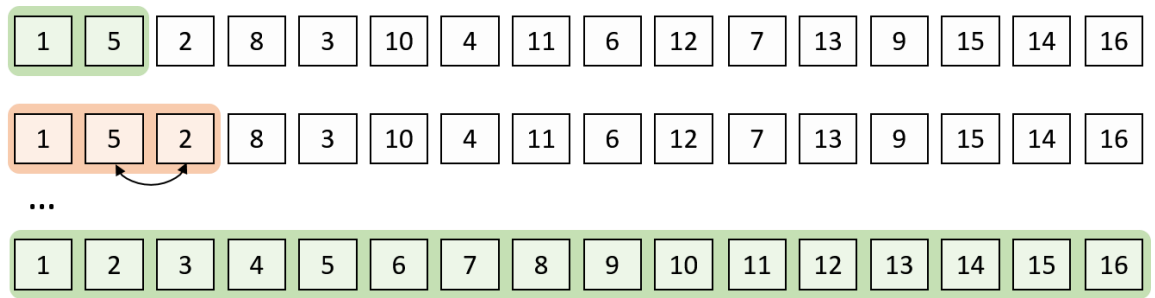
2. Потом сортируем каждую из четырех групп по 4 элемента ( $a[0]$ ,  $a[4]$ ,  $a[8]$ ,  $a[12]$ ), ..., ( $a[3]$ ,  $a[7]$ ,  $a[11]$ ,  $a[15]$ ).



3. Далее сортируем 2 группы по 8 элементов, начиная с ( $a[0]$ ,  $a[2]$ ,  $a[4]$ ,  $a[6]$ ,  $a[8]$ ,  $a[10]$ ,  $a[12]$ ,  $a[14]$ ).



4. В конце сортируем вставками все 16 элементов.



Очевидно, лишь последняя сортировка необходима, чтобы расположить все элементы по своим местам. Так зачем нужны остальные?

На самом деле они продвигают элементы максимально близко к соответствующим позициям, так что в последней стадии число перемещений будет весьма невелико. На последней итерации алгоритм работает с уже почти отсортированной последовательностью, а на таких данных алгоритм сортировки простыми вставками наиболее производителен. Ускорение подтверждено многочисленными исследованиями и на практике оказывается довольно существенным.

Единственной характеристикой сортировки Шелла является приращение — расстояние между сортируемыми элементами, в зависимости от прохода. Именно от него зависит среднее время работы алгоритма. В конце приращение всегда равно единице — метод завершается обычной сортировкой вставками, но именно последовательность приращений определяет рост эффективности.

Использованный в примере набор — 8, 4, 2, 1 — неплохой выбор, особенно, когда количество элементов — степень двойки. Однако гораздо лучший вариант предложил Р. Седжвик. Его последовательность имеет вид:

$$inc[s] = \begin{cases} 9 \cdot 2^s - 9 \cdot 2^{s/2} + 1, & \text{если } s \text{ четно} \\ 8 \cdot 2^s - 6 \cdot 2^{(s+1)/2} + 1, & \text{если } s \text{ нечетно.} \end{cases}$$

При использовании таких приращений среднее количество операций:  $O(n^{7/6})$ , в худшем случае — порядка  $O(n^{4/3})$ .

Обратим внимание на то, что последовательность вычисляется в порядке, противоположном используемому:  $inc[0] = 1$ ,  $inc[1] = 5$ , ... Формула дает сначала меньшие числа, затем все большие и большие, в то время как расстояние между сортируемыми элементами, наоборот, должно уменьшаться. Поэтому массив приращений  $inc$  вычисляется перед запуском собственно сортировки до максимального расстояния между элементами, которое будет первым шагом в сортировке Шелла. Потом его значения используются в обратном порядке.



При использовании формулы Седжвика следует остановиться на значении `inc[s-1]`, если  $3 * inc[s] > size$ .

Часто вместо вычисления последовательности во время каждого запуска процедуры, ее значения рассчитывают заранее и записывают в таблицу, которой пользуются, выбирая начальное приращение по тому же правилу: начинаем с `inc[s-1]`, если  $3 * inc[s] > size$  (см. прил. А).

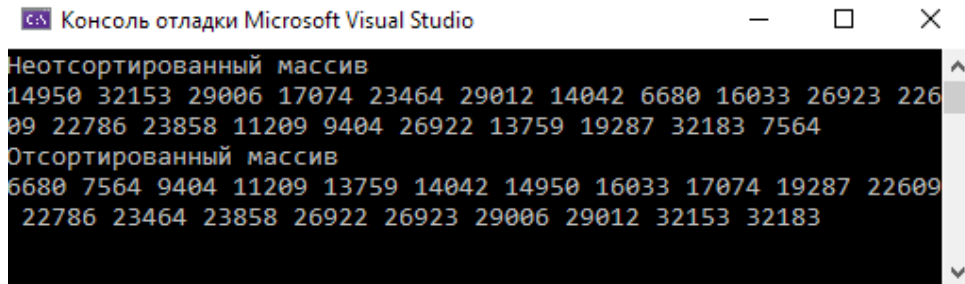
### Задание для алгоритмов сортировки

1. Создайте новый консольный проект.
2. Добавьте в проект `cpp`-файл, в котором будет располагаться функция `main()`.
3. Для удобства будем сортировать последовательности целых положительных чисел. Создайте функцию **generate**, которая будет создавать массив заданного количества произвольных чисел от 0 до 32767, расположенных в произвольном порядке.  
Для генерации чисел можно воспользоваться функцией `int rand(void)` из библиотеки `stdlib.h`. Эта функция возвращает псевдослучайное число из диапазона 0 – 32767. Перед стартом цикла генерации чисел необходимо инициализировать систему псевдослучайных чисел, чтобы каждый раз генерировались разные последовательности. Сделать это можно вызовом функции `srand(steady_clock::now().time_since_epoch().count())`. Аргумент функции `srand()` — это функция из пространства имен стандартной библиотеки `chrono`. Она берет время, прошедшее с начала работы системы, и приводит его к типу `long`.
4. Создайте функцию **print**, которая будет выводить на экран в строку через пробел все числа заданной последовательности.
5. Реализуйте функцию для алгоритма сортировки чисел по возрастанию согласно варианту:

Вариант	Алгоритм сортировки
1	Сортировка пузырьком
2	Сортировка Шелла
3	Сортировка выбором
4	Сортировка вставками со сторожевым элементом
5	Шейкер-сортировка
6	Сортировка простыми вставками

Описания функций даны в приложении А.

6. Составьте функцию `main` так, чтобы генерировалось 20 случайных чисел, затем они выводились на экран, сортировались и снова выводились на экран. Убедитесь, что сортировка работает:



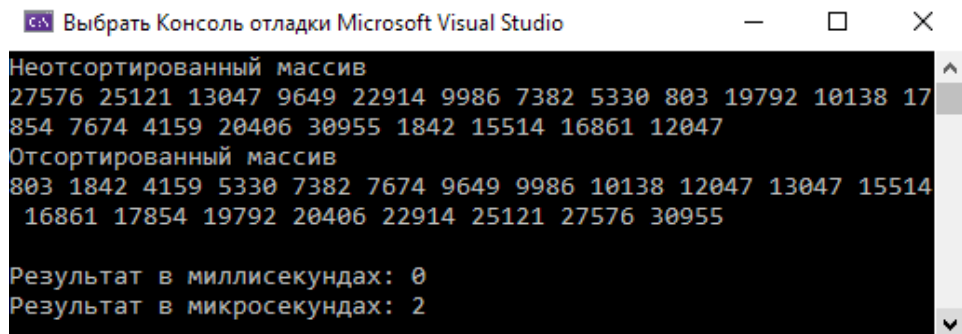
```
Консоль отладки Microsoft Visual Studio
Неотсортированный массив
14950 32153 29006 17074 23464 29012 14042 6680 16033 26923 226
09 22786 23858 11209 9404 26922 13759 19287 32183 7564
Отсортированный массив
6680 7564 9404 11209 13759 14042 14950 16033 17074 19287 22609
22786 23464 23858 26922 26923 29006 29012 32153 32183
```

7. Замерьте время, требуемое для сортировки. Для этого можно воспользоваться уже знакомой функцией `steady_clock::now()` по следующему принципу:

```
time_point<steady_clock> start = steady_clock::now();
// ... Сортировка массива
time_point<steady_clock> fin = steady_clock::now();
long dur = duration_cast<milliseconds>(fin - start).count();
```

Разница (`fin - start`) даст длительность процесса сортировки в миллисекундах, а функция `count()` приведет результат к типу `long`.

8. Результатом замера скорее всего будет 0, т.к. чисел очень мало для того, чтобы сортировка заняла сколько-нибудь ощутимое время, но если установить измерение времени в микросекундах, можно будет узнать время выполнения сортировки точнее.

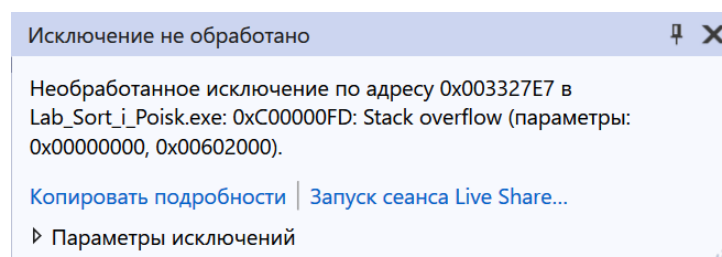


```
Выбрать Консоль отладки Microsoft Visual Studio
Неотсортированный массив
27576 25121 13047 9649 22914 9986 7382 5330 803 19792 10138 17
854 7674 4159 20406 30955 1842 15514 16861 12047
Отсортированный массив
803 1842 4159 5330 7382 7674 9649 9986 10138 12047 13047 15514
16861 17854 19792 20406 22914 25121 27576 30955

Результат в миллисекундах: 0
Результат в микросекундах: 2
```

Количество сортируемых чисел необходимо увеличить.

Сделайте замеры для массива из 10 000 000 элементов. Обратите внимание, что если массив задается статически в стиле `int m[10000000]`, то скорее всего произойдет ошибка переполнения стека:



Для того чтобы можно было работать с большим количеством чисел, воспользуемся динамическим выделением:

```
int* m = new int[10000000];  
// ...  
delete[] m;
```

9. **Сделайте 10 замеров для количества 10000000 чисел.** Посчитайте среднее время работы алгоритма сортировки. Замеры и результаты расчета отразите в отчете.
10. **Сделайте такие же замеры для различного числа элементов** так, чтобы по полученным данным можно было получить 5-7 точек для построения графика зависимости времени работы от количества данных. В отчете приведите таблицу с данными о замерах, о среднем времени и график зависимости.
11. **Реализуйте функцию для алгоритма, следующего за Вашим вариантом** (1 для варианта б) **для сортировки по убыванию.** Описания алгоритмов даны в приложении А Приложение Б. Алгоритмы поиска.
12. Проведите все необходимые замеры, **приведите в отчет таблицы замеров и график,** на котором отражены зависимости для Ваших алгоритмов.

## Часть 2 — Алгоритмы поиска

Поиск — обработка некоторого множества данных с целью выявления подмножества данных, соответствующего критериям поиска.

Поиск является одним из наиболее часто встречаемых действий в программировании. Существует множество различных алгоритмов поиска, которые принципиально зависят от способа организации данных. У каждого алгоритма поиска есть свои преимущества и недостатки. Поэтому важно выбрать тот алгоритм, который лучше всего подходит для решения конкретной задачи.

Рассмотрим основные алгоритмы поиска в линейных структурах.

### *Поиск значения перебором*

Это простейший алгоритм поиска. Он редко используется из-за своей неэффективности, поскольку он уступает другим алгоритмам.

Алгоритм ищет элемент в заданной структуре данных, пока не достигнет конца структуры. При нахождении элемента возвращается его позиция в структуре данных. Если элемент не найден, возвращаем -1.

```
int enumSearch(int a[], long size, int val)
{
    for(int i = 0; i < size; i++)
    {
        if (a[i] == val)
            return i;
    }
    return -1; // значение не найдено
}
```

Искомый элемент: 9

{3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52}



{3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52}



{3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52}



{3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52}



### *Поиск перебором*

Для получения позиции искомого элемента перебирается набор из  $n$  элементов. В худшем сценарии для этого алгоритма искомый элемент оказывается последним в массиве.

В этом случае потребуется  $n$  итераций для нахождения элемента, следовательно, временная сложность линейного поиска равна  $O(n)$ .

Поиск перебором можно использовать для малого, несортированного набора данных, который не увеличивается в размерах. Несмотря на простоту, алгоритм не находит применения в проектах из-за линейного увеличения временной сложности.

### **Бинарный поиск**

В бинарном поиске исходное множество должно быть упорядочено по возрастанию. Иными словами, каждый последующий ключ больше предыдущего, т. е.  $\{K_1 \leq K_2 \leq \dots \leq K_{n-1} \leq K_n\}$ .

Отыскиваемый ключ сравнивается с центральным элементом множества, если он меньше центрального, то поиск продолжается в левом подмножестве, в противном случае — в правом.

Номер центрального элемента  $N$  находится по формуле:

$$N = \lfloor n/2 \rfloor + 1,$$

где квадратные скобки обозначают, что берется только целая часть результата деления (всегда округляется в меньшую сторону). В методе бинарного поиска анализируются только центральные элементы.

Пример. Дано множество:

$\{7, 8, 12, 16, 18, 20, 30, 38, 49, 50, 54, 60, 61, 69, 75, 79, 80, 81, 95, 101, 123, 198\}$ .

Найти во множестве ключ  $K = 61$ .

**Шаг 1.**  $N = \lfloor n/2 \rfloor + 1 = \lfloor 22/2 \rfloor + 1 = 12$ .

$\{7, 8, 12, 16, 18, 20, 30, 38, 49, 50, 54, 60, 61, 69, 75, 79, 80, 81, 95, 101, 123, 198\}$   
левое подмножество                      правое подмножество

$K \sim K_{12}$ ,  $61 > 60$ . Дальнейший поиск будем вести в правом подмножестве.

Значок « $\sim$ » обозначает сравнение элементов (чисел, значений).

**Шаг 2.**  $N = \lfloor n/2 \rfloor + 1 = \lfloor 10/2 \rfloor + 1 = 6$ .

$\{7, 8, 12, 16, 18, 20, 30, 38, 49, 50, 54, 60, 61, 69, 75, 79, 80, 81, 95, 101, 123, 198\}$   
левое подмножество                      правое подмножество

$K \sim K_{18}$ ,  $61 < 81$ . Дальнейший поиск в левом подмножестве относительно предыдущего подмножества.

**Шаг 3.**  $N = \lfloor n/2 \rfloor + 1 = \lfloor 5/2 \rfloor + 1 = 3$ .

{7, 8, 12, 16, 18, 20, 30, 38, 49, 50, 54, 60, 61, 69, 75, 79, 80, 81, 95, 101, 123, 198}

левое подмножество
правое подмножество

$K \sim K_{15}$ ,  $61 < 75$ . Дальнейший поиск в левом подмножестве.

**Шаг 4.**  $N = \lfloor n/2 \rfloor + 1 = \lfloor 2/2 \rfloor + 1 = 2$ .

{7, 8, 12, 16, 18, 20, 30, 38, 49, 50, 54, 60, 61, 69, 75, 79, 80, 81, 95, 101, 123, 198}

левое подмножество

$K \sim K_{14}$ ,  $61 < 69$ . Дальнейший поиск в левом подмножестве.

**Шаг 5.**  $N = \lfloor n/2 \rfloor + 1 = \lfloor 1/2 \rfloor + 1 = 1$ .

{7, 8, 12, 16, 18, 20, 30, 38, 49, 50, 54, 60, 61, 69, 75, 79, 80, 81, 95, 101, 123, 198}

$K \sim K_{13}$ ,  $61 = 61$ .

Вывод: искомый ключ найден под номером 13.

Временная сложность алгоритма двоичного поиска равна  $O(\log(n))$  из-за деления массива пополам. Она превосходит  $O(n)$  линейного алгоритма. Этот алгоритм используется в большинстве библиотек и используется с отсортированными структурами данных.

### **Фибоначчиев поиск**


Поиск Фибоначчи - это вариант бинарного поиска. Его временная сложность  $O(\log_2 n)$ . В этом поиске анализируются элементы, находящиеся в позициях, равных числам Фибоначчи. Числа Фибоначчи получаются по следующему правилу: каждое последующее число равно сумме двух предыдущих чисел, например: {1, 2, 3, 5, 8, 13, 21, 34, 55, ...}.

Поиск продолжается до тех пор, пока не будет найден интервал между двумя ключами, где может располагаться отыскиваемый ключ.

Пример. Дано исходное множество ключей:

{3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52}.

Пусть отыскиваемый ключ равен 42 ( $K = 42$ ). Последовательное сравнение отыскиваемого ключа будет проводиться в позициях, равных числам Фибоначчи: {1, 2, 3, 5, 8, 13, 21, ...}.

{3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52}  


**Шаг 1.**  $K \sim K_1$ ,  $42 > 3 \Rightarrow$  отыскиваемый ключ сравнивается с ключом, стоящим в позиции, равной числу Фибоначчи.

**Шаг 2.**  $K \sim K_2$ ,  $42 > 5 \Rightarrow$  сравнение продолжается с ключом, стоящим в позиции, равной следующему числу Фибоначчи.

**Шаг 3.**  $K \sim K_3$ ,  $42 > 8 \Rightarrow$  сравнение продолжается.

**Шаг 4.**  $K \sim K_5$ ,  $42 > 11 \Rightarrow$  сравнение продолжается.

**Шаг 5.**  $K \sim K_8$ ,  $42 > 19 \Rightarrow$  сравнение продолжается.

**Шаг 6.**  $K \sim K_{13}$ ,  $42 > 35 \Rightarrow$  сравнение продолжается.

**Шаг 7.**  $K \sim K_{18}$ ,  $42 < 52 \Rightarrow$  найден интервал, в котором находится отыскиваемый ключ: от 13 до 18 позиции, т. е. {35, 37, 42, 45, 48, 52}.

В найденном интервале поиск вновь ведется в позициях, равных числам Фибоначчи.

### **Интерполяционный поиск**

Исходное множество должно быть упорядочено по возрастанию весов.

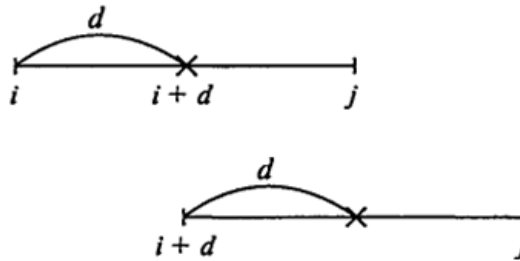
Первоначальное сравнение осуществляется на расстоянии шага  $d$ , который определяется по формуле:

$$d = \left\lfloor \frac{(j - i)(K - K_i)}{(K_j - K_i)} \right\rfloor,$$

где  $i$  — номер первого рассматриваемого элемента;  $j$  — номер последнего рассматриваемого элемента;  $K$  — отыскиваемый ключ;  $K_i$ ,  $K_j$  — значения ключей в  $i$  и  $j$  позициях;  $\lfloor \rfloor$  — целая часть от числа.

Идея метода заключается в следующем: шаг  $d$  меняется после каждого этапа по формуле, приведенной выше. Алгоритм заканчивает работу при  $d = 0$ , при этом

анализируются соседние элементы, после чего делается окончательное решение о результатах поиска.



Этот метод прекрасно работает, если исходное множество представляет собой арифметическую прогрессию или множество, приближенное к ней. В среднем интерполирующий поиск производит  $\log(\log(n))$  операций. Число операций зависит от равномерности распределения значений элементов. В плохом случае (например, когда значения элементов экспоненциально возрастают) интерполяционный поиск может потребовать до  $O(n)$  операций.

Пример. Дано множество ключей:

{2, 9, 10, 12, 20, 24, 28, 30, 37, 40, 45, 50, 51, 60, 65, 70, 74, 76}.

Пусть искомый ключ равен 70 ( $K = 70$ ).

**Шаг 1.** Определим шаг  $d$  для исходного множества ключей:

$$d = [(18 - 1)(70 - 2)/(76 - 2)] = 15.$$

Сравниваем ключ, стоящий под шестнадцатым порядковым номером в данном множестве, с искомым ключом:

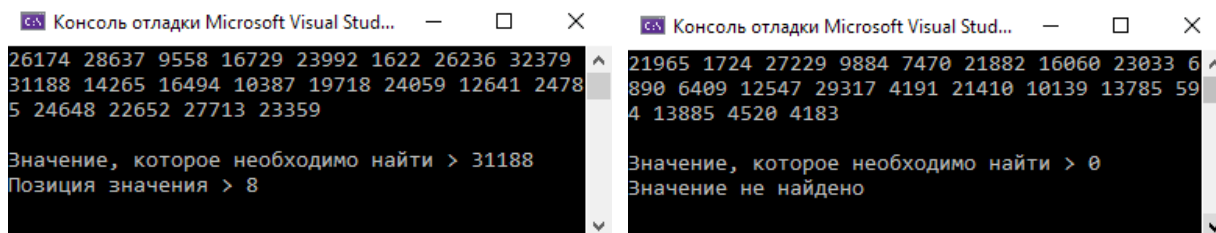
$K_{16} \sim K$ ,  $70 = 70$ , ключ найден.

### Задание для алгоритмов поиска

1. В том же консольном проекте реализуйте функцию поиска заданного значения перебором (приведенную в приложении Б), возвращающую позицию найденного элемента **в неотсортированном массиве** или сообщающую, что искомого элемента нет.
2. Составьте функцию `main` так, чтобы генерировалось 20 случайных чисел, затем они выводились на экран, с клавиатуры запрашивался элемент для поиска, производился



поиск и на экран выводилась позиция найденного элемента. Убедитесь, что поиск работает:



3. **Замерьте время**, требуемое для поиска таким же способом, как и в предыдущем задании.
4. **Сделайте 10 замеров для количества 10000000 чисел.** Посчитайте среднее время работы алгоритма поиска. Замеры и результаты расчета отразите в отчете.
5. **Сделайте такие же замеры для различного числа элементов** так, чтобы по полученным данным можно было получить 5-7 точек для построения графика зависимости времени работы от количества данных. В отчете приведите таблицу с данными о замерах, о среднем времени и график зависимости.
6. **Реализуйте функцию для алгоритма поиска** согласно варианту (заметьте, что не все алгоритмы работают с неотсортированными данными). Описания алгоритмов даны в приложении Б.

Вариант	Алгоритм поиска
1	Бинарный поиск
2	Интерполяционный поиск
3	Фибоначчиев поиск
4	Интерполяционный поиск

7. Проведите все необходимые замеры, **приведите в отчет таблицы замеров и график**, на котором отражены зависимости для алгоритма последовательного поиска и Вашего алгоритма. Замеры производите таким образом, чтобы можно было отразить в отчете время, требуемое на весь алгоритм поиска (включая сортировку любым из двух реализованных способов), а также время непосредственно поиска (исключая сортировку).

## Индивидуальные задания

В новом консольном проекте реализуйте структуру данных и алгоритмы сортировки и поиска согласно варианту.

1. **Успеваемость студентов.** Фамилиям студентов двух групп Вашего потока поставьте в соответствие случайное число из интервала от 25 до 54. Отсортируйте получившийся список по алфавиту, а затем по возрастанию баллов для каждой буквы. Используйте два разных алгоритма сортировки. Найдите в получившемся списке свою фамилию, не используя поиск перебором.
2. **Автомобильные номера и марки.** Создать список из 50 номеров автомобилей в формате «0 XXX 00», где 0 — случайная цифра, а X — случайная буква, которым в соответствие поставить случайную марку из некоторого набора. Отсортировать список номеров сначала по алфавиту по маркам, а потом по номерам для каждой марки. Реализуйте алгоритм поиска автомобиля по марке и номеру для полученного списка.
3. **Очередь.** В каждый из трех кабинетов есть запись по талонам, в которых указана фамилия посетителя, кабинет и время записи. Запись ведется каждые 7 минут в первый, каждые 30 минут во второй и каждые 6 минут в третий кабинет во временном интервале, соответствующем времени проведения лабораторной работы. Информация об очередях — совокупность записей формата «Фамилия N чч:мм», где N — номер кабинета. Объедините очереди в первый и второй кабинеты и отсортируйте по времени записи с приоритетом по алфавиту в случае одинакового времени.  
Найдите фамилию человека с записью на 9:12 (13:44) в третий кабинет.
4. **Список товаров.** Создайте список для 20 товаров из записей вида «Наименование — Цена — Количество». Реализуйте функции сортировки товара по алфавиту и по суммарной стоимости и функцию, принимающую на вход наименование и искомое количество товара. В случае наличия она должна уменьшать количество в исходном списке на указанное значение, иначе — уведомлять об отсутствии необходимого количества.
5. **Телефонная книга.** Создать список из фамилий и соответствующих им телефонных номеров формата «0-00», где 0 — случайная цифра, и номеров кабинетов (номера не повторяются). Реализовать функции сортировки по фамилиям и по номерам кабинетов, а также функцию поиска номера по фамилии или кабинету.
6. **Сортировка букв по алфавиту.** Отсортировать буквы текста двух объединенных панграмм (коротких текстов, использующих все буквы алфавита) — одной на русском языке, другой на английском — по возрастанию и по убыванию соответственно,

используя два разных алгоритма сортировки. В каждой полученной последовательности найти первую букву своей фамилии.

7. **Каждый второй — бесплатно.** В некотором магазине проводится акция: каждый второй товар в чеке покупатель получает бесплатно. На вход программа получает последовательность товаров вида «Наименование — Цена — Количество», а на выходе мы должны получить последовательность этих товаров, оптимальную для того, чтобы сумма чека была максимальной, и итоговую сумму чека. Реализуйте возможность поиска и удаления выбранной позиции по наименованию с последующим пересмотром порядка позиций в чеке.
8. **Визуализация (а).** В окне консоли визуализировать работу алгоритмов сортировки пузырьком и Шелла для последовательности случайных чисел, показывая перестановки на каждом шаге, например:

```
Неотсортированный массив
81 54 84 93 13 0 34 97 77 46 40 16 79 14 40 8

Процесс сортировки пузырьком:
81 54 84 93 13 0 34 97 77 46 40 16 79 14 40 8
                                     ^_
81 54 84 93 13 0 34 97 77 46 40 16 79 14 8 40
                                     ^_
```

Реализовать рекурсивно бинарный поиск.

9. **Визуализация (б).** В окне консоли визуализировать работу алгоритмов сортировки выбором и простыми вставками для последовательности случайных чисел, показывая перестановки на каждом шаге, например:

```
Неотсортированный массив
66 53 67 71 1 57 55 27 18 93 54 63 38 10 90 8

Процесс сортировки выбором:
66 53 67 71 1 57 55 27 18 93 54 63 38 10 90 8
^_____
1 53 67 71 66 57 55 27 18 93 54 63 38 10 90 8
^_____
```

Реализовать фибоначчиев поиск рекурсивно.

10. **Построение по росту.** На вход программы подается список солдат из трех взводов (один взвод от 15 до 60 человек). Записи списка имеют формат «Фамилия – Взвод – Рост». Рост принять от 150 до 210 см. На выходе программа должна «построить» солдат по взводам и по росту. Реализовать алгоритм поиска солдата в списке по взводу и по фамилии.
11. **Письма с ярлыками.** Создайте список писем из записей формата «Тема – Цвет», где один из пяти цветов сигнализирует о принадлежности письма к некоторой группе.

Отсортировать письма по цветам в порядке радуги, например: «красный», «желтый», «зеленый», «синий», «фиолетовый». Для каждой группы отсортировать письма по теме в алфавитном порядке. Реализовать алгоритм поиска письма по цвету и теме.

**12. Сортировка простыми вставками с бинарным поиском.** Модифицировать алгоритм сортировки вставками так, чтобы место вставки определялось методом бинарного поиска. Сравнить временные показатели такого алгоритма с обычной сортировкой вставками. Реализовать бинарный поиск рекурсивно.

**13. Сортировка расчёской.** Модифицировать базовый алгоритм пузырьковой сортировки так, чтобы сначала сравнивалась пара элементов из противоположных концов массива, а затем расстояние с каждым проходом уменьшалось, пока не будут сравниваться пары соседних элементов. Выбрать наиболее оптимальную стратегию уменьшения расстояния между сравниваемыми элементами. Сравнить временные показатели алгоритма с обычной пузырьковой сортировкой. На отсортированной последовательности отработать алгоритм поиска, который не отработывался в общей части.

**14. Парная сортировка простыми вставками.** Модифицировать алгоритм сортировки простыми вставками так, чтобы к отсортированной части добавлялся не один, а сразу два рядом стоящих элемента. Сначала вставляется больший элемент из пары и сразу после него метод простой вставки применяется к меньшему элементу из пары. Сравнить с сортировкой вставками из общей части работы по времени обработки массивов малой и большой длины. На отсортированной последовательности отработать алгоритм поиска, который не отработывался в общей части.

**15. Быстрая сортировка.** Реализовать функцию для алгоритма быстрой сортировки:

- 1) Движемся от левого края массива до тех пор, пока не обнаружим элемент больше элемента посередине, а от правого — пока не обнаружим элемент меньше элемента посередине.
- 2) Меняем местами два найденных элемента.
- 3) Продолжаем выполнять пункты 1–2, пока не дойдем до середины, где поделим массив пополам и к каждой части рекурсивно применим алгоритм быстрой сортировки.

Сравнить время выполнения с одним из реализованных алгоритмов. На отсортированной последовательности отработать алгоритм поиска, который не отработывался в общей части.

## Приложения

### Приложение А. Алгоритмы сортировки

Сортировка пузырьком	Шейкер-сортировка
<pre>void bubbleSort(int a[], int n) {     int buff = 0;      for (int i = 0; i &lt; n - 1; i++)     {         for (int j = n - 1; j &gt; i; j--)         {             if (a[j] &lt; a[j - 1])             {                 buff = a[j - 1];                 a[j - 1] = a[j];                 a[j] = buff;             }         }     } }</pre>	<pre>void shakerSwap(int a[], int i) {     int buff = a[i];     a[i] = a[i - 1];     a[i - 1] = buff; }  void shakerSort(int a[], int n) {     int leftMark = 1;     int rightMark = n - 1;     while (leftMark &lt;= rightMark)     {         for (int i = rightMark; i &gt;= leftMark; i--)             if (a[i - 1] &gt; a[i]) shakerSwap(a, i);         leftMark++;          for (int i = leftMark; i &lt;= rightMark; i++)             if (a[i - 1] &gt; a[i]) shakerSwap(a, i);         rightMark--;     } }</pre>
Сортировка простыми вставками	Сортировка вставками со сторожевым элементом
<pre>void insertSort(int a[], int n) {     int buff = 0, j;      for (int i = 1; i &lt; n; i++)     {         buff = a[i];         for (j = i - 1;              j &gt;= 0 &amp;&amp; a[j] &gt; buff; j--)             a[j + 1] = a[j];         a[j + 1] = buff;     } }</pre>	<pre>void insertGuardSort(int a[], int n) {     int buff;     int i, j;     int backup = a[0];     a[0] = INT_MIN;     for (i = 1; i &lt; n; i++)     {         buff = a[i];         for (j = i - 1; a[j] &gt; buff; j--)             a[j + 1] = a[j];         a[j + 1] = buff;     }     for (j = 1; j &lt; n &amp;&amp; a[j] &lt; backup; j++)         a[j - 1] = a[j];     a[j - 1] = backup; }</pre>
Сортировка Шелла	Сортировка выбором
<pre>int increment(long inc[], long size) {     int p1, p2, p3, s;     p1 = p2 = p3 = 1;     s = -1;     do     {         if (++s % 2) {             inc[s] = 8 * p1 - 6 * p2 + 1;         }         else {             inc[s] = 9 * p1 - 9 * p3 + 1;             p2 *= 2;             p3 *= 2;         }         p1 *= 2;     } }</pre>	<pre>void chooseSort(int a[], int n) {     int min = 0;     int buf = 0;      for (int i = 0; i &lt; n; i++)     {         min = i;         for (int j = i + 1; j &lt; n; j++)             min = (a[j] &lt; a[min]) ? j : min;          if (i != min)         {             buf = a[i];             a[i] = a[min];             a[min] = buf;         }     } }</pre>

<pre> } while (3 * inc[s] &lt; size); return s &gt; 0 ? --s : 0; }  void shellSort(int a[], long size) { long inc, i, j, seq[40]; int s; // вычисление последовательности приращений s = increment(seq, size); while (s &gt;= 0) { // сортировка вставками с инкрементами inc inc = seq[s--]; for (i = inc; i &lt; size; i++) { int temp = a[i]; for (j = i - inc; (j &gt;= 0) &amp;&amp; (a[j] &gt; temp); j -= inc) a[j + inc] = a[j]; a[j + inc] = temp; } } } </pre>	<pre> } } </pre>
--	------------------

## Приложение Б. Алгоритмы поиска

Поиск перебором	Бинарный поиск
<pre> int enumSearch(int a[], long size, int val) { for(int i = 0; i &lt; size; i++) { if (a[i] == val) return i; } return -1; } </pre>	<pre> int binarySearch(int a[], long size, int val) { int lastIndex = size - 1; int firstIndex = 0;  while (lastIndex &gt;= firstIndex) { int middleIndex = (firstIndex + lastIndex) / 2; if (a[middleIndex] == val) { return middleIndex; } else if (a[middleIndex] &lt; val) firstIndex = middleIndex + 1; else if (a[middleIndex] &gt; val) lastIndex = middleIndex - 1; } return -1; } </pre>
Фибоначчиев поиск	Интерполяционный поиск
<pre> int fibSearch(int a[], int start, int finish, int size, int val) { int fibm2 = 0; int fibm1 = 1; int fib = 1;  while (fibm1 + shift &lt; size) { if (a[fib + shift - 1] == val) </pre>	<pre> int interpolationSearch(int a[], long size, int val) {  int middleIndex; int firstIndex = 0; int lastIndex = size - 1; int d; </pre>

```

return fib + shift - 1;
if (a[fib + shift - 1] < val)
{
    fibm2 = fibm1;
    fibm1 = fib;
    fib = (fibm1 + fibm2 + shift <
size) ? fibm1 + fibm2 : size - shift;
}
else if (a[fib + shift - 1] > val)
{
    shift += fibm1;
    fibm2 = 0;
    fibm1 = 1;
    fib = 1;
}
}
}
return -1;
}

```

```

while (a[firstIndex] <= val &&
a[lastIndex] >= val && firstIndex <
lastIndex) {
    d = ((lastIndex - firstIndex)
* (val - a[firstIndex])) / (a[lastIndex] -
a[firstIndex]);
    middleIndex = firstIndex + d;

    if (a[middleIndex]==val)
        return middleIndex;
    else if (a[middleIndex] < val)
        firstIndex = middleIndex + 1;
    else
        lastIndex = middleIndex - 1;
}
return -1;
}

```