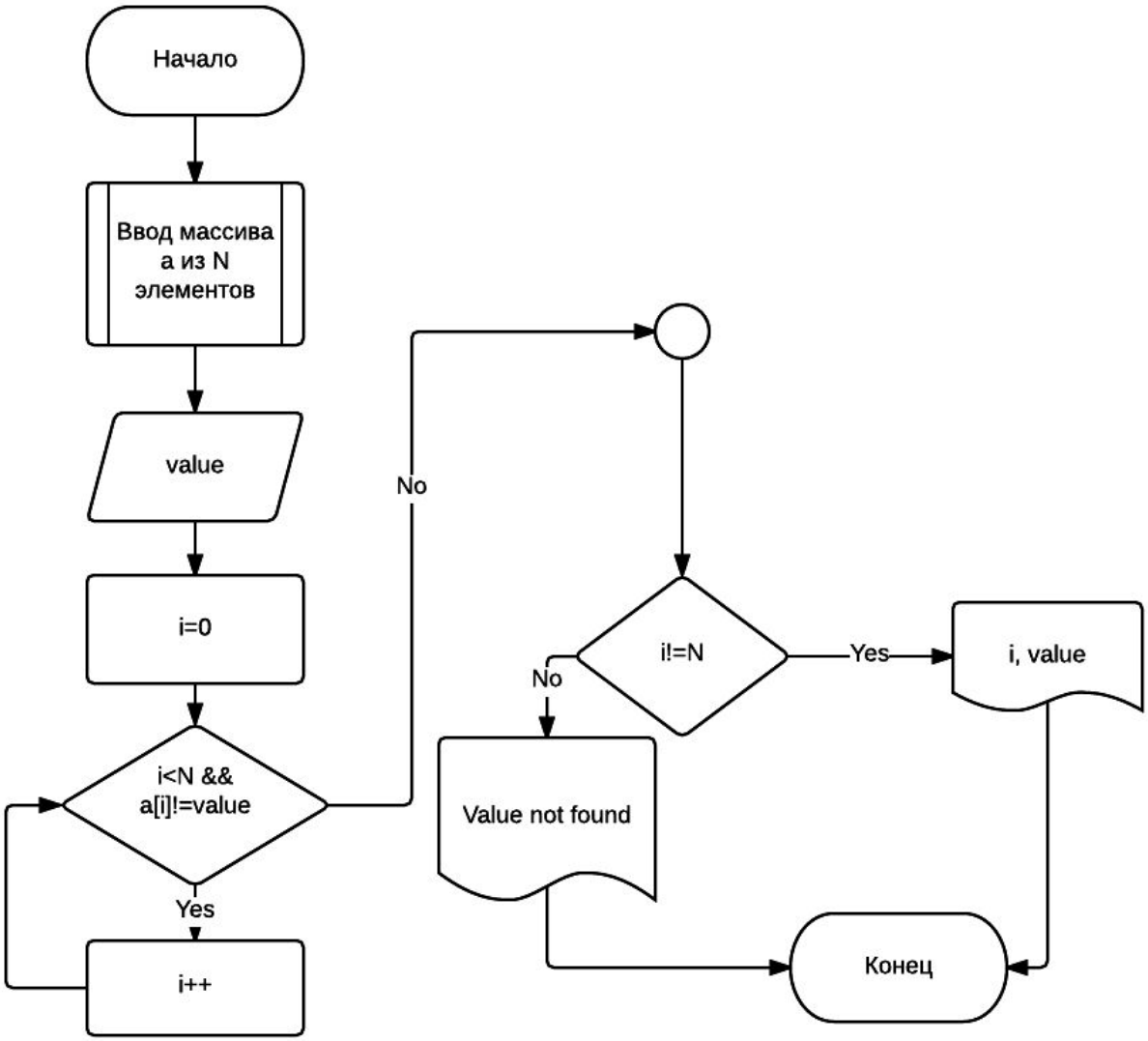
Урок 3

# Поиск в массиве. Простые сортировки

# Поиск в одномерном массиве

## Линейный поиск в массиве

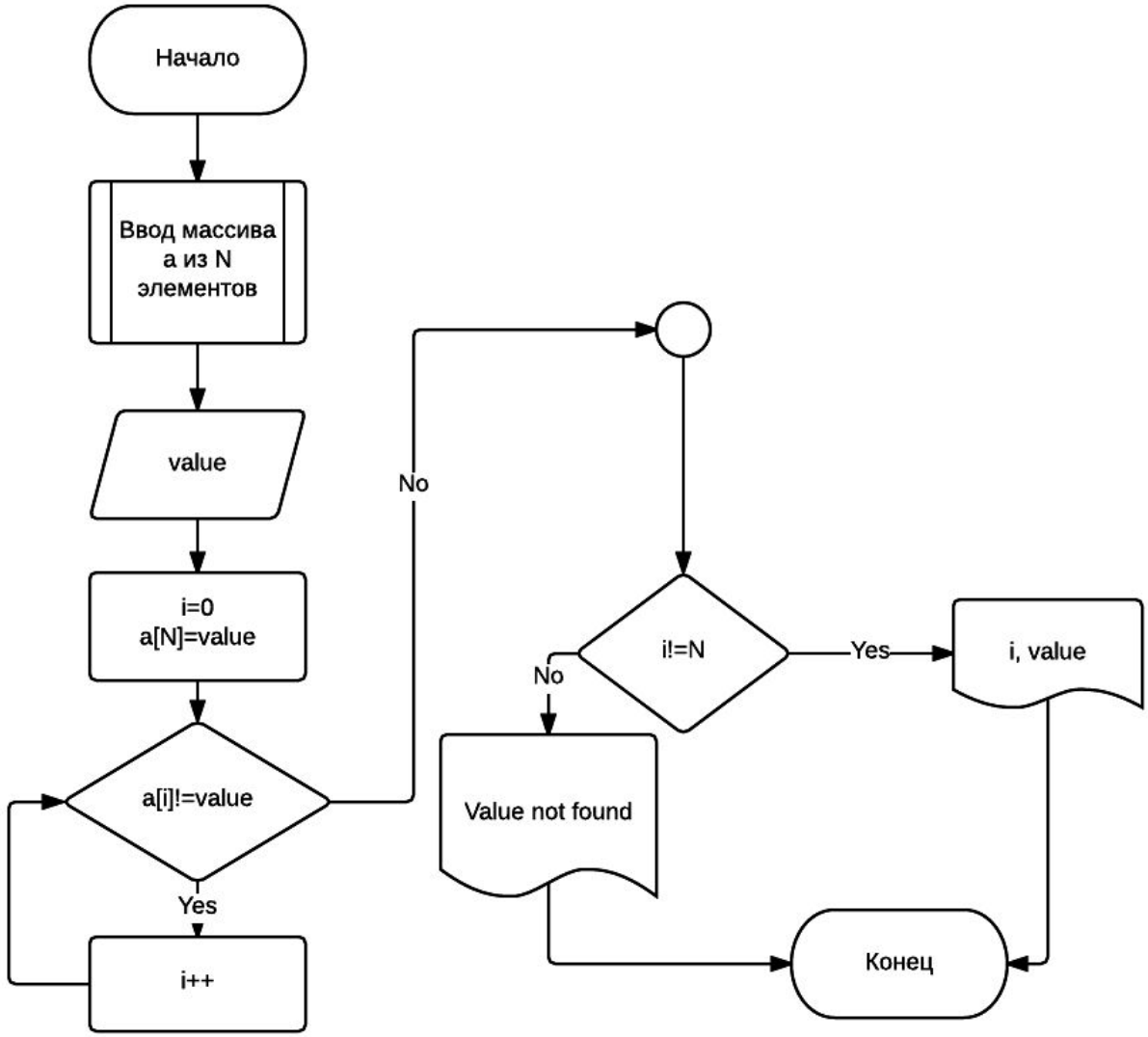
Рассмотрим алгоритм линейного поиска. Это самый простой поиск, который сводится к перебору всех вариантов массива, пока не будет найден искомый. Средняя сложность алгоритма N/2, где N — это размер массива.



```c
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>
#include <malloc.h>

int main(int argc, char *argv[])
{
	int* a;
	int N;
	printf("Input N:");
	scanf("%i", &N);
	a =(int*) malloc(N * sizeof(int));
	int i;
	for (i = 0; i < N; i++)
	{
		printf("Input %d:", i);
		scanf("%d", &a[i]);
	}
	int value;
	printf("Input value for search:");
	scanf("%d", &value);
	i = 0;
	while (i < N && a[i] != value) i++;  // Алгоритм поиска

	if (i != N)
		printf("Index:%d Value:%d", i, a[i]);
	else
		printf("Value not found");

	return 0;
}
```

### Поиск с барьером

Мы можем немного ускорить поиск, если упростим логическое выражение цикла. Правда, для этого нужно, чтобы искомое значение всегда находилось. В этом случае мы должны сами поместить искомое значение в конец нашей последовательности. Не забывайте сделать массив на один элемент больше. Если мы достигли барьера (i == N), значит искомого значения нет в массиве.

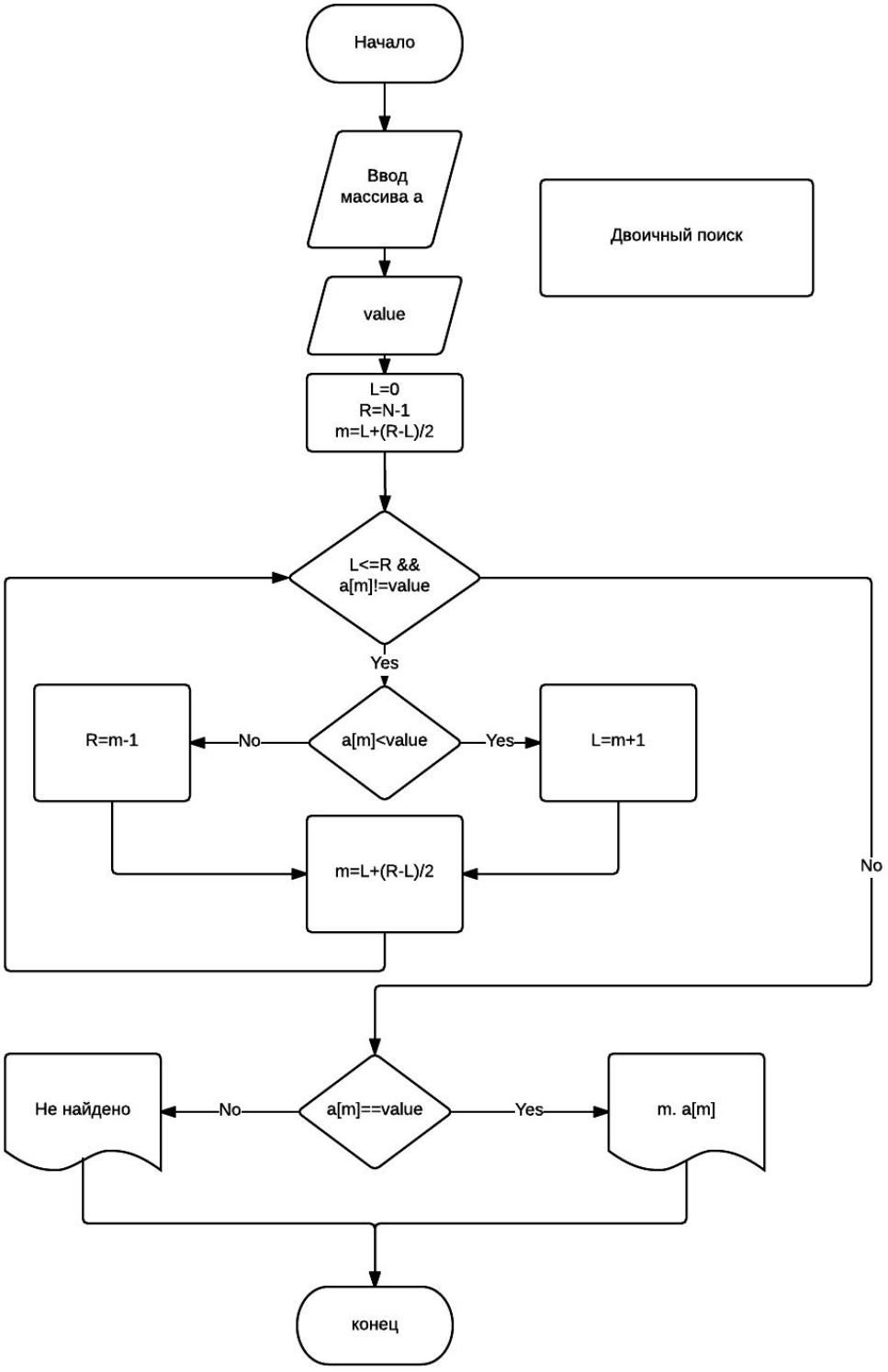



```
    int a[MaxN + 1];
...
    int value;
    printf("Input value for search:");
    scanf("%d", &value);
    a[N] = value;
    i = 0;
    while(a[i] != value) i++;
    if (i != N)
        printf("Index:%d Value:%d", i, a[i]);  // Алгоритм поиска
    else
        printf("Value not found");

    return 0;
```

## Поиск методом половинного деления

Ускорить поиск невозможно, если нет дополнительных данных. Хорошо известно, что поиск может быть гораздо более эффективным, если данные упорядочены. Достаточно представить телефонный справочник.

Выбираем центральный элемент массива с индексом m, сравниваем его с искомым значением. Если он равен искомому значению, то поиск прекращается, если он меньше, то заключаем, что все элементы с индексами, равными или меньшими m, можно игнорировать в дальнейшем поиске, а если он больше, то можно игнорировать все значения индекса, большие или равные m. Такой поиск носит название «поиск делением пополам» (binary search). В нём используются две индексные переменные L и R, отмечающие в массиве левый и правый концы отрезка, в котором искомое значение все ещё может быть найдено:


Начало
Ввод
массива a
value
L=0
R=N-1
m=L+(R-L)/2
L<=R &&
a[m]!=value
Yes
No
a[m]<value
Yes
R=m-1
L=m+1
m=L+(R-L)/2
No
Не найдено
No
a[m]==value
Yes
m. a[m]
конец
Двоичный поиск

```
int value;
printf("Input value for search:");
scanf("%i",&value);
int L = 0, R = N - 1;
int m = L + (R – L) / 2;
while(L <= R && a[m] != value)
{
    if (a[m] < value)
        L = m + 1;
    else
        R = m - 1;
    m = L + (R - L) / 2;
}
if (a[m] == value)
    printf("Index:%d Value:%d", m, a[m]);
else
    printf("Value not found");
```

Подразумевается, что поиск производится в отсортированном массиве.

# Интерполяционный поиск

Бинарный поиск каждый раз ищет целевой элемент массива в середине раздела, интерполяционный пытается ускорить процесс — он старается угадать расположение целевого элемента в массиве по его значению. Предположим, в массиве содержится 1000 элементов со значениями от 1 до 100. Если наша цель — найти число 30, то его нужно искать в районе первой трети массива, где-то рядом с индексом 300. Общее распределение чисел не всегда позволяет получить результат с прицельной точностью, но он может оказаться довольно близким.

Следующий код описывает этот метод:

```
int InterpolationSearch(int *a, int length, int value)
{
    int min = 0;
    int max = length - 1;
    while (min <= max)
    {
    // Находим разделяющий элемент.
        int mid = min + (max - min) * (value - a[min]) / (a[max] - a[min]);
        if (a[mid] == value)
            return mid;
        else if (a[mid] < value)
            min = mid + 1;
        else if (a[mid] > value)
            max = mid - 1;
    }
    return - 1// Элемент не найден
}
```

В данном алгоритме некоторые проблемы остаются нерешёнными. Например, расчет mid может привести к выходу за пределы массива или к тому, что его значение не будет находиться между min и max. Эти задачи вам предлагается решить в домашней работе.

Самая сложная часть приведенного кода — оператор для расчета mid. Чтобы найти его значение, к текущему значению min добавляется расстояние от min до max, масштабируемое по ожидаемой доле расстояния от a[min] до a[max], где должно находиться искомое значение values.

Например, если значение a[min] равно 100, a[max] — 200, a values — 125, определить место для поиска целевого элемента помогут следующие расчеты:

(values - a[min]) / (a[max] - a[min]) = (125 - 100) / (200 - 100) = 25 / 100 = 0.25

Это значит, что новое значение mid должно располагаться на четверти пути от min до max. Если данные распределены очень неравномерно, и вы ищете наихудшее целевое значение, данный алгоритм обладает производительностью O(N). Если распределение относительно равномерное, ожидаемая производительность составит O(log (log N)).

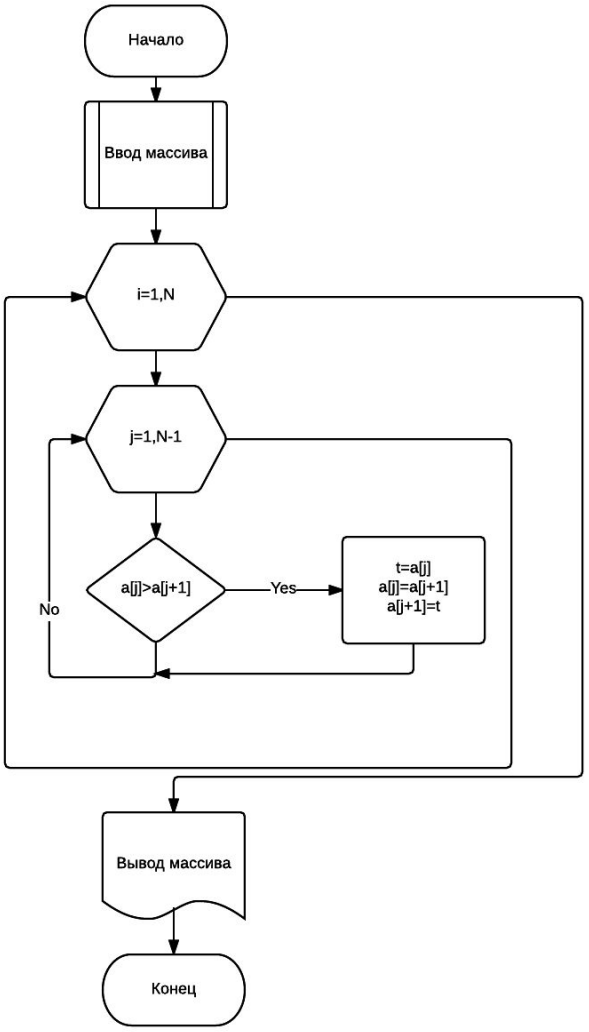
# Сортировка массива

Сортировка — это упорядочение массива по какому-либо признаку. Существует множество различных сортировок, отличающихся друг от друга эффективностью. Давайте познакомимся с некоторыми из них.

## Сортировки

### Пузырьковая сортировка

Пузырьковая сортировка — одна из самых простых для запоминания. В этой сортировке сравниваются два соседних элемента, и, при необходимости, они меняются местами. Таким образом, одни элементы поднимаются наверх, а другие — спускаются вниз.


Начало
Ввод массива
i=1,N
j=1,N-1
a[j]>a[j+1]
Yes
t=a[j]
a[j]=a[j+1]
a[j+1]=t
No
Вывод массива
Конец

```c
#include <stdio.h>
// Определяем максимальный размер массива
#define MaxN 100
void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}
// Распечатываем массив
void print(int N, int *a)
{
    int i;
    for(i = 0; i < N; i++)
        printf("%6i", a[i]);
    printf("\n");
}
int main(int argc, char *argv[]) {
    int a[MaxN];          // создаём массив максимального размера
    int N;
    FILE *in;
    in = fopen("d:\\temp\\data.txt","r");
    fscanf(in, "%i", &N);
    int i;
    for(i = 0; i < N; i++)
    {
        fscanf(in, "%i", &a[i]);
    }
    fclose(in);
    puts("Array before sort");
    print(N, a);
    int j = 0;

    for(i = 0; I < N; i++)
        for(j = 0; j < N - 1; j++)
            if (a[j] > a[j + 1])
            {
                swap(&a[j], &a[j + 1]);
            }
    puts("Array after sort");
    print(N, a);

    return 0;
}
```

## Шейкерная сортировка

Пузырьковая сортировка допускает несложное улучшение. После первого прохода этим методом самый большой элемент массива встанет на своё место. Выполним второй проход наоборот — от предпоследнего элемента до первого. После него встанет на своё место самый маленький элемент. Так и будем выполнять наши проходы массива: нечётные — слева направо и чётные — справа налево. При этом на нечётных проходах будет занимать свое место самый большой элемент (из оставшихся), а при чётных — самый маленький (также из оставшихся).

## Сортировка методом выбора

На первом проходе цикла выбирается минимальный элемент из текущей последовательности, и он меняется местами с первым элементом. На следующей итерации цикла поиск минимального элемента осуществляется со второй позиции, после найденный минимальный элемент меняется местами со вторым в списке. Такую процедуру выполняем до конца массива, пока он весь не будет отсортирован.

```
#include <stdio.h>
#define MaxN 100
void swap(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
void print(int N, int *a)
{
    int i;
    for(i = 0; i < N; i++)
        printf("%6i", a[i]);
    printf("\n");
}
int main(int argc, char *argv[])
{
    int a[MaxN]; // создаём массив максимального размера
    int N;
    FILE *in;
    in = fopen("d:\\temp\\data.txt","r");
    fscanf(in, "%i", &N);
    int i;
    for(i = 0; i < N; i++)
    {
        fscanf(in, "%i", &a[i]);
    }
    fclose(in);
    puts("Array before sort");
    print(N, a);
    int j = 0;
    int jmin;
    for(i = 0; i < N; i++)
    {
        jmin = i;
        for(j = i + 1; j < N; j++)
            if (a[j] < a[jmin])
            {
                jmin = j;
            }
        swap(&a[i], &a[jmin]);
    }
    puts("Array after sort");
    print(N, a);
    return 0;
}
```

### Сортировка вставками

Сортировка вставками (*Insertion sort*) — простой алгоритм сортировки, при которой очередной элемент всегда добавляется в конец списка, а затем перемещается в начало до тех пор, пока он меньше предыдущего элемента.

Таким образом находится правильное место для вставки (пример — сортировка карточек в библиотеке). Этот алгоритм будет иметь следующую последовательность шагов.

1. **Шаг инициализации.** Массив из одного элемента является уже отсортированным по возрастанию и выступает в виде начального списка.

2. **Шаг итерации.** Для всех последующих элементов со второго по n−1 очередной элемент i сравнивается со всеми предыдущими от 0 до (i − 1), пока текущий элемент i меньше или равен предыдущему. После этого будет найдена позиция для вставки или достигнуто начало списка. Далее элемент вставляется в найденную позицию.

3. **Шаг выхода**. Шаг итерации продолжается до тех пор, пока не будут просмотрены все элементы массива.

```
for (int i = 0; i < N; i++)
{
    int temp = a[i];
    int j = i;
    while (j > 0 && a[j - 1] > temp)
    {
        swap(&a[j], &a[j - 1]);
        j--;
    }
}
```

## Простейший подсчёт производительности программы

Производительность в чистом виде — это количество элементарных операций, которые необходимо выполнить для получения результата в зависимости от размера обрабатываемых данных. Для сортировки размер обрабатываемых данных — это длина сортируемого массива. В качестве элементарных операций разумно взять операции сравнения и присваивания. Суммарное количество таких операций и будет характеризовать производительность.

Для простейшего подсчёта производительности можно ввести переменную, которую следует увеличивать на единицу при каждой операции, которые мы хотим подсчитать. Мы можем легко подсчитать количество операций сравнений в пузырьковой сортировке:

```
int count = 0;                  // Ввели счётчик количества операций
for (i = 0; i < N; i++)
   for (j = 0; j < N - 1; j++)
   {
      count++;
      if (a[j] > a[j + 1])
      {
         count++;
         swap(&a[j], &a[j + 1]);
      }
   }
puts("Array after sort");
Print(N, a);
printf("Count:%d", count); // Выводим счётчик на экран
```

## *Заполнение массива по строкам и по столбцам

Это неочевидно, но оказывается, что скорость заполнения массива зависит от того, каким способом мы его заполняем — по строкам или по столбцам.

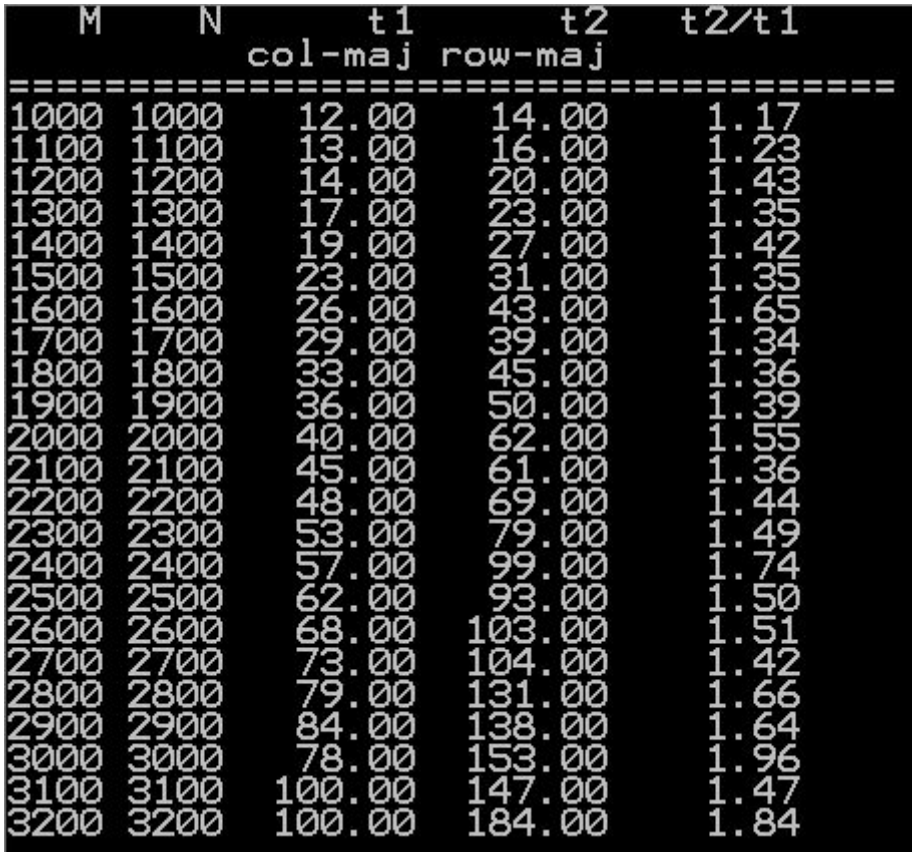
Рассмотрим это на примере:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h>
#include <math.h>

#define MINDIM_M    1000
#define MINDIM_N    1000
#define MINDIM_K    1000
#define MAXDIM_M    3300
#define MAXDIM_N    3300
#define MAXDIM_K    3300
#define INC_M   100
#define INC_N   100
#define INC_K   100
#define MIN_T   1

struct timeval tv1, tv2, dtv;
struct timezone tz;

// Запоминаем в глобальных переменных текущее время
void time_start()
{
   gettimeofday(&tv1, &tz);
}
// Рассчитываем время прошедшее с момента запуска time_start()
long time_stop()
{
   gettimeofday(&tv2, &tz);
   dtv.tv_sec = tv2.tv_sec - tv1.tv_sec;
   dtv.tv_usec = tv2.tv_usec - tv1.tv_usec;
```

```
    if(dtv.tv_usec<0)
    {
        dtv.tv_sec--;
        dtv.tv_usec += 1000000;
    }
    return dtv.tv_sec * 1000 + dtv.tv_usec / 1000;
}

double buffer1[MAXDIM_M*MAXDIM_N];
double buffer2[MAXDIM_M*MAXDIM_N];

void
initMatrix(size_t m, size_t n, double *A, size_t incRowA, size_t incColA)
{
    for (size_t j = 0; j < n; ++j)
        for (size_t i = 0; i < m; ++i)
            A[I * incRowA + j * incColA] = j * n + I + 1;
}

void printMatrix(size_t m, size_t n, const double *A,
                 size_t incRowA, size_t incColA)
{
    for (size_t i = 0; i < m; ++i)
    {
        printf("   ");
        for (size_t j = 0; j < n; ++j)
            printf("%4.1lf ", A[i*incRowA+j*incColA]);
        printf("\n");
    }
    printf("\n");
}

int main() {
    printf("   M    N      t1      t2   t2/t1\n");
    printf("          col-maj row-maj\n");
    printf("=====================================\n");

    for (size_t m = MINDIM_M, n = MINDIM_N; m < MAXDIM_M && n < MAXDIM_N;
         m += INC_M, n += INC_N)
    {
        size_t runs = 0;
        double t1 = 0;
        do {
            time_start();
            initMatrix(m, n, buffer1, 1, m);
            t1 += time_stop();
            ++runs;
        } while (t1 < MIN_T);
        t1 /= runs;

        runs = 0;
        double t2 = 0;
        do {
            time_start();
            initMatrix(m, n, buffer2, n, 1);
            t2 += time_stop();
            ++runs;
        } while (t2 < MIN_T);
```

```
        t2 /= runs;

        printf("%4d %4d %7.2lf %7.2lf %7.2lf\n", m, n, t1, t2, t2/t1);
    }
}
```

Результат выполнения программы:



| M | N | t1 col-maj | t2 row-maj | t2/t1 |
|---|---|---|---|---|
| 1000 | 1000 | 12.00 | 14.00 | 1.17 |
| 1100 | 1100 | 13.00 | 16.00 | 1.23 |
| 1200 | 1200 | 14.00 | 20.00 | 1.43 |
| 1300 | 1300 | 17.00 | 23.00 | 1.35 |
| 1400 | 1400 | 19.00 | 27.00 | 1.42 |
| 1500 | 1500 | 23.00 | 31.00 | 1.35 |
| 1600 | 1600 | 26.00 | 43.00 | 1.65 |
| 1700 | 1700 | 29.00 | 39.00 | 1.34 |
| 1800 | 1800 | 33.00 | 45.00 | 1.36 |
| 1900 | 1900 | 36.00 | 50.00 | 1.39 |
| 2000 | 2000 | 40.00 | 62.00 | 1.55 |
| 2100 | 2100 | 45.00 | 61.00 | 1.36 |
| 2200 | 2200 | 48.00 | 69.00 | 1.44 |
| 2300 | 2300 | 53.00 | 79.00 | 1.49 |
| 2400 | 2400 | 57.00 | 99.00 | 1.74 |
| 2500 | 2500 | 62.00 | 93.00 | 1.50 |
| 2600 | 2600 | 68.00 | 103.00 | 1.51 |
| 2700 | 2700 | 73.00 | 104.00 | 1.42 |
| 2800 | 2800 | 79.00 | 131.00 | 1.66 |
| 2900 | 2900 | 84.00 | 138.00 | 1.64 |
| 3000 | 3000 | 78.00 | 153.00 | 1.96 |
| 3100 | 3100 | 100.00 | 147.00 | 1.47 |
| 3200 | 3200 | 100.00 | 184.00 | 1.84 |

Когда мы пробегаем по колонкам, процессор забирает к себе сразу несколько рядом стоящих значений (кеширует), а когда по строкам, то приходится каждый раз обращаться к памяти, чтобы взять следующее значение.

# Домашнее задание

1. Попробовать оптимизировать пузырьковую сортировку. Сравнить количество операций сравнения оптимизированной и не оптимизированной программы. Написать функции сортировки, которые возвращают количество операций.

2. *Реализовать шейкерную сортировку.

3. Реализовать бинарный алгоритм поиска в виде функции, которой передается отсортированный массив. Функция возвращает индекс найденного элемента или -1, если элемент не найден.

4. *Подсчитать количество операций для каждой из сортировок и сравнить его с асимптотической сложностью алгоритма.

*Записывайте в начало программы условие и свою фамилию. Все решения создавайте в одной программе. Со звёздочками выполняйте в том случае, если вы решили задачи без звёздочек.*

# Дополнительные материалы

1. Сортировка пузырьком
2. Сортировка выбором
3. Сортировка перемешиванием
4. Вычислительная сложность

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:


1. Род Стивенс, Алгоритмы. Теория и практическое применение. Издательство «Э», 2016.
2. Никлаус Вирт, Алгоритмы и структуры данных. Новая версия для Оберона, ДМК, Москва, 2010.
3. Пол Дейтел, Харви Дейтел. C для программистов. C введением в C11, ДМК, Москва, 2014.