МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

**Реферат на тему: “Виды сортировок и анализ их эффективности”**

Выполнили:

Студентки 1 курса, 5 группы,

Буданова К. А. и Николаева Е. В.

Специальность ПОИТ.

Преподаватель:

Белодед Николай Иванович

Оглавление

[Введение. 3](#_Toc72422779)

[Виды сортировок. 3](#_Toc72422780)

[Пузырьковая (Bubble) 5](#_Toc72422781)

[Шейкерная (Shaker) 5](#_Toc72422782)

[Чёт-нечёт (Odd-even) 6](#_Toc72422783)

[Сортировка выбором (Selection Sort) 7](#_Toc72422784)

[Сортировка вставками (Insertion Sort) 7](#_Toc72422785)

[Пирамидальная сортировка (Heap Sort) 8](#_Toc72422786)

[Сортировка слиянием (Merge Sort) 8](#_Toc72422787)

[Сортировка Шелла (ShellSort) 9](#_Toc72422788)

[Сортировка расчёской (Comb Sort) 10](#_Toc72422789)

[Быстрая сортировка (Quick Sort) 10](#_Toc72422790)

[Проверка эффективности 11](#_Toc72422791)

[Вывод 11](#_Toc72422792)

# Введение.

Очень часто программисту и не только необходимо быстро найти какие-либо данные или значения в массивах, либо отсортировать массив по некоторым параметрам (по убыванию или по возрастанию, например). Можно, конечно, и самому подумать и набросать код, который будет работать. Но почти всё уже придумано до вас. Более того  – все известные алгоритмы поиска и сортировки много раз улучшены и оптимизированы, как по эффективности так и по скорости работы.

В данном реферате мы рассмотрим и сравним несколько видов сортировок, но для начала введем его понятие и его цели.

**Сортировка массива** — это процесс распределения всех элементов [**массива**](https://ravesli.com/urok-74-massivy-chast-1/) в определенном порядке. Очень часто это бывает полезным. Например, в вашем почтовом ящике электронные письма отображаются в зависимости от времени получения; новые письма считаются более релевантными, чем те, которые вы получили полчаса, час, два или день назад; когда вы переходите в свой список контактов, имена обычно находятся в алфавитном порядке, потому что так легче что-то найти. Все эти случаи включают в себя сортировку данных перед их фактическим выводом.  
От сюда и вытекает ***цель сортировки*** — облегчить последующий поиск элементов в отсортированном множестве при обработке данных.

# Виды сортировок.

На данный момент существует множество алгоритмов сортировок и с каждым днём их количество увеличивается.

Ниже представлен список рассматриваемых типов сортировок, которые будут рассмотрены далее:

1. **Пузырьковая** (Bubble)
2. **Шейкерная** (Shaker)
3. **Чёт-нечет** (Odd-even)
4. **Выбором** (Selection)
5. **Вставками** (Insertion)
6. **Пирамидная** (Heap)
7. **Слиянием** (Merge)
8. **Шелла** (Shell)
9. **Расчёской** (Comb)
10. Быстрая **сортировка Хоара** (Quick)

# Пузырьковая (Bubble)

**Сортировка пузырьком** — один из самых известных алгоритмов сортировки. Здесь нам нужно последовательно сравнивать значения соседних элементов и менять числа местами, если предыдущее оказывается больше последующего. Таким образом элементы с большими значениями оказываются в конце списка, а с меньшими остаются в начале.

Этот алгоритм считается учебным и почти не применяется на практике из-за низкой эффективности: он медленно работает на тестах, в которых маленькие элементы стоят в конце массива. Однако на нём основаны многие другие методы, например, шейкерная сортировка и сортировка расчёской.

Пример №1:

void BubbleSort(int a[], int n) {

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

for (int j = 0; j < n - i; ++j) //with ++j

if (a[j] > a[j + 1])

swap(a[j], a[j + 1]);}

Пример №2:

void BubbleSort1(int a[], int n) {

for (int i = 0; i < n - 1; ++i)

for (int j = n - 1; j > i; --j) //with --j

if (a[j] < a[j - 1])

swap(a[j], a[j - 1]);}

Модификация:

void BubbleModificated(int a[], int n) {

for (int i = 0; i < n - 1; ++i)

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

if (a[j] < a[i])

swap(a[j], a[i]);}

Данная модификация представляет из себя скрещение пузырьковой сортировки и сортировки выбором.

При пузырьковой сортировке каждый элемент во внутреннем цикле сравнивается со своим соседом слева и при необходимости переставляется. В модификации сравнение происходит с текущим элементом внешнего цикла. Как показали практические замеры, в результате для больших массивов случайных чисел (примерно >6000) сортировка проходит в разы быстрее, лишь немногим уступая более сложным сортировкам выбора и вставками. Однако на массивах менее 5000 элементов классическая эффективнее.

# Шейкерная (Shaker)

Шейкерная сортировка отличается от пузырьковой тем, что она двунаправленная: алгоритм перемещается не строго слева направо, а сначала слева направо, затем справа налево.

Преимущество данной сортировки перед пузырьковой в том, что она одинаково легко справляется с большими элементами в начале исходного массива и маленькими элементами в конце. Минус: потеря времени на служебные операции и более сложная организация. Теоретически может быть применена для обработки некоторых экзотических случаев, но в целом считается учебной. Эффективность значительно ниже сортировки выбором. Интереса не представляет.

void ShakerSort(int a[], int n){

int left, right, i;

left = 0;

right = n - 1;

while (left <= right){

for (i = right; i >= left; i--){

if (a[i - 1] > a[i]){

int temp = a[i - 1];

a[i - 1] = a[i];

a[i] = temp;}}

left++;

for (i = left; i <= right; i++){

if (a[i - 1] > a[i]){

int temp = a[i - 1];

a[i - 1] = a[i];

a[i] = temp;}}

right--;}}

# Чёт-нечёт (Odd-even)

Является модификацией пузырьковой сортировки, разработана для использования на параллельных процессорах. Но если у нас нет соответствующего инструмента управления распределением нагрузки, то получающаяся скорость лишь немного выше пузырьковой, заметно проигрывая сортировке выбором.  
Суть в том, чтобы независимо на разных ядрах прокручивать циклы по элементам с чётными и нечётными индексами.

Пример №1:

void OddEvenSort(int a[], int n) {

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

for (int j = (i & 1) ? 1 : 2; j < n; j += 2) //with (i&1)

if (a[j - 1] > a[j])

swap(a[j - 1], a[j]);}

Пример №2:

void OddEvenSort(int a[], int n) {

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

for (int j = (i % 2) ? 1 : 2; j < n; j += 2) //with (i%2)

if (a[j - 1] > a[j])

swap(a[j - 1], a[j]);}

# Сортировка выбором (Selection Sort)

Суть сортировки сводится к алгоритму, что пробегает по всем ещё не упорядоченным элементам, ищет среди них минимальный (сохраняя его значение и индекс) и переставляет в конец выстриваемого списка. Этот нужно повторять до тех пор, пока в массиве не закончатся неотсортированные подмассивы.  
void selectSort(int a[], int size)

{int k, i, j;

for (i = 0; i < size - 1; i++)

{for (j = i + 1, k = i; j < size; j++)

if (a[j] < a[k])

k = j;

int c = a[k];

a[k] = a[i];

a[i] = c;}}

# Сортировка вставками (Insertion Sort)

Этот алгоритм состоит из трёх шагов. Сначала из массива нужно выбрать один элемент — его обычно называют опорным. Затем другие элементы в массиве перераспределяют так, чтобы элементы меньше опорного оказались до него, а большие или равные — после. А дальше рекурсивно применяют первые два шага к подмассивам справа и слева от опорного значения.

Данную сортировку изобрели в 1960 году для машинного перевода: тогда словари хранились на магнитных лентах, а сортировка слов обрабатываемого текста позволяла получить переводы за один прогон ленты, без перемотки назад.

void insertSort(int\* a, int size)

{ int t, i, j;

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

{

t = a[i];

for (j = i - 1; j >= 0 && a[j] > t; j--)

a[j + 1] = a[j];

a[j + 1] = t; }}

# Пирамидальная сортировка (Heap Sort)

Работает на принципе сортирующего дерева, которое организуется прямо в исходном массиве. Состоит из двух шагов:  
**На первом** в массиве происходит относительно небольшое число перестановок, чтобы сформировать исходное дерево, с которым далее можно работать. В результате на вершине дерева (на первом месте массива) оказывается самый большой элемент.  
**На втором** шаге вершина дерева переставляется в конец, откуда и будет выстраиваться отсортированный подмассив. А среди оставшихся элементов происходит ряд перестановок с целью восстановить дерево, чтобы в его вершине вновь оказался самый большой элемент из ещё не отсортированной части. Таким образом на первом месте постоянно оказывается один из самых маленьких элементов, который затем серией перестановок отправляется назад, но несмотря на кажущуюся неказистость в целом алгоритм высокоэффективен.

void heapify(int A[], int pos, int n) {

int t, tm;

while (2 \* pos + 1 < n) {

t = 2 \* pos + 1;

if (2 \* pos + 2 < n && A[2 \* pos + 2] >= A[t])

t = 2 \* pos + 2;

if (A[pos] < A[t]) {

tm = A[pos];

A[pos] = A[t];

A[t] = tm;

pos = t;

}

else break;

}

}

void PiramSort(int A[], int n)

{ for (int i = n - 1; i >= 0; i--)

heapify(A, i, n);

while (n > 0)

{ int tm = A[0];

A[0] = A[n - 1];

A[n - 1] = tm;

n--;

heapify(A, 0, n);}}

# Сортировка слиянием (Merge Sort)

Исходный массив делится надвое на всё меньшие подмассивы, пока количество элементов в очередных не станет равным 2 или 1. Если в подмассиве 2 элемента, то он упорядочивается банальным сравнением. А подмассив из одного элемента по своей сути является упорядоченным. Затем происходит обратный процесс - слияние подмассивов. Поскольку подмассивы к этому времени являются упорядоченными, то можем сравнивать лишь элементы, стоящие в их начале. И отслеживаем, чтобы подмассив не иссяк.

void insOrd(int m[], int sm, int em, int e)

{ int i = sm;

while (i <= em && m[i] < e)

{ if (i - 1 >= sm)

m[i - 1] = m[i];

i++;}   
 m[i - 1] = e;}

void Merge(int m[], int sm, int cm, int em)

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

{ if (m[i] > m[cm + 1])

{ int buf = m[i];

m[i] = m[cm + 1];

insOrd(m, cm + 1, em, buf);}}}

void SortMerge(int m[], int lm, int sm = 0)

{ if (lm > 1)

{ SortMerge(m, lm / 2, sm);

SortMerge(m, lm - lm / 2, sm + lm / 2);

Merge(m, sm, sm + lm / 2 - 1, sm + lm - 1);};}

# Сортировка Шелла (ShellSort)

Существенно улучшенная модификация сортировки вставками. Получается из неё таким же образом, как сортировка расчёской из пузырьковой.  
  
В алгоритме два цикла. Внутренний переставляет элементы. Внешний служит для изменения промежутка, через который внутренний цикл элементы будет переставлять.  Внутренний цикл функционирует по принципу сортировки вставками, но обмен происходит не между соседними элементами, а между элементами, отстоящими друг от друга на промежутке. Промежутки постепенно сокращаются до 1 - и тогда алгоритм Шелла превращается в обычную сортировку вставками.  
  
Общий смысл в том, что прежде чем запустить полноценную сплошную сортировку, массив следует привести в относительный порядок, позволяя элементам с большими значениями легко перепрыгивать к концу списка, а с маленькими значениями к его началу. По мере того, как массив становится всё более и более упорядоченным, величины прыжков сокращаются. Почти отсортированный массив легко отшлифовывается.

int\* SortShell(int m[], int lm)

{ int buf; bool sort;

for (int g = lm / 2; g > 0; g /= 2)

do {

sort = false;

for (int i = 0, j = g; j < lm; i++, j++)

if (m[i] > m[j]) {

sort = true; buf = m[i];

m[i] = m[j]; m[j] = buf;}

} while (sort); return m;};

# Сортировка расчёской (Comb Sort)

Сортировка расчёской — улучшение сортировки пузырьком. Её идея состоит в том, чтобы «устранить» элементы с небольшими значения в конце массива, которые замедляют работу алгоритма. Если при пузырьковой и шейкерной сортировках при переборе массива сравниваются соседние элементы, то при «расчёсывании» сначала берётся достаточно большое расстояние между сравниваемыми значениями, а потом оно сужается вплоть до минимального.

Первоначальный разрыв нужно выбирать не случайным образом, а с учётом специальной величины — фактора уменьшения, оптимальное значение которого равно 1,247. Сначала расстояние между элементами будет равняться размеру массива, поделённому на 1,247; на каждом последующем шаге расстояние будет снова делиться на фактор уменьшения — и так до окончания работы алгоритма.

void CombSort(int\* m, int n) {

int tmp, k;

int s = n; // готовим начальный шаг

long long cnt = 0;

while (n > 1) {s /= 1.247f; // шаг на этом проходе. На первом проходе получается примерно 80% от размера массива,

// и легкие элементы в хвосте переносятся в первые 20%

if (s < 1) s = 1; // 0 быть не может, присвоим 1

k = 0; // нет перестановок

for (int i = 0; i + s < n; ++i) { // двигаемся, пока сравниваемый элемент (на s от текущего) в массиве

if (m[i] / 10 > m[i + s] / 10) {

tmp = m[i];

m[i] = m[i + s];

m[i + s] = tmp;

k = i;}

++cnt;}

if (s == 1) // шаг 1, как в обычном пузырьке. Включаем контроль

n = k + 1;}}

# Быстрая сортировка Хоара (Quick Sort)

Существенно улучшенная модификация пузырьковой сортировки.  Любопытный факт: усовершенствование самого неэффективного метода в результате дало один из самых эффективных.  
  
Суть: Один из элементов массива назначается опорным. Элементы массивы переставляются таким образом, чтобы все, которые меньше опорного, оказались левее от него, а которые больше - правее. Для каждого из подмассивов операция рекурсивно повторяется. Эффективность алгоритма зависит главным образом от того, насколько удачно будет подбираться этот опорный элемент. Идеален случай, когда он постоянно делит очередные подмассивы на равные части, но если каждый раз высчитывать, то время будет теряться на сами рассчёты. Различные модификации в основном отличаются друг от друга способом его выбора.

int getHoarBorder(int A[], int sm, int em)

{ int i = sm - 1, j = em + 1;

int brd = A[sm];

int buf;

while (i < j)

{ while (A[--j] > brd);

while (A[++i] < brd);

if (i < j)

{ buf = A[j];

A[j] = A[i];

A[i] = buf;};}

return j;}

void sortHoar(int A[], int sm, int em)

{ if (sm < em)

{ int hb = getHoarBorder(A, sm, em);

sortHoar(A, sm, hb);

sortHoar(A, hb + 1, em);}};

# Проверка эффективности

# Алгоритмы сортировки оцениваются по скорости выполнения и эффективности использования памяти: Время — основной параметр, характеризующий быстродействие алгоритма. Называется также [вычислительной сложностью](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C). Для упорядочения важны *худшее*, *среднее* и *лучшее* поведение алгоритма в терминах мощности входного множества A. Память — ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных. При оценке не учитывается место, которое занимает исходный массив и независящие от входной последовательности затраты, например, на хранение кода программы. Алгоритмы сортировки, не потребляющие дополнительной памяти, относят к *сортировкам на месте*.

Приведённый ниже код будет реализовывать функцию подсчёта времени выполнения сортировки в зависимости от количества элементов, которые с каждым шагом будут увеличиваться на 1000. Диапазон элементов от 10000 до 50000.

#include <iostream>

#include <time.h>

#include <stdlib.h>

# define N 10

using namespace std;

clock\_t start;

double duration;

void BubbleSort(int a[], int n) {

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

for (int j = 0; j < n - i; ++j) //with ++j

if (a[j] > a[j + 1])

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

}

void BubbleSort1(int a[], int n) {

for (int i = 0; i < n - 1; ++i)

for (int j = n - 1; j > i; --j) //with --j

if (a[j] < a[j - 1])

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

}

void BubbleModificated(int a[], int n) {

for (int i = 0; i < n - 1; ++i)

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

if (a[j] < a[i])

swap(a[j], a[i]);

}

void ShakerSort(int a[], int n)

{

int left, right, i;

left = 0;

right = n - 1;

while (left <= right)

{

for (i = right; i >= left; i--)

{

if (a[i - 1] > a[i])

{

int temp = a[i - 1];

a[i - 1] = a[i];

a[i] = temp;

}

}

left++;

for (i = left; i <= right; i++)

{

if (a[i - 1] > a[i])

{

int temp = a[i - 1];

a[i - 1] = a[i];

a[i] = temp;

}

}

right--;

}

}

void OddEvenSort(int a[], int n) {

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

for (int j = (i & 1) ? 1 : 2; j < n; j += 2) //with (i&1)

if (a[j - 1] > a[j])

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

}

void OddEvenSort1(int a[], int n) {

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

for (int j = (i % 2) ? 1 : 2; j < n; j += 2) //with (i%2)

if (a[j - 1] > a[j])

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

}

void selectSort(int a[], int size)

{

int k, i, j;

for (i = 0; i < size - 1; i++)

{

for (j = i + 1, k = i; j < size; j++)

if (a[j] < a[k])

k = j;

int c = a[k];

a[k] = a[i];

a[i] = c;

}

}

void insertSort(int\* a, int size)

{

int t, i, j;

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

{

t = a[i];

for (j = i - 1; j >= 0 && a[j] > t; j--)

a[j + 1] = a[j];

a[j + 1] = t;

}

}

void heapify(int A[], int pos, int n)

{

int t, tm;

while (2 \* pos + 1 < n)

{

t = 2 \* pos + 1;

if (2 \* pos + 2 < n && A[2 \* pos + 2] >= A[t])

t = 2 \* pos + 2;

if (A[pos] < A[t])

{

tm = A[pos];

A[pos] = A[t];

A[t] = tm;

pos = t;

}

else break;

}

}

void PiramSort(int A[], int n)

{

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

heapify(A, i, n);

while (n > 0)

{

int tm = A[0];

A[0] = A[n - 1];

A[n - 1] = tm;

n--;

heapify(A, 0, n);

}

}

void insOrd(int m[], int sm, int em, int e)

{

int i = sm;

while (i <= em && m[i] < e)

{

if (i - 1 >= sm)

m[i - 1] = m[i];

i++;

} m[i - 1] = e;

}

void Merge(int m[], int sm, int cm, int em)

{

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

{

if (m[i] > m[cm + 1])

{

int buf = m[i];

m[i] = m[cm + 1];

insOrd(m, cm + 1, em, buf);

}

}

}

void SortMerge(int m[], int lm, int sm = 0)

{

if (lm > 1)

{

SortMerge(m, lm / 2, sm);

SortMerge(m, lm - lm / 2, sm + lm / 2);

Merge(m, sm, sm + lm / 2 - 1, sm + lm - 1);

};

}

void SortShell(int m[], int lm)

{

int buf; bool sort;

for (int g = lm / 2; g > 0; g /= 2)

do {

sort = false;

for (int i = 0, j = g; j < lm; i++, j++)

if (m[i] > m[j]) {

sort = true; buf = m[i];

m[i] = m[j]; m[j] = buf;

}

} while (sort);

};

void CombSort(int\* m, int n)

{

int tmp, k;

int s = n; // готовим начальный шаг

long long cnt = 0;

while (n > 1) {

s /= 1.247f; // шаг на этом проходе. На первом проходе получается примерно 80% от размера массива,

// и легкие элементы в хвосте переносятся в первые 20%

if (s < 1) s = 1; // 0 быть не может, присвоим 1

k = 0; // нет перестановок

for (int i = 0; i + s < n; ++i) { // двигаемся, пока сравниваемый элемент (на s от текущего) в массиве

if (m[i] / 10 > m[i + s] / 10) {

tmp = m[i];

m[i] = m[i + s];

m[i + s] = tmp;

k = i;

}

++cnt;

}

if (s == 1) // шаг 1, как в обычном пузырьке. Включаем контроль

n = k + 1;

}

}

int getHoarBorder(int A[], int sm, int em)

{

int i = sm - 1, j = em + 1;

int brd = A[sm];

int buf;

while (i < j)

{

while (A[--j] > brd);

while (A[++i] < brd);

if (i < j)

{

buf = A[j];

A[j] = A[i];

A[i] = buf;

};

}

return j;

}

void sortHoar(int A[], int sm, int em)

{

if (sm < em)

{

int hb = getHoarBorder(A, sm, em);

sortHoar(A, sm, hb);

sortHoar(A, hb + 1, em);

}

};

void main()

{

setlocale(LC\_CTYPE, "Russian");

int A[N];

int\* C = new int[N];

for (int p = 10000; p <= 50000; p += 10000)

{

int\* a = new int[p], \* b = new int[p];

srand(time(0));

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

{

a[i] = rand() % (p \* 10);

}

memcpy(b, a, p \* sizeof(int));

start = clock(); BubbleSort(b, p);

duration = (clock() - start) / (double)CLOCKS\_PER\_SEC;

cout << "Пузырьковая сортировка (1). Прошло " << duration << " секунд" << endl;

cout << endl;

memcpy(b, a, p \* sizeof(int));

start = clock(); BubbleSort1(b, p);

duration = (clock() - start) / (double)CLOCKS\_PER\_SEC;

cout << "Пузырьковая сортировка (2). Прошло " << duration << " секунд" << endl;

cout << endl;

memcpy(b, a, p \* sizeof(int));

start = clock(); BubbleModificated(b, p);

duration = (clock() - start) / (double)CLOCKS\_PER\_SEC;

cout << "Пузырьковая сортировка (3). Прошло " << duration << " секунд" << endl;

cout << endl;

memcpy(b, a, p \* sizeof(int));

start = clock(); ShakerSort(b, p);

duration = (clock() - start) / (double)CLOCKS\_PER\_SEC;

cout << "Шейкерная сортировка. Прошло " << duration << " секунд" << endl;

cout << endl;

memcpy(b, a, p \* sizeof(int));

start = clock(); OddEvenSort(b, p);

duration = (clock() - start) / (double)CLOCKS\_PER\_SEC;

cout << "Сортировка чёт-нечёт (1). Прошло " << duration << " секунд" << endl;

cout << endl;

memcpy(b, a, p \* sizeof(int));

start = clock(); OddEvenSort1(b, p);

duration = (clock() - start) / (double)CLOCKS\_PER\_SEC;

cout << "Сортировка чёт-нечёт (2). Прошло " << duration << " секунд" << endl;

cout << endl;

memcpy(b, a, p \* sizeof(int));

start = clock(); selectSort(b, p);

duration = (clock() - start) / (double)CLOCKS\_PER\_SEC;

cout << "Сортировка выбором. Прошло " << duration << " секунд" << endl;

cout << endl;

memcpy(b, a, p \* sizeof(int));

start = clock(); insertSort(b, p);

duration = (clock() - start) / (double)CLOCKS\_PER\_SEC;

cout << "Сортировка вставками. Прошло " << duration << " секунд" << endl;

cout << endl;

memcpy(b, a, p \* sizeof(int));

start = clock(); PiramSort(b, p);

duration = (clock() - start) / (double)CLOCKS\_PER\_SEC;

cout << "Пирамидальная сортировка. Прошло " << duration << " секунд" << endl;

cout << endl;

memcpy(b, a, p \* sizeof(int));

start = clock(); SortMerge(b, p);

duration = (clock() - start) / (double)CLOCKS\_PER\_SEC;

cout << "Сортировка слиянием. Прошло " << duration << " секунд" << endl;

cout << endl;

memcpy(b, a, p \* sizeof(int));

start = clock(); SortShell(b, p);

duration = (clock() - start) / (double)CLOCKS\_PER\_SEC;

cout << "Сортировка Шелла. Прошло " << duration << " секунд" << endl;

cout << endl;

memcpy(b, a, p \* sizeof(int));

start = clock(); CombSort(b, p);

duration = (clock() - start) / (double)CLOCKS\_PER\_SEC;

cout << "Сортировка расчёской. Прошло " << duration << " секунд" << endl;

cout << endl;

memcpy(b, a, p \* sizeof(int));

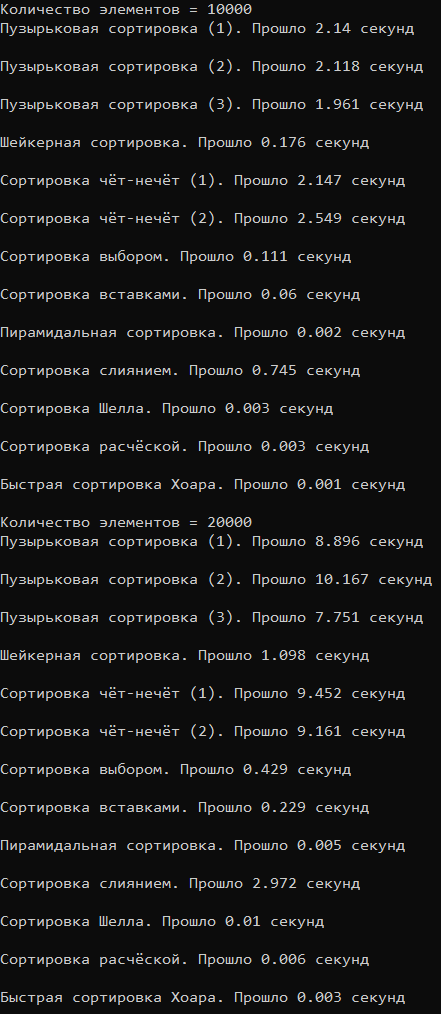
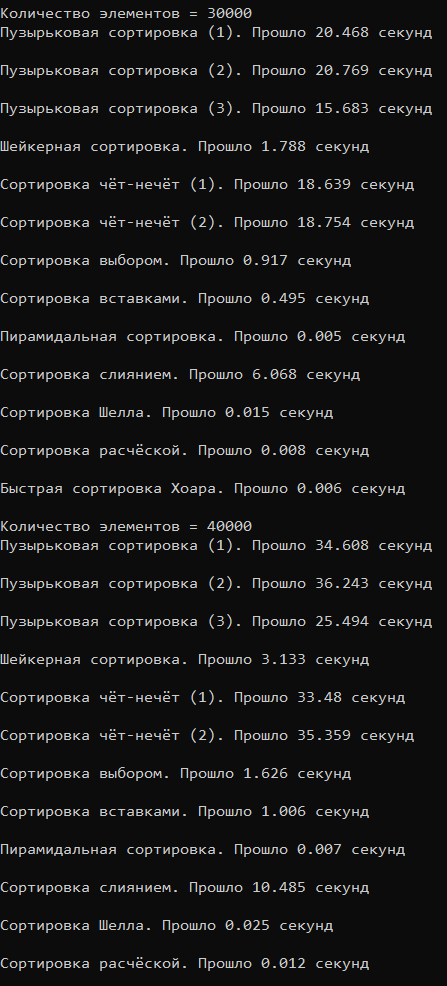
start = clock(); sortHoar(b, 1, p - 1);

duration = (clock() - start) / (double)CLOCKS\_PER\_SEC;

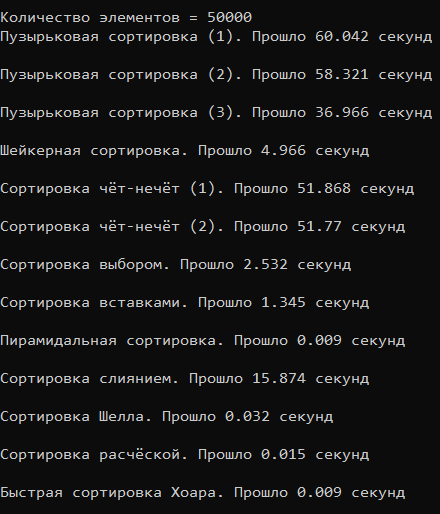
cout << "Быстрая сортировка Хоара. Прошло " << duration << " секунд" << endl;

cout << endl;}}

**Результаты:**



# 



# Вывод

Составим график с количеством элементов массива по горизонтали и времени сортировки в секундах по вертикали:

Как видно из графика, вначале показатели не сильно и отличаются, но с увеличением количества элементов видно разделение и можно понять, какой из типов сортировки является самым быстрым.

Составим рейтинг(топ):

1. Сортировка Хоара
2. Пирамидальная
3. Расческой
4. Шелла
5. Сортировка Вставками
6. Сортировка Выбором
7. Шейкерная
8. Сортировка Слиянием
9. Пузырьковая(3)
10. Чет-нечет(2)
11. Черт-нечет(1)
12. Пузырьковая(2)
13. Пузырьковая(1)

После 20000 элементов выделяются такие типы сортировки как Сортировка Хоара, Пирамидальная, Расческой, Шелла, Сортировка Вставками, Сортировка Выбором, Шейкерная, именно их стоит использовать для обеспечения максимально возможного быстродействия кода.

Для ознакомления можно использовать и другие виды сортировки, ведь те вполне устойчивы и справляются со своей функцией, но они мало где используются, и являются неэффективными при работе, даже если использовать модификации.