# Отчет

Для рассмотрения нам были представлены следующие виды сортировок: пузырьковая, расческой, шейкерная, вставками, слиянием, выбором, подсчетом, Bogosort и быстрая сортировка. Рассмотрим каждые из них и сравним по времени.

Для более точного определения эффективности сортировки мы рассмотрим работу сортировок с большим (1000 эл) и маленьким (10 эл) массивами. В качестве примера работы будет представлена сортировка по возрастанию а в качестве тестируемого массива будет представлен целочисленный массив.

# Пузырьковая сортировка:

В общем случае без проверки на отсортированность выполняется n - 1 проходов по массиву. Проходя массив каждый раз сравниваются соседние элементы. При первом проходе самый большой элемент становится на свое место. И словно пузыри всплывают на свои места все остальные элементы. Сортировка проста для написания. Эффективна для небольших массивов. В основном используется только для изучения.

Пример реализации сортировки:

bool swap{ false };

for (int32\_t i{}; i < size; ++i)

{

swap = false;

for (int32\_t j{}; j < size - i - 1; ++j)

{

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

{

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

swap = true;

}

}

if (!swap)

{

break;

}

}

Сложность сортировки в среднем: O(n^2)

Время для 10: 0.0005516сек

Время для 1000: 0.0693693сек

Пример пошаговой работы сортировки (10 эл):

68 35 25 17 52 98 37 83 44 64

35 25 17 52 68 37 83 44 64 98

25 17 35 52 37 68 44 64 83 98

17 25 35 37 52 44 64 68 83 98

17 25 35 37 44 52 64 68 83 98

# Сортировка расческой:

Улучшенный вариант сортировки пузырьком. Вместо сравнивания соседних элементов сравниваются элементы на расстоянии одного шага, который с каждым проходом массива уменьшается. Начиная с большого шага массив словно расчесывается, перебрасывая большие элементы в конец, а маленькие в начало. Постепенное уменьшение шага до 1 позволяет быстрее распределить элементы по своим местам. Начальное значение шага — это размер исходного массива, поделенный на оптимальное значение 1.25. Эффективна для небольших массивов.

Пример реализации сортировки:

int32\_t step{ size \*4/5};

while (step > 1 || swap)

{

if (step < 1)

{

step = 1;

swap = false;

}

for (int32\_t i{}; i + step < size; ++i)

{

if (a[i] > a[i + step])

{

std::swap(a[i], a[i + step]);

swap = true;

}

}

step = step \*4/5;

}

Сложность сортировки в среднем: O(nlogn)

Время для 10: 1.8e-05сек

Время для 1000: 0.0001019сек

Пример пошаговой работы сортировки (10 эл):

96 93 68 46 65 69 97 99 5 53

5 53 68 46 65 69 97 99 96 93

5 53 68 46 65 69 97 99 96 93

5 53 68 46 65 69 97 99 96 93

5 53 68 46 65 69 93 99 96 97

5 46 65 53 68 69 93 97 96 99

5 46 53 65 68 69 93 96 97 99

# Сортировка вставками:

Такая сортировка пошагово строит отсортированный массив. Массив разделяется на отсортированную и неотсортированную части. Изначально отсортированным будет считаться один элемент. На каждом шаге берется следующий элемент из второй части и сравнивается с элементами первой. После чего он занимает свое место. Данный вид сортировки не меняет местами одинаковые элементы. Хороша для маленьких массивов.

Пример реализации сортировки:

for (int32\_t i{ 1 }; i < size; i++)

{

int32\_t j{ i - 1 };

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

{

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

--j;

}

}

Сложность сортировки в среднем: O(n^2)

Время для 10: 5.3e-06сек

Время для 1000: 0.0016628 сек

Пример пошаговой работы сортировки (10 эл):

8 20 8 35 3 86 41 87 61 27

8 8 20 35 3 86 41 87 61 27

8 8 20 3 35 86 41 87 61 27

8 8 3 20 35 86 41 87 61 27

8 3 8 20 35 86 41 87 61 27

3 8 8 20 35 86 41 87 61 27

3 8 8 20 35 41 86 87 61 27

3 8 8 20 35 41 86 61 87 27

3 8 8 20 35 41 61 86 87 27

3 8 8 20 35 41 61 86 27 87

3 8 8 20 35 41 61 27 86 87

3 8 8 20 35 41 27 61 86 87

3 8 8 20 35 27 41 61 86 87

3 8 8 20 27 35 41 61 86 87

# Сортировка выбором:

Каждый раз для сортировки массива алгоритм ищет самый маленький элемент из неотсортированной части и перемещает его в конец отсортированной части. Вполне простой алгоритм, но неэффективен для массивов больших размеров.

Пример реализации сортировки:

for (int32\_t i{ 0 }; i < size; ++i)

{

int32\_t small\_position{ FindSmalFromBegin(a, i, size) };

std::swap(a[i], a[small\_position]);

}

Дополнительная функция:

int32\_t FindSmalFromBegin(T\* a, int32\_t i, int32\_t n)

{

int32\_t position{ i };

for (int32\_t k{ i }; k < n; ++k)

{

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

position = k;

}

return position;

}

Сложность сортировки в среднем: O(nlogn)

Время для 10: 1.14e-05 сек

Время для 1000: 0.0006829 сек

Пример пошаговой работы сортировки (10 эл):

71 34 58 14 1 50 59 37 64 53

1 34 58 14 71 50 59 37 64 53

1 14 58 34 71 50 59 37 64 53

1 14 34 58 71 50 59 37 64 53

1 14 34 37 71 50 59 58 64 53

1 14 34 37 50 71 59 58 64 53

1 14 34 37 50 53 59 58 64 71

1 14 34 37 50 53 58 59 64 71

# Быстрая сортировка:

Алгоритм выбирает опорный элемент. Затем все элементы меньшие опорного перебрасываются в лево, а большие в право. Затем рекурсивно процедура применяется к получившимся частям массива. Является самой эффективной сортировкой.

Пример реализации сортировки:

void QuickSort(T\* a, int32\_t start, int32\_t end, bool sort, int32\_t size)

{

if (start < end)

{

int32\_t sup\_elem = Divide(a, start, end, sort);;

QuickSort(a, start, sup\_elem - 1, sort, size);

QuickSort(a, sup\_elem + 1, end, sort, size);

}

}

Дополнительная функция:

int32\_t Divide(T\* a, int32\_t start, int32\_t sup\_elem, bool sort)

{

int32\_t i{ start };

while (i < sup\_elem)

{

if (a[i] > a[sup\_elem] && i == sup\_elem - 1)

{

std::swap(a[i], a[sup\_elem]);

--sup\_elem;

}

else if (a[i] > a[sup\_elem])

{

std::swap(a[sup\_elem - 1], a[sup\_elem]);

std::swap(a[i], a[sup\_elem]);

--sup\_elem;

}

else

{

++i;

}

}

}

Сложность алгоритма в среднем: O(nlogn)

Время для 10: 9e-06 сек

Время для 1000: 0.0001229 сек

Пример пошаговой работы сортировки (10 эл):

13 13 9 19 10 20 67 84 58 92

9 10 13 19 13 20 67 84 58 92

9 10 13 13 19 20 67 84 58 92

9 10 13 13 19 20 67 84 58 92

9 10 13 13 19 20 58 84 67 92

9 10 13 13 19 20 58 67 84 92

# Сортировка слиянием:

Алгоритм рекурсивно разделяет массив на мелкие части вплоть до того пока части не будут состоять из одного элемента. Затем все части объединяются в рекурсивном порядке. Алгоритм работает с постоянной скоростью. Однако данный вид сортировки требует дополнительной памяти на новый массив.

Пример реализации сортировки:

void MergeSort(T\* a, int32\_t start, int32\_t end, bool sort, int32\_t size)

{

int32\_t mid{};

if (start < end)

{

mid = (start + end) / 2;

MergeSort(a, start, mid, sort, size);

MergeSort(a, mid + 1, end, sort, size);

Merge(a, start, end, mid, sort);

}

}

Дополнительная функция

void Merge(T\* a, int32\_t start, int32\_t end, int32\_t mid, bool sort)

{

T\* merge\_array{ new T[end + 1] };

int32\_t i{ start };

int32\_t k{ start };

int32\_t j{ mid + 1 };

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

{

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

{

merge\_array[k++] = a[i++];

}

else

{

merge\_array[k++] = a[j++];

}

}

while (i <= mid)

{

merge\_array[k++] = a[i++];

}

while (j <= end)

{

merge\_array[k++] = a[j++];

}

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

{

a[i] = merge\_array[i];

}

}

Сложность сортировки: O(nlogn)

Время для 10: 9.7e-06 сек

Время для 1000: 0.0017448 сек

Пример пошаговой работы сортировки (10 эл):

79 9 19 22 91 65 26 62 4 71

9 79 19 22 91 65 26 62 4 71

9 19 79 22 91 65 26 62 4 71

9 19 79 22 91 65 26 62 4 71

9 19 22 79 91 65 26 62 4 71

9 19 22 79 91 26 65 62 4 71

9 19 22 79 91 26 62 65 4 71

9 19 22 79 91 26 62 65 4 71

9 19 22 79 91 4 26 62 65 71

4 9 19 22 26 62 65 71 79 91

# Шейкерная сортировка:

Алгоритм повторяет работу пузырьковой сортировки. Проход по массиву идет в две стороны с лева на право и наоборот. Таким образом сразу на свои места встают самый маленький и самый большой элемент массива. При этом диапазон перемешивания элементов с каждым проходом сужается чтобы не проверять уже отсортированные элементы. Движение в двух направлениях позволяет алгоритму работать быстрее чем простой пузырек.

Пример реализации сортировки:

int32\_t left{};

int32\_t right{ size };

bool swap{ true };

while (swap)

{

swap = false;

for (int32\_t i{ left }; i < right - 1; ++i)

{

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

{

std::swap(a[i], a[i + 1]);

swap = true;

}

}

--right;

for (int32\_t i{ right }; i > left; --i)

{

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

{

std::swap(a[i], a[i - 1]);

swap = true;

}

}

++left;

}

Сложность сортировки в среднем:O(n^2)

Время для 10: 1.35e-05 сек

Время для 1000: 0.0018027 сек

Пример пошаговой работы сортировки (10 эл):

52 69 16 55 46 96 56 75 58 72

52 16 55 46 69 56 75 58 72 96

16 52 46 55 56 69 58 75 72 96

16 46 52 55 56 58 69 72 75 96

# Сортировка подсчетом:

Алгоритм только отдаленно напоминает сортировку. На деле высчитывается диапазон, в котором находятся элементы массива. Затем отдельно считается сколько раз то или иное число встречается в массиве. И затем заполняется заново массив в соответствии с тем сколько чисел и какие они. Является вполне эффективным. Однако такая сортировка работает только для целых чисел и символов. Это связано с тем, что вычислить диапазон для вещественных чисел нельзя если не знать заранее с какой точностью считать числа после запятой. Что касаемо строк и прочего, то данная сортировка не возможна для реализации.

Пример реализации сортировки:

T min{ FindMin(a,size) };

T max{ FindMax(a,size) };

int32\_t range{ static\_cast<int32\_t>(max - min) + 1 };

T\* counter = new T[range];

for (int32\_t i{}; i < range; ++i)

{

counter[i] = 0;

}

for (int32\_t i{}; i < size; ++i)

{

counter[static\_cast<int32\_t>(a[i] - min)]++;

}

int32\_t k{};

for (int32\_t i{}; i < range; ++i)

{

while (counter[i] > 0)

{

--counter[i];

a[k++] = i - min;

}

}

Дополнительные функции:

T FindMin(T\* a, int32\_t size)

{

T min = a[0];

for (int32\_t i{}; i < size; ++i)

{

if (a[i] < min)

{

min = a[i];

}

}

return min;

}

T FindMax(T\* a, int32\_t size)

{

T max = a[0];

for (int32\_t i{}; i < size; ++i)

{

if (a[i] > max)

{

max = a[i];

}

}

return max;

}

Сложность сортировки в среднем: O(n + k)

Время для 10: 1.3e-05 сек

Время для 1000: 2.59e-05 сек

Пример пошаговой работы сортировки (10 эл):

89 70 61 90 80 30 7 89 78 11

7 70 61 90 80 30 7 89 78 11

7 11 61 90 80 30 7 89 78 11

7 11 30 90 80 30 7 89 78 11

7 11 30 61 80 30 7 89 78 11

7 11 30 61 70 30 7 89 78 11

7 11 30 61 70 78 7 89 78 11

7 11 30 61 70 78 80 89 78 11

7 11 30 61 70 78 80 89 78 11

7 11 30 61 70 78 80 89 89 11

7 11 30 61 70 78 80 89 89 90

# Богосорт

Самый не эффективный алгоритм. На деле используется в образовательных целях. В представленной версии, алгоритм сравнивает случайные элементы и меняет их местами если нужно, до тех пор, пока массив не будет отсортирован.

Пример реализации сортировки:

srand(static\_cast<unsigned>(time(0)));

while (!IsSorted(arr, size))

{

ShuffleElements(arr, size);

}

Дополнительные функции:

void ShuffleElements(T\* arr, int32\_t size)

{

int32\_t j{};

T temp{};

for (int32\_t i{}; i < size; ++i)

{

j = rand() % size;

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

}

}

bool IsSorted(T\* arr, int32\_t size)

{

for (int32\_t i{ 1 }; i < size; ++i)

{

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

{

return false;

}

}

return true;

}

Сложность сортировки в среднем:O(n\*n!)

Время для 10: 0.0355524 сек

Время для 1000: считать не следует

Пример пошаговой работы сортировки (5 эл):

50 61 43 15 33

33 50 15 43 61

………………………

43 15 61 50 33

61 15 50 33 43

61 15 43 33 50

………………………..

61 50 43 33 15

50 43 61 15 33

15 33 43 50 61

(таких повторений в общей сложности было 41)

# Вывод:

По результатам проверок получается, что быстрее всех сработала сортировка подсчетом. Это было вполне ясно. Данный алгоритм сложно назвать полноценной сортировкой, сортировки то тут и нет. Взяв в расчет еще тот факт, что она работает только с целыми числами и символами то тяжело назвать ее хорошей. После нее идут сортировка расческой и быстрая сортировка. В сортировке расческой для шага выбрано заранее посчитанное оптимальное значение что и делает ее такой быстрой. Так что, на мой взгляд не учитывая все вышесказанные нюансы самой быстрой и эффективной можно считать быструю сортировку что уже ясно из ее названия. Самой неэффективной конечно можно назвать богосорт. Для больших чисел время работы может перейти на дни. Но среди контролируемых сортировок малоэффективной оказалась пузырьковая сортировка. В представленном алгоритме идет проверка на отсортированность массива. В исходном варианте он бы работать и дольше.