**ОТЧЕТ**

Рассмотрим различные алгоритмы сортировки. Оценим их временную сложность и сравним их производительность. Для каждого алгоритма будет приведено описание, фрагмент кода, его средняя временная сложность, а также примеры пошаговой работы и времена выполнения для массивов разного размера (10 и 1000 элементов). Для расчета времени выполнения будем использовать целочисленный массив и сортировать его по возрастанию.

**1. Пузырьковая сортировка (Bubble Sort)**

Это простой алгоритм сортировки, который многократно проходит по списку, сравнивает соседние элементы и меняет их местами, если они находятся в неправильном порядке. Проходы по списку повторяются до тех пор, пока не останется ни одной пары, которую нужно поменять местами, что означает, что список отсортирован. На каждом проходе самый большой (или самый маленький) элемент "всплывает" на свою конечную позицию.

void BubbleSort(int32\_t\* arr, int32\_t size, bool sort)

{

bool swap{ false };

if (sort)

{

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

{

swap = false;

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

{

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

{

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

swap = true;

}

}

if (!swap)

{

break;

}

ShowArray(arr, size);

}

ShowArray(arr, size);

}

}

Сложность сортировки в среднем: O(n2)

Время для 10 элементов: 3e-06 сек

Время для 1000 элементов: 0.0023256 сек

Пример пошаговой работы сортировки (10 эл.):

-794 117 517 51 -656 872 -170 843 618 -662

-794 117 51 -656 517 -170 843 618 -662 872

-794 51 -656 117 -170 517 618 -662 843 872

-794 -656 51 -170 117 517 -662 618 843 872

-794 -656 -170 51 117 -662 517 618 843 872

-794 -656 -170 51 -662 117 517 618 843 872

-794 -656 -170 -662 51 117 517 618 843 872

-794 -656 -662 -170 51 117 517 618 843 872

-794 -662 -656 -170 51 117 517 618 843 872

-794 -662 -656 -170 51 117 517 618 843 872

**2. Сортировка расческой (Comb Sort)**

Улучшенный вариант пузырьковой сортировки. Она устраняет "черепах" (маленькие значения в конце списка), которые сильно замедляют пузырьковую сортировку. Вместо сравнения соседних элементов, она сравнивает элементы, разделенные "шагом", который уменьшается на каждом проходе. Начальное значение шага обычно равно размеру массива, деленному на коэффициент уменьшения.

void CombSort(int32\_t\* arr, int32\_t size, bool sort)

{

int32\_t step = size;

bool swapped = true;

if (sort)

{

while (step > 1 || swapped)

{

step = std::max(1, step \* 4 / 5);

swapped = false;

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

{

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

{

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

swapped = true;

}

}

ShowArray(arr, size);

}

}

}

Сложность сортировки в среднем: O(n log n)

Время для 10 элементов: 1.8e-06 сек

Время для 1000 элементов: 0.0001947 сек

Пример пошаговой работы сортировки (10 эл.):

-794 -662 517 51 -656 872 -170 843 618 117

-794 -662 517 51 -656 872 -170 843 618 117

-794 -662 -170 51 -656 117 517 843 618 872

-794 -662 -170 51 -656 117 517 843 618 872

-794 -662 -656 51 -170 117 517 843 618 872

-794 -662 -656 -170 51 117 517 618 843 872

-794 -662 -656 -170 51 117 517 618 843 872

**3. Сортировка вставками (Insertion Sort)**

Алгоритм строит отсортированный массив (или список) по одному элементу за раз. Он итерирует по входному массиву, берет по одному элементу и вставляет его в правильное место уже отсортированной части массива.

void InsertionSort(int32\_t\* arr, int32\_t size, bool sort) {

ShowArray(arr, size);

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

int32\_t j{ i - 1 };

if (sort) {

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

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

--j;

ShowArray(arr, size);

}

}

}

Сложность сортировки в среднем: O(n2)

Время для 10 элементов: 1.71e-05 сек

Время для 1000 элементов: 0.166787 сек

Пример пошаговой работы сортировки (10 эл.):

-794 117 517 51 -656 872 -170 843 618 -662

-794 117 51 517 -656 872 -170 843 618 -662

-794 51 117 517 -656 872 -170 843 618 -662

-794 51 117 -656 517 872 -170 843 618 -662

-794 51 -656 117 517 872 -170 843 618 -662

-794 -656 51 117 517 872 -170 843 618 -662

-794 -656 51 117 517 -170 872 843 618 -662

-794 -656 51 117 -170 517 872 843 618 -662

-794 -656 51 -170 117 517 872 843 618 -662

-794 -656 -170 51 117 517 872 843 618 -662

-794 -656 -170 51 117 517 843 872 618 -662

-794 -656 -170 51 117 517 843 618 872 -662

-794 -656 -170 51 117 517 618 843 872 -662

-794 -656 -170 51 117 517 618 843 -662 872

-794 -656 -170 51 117 517 618 -662 843 872

-794 -656 -170 51 117 517 -662 618 843 872

-794 -656 -170 51 117 -662 517 618 843 872

-794 -656 -170 51 -662 117 517 618 843 872

-794 -656 -170 -662 51 117 517 618 843 872

-794 -656 -662 -170 51 117 517 618 843 872

-794 -662 -656 -170 51 117 517 618 843 872

**4. Сортировка выбором (Selection Sort)**

Алгоритм делит входной список на две части: отсортированную и неотсортированную. Он многократно находит наименьший (или наибольший) элемент из неотсортированной части и помещает его в конец отсортированной части.

int32\_t GetSmallOfBegin(int32\_t\* arr, int32\_t i, int32\_t size) {

int32\_t pos{ i };

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

if (arr[k] < arr[pos])

pos = k;

}

return pos;

}

int32\_t GetSmallOfEnd(int32\_t\* arr, int32\_t i, int32\_t size) {

int32\_t pos{ i };

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

if (arr[k] < arr[pos])

pos = k;

}

return pos;

}

template<typename T>

void SelectionSort(T\* arr, int32\_t size, bool sort)

{

if (sort)

{

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

{

int32\_t small\_pos{ GetSmallOfBegin(arr, i, size) };

std::swap(arr[i], arr[small\_pos]);

ShowArray(arr, size);

}

}

else

{

for (int32\_t i{ size - 1 }; i >= 0; --i)

{

int32\_t small\_pos{ GetSmallOfEnd(arr, i, size) };

std::swap(arr[i], arr[small\_pos]);

ShowArray(arr, size);

}

}

}

Сложность сортировки в среднем: O(n2)

Время для 10 элементов: 3.7e-06 сек

Время для 1000 элементов: 0.0015358сек

Пример пошаговой работы сортировки (10 эл.):

-794 117 517 51 -656 872 -170 843 618 -662

-794 -662 517 51 -656 872 -170 843 618 117

-794 -662 -656 51 517 872 -170 843 618 117

-794 -662 -656 -170 517 872 51 843 618 117

-794 -662 -656 -170 51 872 517 843 618 117

-794 -662 -656 -170 51 117 517 843 618 872

-794 -662 -656 -170 51 117 517 843 618 872

-794 -662 -656 -170 51 117 517 618 843 872

-794 -662 -656 -170 51 117 517 618 843 872

-794 -662 -656 -170 51 117 517 618 843 872

**5. Быстрая сортировка (Quick Sort)**

Это эффективный алгоритм сортировки, основанный на принципе "разделяй и властвуй". Он выбирает "опорный" элемент из массива и разбивает другие элементы на две подмассива, согласно тому, меньше они или больше опорного. Затем подмассивы рекурсивно сортируются.

int32\_t Divide(int32\_t\* arr, int32\_t start, int32\_t sup\_elem, int32\_t size, bool sort) {

int32\_t i{ start };

if (sort) {

while (i < sup\_elem) {

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

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

--sup\_elem;

ShowArray(arr, size);

}

else if (arr[i] > arr[sup\_elem]) {

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

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

--sup\_elem;

ShowArray(arr, size);

}

else {

++i;

}

}

}

}

template<typename T>

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

{

if (start < end)

{

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

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

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

}

}

Сложность алгоритма в среднем: O(n log n)

Время для 10 элементов: 2.6e-05 сек

Время для 1000 элементов: 0.000133 сек

Пример пошаговой работы сортировки (10 эл.):

-794 117 517 51 -656 872 -170 843 618 -662

-794 618 517 51 -656 872 -170 843 -662 117

-794 843 517 51 -656 872 -170 -662 618 117

-794 -170 517 51 -656 872 -662 843 618 117

-794 872 517 51 -656 -662 -170 843 618 117

-794 -656 517 51 -662 872 -170 843 618 117

-794 51 517 -662 -656 872 -170 843 618 117

-794 517 -662 51 -656 872 -170 843 618 117

-794 -662 517 51 -656 872 -170 843 618 117

-794 -662 618 51 -656 872 -170 843 117 517

-794 -662 843 51 -656 872 -170 117 618 517

-794 -662 -170 51 -656 872 117 843 618 517

-794 -662 -170 51 -656 117 872 843 618 517

-794 -662 51 -656 -170 117 872 843 618 517

-794 -662 -656 51 -170 117 872 843 618 517

-794 -662 -656 -170 51 117 872 843 618 517

-794 -662 -656 -170 51 117 618 843 517 872

-794 -662 -656 -170 51 117 843 517 618 872

-794 -662 -656 -170 51 117 517 843 618 872

-794 -662 -656 -170 51 117 517 618 843 872

**6. Сортировка слиянием (Merge Sort)**

Это эффективный, стабильный алгоритм сортировки, основанный на принципе "разделяй и властвуй". Он рекурсивно делит массив на две половины, пока не останутся подмассивы из одного элемента. Затем эти подмассивы многократно сливаются (объединяются) в отсортированном порядке, пока не будет получен один отсортированный массив.

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

{

int32\_t\* merge\_arr{ new int32\_t[end + 1] };

int32\_t i{ start };

int32\_t k{ start };

int32\_t j{ mid + 1 };

if (sort) {

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

{

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

{

merge\_arr[k++] = arr[i++];

}

else

{

merge\_arr[k++] = arr[j++];

}

}

while (i <= mid)

{

merge\_arr[k++] = arr[i++];

}

while (j <= end)

{

merge\_arr[k++] = arr[j++];

}

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

{

arr[i] = merge\_arr[i];

}

}

}

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

{

int32\_t mid{};

if (start < end)

{

mid = (start + end) / 2;

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

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

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

ShowArray(arr, size);

}

}

Сложность сортировки: O(n log n)

Время для 10 элементов: 3.21e-05 сек

Время для 1000 элементов: 0.0031031 сек

Пример пошаговой работы сортировки (10 эл.):

-794 117 517 51 -656 872 -170 843 618 -662

-794 117 517 51 -656 872 -170 843 618 -662

-794 117 517 -656 51 872 -170 843 618 -662

-794 -656 51 117 517 872 -170 843 618 -662

-794 -656 51 117 517 -170 872 843 618 -662

-794 -656 51 117 517 -170 843 872 618 -662

-794 -656 51 117 517 -170 843 872 -662 618

-794 -656 51 117 517 -662 -170 618 843 872

-794 -662 -656 -170 51 117 517 618 843 872

**7. Шейкерная сортировка (Cocktail Sort)**

Двунаправленная версия пузырьковой сортировки. Она проходит по списку в обоих направлениях (сначала слева направо, затем справа налево), "встряхивая" элементы, чтобы они быстрее перемещались к своим конечным позициям. Это позволяет быстрее перемещать как большие, так и маленькие элементы.

void CocktailSort(int32\_t\* arr, int32\_t size, bool sort)

{

int32\_t left{};

int32\_t right{ size };

bool swap{ true };

ShowArray(arr, size);

if (sort) {

while (swap) {

swap = false;

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

if (arr[i] > arr[i + 1]) {

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

swap = true;

ShowArray(arr, size);

}

}

--right;

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

if (arr[i] < arr[i - 1]) {

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

swap = true;

ShowArray(arr, size);

}

}

++left;

}

}

}

Сложность сортировки в среднем: O(n2)

Время для 10 элементов: 1.76e-05 сек

Время для 1000 элементов: 0.0037794 сек

Пример пошаговой работы сортировки (10 эл.):

-794 117 517 51 -656 872 -170 843 618 -662

-794 117 51 517 -656 872 -170 843 618 -662

-794 117 51 -656 517 872 -170 843 618 -662

-794 117 51 -656 517 -170 872 843 618 -662

-794 117 51 -656 517 -170 843 872 618 -662

-794 117 51 -656 517 -170 843 618 872 -662

-794 117 51 -656 517 -170 843 618 -662 872

-794 117 51 -656 517 -170 843 -662 618 872

-794 117 51 -656 517 -170 -662 843 618 872

-794 117 51 -656 517 -662 -170 843 618 872

-794 117 51 -656 -662 517 -170 843 618 872

-794 117 51 -662 -656 517 -170 843 618 872

-794 117 -662 51 -656 517 -170 843 618 872

-794 -662 117 51 -656 517 -170 843 618 872

-794 -662 51 117 -656 517 -170 843 618 872

-794 -662 51 -656 117 517 -170 843 618 872

-794 -662 51 -656 117 -170 517 843 618 872

-794 -662 51 -656 117 -170 517 618 843 872

-794 -662 51 -656 -170 117 517 618 843 872

-794 -662 -656 51 -170 117 517 618 843 872

-794 -662 -656 -170 51 117 517 618 843 872

**8. Сортировка подсчетом (Countsort)**

Этот алгоритм - не сортировка. Его суть заключается не в сравнении элементов (как в сортировках), а в определении диапазона значений, присутствующих в массиве. Затем подсчитывается количество вхождений каждого элемента. На основе этих данных формируется новый массив, в котором элементы располагаются в порядке их значений и с соответствующей частотой. Такая сортировка эффективно работает для целых чисел и символов. Для строк и студентов данная сортировка не возможна для реализации.

template <typename T>

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

{

T min = arr[0];

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

{

if (arr[i] < min)

{

min = arr[i];

}

}

return min;

}

template <typename T>

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

{

T max = arr[0];

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

{

if (arr[i] > max)

{

max = arr[i];

}

}

return max;

}

template <typename T>

void CountSort(T\* arr, int32\_t size, bool sort)

{

if constexpr (std::is\_same\_v<T, int> || std::is\_same\_v<T, char>)

{

ShowArray(arr, size);

T min{ FindMin(arr,size) };

T max{ FindMax(arr,size) };

int32\_t range{ static\_cast<int32\_t>(max) - static\_cast<int32\_t>(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>(arr[i]) - static\_cast<int32\_t>(min)];

}

int32\_t k{};

if (sort) {

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

{

while (counter[i] > 0)

{

counter[i]--;

arr[k++] = i + static\_cast<int32\_t>(min);

ShowArray(arr, size);

}

}

}

else

{

for (int32\_t i{ range - 1 }; i >= 0; --i)

{

while (counter[i] > 0)

{

counter[i]--;

arr[k++] = i + static\_cast<int32\_t>(min);

ShowArray(arr, size);

}

}

}

delete[] counter;

}

else

{

throw std::invalid\_argument("unsupported array type\n");

}

}

Сложность сортировки в среднем: O(n+k)

Время для 10 элементов: 3.2e-05 сек

Время для 1000 элементов: 4.97e-05 сек

Пример пошаговой работы сортировки (10 эл.):

-794 117 517 51 -656 872 -170 843 618 -662

-794 117 517 51 -656 872 -170 843 618 -662

-794 -662 517 51 -656 872 -170 843 618 -662

-794 -662 -656 51 -656 872 -170 843 618 -662

-794 -662 -656 -170 -656 872 -170 843 618 -662

-794 -662 -656 -170 51 872 -170 843 618 -662

-794 -662 -656 -170 51 117 -170 843 618 -662

-794 -662 -656 -170 51 117 517 843 618 -662

-794 -662 -656 -170 51 117 517 618 618 -662

-794 -662 -656 -170 51 117 517 618 843 -662

-794 -662 -656 -170 51 117 517 618 843 872

**9. Богосорт (Bogo Sort)**

Крайне неэффективный алгоритм сортировки, используемый в образовательных целях для демонстрации худшего случая. Он работает путем случайного перемешивания элементов массива до тех пор, пока массив не окажется отсортированным.

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

{

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

{

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

{

return false;

}

}

return true;

}

template <class T>

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

{

int32\_t j{};

T temp{};

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

{

j = rand() % size;

temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

template <class T>

void BogoSort(T\* arr, int32\_t size, bool sort)

{

ShowArray(arr, size);

if (sort)

{

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

while (!IsSorted(arr, size))

{

ShuffleElements(arr, size);

ShowArray(arr, size);

}

}

else

{

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

while (IsSorted(arr, size))

{

ShuffleElements(arr, size);

ShowArray(arr, size);

}

}

}

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

Время для 10 элементов: 0.508907 сек

Время для 1000 элементов: практически бесконечно

Пример пошаговой работы сортировки (5 эл.):

335 -572 -254 -786 592

-572 -254 335 592 -786

592 -572 -786 -254 335

-786 335 -572 592 -254

335 592 -786 -254 -572

-572 -786 335 592 -254

592 -572 -254 335 -786

-572 592 -254 335 -786

... (много шагов)

-786 335 -572 592 -254

-254 335 -786 -572 592

-786 -254 -572 335 592

-786 -254 592 335 -572

-786 -572 -254 335 592

**Вывод**

На основе анализа и предоставленных данных можно сделать следующие выводы:

1. Наиболее эффективные алгоритмы:

Быстрая сортировка (Quick Sort): Обладает средней временной сложностью O(n log n) и показала очень хорошие результаты (0.000133 сек для 1000 элементов). Она является одной из самых быстрых универсальных сортировок на практике.

Сортировка расческой (Comb Sort): Также имеет среднюю сложность O(n log n) и показала отличные результаты (0.0001947 сек для 1000 элементов), опередив даже быструю сортировку в данном тесте. Это подчеркивает эффективность ее оптимизации по сравнению с пузырьковой сортировкой.

Сортировка слиянием (Merge Sort): Также имеет сложность O(n log n) и является стабильной сортировкой. Ее производительность (0.0031031 сек для 1000 элементов) хороша, но она требует дополнительной памяти.

2. Менее эффективные алгоритмы (O(n2)):

Пузырьковая сортировка (Bubble Sort), Сортировка вставками (Insertion Sort), Сортировка выбором (Selection Sort), и Шейкерная сортировка (Cocktail Sort): Все эти алгоритмы имеют квадратичную временную сложность O(n2), что делает их неэффективными для больших массивов. Их времена выполнения значительно выше, чем у алгоритмов O(n log n). Шейкерная сортировка немного быстрее пузырьковой благодаря двунаправленным проходам.

3. Наименее эффективный алгоритм:

Богосорт (Bogo Sort): Это демонстрационный алгоритм с крайне высокой временной сложностью O(n\*n!), что делает его абсолютно непригодным для практического использования даже для небольших массивов.

Для универсального применения к различным типам данных и без ограничений по диапазону значений, Быстрая сортировка и Сортировка расческой являются наилучшими вариантами. Они демонстрируют высокую производительность благодаря своей временной сложности O(n log n).