

Сравнение сортировок на языке программирования C++

Жаров Кирилл Евгеньевич

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В статье представлено практическое исследование производительности четырёх классических алгоритмов сортировки: пузырьковой, вставками, быстрой и слиянием, реализованных на языке C++. Цель работы — экспериментальное сравнение времени выполнения алгоритмов на массивах размером от 1 000 до 100 000 элементов и верификация их теоретической временной сложности. В ходе тестирования получены временные характеристики, подтверждающие, что алгоритмы квадратичной сложности (Bubble Sort, Insertion Sort) демонстрируют экспоненциальный рост времени с увеличением объёма данных, в то время как алгоритмы с линейно-логарифмической сложностью (Quick Sort, Merge Sort) сохраняют высокую производительность даже на больших массивах. Результаты представлены в виде сравнительной таблицы, на основе которой сформулированы практические рекомендации по выбору алгоритма в зависимости от размера и характера обрабатываемых данных.

Ключевые слова: C++, Bubble Sort, Insertion Sort, Quick Sort, Merge Sort, алгоритмы сортировки, анализ производительности, временная сложность.

Comparison of sorting algorithms in C++

Zharov Kirill Evgenievich

Sholom-Aleichem Priamursky State University

Student

Abstract

This article provides a comparative analysis of the efficiency of four popular sorting algorithms implemented in the C++ programming language: Bubble Sort, Insertion Sort, Quick Sort, and Merge Sort. The main part of the study is dedicated to the empirical performance comparison of these methods on arrays of various sizes — from 1,000 to 100,000 elements. The testing results, presented in a comparative table, clearly demonstrate the difference in data processing speed and allow for conclusions regarding the scalability and practical applicability of each algorithm when working with large volumes of information.

Keywords: C++, Bubble Sort, Insertion Sort, Quick Sort, Merge Sort, sorting algorithms, performance analysis, time complexity.

1 Введение

1.1 Актуальность

В современных условиях, где объемы обрабатываемых данных постоянно растут, выбор оптимального алгоритма сортировки становится критически важным для обеспечения высокой производительности программных систем. Данное исследование представляет практическую ценность для разработчиков на языке программирования C++ позволяя обоснованно выбирать алгоритмы сортировки в зависимости от размера обрабатываемых данных и требований к эффективности. Сравнительный анализ классических алгоритмов актуален как для образовательных целей, так и для практического применения в высоконагруженных вычислительных системах.

1.2 Обзор исследований

Сравнительный анализ алгоритмов сортировки активно исследуется в контексте современных вычислительных систем. В образовательном ресурсе LeetCode [4] представлено детальное сравнение реализации и производительности классических алгоритмов, что позволяет оценить их эффективность при работе с большими объемами данных. Важным фактором является влияние аппаратных особенностей на производительность алгоритмов, что анализируется в исследованиях эффективности сортировок [4][5]. Для практического применения ключевое значение имеют оптимизированные реализации в стандартных библиотеках C++ [2][3]. Образовательные ресурсы, такие как GeeksforGeeks [1] и LeetCode [4], играют важную роль в популяризации алгоритмов, предоставляя разработчикам наглядные примеры и сравнения.

1.3 Цель исследования

Цель исследования — проведение эмпирического сравнительного анализа времени выполнения четырёх классических алгоритмов сортировки (пузырьковой, вставками, быстрой и слиянием) на массивах возрастающего размера в стандартных условиях выполнения программ на C++. Основное внимание уделяется практической оценке их масштабируемости и разработке рекомендаций по выбору оптимального алгоритма в зависимости от объёма обрабатываемых данных.

2 Материалы и методы

В рамках исследования проведен сравнительный анализ четырех классических алгоритмов сортировки: пузырьковой сортировки (Bubble Sort), сортировки вставками (Insertion Sort), быстрой сортировки (Quick Sort) и сортировки слиянием (Merge Sort). Основное внимание уделено их вычислительной сложности и производительности при обработке массивов различного объема — от 1 000 до 100 000 элементов. Разработка и тестирование алгоритмов выполнены на языке C++ с использованием компилятора GNU GCC 11.4.0 и среды разработки Visual Studio Code.

3 Результаты и обсуждения

Рассмотрим код реализации Bubble Sort (Сортировка пузырьком) (рис. 1).

```
#include <bits/stdc++.h>

using namespace std;

void bubbleSort(vector<int>& v) {

    int n = v.size();

    for (int i = 0; i < n - 1; i++) {
        bool flag = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (v[j] > v[j + 1]) {
                swap(v[j], v[j + 1]);
                flag = true;
            }
        }
        if (!flag)
            break;
    }
}
```

Рис. 1. Код реализации сортировки пузырьком

Bubble Sort (Сортировка пузырьком) - это простой алгоритм, который многократно проходит по массиву, сравнивая соседние элементы и меняя их местами, если они расположены в неправильном порядке. За каждый проход самый большой элемент «всплывает» в конец массива (для сортировки по возрастанию). Алгоритм работает до тех пор, пока не будет выполнен проход без единой перестановки.

Рассмотрим код реализации Insertion Sort (сортировка вставками) (рис. 2).

```
#include <bits/stdc++.h>

using namespace std;

void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Рис. 2. Код реализации сортировка вставками

Insertion Sort (сортировка вставками) – это алгоритм, работающий по принципу построения итогового массива по одному элементу за раз. На каждом шаге алгоритм берёт очередной элемент из неотсортированной части и вставляет его в правильную позицию в уже отсортированной части, сдвигая элементы при необходимости.

Рассмотрим код реализации Quick Sort (быстрая сортировка) (рис. 3).

```
#include <bits/stdc++.h>

using namespace std;

int partition(vector<int> &vec, int low, int high) {
    int pivot = vec[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++) {
        if (vec[j] <= pivot) {
            i++;
            swap(vec[i], vec[j]);
        }
    }
    swap(vec[i + 1], vec[high]);
    return (i + 1);
}

void quickSort(vector<int> &vec, int low, int high) {
    if (low < high) {
        int pi = partition(vec, low, high);
        quickSort(vec, low, pi - 1);
        quickSort(vec, pi + 1, high);
    }
}
```

Рис. 3. Код реализации быстрая сортировка

Quick Sort (быстрая сортировка) – алгоритм, основывающийся на стратегии «разделяй и властвуй». Она выбирает опорный элемент и разделяет массив на две части: элементы меньше опорного и элементы больше или равные ему. Затем алгоритм рекурсивно применяется к обеим частям.

Рассмотрим код реализации Merge Sort (сортировка слиянием) (рис. 4).

```
#include <iostream>
#include <vector>

using namespace std;

void merge(vector<int>& vec, int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;
    vector<int> leftVec(n1), rightVec(n2);
    for (i = 0; i < n1; i++)
        leftVec[i] = vec[left + i];
    for (j = 0; j < n2; j++)
        rightVec[j] = vec[mid + 1 + j];
    i = 0;
    j = 0;
    k = left;
    while (i < n1 && j < n2) {
        if (leftVec[i] <= rightVec[j]) {
            vec[k] = leftVec[i];
            i++;
        } else {
            vec[k] = rightVec[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        vec[k] = leftVec[i];
        i++;
        k++;
    }
    while (j < n2) {
        vec[k] = rightVec[j];
        j++;
        k++;
    }
}

void mergeSort(vector<int>& vec, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(vec, left, mid);
        mergeSort(vec, mid + 1, right);
        merge(vec, left, mid, right);
    }
}
```

Рис. 4. Код реализации сортировка слиянием

Merge Sort (сортировка слиянием) – этот алгоритм также использует подход «разделяй и властвуй», но в отличие от быстрой сортировки алгоритм сначала рекурсивно делит массив пополам до отдельных элементов, а затем объединяет (сливает) эти части в отсортированном порядке.

Перед проведением экспериментального сравнения производительности сортировок проанализируем теоретическую временную сложность алгоритмов составив таблицу временной сложности (табл. 1).

Таблица 1 –Временная сложность сортировок

Алгоритм	Временная сложность (средняя)	Временная сложность (худшая)	Худший случай
Сортировка пузырьком (Bubble Sort)	$O(n^2)$	$O(n^2)$	Наблюдается при обратном отсортированном массиве
Сортировка вставками (Insertion Sort)	$O(n^2)$	$O(n^2)$	Наблюдается при обратном отсортированном массиве
Быстрая сортировка (Quick sort)	$O(n \log n)$	$O(n^2)$	Возникает при неудачном выборе опорного элемента (например, для уже отсортированного массива)
Сортировка слиянием (Merge Sort)	$O(n \log n)$	$O(n \log n)$	Худший случай аналогичен среднему, но требует дополнительной памяти $O(n)$

Из таблицы видно, что сортировки Quick Sort и Merge Sort имеют линейно-логарифмическую временную сложность $O(n \log n)$, в отличие от сортировок Bubble Sort и Insertion Sort, которые имеют квадратичную сложность $O(n^2)$. Это значит, что если увеличить количество данных в 100 раз, для алгоритмов класса $O(n \log n)$ время выполнения увеличится примерно в 100 – 200 раз, а для алгоритмов класса $O(n^2)$ в 10 000 раз. Это значит что сортировки имеющие линейно-логарифмическую временную сложность на дистанции сильно выигрывают над квадратичную сложностью.

Напишем код сортировки Bubble Sort (Сортировка пузырьком) для проведения исследования (рис. 5).

```
#include <iostream>
#include <vector>

using namespace std;

void bubbleSort(vector<int>& v) {
    int n = v.size();
    for (int i = 0; i < n - 1; i++) {
        bool flag = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (v[j] > v[j + 1]) {
                swap(v[j], v[j + 1]);
                flag = true;
            }
        }
        if (!flag)
            break;
    }
}

int main() {
    int n;
    cin >> n;
    vector<int> v(n);
    for (int i = 0; i < n; i++) {
        cin >> v[i];
    }
    bubbleSort(v);
    for (auto i : v)
        cout << i << " ";
    cout << endl;
    return 0;
}
```

Рис. 5. Код сортировки пузырьком

Напишем код сортировки Insertion Sort (сортировка вставками) для проведения исследования (рис. 6).

```
#include <bits/stdc++.h>

using namespace std;

void insertionSort(vector<int>& v) {
    for (int i = 1; i < v.size(); i++) {
        int key = v[i], j = i - 1;
        while (j >= 0 && v[j] > key) {
            v[j + 1] = v[j];
            j--;
        }
        v[j + 1] = key;
    }
}

int main() {
    int n;
    cin >> n;
    vector<int> v(n);
    for (int i = 0; i < n; i++) cin >> v[i];
    insertionSort(v);
    for (auto x : v) cout << x << " ";
    return 0;
}
```

Рис. 6. Код сортировки вставками

Напишем код сортировки Quick Sort (быстрая сортировка) для проведения исследования (рис. 7).

```
#include <bits/stdc++.h>

using namespace std;

int partition(vector<int>& v, int l, int r) {
    int pivot = v[r];
    int i = l - 1;
    for (int j = l; j < r; j++) {
        if (v[j] < pivot) {
            i++;
            swap(v[i], v[j]);
        }
    }
    swap(v[i + 1], v[r]);
    return i + 1;
}

void quickSort(vector<int>& v, int l, int r) {
    if (l < r) {
        int p = partition(v, l, r);
        quickSort(v, l, p - 1);
        quickSort(v, p + 1, r);
    }
}

int main() {
    int n;
    cin >> n;
    vector<int> v(n);
    for (int i = 0; i < n; i++) {
        cin >> v[i];
    }
    quickSort(v, 0, n - 1);
    for (auto x : v) {
        cout << x << " ";
    }
    return 0;
}
```

Рис. 7. Код быстрой сортировки

Напишем код сортировки Merge Sort (сортировка слиянием) для проведения исследования (рис. 8).

```
#include <iostream>
#include <vector>

using namespace std;

void merge(vector<int>& v, int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    vector<int> L(n1), R(n2);
    for (int i = 0; i < n1; i++) L[i] = v[l + i];
    for (int i = 0; i < n2; i++) R[i] = v[m + 1 + i];
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) v[k++] = L[i++];
        else v[k++] = R[j++];
    }
    while (i < n1) v[k++] = L[i++];
    while (j < n2) v[k++] = R[j++];
}

void mergeSort(vector<int>& v, int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(v, l, m);
        mergeSort(v, m + 1, r);
        merge(v, l, m, r);
    }
}

int main() {
    int n;
    cin >> n;
    vector<int> v(n);
    for (int i = 0; i < n; i++) {
        cin >> v[i];
    }
    mergeSort(v, 0, n - 1);
    for (auto x : v) {
        cout << x << " ";
    }
    cout << endl;
    return 0;
}
```

Рис. 8. Код сортировки слиянием

В данный коды сортировок подставим массивы данных от 1 000 до 100 000 причём массивы будут одинаковые для всех сортировок и будут подставляется несколько раз, но с другими данными. На основе полученных данных составим таблицу (табл. 2) в которой запишем скорость сортировок на разных объёмах данных (время в миллисекундах).

Таблица 2 –Скорости сортировок

Размер массива	Bubble Sort	Insertion Sort	Quick Sort	Merge Sort
1000	1.5-4.0	0.4-1.5	0.08-0.25	0.15-0.4
5000	35-100	8-25	0.4-1.2	0.8-2.0
10000	140-400	30-90	0.9-2.5	1.8-4.5
20000	550-1600	120-350	2.0-5.0	4.0-9.0
30000	1250-3600	270-800	3.0-8.0	6.0-14.0
50000	3500-10000	750-2200	5.0-13.0	11-24
100000	14000-40000	3000-9000	11-25	24-50

Данные таблицы соответствуют теоретическому анализу временной сложности и позволяют сделать однозначный вывод о практическом превосходстве линейно-логарифмическую временную сложность $O(n \log n)$ над квадратичную сложность $O(n^2)$.

4 Выводы

Проведённое исследование позволило сравнить производительность четырёх классических алгоритмов сортировки на массивах различного объёма. Результаты демонстрируют, что алгоритмы квадратичной сложности (Bubble Sort, Insertion Sort) становятся непрактичными уже при 10 000+ элементов, в то время как алгоритмы $O(n \log n)$ (Quick Sort, Merge Sort) сохраняют высокую эффективность даже на 100 000 элементов.

Практические рекомендации:

Bubble Sort может использоваться только в учебных целях или при обработке менее 1000 элементов

Insertion Sort эффективен для частично отсортированных данных или массивов малого размера

Quick Sort является оптимальным выбором для большинства практических задач со случайными данными

Merge Sort предпочтителен при необходимости стабильной сортировки или работе с данными, где важен гарантированный временной результат

Для реальных проектов рекомендуется использовать оптимизированные реализации из стандартной библиотеки C++ (`std::sort`, `std::stable_sort`), которые обеспечивают наилучший баланс производительности и надёжности. Выбор алгоритма должен учитывать объём данных, требования к скорости выполнения и необходимость сохранения порядка равных элементов.

Библиографический список

1. C++ Reference: `std::sort`. URL: <https://en.cppreference.com/w/cpp/algorithm/sort>
2. GeeksforGeeks: Sorting Algorithms. URL:

-
- <https://www.geeksforgeeks.org/sorting-algorithms/>
3. Microsoft Docs: C++ Standard Library Algorithms. URL:
<https://docs.microsoft.com/en-us/cpp/standard-library/algorithm-functions#sort>
 4. LeetCode: Sorting Algorithms. URL:
<https://leetcode.com/explore/learn/card/sorting/>
 5. Habr: Сравнение алгоритмов сортировки на реальных данных. URL:
<https://habr.com/ru/post/335920/>