Отчет по задаче A2

Гуршумов Даниил Бахтиятович
БПИ 239

Аннотация

В данном отчете представлено сравнение стандартного алгоритма сортировки слиянием (MERGE SORT) и гибридного алгоритма MERGE+INSERTION SORT. Выполнены эмпирические замеры временных затрат для различных типов массивов и проведен сравнительный анализ эффективности алгоритмов.

1 Задача

Цель работы: провести эмпирическое сравнение двух реализаций алгоритмов сортировки на основе сортировки слиянием: стандартного MERGE SORT и гибридного MERGE+INSERTION SORT.

Необходимо:

- Реализовать генератор тестовых данных для экспериментов.
- Замерить время работы стандартного алгоритма MERGE SORT на массивах разных типов.
- Замерить время работы гибридного алгоритма MERGE+INSERTION SORT на тех же массивах.
- Выполнить сравнительный анализ результатов.

2 Решение

2.1 Реализация алгоритмов

В данном разделе представлен код, используемый для генерации тестовых данных, реализации алгоритмов и замера времени.

```
#include <iostream>
#include <cmath>
#include <random>
#include <vector>
#include <fstream>
#include <iomanip>

//
class ArrayGenerator {
public:
    void generate_array() {
        std::random_device rd;
        std::mt19937 generator(rd());
        std::uniform_int_distribution<> dist(_min_value, _max_value);
        arr.resize(_size);
        for (int i = 0; i < _size; ++i) {
            arr[i] = dist(generator);
        }
    }
    std::vector<int> gen_1(int n) {
        return {arr.begin(), arr.begin() + n};
    }
    std::vector<int> gen_2(int n) {
```

```
        std::vector<int> temp = {arr.begin(), arr.begin() + n};
        std::sort(temp.rbegin(), temp.rend());
        return temp;
    }
    std::vector<int> gen_3(int n) {
        std::vector<int> temp = {arr.begin(), arr.begin() + n};
        std::sort(temp.begin(), temp.end());
        std::random_device rd;
        std::mt19937 generator(rd());
        std::uniform_int_distribution<> dist(0, n);
        for (int i = 0; i < 10; ++i) {
            std::swap(temp[dist(generator)], temp[dist(generator)]);
        }
        return temp;
    }
    ArrayGenerator() {
        generate_array();
    }

private:
    int _max_value = 6000;
    int _min_value = 0;
    int _size = 10000;
    std::vector<int> arr;
};

//                      Insertion Sort
void insertion_sort(std::vector<int>& a, int s, int e) {
    for (int i = s + 1; i <= e; ++i) {
        int key = a[i];
        int j = i - 1;

        while (j >= s && a[j] > key) {
            a[j + 1] = a[j];
            j--;
        }

        a[j + 1] = key;
    }
}

//                      Merge Sort
void merge_sort(std::vector<int>& a, int s, int m, int e) {
    int n1 = m - s + 1;
    int n2 = e - m;
    std::vector<int> left(n1), right(n2);

    for (int i = 0; i < n1; i++) {
        left[i] = a[s + i];
    }
    for (int i = 0; i < n2; i++) {
        right[i] = a[m + 1 + i];
    }

    int l = 0, r = 0, k = s;
    while (l < n1 && r < n2) {
        if (left[l] <= right[r]) {
            a[k] = left[l];
            l++;
        } else {
            a[k] = right[r];
            r++;
        }
```

```
87          k++;
88      }
89
90      while (l < n1) {
91          a[k] = left[l];
92          l++;
93          k++;
94      }
95
96      while (r < n2) {
97          a[k] = right[r];
98          r++;
99          k++;
100     }
101 }
102
103 //                                          MERGE+INSERTION SORT
104 void merge(std::vector<int>& a, int s, int e) {
105     if (e - s <= 15) {
106         insertion_sort(a, s, e);
107         return;
108     }
109
110     int mid = s + (e - s) / 2;
111     merge(a, s, mid);
112     merge(a, mid + 1, e);
113     merge_sort(a, s, mid, e);
114 }
115
116 //                              MERGE SORT
117 void standart_merge(std::vector<int>& a, int l, int r) {
118     if (l >= r) {
119         return;
120     }
121     int mid = l + (r - l) / 2;
122     standart_merge(a, l, mid);
123     standart_merge(a, mid + 1, r);
124     merge_sort(a, l, mid, r);
125 }
126
127 //

128 class Test {
129 public:
130     long long measureMergeSortTime(std::vector<int> array, int repetitions =
        10) {
131         long long totalTime = 0;
132         for (int i = 0; i < repetitions; ++i) {
133             auto start = std::chrono::high_resolution_clock::now();
134             standart_merge(array, 0, array.size() - 1);
135             auto elapsed = std::chrono::high_resolution_clock::now() - start;
136             totalTime += std::chrono::duration_cast<std::chrono::microseconds>(
                elapsed).count();
137         }
138         return totalTime / repetitions;
139     }
140
141     long long measureHybridSortTime(std::vector<int> array, int repetitions =
        10) {
142         long long totalTime = 0;
143         for (int i = 0; i < repetitions; ++i) {
144             auto start = std::chrono::high_resolution_clock::now();
145             merge(array, 0, array.size() - 1);
```

```
            auto elapsed = std::chrono::high_resolution_clock::now() - start;
            totalTime += std::chrono::duration_cast<std::chrono::microseconds>(
                elapsed).count();
        }
        return totalTime / repetitions;
    }
};

int main() {
    ArrayGenerator generator;
    Test tester;
    std::vector<int> sizes;
    for (int size = 500; size <= 10000; size += 100) {
        sizes.push_back(size);
    }

    std::ofstream csvFileMerge("merge_results.csv");
    std::ofstream csvFileHybrid("hybrid_results.csv");

    csvFileMerge << "Size,Random,gen_2,gen_3\n";
    csvFileHybrid << "Size,Random,gen_2,gen_3\n";
    for (int size : sizes) {
        std::vector<int> randomArray = generator.gen_1(size);
        std::vector<int> gen_2Array = generator.gen_2(size);
        std::vector<int> gen_3Array = generator.gen_3(size);

        long long randomTimeMerge = tester.measureMergeSortTime(randomArray);
        long long gen_2TimeMerge = tester.measureMergeSortTime(gen_2Array);
        long long gen_3TimeMerge = tester.measureMergeSortTime(gen_3Array);

        long long randomTimeHybrid = tester.measureHybridSortTime(randomArray);
        long long gen_2TimeHybrid = tester.measureHybridSortTime(gen_2Array);
        long long gen_3TimeHybrid = tester.measureHybridSortTime(gen_3Array);

        csvFileMerge << size << "," << randomTimeMerge << "," << gen_2TimeMerge
             << "," << gen_3TimeMerge << "\n";
        csvFileHybrid << size << "," << randomTimeHybrid << "," <<
            gen_2TimeHybrid << "," << gen_3TimeHybrid << "\n";

        std::cout << "Size: " << size
                  << " Random Merge: " << randomTimeMerge << " microsec"
                  << " gen_2 Merge: " << gen_2TimeMerge << " microsec"
                  << " gen_3 Merge: " << gen_3TimeMerge << " microsec"
                  << "\n";
    }
    csvFileMerge.close();
    csvFileHybrid.close();
}
```

2.2 Результаты экспериментов

На основании проведенных экспериментов были построены следующие графики, отображающие зависимость времени выполнения алгоритмов от размера массива N.

3 Анализ и выводы

На основании графиков можно сделать следующие выводы:

1. **Эффективность гибридного алгоритма на случайных данных:** Для случайных массивов N, гибридный алгоритм MERGE+INSERTION SORT демонстрирует преимущество при $N \leq 1000$. Однако с увеличением размера массива стандартный MERGE SORT начинает показывать сопоставимые результаты.

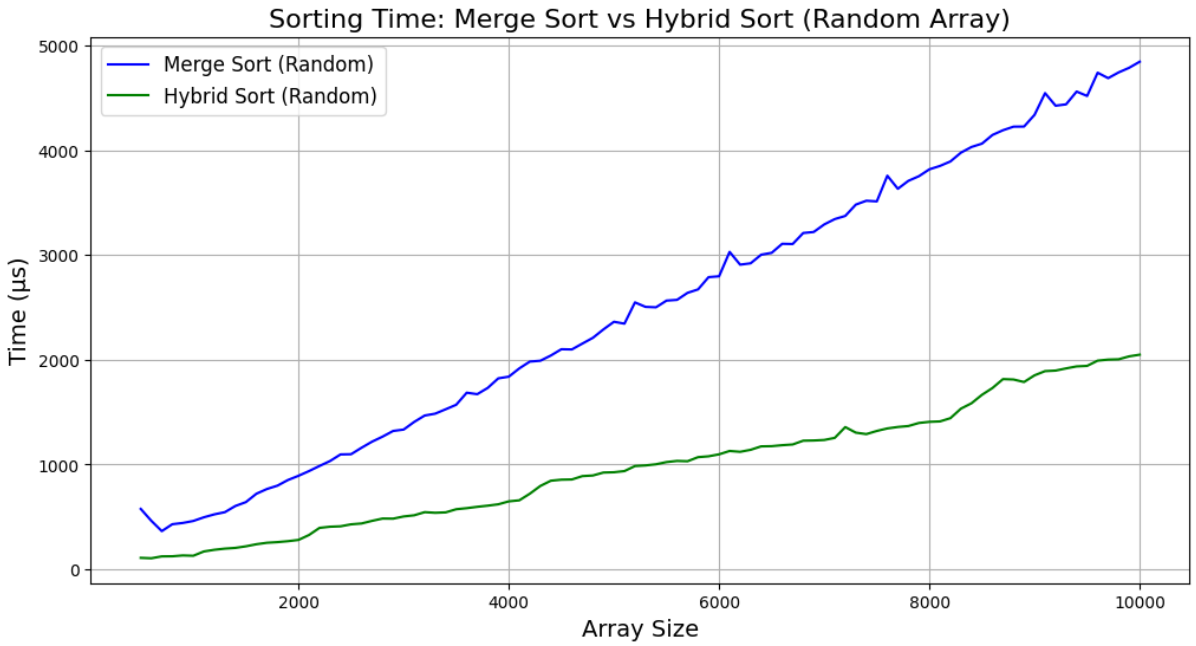

Рис. 1: Сравнение MERGE SORT и MERGE+INSERTION SORT на случайном массиве.

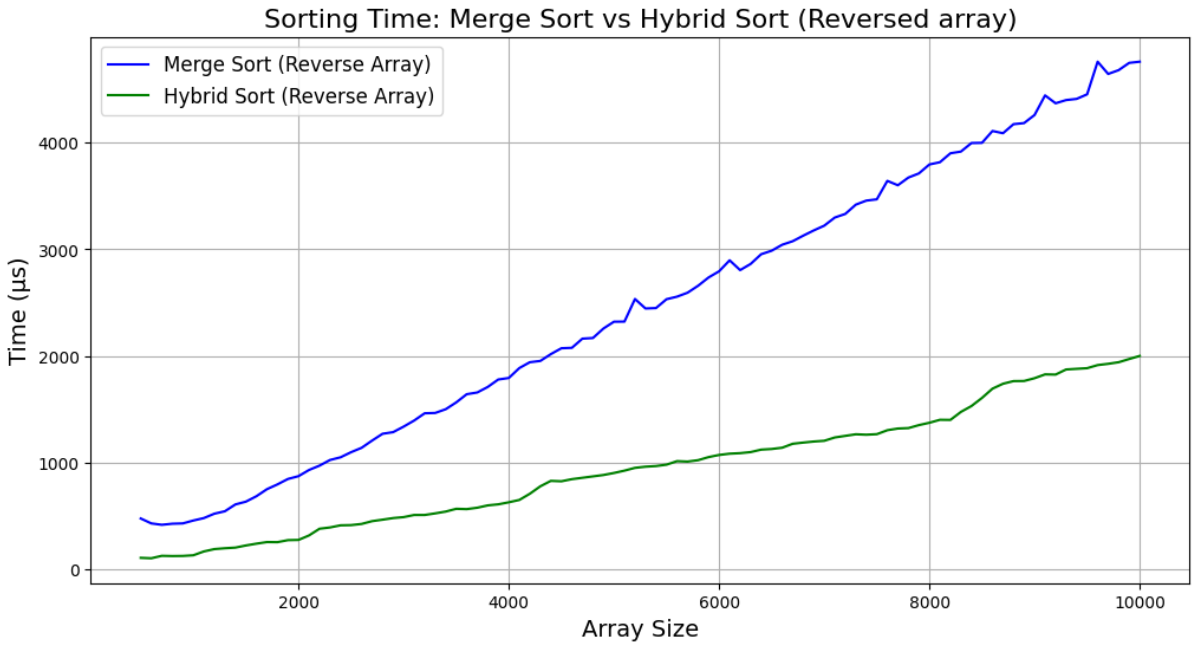

Рис. 2: Сравнение MERGE SORT и MERGE+INSERTION SORT на массиве с обратным порядком элементов.

2. **Эффективность на массиве с обратным порядком:** Для массивов, отсортированных в обратном порядке, гибридный алгоритм также выигрывает на малых размерах N, что обусловлено снижением затрат на разбиение массивов за счет вставки.

3. **Эффективность на почти отсортированных данных:** Гибридный алгоритм оказывается заметно быстрее, так как вставка работает практически за линейное время в таких условиях.

4. **Рекомендации по выбору алгоритма:**

   - Для малых объемов данных (до 1000 элементов) целесообразно использовать MERGE+INSERTION SORT.

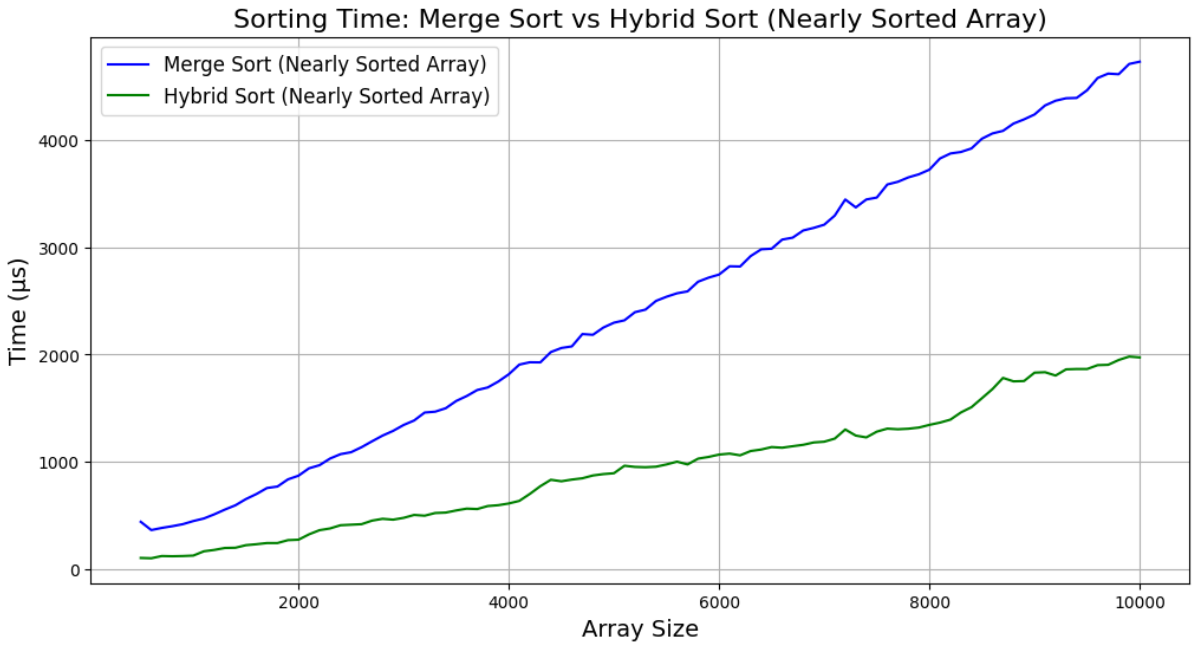

Рис. 3: Сравнение MERGE SORT и MERGE+INSERTION SORT на почти отсортированном массиве.

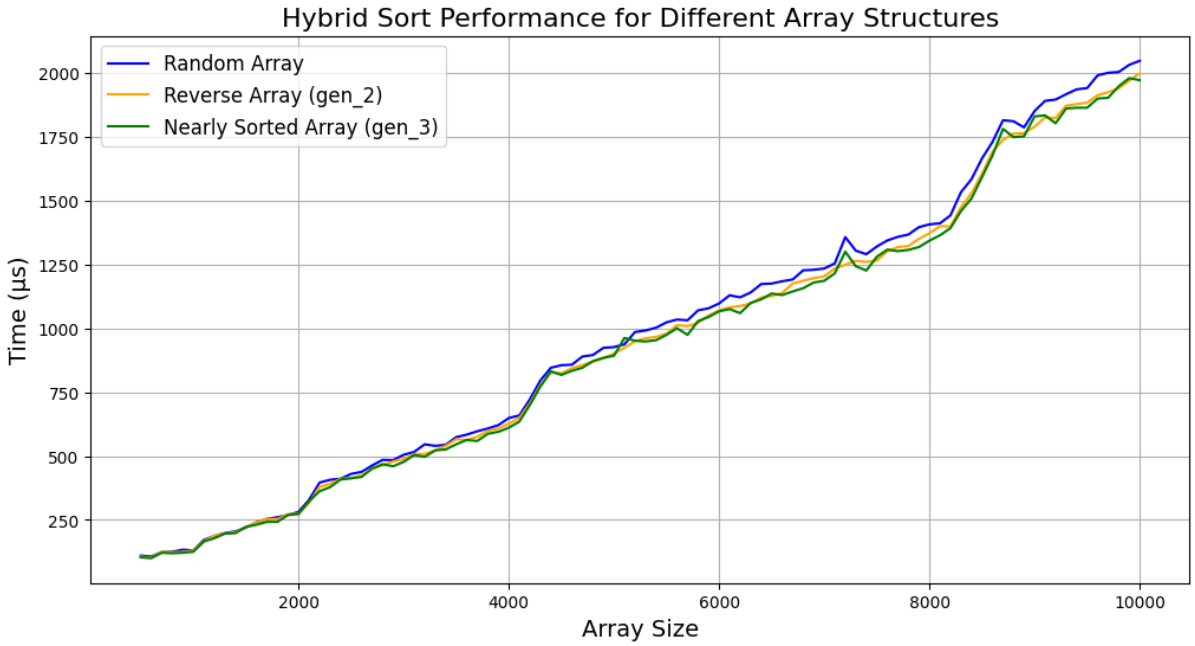

Рис. 4: Сравнение Hybrid sort для различных векторов.

- Для больших объемов данных и массивов со случайной структурой стандартный MERGE SORT показывает себя лучше за счет меньшей сложности на разбиении.
- Для массивов с частичным упорядочиванием гибридный алгоритм предпочтителен.

4 Заключение

В ходе работы проведено сравнение стандартного MERGE SORT и гибридного MERGE+INSERTION SORT. Было выявлено, что гибридный алгоритм имеет преимущество на малых объемах данных и для массивов с частичной упорядоченностью. Стандартный алгоритм более универсален и пока-

зывает стабильные результаты на больших объемах данных.