

Id_cf: 348378062

Github: https://github.com/leolazzz/dsa_hse_set3

Вывод:

Гибридный алгоритм работает быстрее стандартного на всех типах данных. Оптимальный порог переключения - 20-30 элементов. При порогах больше 50 элементов гибридный алгоритм замедляется из-за квадратичной сложности Insertion Sort.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <set>
#include <string>
#include <stack>
#include <cmath>
#include <numeric>
#include <map>
#include <iomanip>
#include <random>
#include <chrono>

using ll = long long;
using ld = long double;

class ArrayGenerator {
public:
    std::vector<int> genAr(int n) {
        std::vector<int> a(n);
        for (int i = 0; i < n; ++i) {
            a[i] = rand() % 6000;
        }
        return a;
    }
}
```

```

    }

std::vector <int> genRevAr(int n) {
    std::vector <int> a = genAr(n);
    std::sort(a.begin(), a.end());
    std::reverse(a.begin(), a.end());
    return a;
}

std::vector <int> genAlmostAr(int n) {
    std::vector <int> a = genAr(n);
    std::sort(a.begin(), a.end());
    for (int i = 0; i < n / 100; ++i) {
        int i1 = rand() % n, i2 = rand() % n;
        std::swap(a[i1], a[i2]);
    }
    return a;
}
};


```

```

void insertionSort(std::vector <int>& a, int l, int r) {
    for (int i = l; i < r; ++i) {
        int tmp = a[i];
        int j = i - 1;
        for (; j >= l && a[j] > tmp; j--) {
            a[j + 1] = a[j];
        }
        a[j + 1] = tmp;
    }
}


```

```

void merge(std::vector <int>& a, int l, int m, int r) {
    std::vector <int> res(r - l);
    int i1 = 0, i2 = 0;
    while (l + i1 < m && m + i2 < r) {
        if (a[l + i1] <= a[m + i2]) {
            res[i1 + i2] = a[l + i1];

```

```

        i1++;
    } else {
        res[i1 + i2] = a[m + i2];
        i2++;
    }
}

while (l + i1 < m) {
    res[i1 + i2] = a[l + i1];
    i1++;
}
while (m + i2 < r) {
    res[i1 + i2] = a[m + i2];
    i2++;
}
for (int i = 0; i < i1 + i2; ++i) {
    a[l + i] = res[i];
}
}

void mergeSort(std::vector<int>& a, int l, int r) {
    if (l >= r - 1) {
        return;
    }
    int m = (l + r) >> 1;
    mergeSort(a, l, m);
    mergeSort(a, m, r);
    merge(a, l, m, r);
}

void combineSort(std::vector<int>& a, int l, int r, int threshold) {
    if (r - l <= threshold) {
        insertionSort(a, l, r);
        return;
    }
    int m = (l + r) >> 1;
    combineSort(a, l, m, threshold);
    combineSort(a, m, r, threshold);
}
```

```

combineSort(a, m, r, threshold);
merge(a, l, m, r);
}

class SortTester {
public:
    ll measureMerge(std::vector <int>& a) {
        auto start = std::chrono::high_resolution_clock::now();
        mergeSort(a, 0, a.size());
        auto elapsed = std::chrono::high_resolution_clock::now() - start;
        long long msec = std::chrono::duration_cast<std::chrono::milliseconds>(elapsed).count();
        return msec;
    }
    ll measureInser(std::vector <int>& a) {
        auto start = std::chrono::high_resolution_clock::now();
        insertionSort(a, 0, a.size());
        auto elapsed = std::chrono::high_resolution_clock::now() - start;
        long long msec = std::chrono::duration_cast<std::chrono::milliseconds>(elapsed).count();
        return msec;
    }
    ll measureCombine(std::vector <int>& a, int threshold) {
        auto start = std::chrono::high_resolution_clock::now();
        combineSort(a, 0, a.size(), threshold);
        auto elapsed = std::chrono::high_resolution_clock::now() - start;
        long long msec = std::chrono::duration_cast<std::chrono::milliseconds>(elapsed).count();
        return msec;
    }
};

int main() {
    ArrayGenerator gen;
    SortTester tes;

    std::vector<int> ar = gen.genAr(100000);
    std::vector<int> revAr = gen.genRevAr(100000);
}

```

```

std::vector<int> alAr = gen.genAlmostAr(100000);

std::vector<int> thresholds={5, 10, 20, 30, 50};
std::vector<int> sizes;
for (int size = 500; size <= 100000; size += 100) {
    sizes.push_back(size);
}

std::ofstream arOut("ar.csv");
std::ofstream revOut("revAr.csv");
std::ofstream alOut("alAr.csv");

arOut << "Size,Merge";
revOut << "Size,Merge";
alOut << "Size,Merge";
for (int th : thresholds) {
    arOut << ",Combine_Th" << th;
    revOut << ",Combine_Th" << th;
    alOut << ",Combine_Th" << th;
}
arOut << "\n";
revOut << "\n";
alOut << "\n";

for (int size : sizes) {
    std::vector<int> subAr(ar.begin(), ar.begin() + size);
    std::vector<int> subRev(revAr.begin(), revAr.begin() + size);
    std::vector<int> subAl(alAr.begin(), alAr.begin() + size);
    arOut << size << "," << tes.measureMerge(subAr);
    revOut << size << "," << tes.measureMerge(subRev);
    alOut << size << "," << tes.measureMerge(subAl);
    for (int th : thresholds) {
        subAr = std::vector<int>(ar.begin(), ar.begin() + size);
        subRev = std::vector<int>(revAr.begin(), revAr.begin() + size);

```

```

subAl = std::vector<int>(alAr.begin(), alAr.begin() + size);

arOut << "," << tes.measureCombine(subAr, th);
revOut << "," << tes.measureCombine(subRev, th);
alOut << "," << tes.measureCombine(subAl, th);

}

arOut << "\n";
revOut << "\n";
alOut << "\n";

}

return 0;
}

```

```

import pandas as pd
import matplotlib.pyplot as plt

ar_df = pd.read_csv('ar.csv')
rev_df = pd.read_csv('revAr.csv')
al_df = pd.read_csv('alAr.csv')

def plot_comparison(df, title):
    plt.figure(figsize=(12, 6))

    for column in df.columns[1:]:
        plt.plot(df['Size'], df[column], label=column)

    plt.xlabel('Размер массива')
    plt.ylabel('Время (мс)')
    plt.title(title)
    plt.legend()
    plt.grid(True)
    plt.show()

plot_comparison(ar_df, 'Случайные массивы')
plot_comparison(rev_df, 'Обратно отсортированные массивы')
plot_comparison(al_df, 'Почти отсортированные массивы')

def find_best_threshold(df, title):
    threshold_cols = [col for col in df.columns if 'Th' in col]

    best_times = []
    for threshold in threshold_cols:
        avg_time = df[threshold].mean()
        best_times.append((threshold, avg_time))

    best_times.sort(key=lambda x: x[1])

```

```
print(f"\n{title}:")
for threshold, time in best_times[:3]:
    print(f"  {threshold}: {time:.2f} мс в среднем")

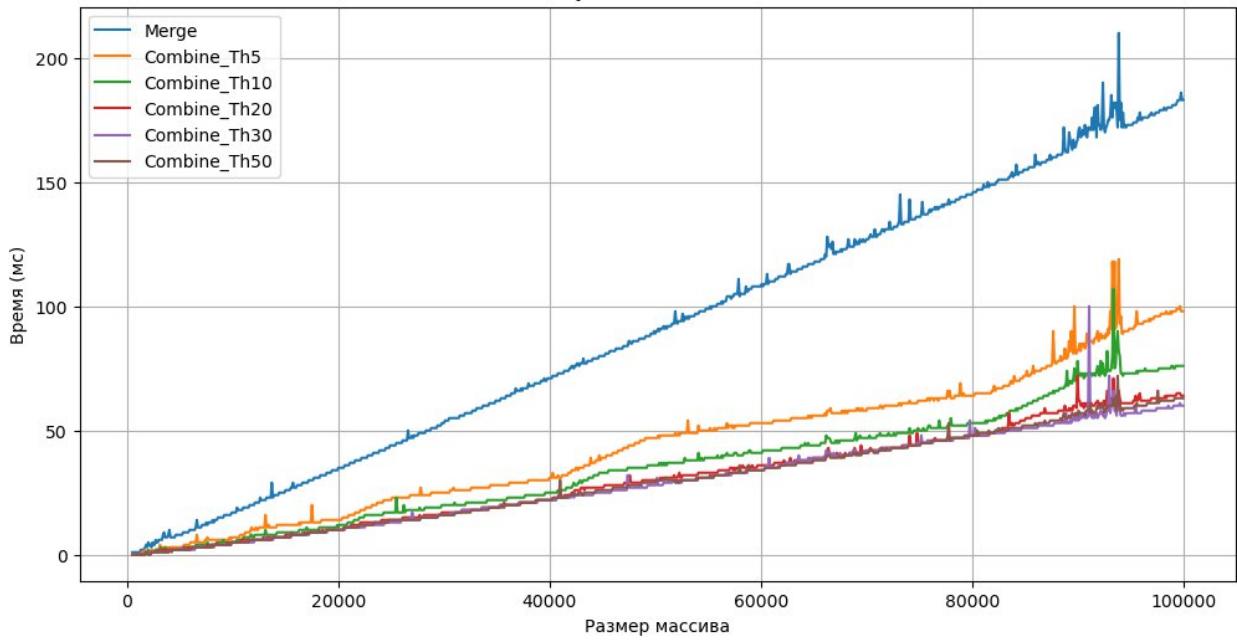
find_best_threshold(ar_df, "Случайные массивы")
find_best_threshold(rev_df, "Обратно отсортированные массивы")
find_best_threshold(al_df, "Почти отсортированные массивы")

def compare_standard_vs_best(df, title):
    standard_times = df['Merge']
    threshold_cols = [col for col in df.columns if 'Th' in col]
    best_hybrid_times = df[threshold_cols].min(axis=1)
    improvement = ((standard_times - best_hybrid_times) / standard_times * 100).mean()

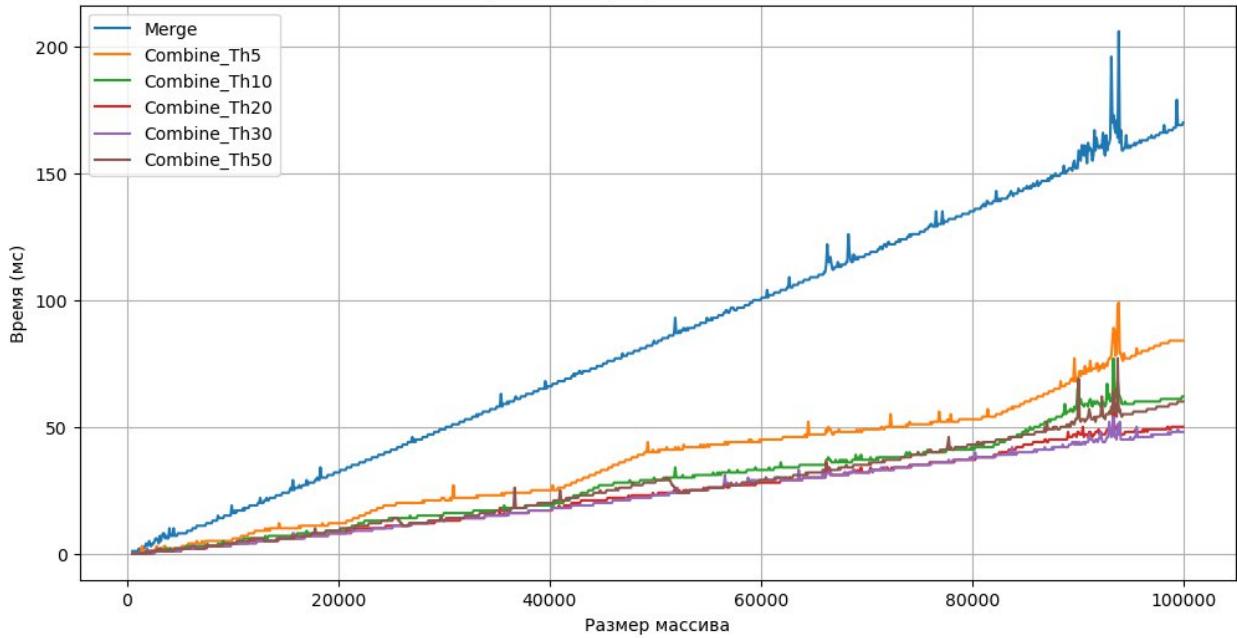
    print(f"\n{title}")
    print(f"  Гибридный алгоритм быстрее на {improvement:.1f}% в среднем")

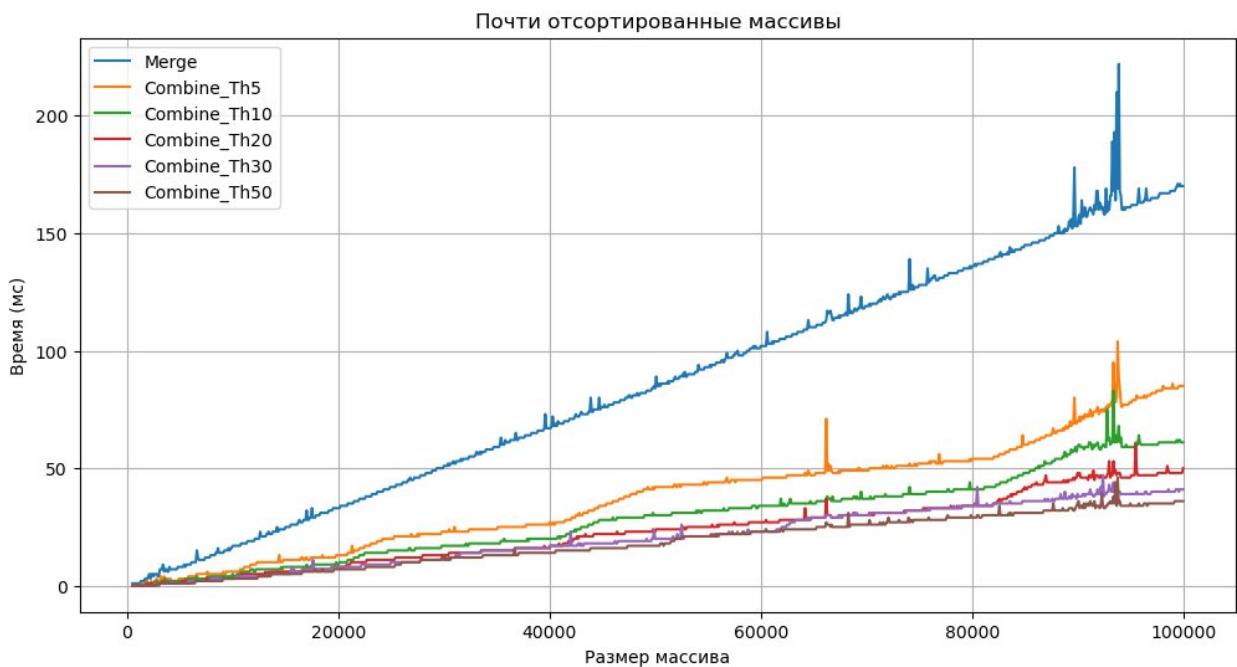
compare_standard_vs_best(ar_df, "Случайные массивы")
compare_standard_vs_best(rev_df, "Обратно отсортированные массивы")
compare_standard_vs_best(al_df, "Почти отсортированные массивы")
```

Случайные массивы



Обратно отсортированные массивы





Случайные массивы:

Combine_Th30: 29.19 мс в среднем

Combine_Th50: 29.49 мс в среднем

Combine_Th20: 30.44 мс в среднем

Обратно отсортированные массивы:

Combine_Th30: 23.17 мс в среднем

Combine_Th20: 23.68 мс в среднем

Combine_Th50: 26.39 мс в среднем

Почти отсортированные массивы:

Combine_Th50: 18.37 мс в среднем

Combine_Th30: 20.71 мс в среднем

Combine_Th20: 22.97 мс в среднем

Случайные массивы:

Гибридный алгоритм быстрее на 69.9% в среднем

Обратно отсортированные массивы:

Гибридный алгоритм быстрее на 74.3% в среднем

Почти отсортированные массивы:

Гибридный алгоритм быстрее на 79.7% в среднем