ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИТМО»

Отчёт по лабораторной работе № 5

«Алгоритмы сортировок»

Выполнил работу

Фиолетов Э.А.

Академическая группа №3114

Принял

Дунаев М.В.

Санкт-Петербург

2024

Отчет по лабораторной работе

1. Введение

Цель работы: Разработка и тестирование трех различных алгоритмов сортировки: Comb Sort, Quick Sort и Tim Sort. Оценка их производительности на массивах различного размера и типов данных.

Задачи:

1. Реализовать алгоритмы Comb Sort, Quick Sort и Tim Sort.
2. Выполнить анализ производительности каждого алгоритма с использованием разных входных данных.
3. Построить графики зависимости времени выполнения от размера массива.
4. Оценить теоретическую и практическую сложность алгоритмов.

2. Теоретическая подготовка

Comb Sort:

- Основан на улучшении пузырьковой сортировки.
- Основная идея: использование переменного “шага” для сравнения элементов, который уменьшается на каждом шаге.
- Сложность: в среднем , в худшем случае .

Quick Sort:

- Рекурсивный алгоритм, основанный на разбиении массива на две части относительно опорного элемента (pivot).
- Сложность: в среднем , в худшем случае .

Tim Sort:

- Гибридная сортировка, объединяющая Merge Sort и Insertion Sort.
- Используется для сортировки больших массивов. Часто применяется в стандартных библиотеках языков программирования.
- Сложность: в среднем и худшем случае.

Оценка сложности:

Name	Best	Average	Worst	Memory	Stable
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$	Depends
Merge sort	$n \log n$	$n \log n$	$n \log n$	Depends	Yes
In-place Merge sort	—	—	$n (\log n)^2$	1	Yes
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No
Insertion sort	n	n^2	n^2	1	Yes
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No
Selection sort	n^2	n^2	n^2	1	Depends
Timsort	n	$n \log n$	$n \log n$	n	Yes
Shell sort	n	$n(\log n)^2$	$O(n\log^2 n)$	1	No
Bubble sort	n	n^2	n^2	1	Yes
Binary tree sort	n	$n \log n$	$n \log n$	n	Yes
Cycle sort	—	n^2	n^2	1	No
Library sort	—	$n \log n$	n^2	n	Yes
Patience sorting	—	—	$n \log n$	n	No
Smoothsort	n	$n \log n$	$n \log n$	1	No
Strand sort	n	n^2	n^2	n	Yes
Tournament sort	—	$n \log n$	$n \log n$		
Cocktail sort	n	n^2	n^2	1	Yes
Comb sort	—	—	n^2	1	No
Gnome sort	n	n^2	n^2	1	Yes
Bogosort	n	$n \cdot n!$	$n \cdot n! \to \infty$	1	No

3. Реализация

Этапы выполнения:

1. Реализованы алгоритмы Comb Sort, Quick Sort и Tim Sort в виде отдельных функций:
   1.1. Comb Sort использует уменьшение шага и проверку на завершение сортировки.
   1.2. Quick Sort реализован с помощью разбиения (partition) и рекурсивного вызова для двух подмассивов.
   1.3. Tim Sort состоит из двух частей: сортировка вставкой для подмассивов и объединение (merge) уже отсортированных частей.
2. Проведено тестирование на заранее заданных наборах данных, чтобы убедиться в корректности реализации.
3. Написан бенчмарк, который измеряет время выполнения алгоритмов на случайных массивах различного размера.
4. Сгенерированы два CSV-файла с результатами экспериментов.

Фрагменты кода:

- **Comb Sort**:

```
void combSort(std::vector<int>& arr) {
   int n = arr.size();
   if (n < 2) return;

   int gap = n;
   bool swapped = true;

   while (gap != 1 || swapped) {
       gap = std::max(1, (int)(gap / 1.3));
       swapped = false;

       for (int i = 0; i < n - gap; ++i) {
           if (arr[i] > arr[i + gap]) {
               std::swap(arr[i], arr[i + gap]);
               swapped = true;
           }
       }
   }
}
```

- **Quick Sort**:

```
int partition(std::vector<int>& arr, int low, int high) {
   int pivot = arr[high];
   int i = low - 1;

   for (int j = low; j < high; ++j) {
       if (arr[j] < pivot) {
           ++i;
           std::swap(arr[i], arr[j]);
       }
   }
   std::swap(arr[i + 1], arr[high]);
   return i + 1;
}

void quickSort(std::vector<int>& arr, int low, int high) {
   if (low >= high) return;

   int pi = partition(arr, low, high);

   quickSort(arr, low, pi - 1);
   quickSort(arr, pi + 1, high);
}
```

- **Tim Sort**:

```
void timSort(std::vector<int>& arr) {
   int n = arr.size();
   int RUN = std::max(16, static_cast<int>(std::log2(n) * 2));
```

```
auto insertionSort = [](std::vector<int>& arr, int left, int right) {
    for (int i = left + 1; i <= right; ++i) {
        int temp = arr[i];
        int j = i - 1;
        while (j >= left && arr[j] > temp) {
            arr[j + 1] = arr[j];
            --j;
        }
        arr[j + 1] = temp;
    }
};

auto merge = [](std::vector<int>& arr, int left, int mid, int right) {
    int len1 = mid - left + 1, len2 = right - mid;
    std::vector<int> leftArr(len1), rightArr(len2);

    for (int i = 0; i < len1; ++i) leftArr[i] = arr[left + i];
    for (int i = 0; i < len2; ++i) rightArr[i] = arr[mid + 1 + i];

    int i = 0, j = 0, k = left;
    while (i < len1 && j < len2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k] = leftArr[i];
            ++i;
        } else {
            arr[k] = rightArr[j];
            ++j;
        }
        ++k;
    }

    while (i < len1) {
        arr[k] = leftArr[i];
        ++i;
        ++k;
    }

    while (j < len2) {
        arr[k] = rightArr[j];
        ++j;
        ++k;
    }
};

for (int i = 0; i < n; i += RUN) {
    insertionSort(arr, i, std::min(i + RUN - 1, n - 1));
}

for (int size = RUN; size < n; size = 2 * size) {
    for (int left = 0; left < n; left += 2 * size) {
        int mid = std::min(left + size - 1, n - 1);
        int right = std::min(left + 2 * size - 1, n - 1);

        if (mid < right) {
            merge(arr, left, mid, right);
        }
    }
}
```

4. Экспериментальная часть

Условия эксперимента:

- Размеры массивов: от 1,000 до 1,000,000 элементов.
- Для каждого размера массивов проводились замеры времени выполнения.
- Каждому алгоритму задавались одинаковые случайные входные данные.

Результаты:

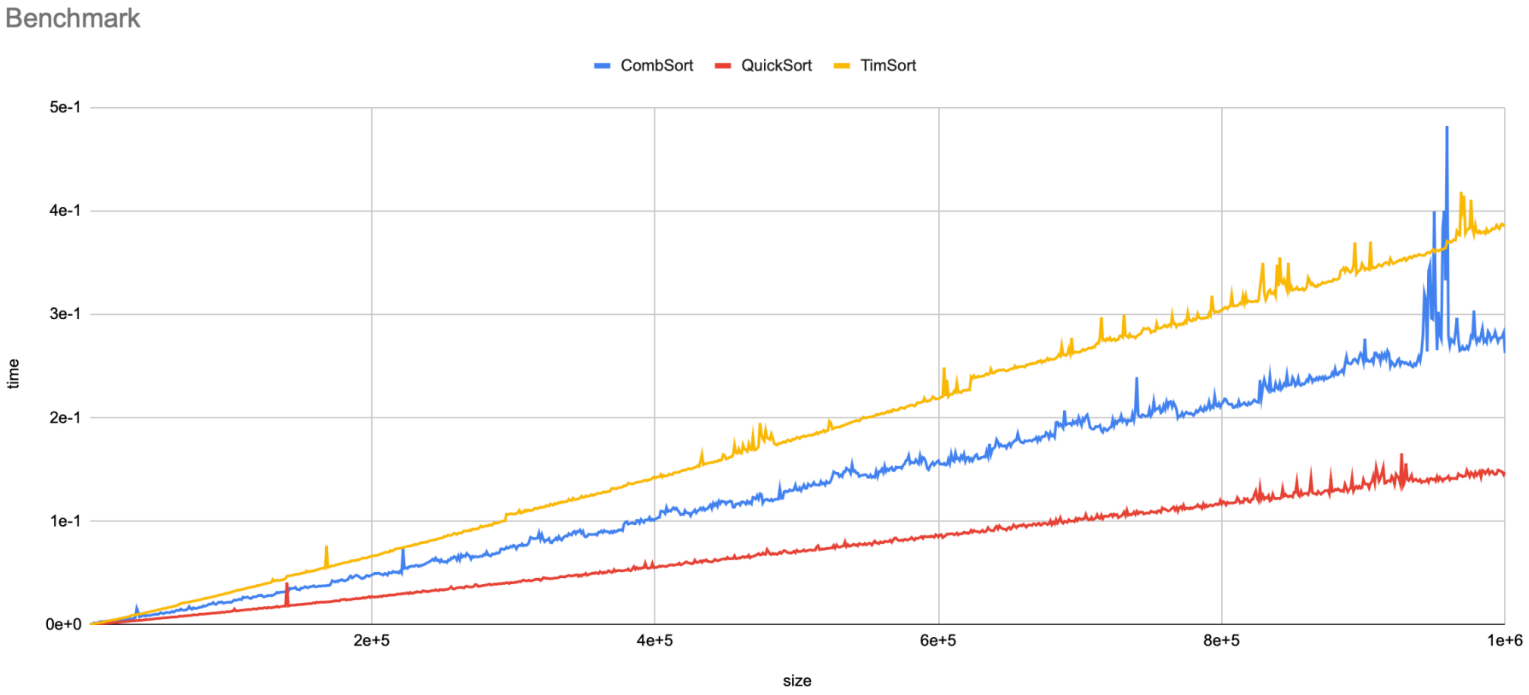
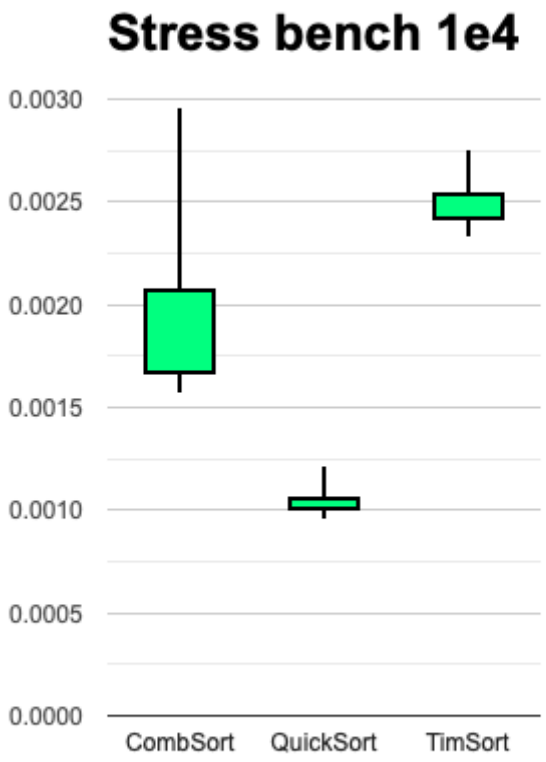
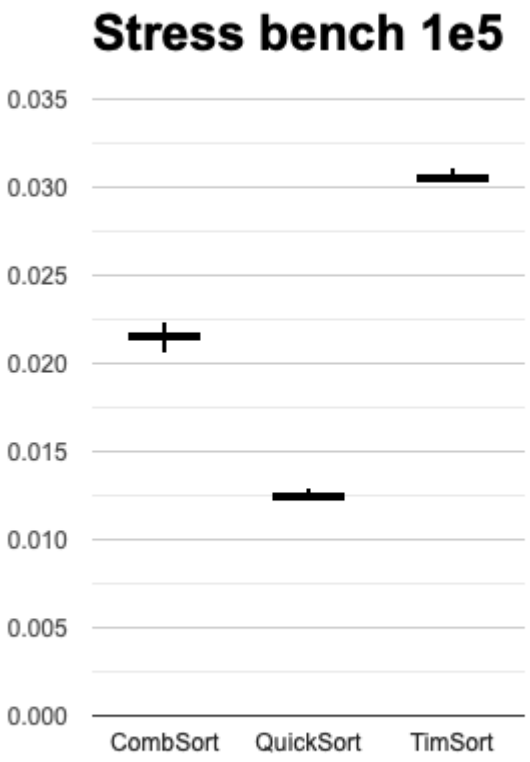


График стресс теста для 1e4 и 1e5





5. Заключение

В ходе выполнения работы были реализованы и протестированы три алгоритма сортировки. Проведенные эксперименты показали:

1. Quick Sort продемонстрировал наилучшую производительность среди всех алгоритмов.
2. Tim Sort также показал хорошие результаты, особенно на средних объемах данных.
3. Comb Sort оказался менее эффективным из-за худшей асимптотики.

Дальнейшие направления исследования:

1. Оптимизация реализации Quick Sort для уменьшения потребления памяти.
2. Исследование производительности алгоритмов при сортировке данных с различной степенью упорядоченности.
3. Реализация параллельных версий алгоритмов для ускорения работы на больших объемах данных.

6. Приложения

Полный исходный код программы: https://github.com/ITMO-ML-algorithms-and-data-structures/polygon/pull/624