Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: “Объектно-ориентированное программирование”
Тема: “Сортировки”

Студенты гр. 3331506/20401 Орлов М. С.

Преподаватель Ананьевский М. С.

Санкт-Петербург
2025

Оглавление

Введение

Сортировка — алгоритм упорядочивания элементов в массиве по определенному признаку. Это одна из самых важных задач при работе с данными, так как служит для оптимизации работы алгоритмов обработки данных за счет упрощения поиска нужных элементов.

В работе будут рассмотрены основные алгоритмы сортировки, такие как: Bubble Sort, Heap Sort, Insertion Sort, Merge Sort, Quick Sort. Все из перечисленных алгоритмов принимают на вход массив элементов (список элементов) и возвращают массив элементов упорядоченный в соответствие с заданным признаком сравнения, например массив целых чисел, расположенных в порядке возрастания.

Задача курсовой работы заключается в том, чтобы описать принципы работы данных алгоритмов, реализовать их на языке C++, провести исследование скорости работы алгоритмов при различных размерах входных данных, а также сравнить их и отметить для каких случаев алгоритмы лучше применимы.

Bubble Sort

Bubble Sort (сортировка пузырьком) — простейший алгоритм сортировки, часто считается учебным из-за малой эффективности, на практике не применяется.

Алгоритм работает по следующему принципу: выполняется некоторое количество проходов по всему массиву элементов, в каждом перебираются с начала массива n-1 пар соседних элементов (где n — количество элементов в массиве). Элементы в паре сравниваются, и производится обмен элементов, если порядок в паре неправильный. Сортировка завершается либо после n-1 проходов, либо в случае если массив полностью отсортирован (то есть за проход по массиву не потребовалось совершать обмен элементов).

Вычислительная сложность: $O(n^2)$ — для наихудшего случая и для среднего. Выделение дополнительной памяти под элементы не требуется, массив сортируется «на месте».

Heap Sort

Heap Sort (пирамидальная сортировка) — алгоритм, использующий для сортировки расположение элементов в виде бинарного сортирующего дерева. Был предложен Дж. Уильямсом в 1963 году.

Бинарное сортирующее дерево (куча) — это бинарное дерево, для которого выполняются следующие условия: глубина всех листьев отличается не более чем на один слой, значение в любой вершине не меньше (max-heap) значений его потомков. Алгоритм работы сортировки заключается в следующем: сначала составляем кучу из элементов входного массива, затем удаляем из кучи наибольший элемент — корень, после чего переставляем оставшиеся элементы так, чтобы восстановить свойства кучи и повторяем процесс.

Вычислительная сложность: O(n*log(n)) — независимо от входных данных. Выделение дополнительной памяти под элементы не требуется.

Вычислительная сложность постоянная, что важно, когда требуется гарантировать скорость выполнения на любых данных, однако это означает что на почти отсортированных массивах алгоритм работает так же долго, алгоритм плохо сочетается с кэшированием, так как требует хаотичного доступа по всей длине массива. Также данный алгоритм не распараллеливается.

Insertion Sort

Insertion Sort (сортировка вставками) — алгоритм заключающийся во вставке элементов на нужную позицию в отсортированной части массива.

Алгоритм работает по следующему принципу: отсортированная последовательность элементов хранится в начале массива. На каждой итерации для очередного элемента из неотсортированной части линейным поиском находится позиция в отсортированной последовательности. После чего часть последовательности после этой позиции сдвигается и новый элемент вставляется на освободившееся место.

Вычислительная сложность: $O(n^2)$ — для наихудшего случая и для среднего, однако из-за константных множителей на малых значениях n данный алгоритм будет работать быстрее, чем алгоритм с более низким порядком роста. Выделение дополнительной памяти под элементы не требуется.

Merge Sort

Merge Sort (сортировка слиянием) — алгоритм работающий по принципу «разделяй и властвуй».

Алгоритм работает по следующему принципу: исходный массив разбивается на отдельные списки, в каждом из которых в начале расположен только один элемент. Затем пары уже отсортированных списков объединяются слиянием: на каждой итерации из начала первого или второго списка пары выбирается наименьший элемент и прибавляется в конец нового списка.

Вычислительная сложность: O(n*log(n)) — независимо от входных данных. Для работы с массивом требуется выделение памяти по размеру входного массива, однако при работе со связными списками выделение дополнительной памяти не требуется.

Вычислительная сложность постоянная, также алгоритм хорошо сочетается с кэшированием, а также возможно сохранение промежуточных результатов сортировки на диск, что полезно при работе с большими базами данных, также есть возможность распараллеливания.

Quick Sort

Quick Sort (быстрая сортировка) — один из самых быстрых универсальных алгоритмов сортировки, аналогично построен по принципу «разделяй и властвуй». Предложен Тони Хоаром в 1960 году.

Алгоритм работает по следующему принципу: в массиве выбирается опорный элемент, затем происходит разбиение массива на две части так, чтобы в одной оказались элементы меньше опорного, а в другой части соответственно больше. После чего аналогичные действия рекурсивно повторяются для каждой из полученных частей. Опорный элемент выбирается случайным образом, чтобы исключить возможность подать на вход массив, с наихудшим для алгоритма расположением элементов.

Вычислительная сложность: $O(n^2)$ — для наихудшего случая и $O(n*\log(n))$ — для среднего. Выделение дополнительной памяти под элементы не требуется.

Алгоритм хорошо сочетается с кэшированием, допускает распараллеливание. Однако в худшем случае может деградировать до квадратичной сложности, что может повлечь за собой переполнение стека из-за большой глубины рекурсии, нет гарантий времени выполнения.

Тестирование алгоритмов

Для тестирования сортировок напишем небольшую программу. Длины массивов входных данных для измерений записаны в массиве, тестирующая программа будет брать длины входных данных из массива, генерировать массив со случайными числами указанной длины и запускать сортировку. Время работы алгоритмов будет измеряться с помощью функции clock(), так как она измеряет процессорное время, потраченное на текущую задачу, что позволит исключить влияние загруженности компьютера на результаты эксперимента. Результаты измерений будут выводиться в стандартный поток вывода в формате csv, тогда останется лишь перенаправить вывод в соответствующий файл.

Полученные данные (в ячейках — время работы алгоритмов в микросекундах):

Data length	bubble	heap	insertion	merge	quick
5	2	4	2	9	3

10	2	3	1	5	2
20	8	7	4	12	7
50	46	21	12	19	23
100	187	45	47	41	48
200	755	104	170	93	117
500	2204	295	1021	256	237
1000	4976	649	1755	598	457
5000	140963	1502	28080	3183	1540
10000	617124	2297	117718	3690	1388
20000	2557030	5093	487435	18576	2878

Построим графики в программе LibreOffice Calc — зависимость времени в микросекундах от числа элементов исходного массива.

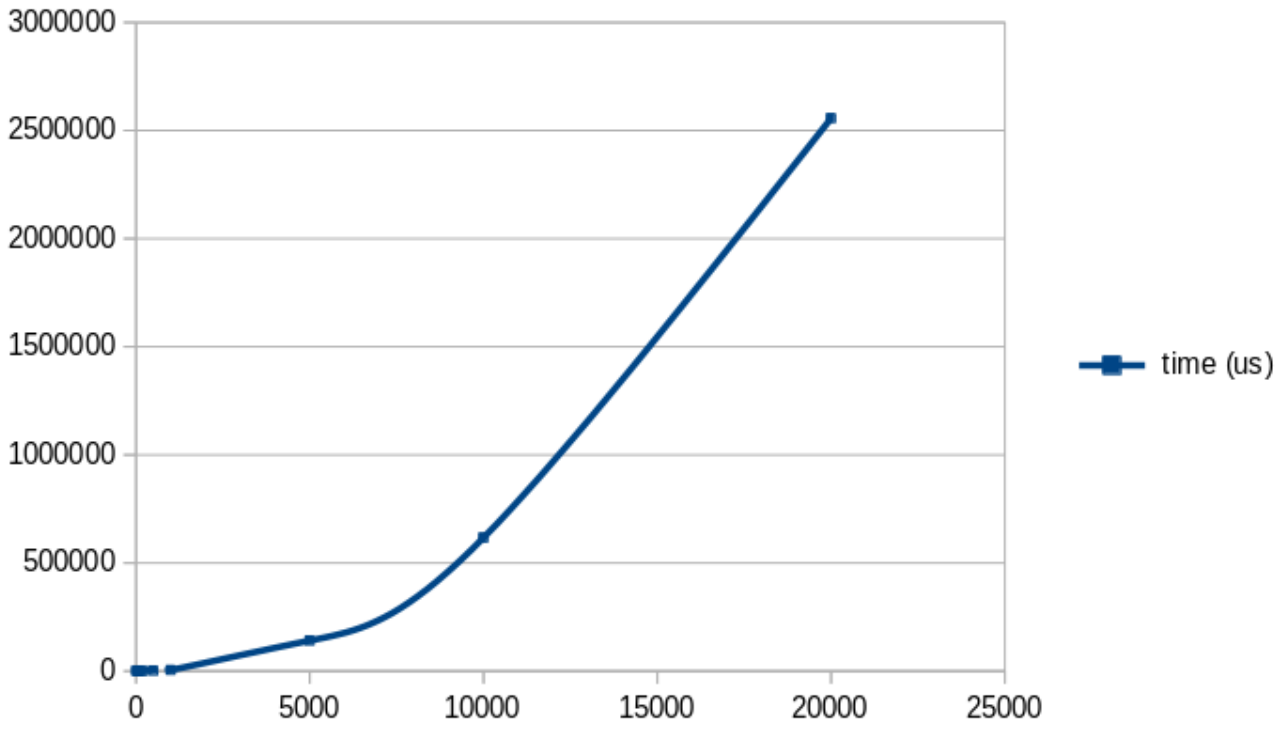

Рисунок 1. Bubble Sort

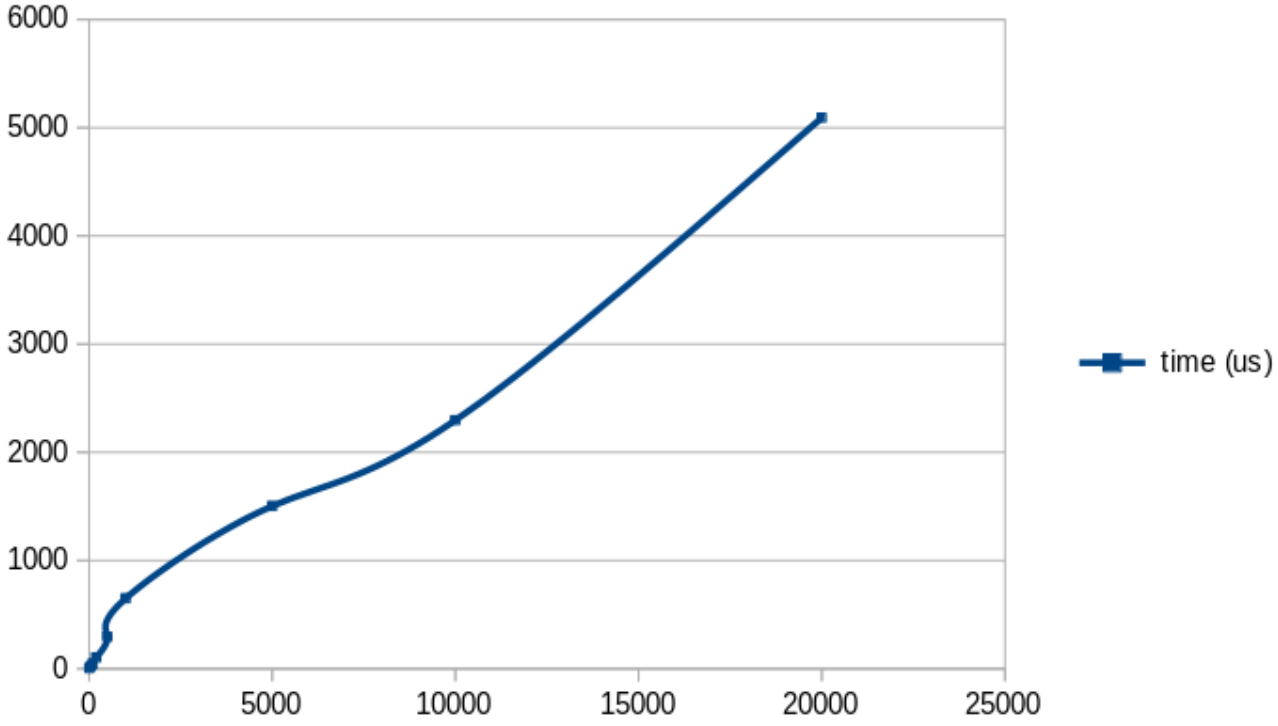

Рисунок 2. Heap Sort

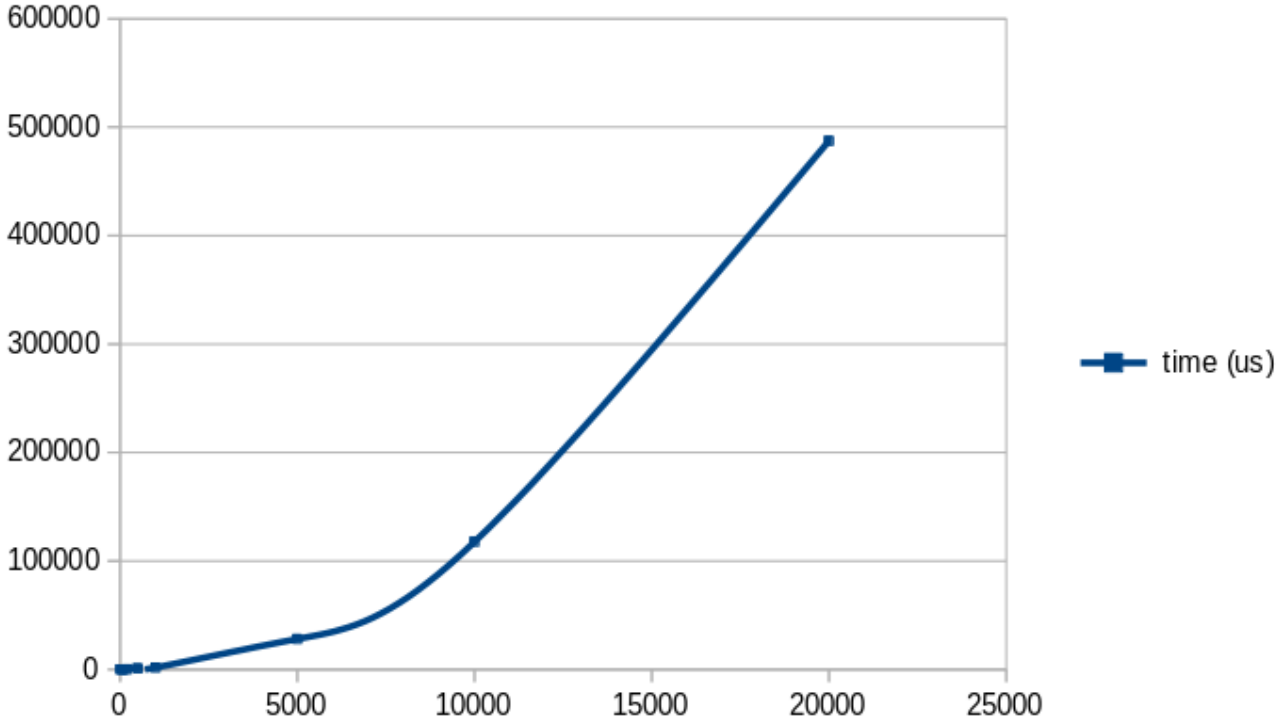

Рисунок 3. Insertion Sort

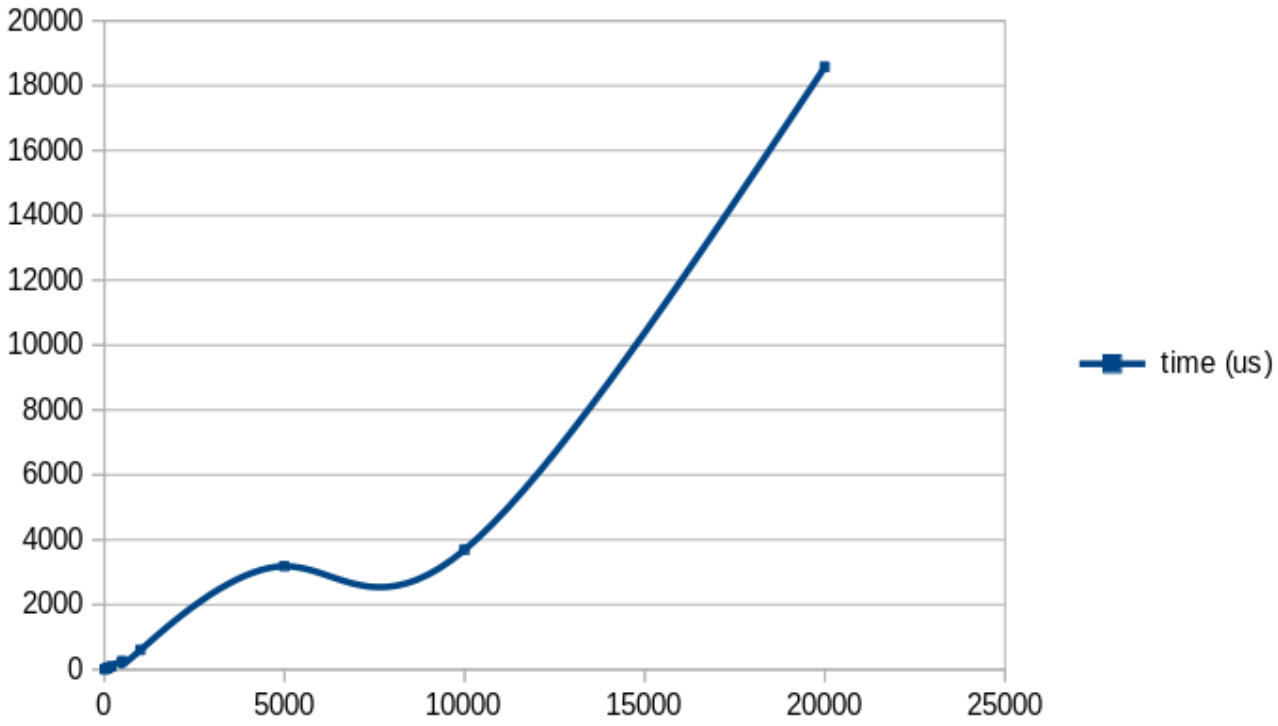

Рисунок 4. Merge Sort

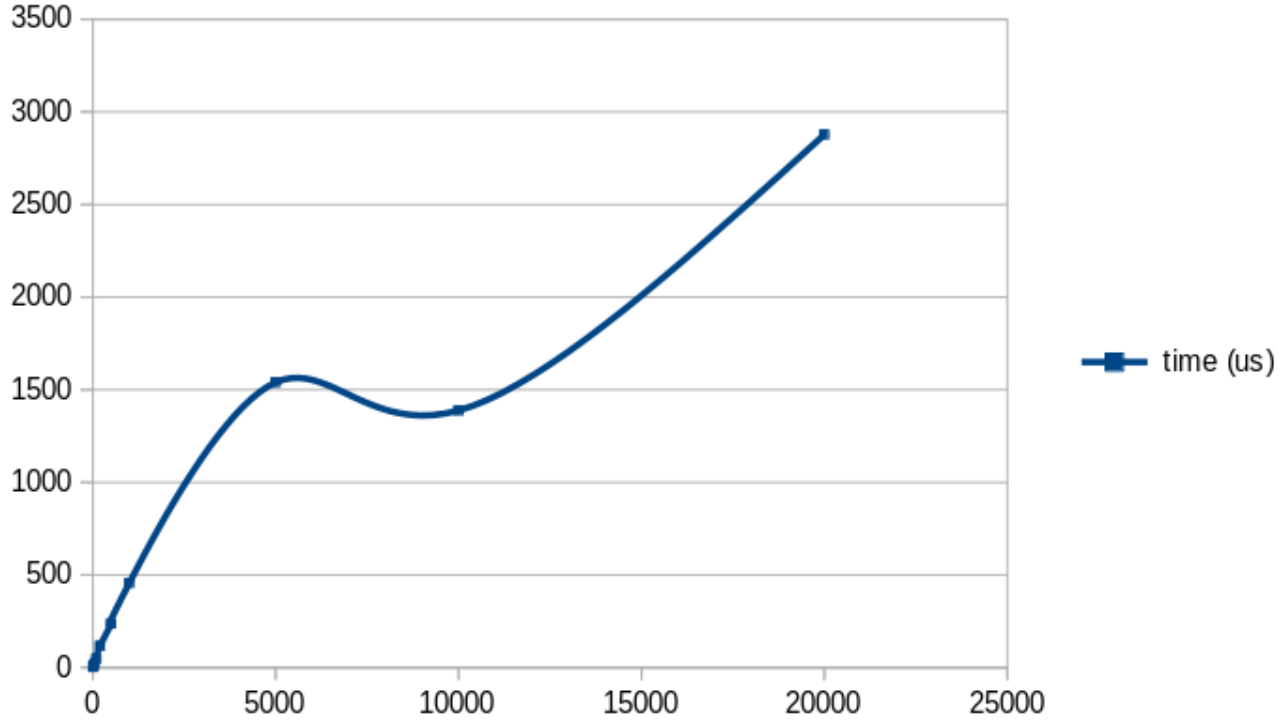

Рисунок 5. Quick Sort

Также построим все зависимости на одном графике:

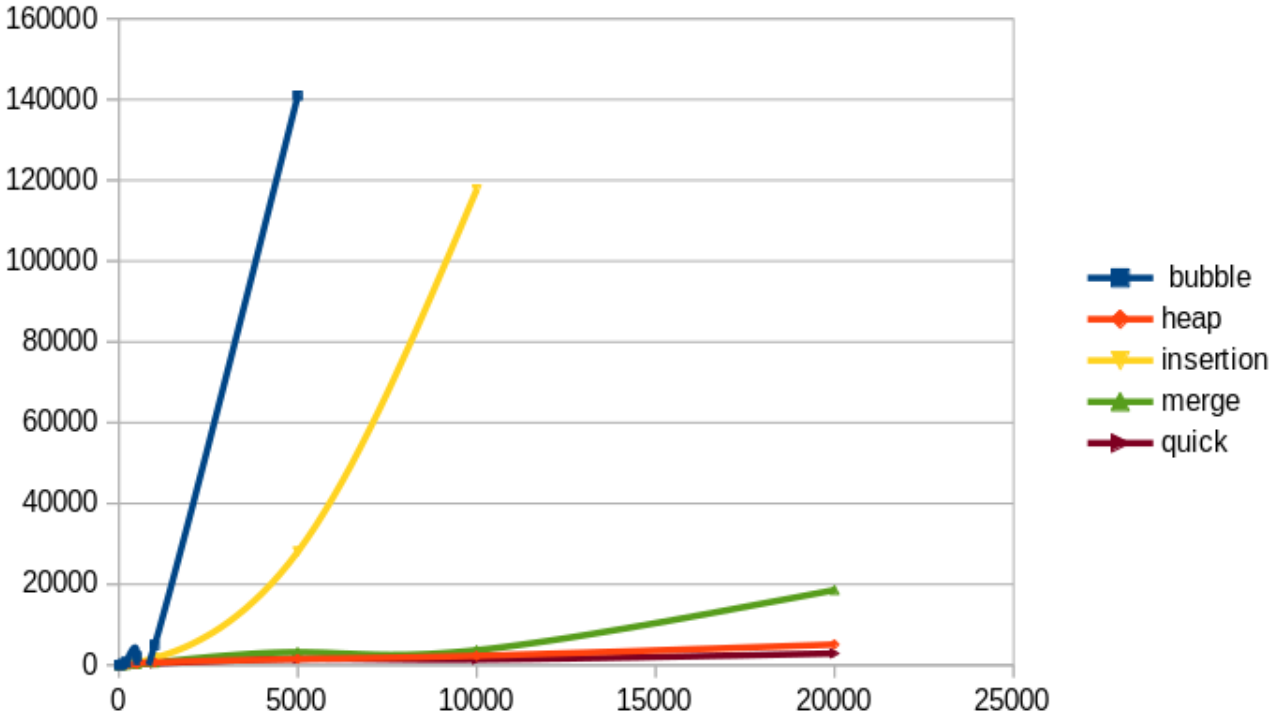

Рисунок 6. Все алгоритмы сортировок

Вывод

В работе были реализованы и исследованы различные алгоритмы сортировки.

Bubble Sort показал себя очень неэффективным, и поэтому на практике не применяется.

Heap Sort сравним по скорости с Quick Sort, и обладает преимуществом в виде постоянной вычислительной сложности, то есть не деградирует при любых входных данных, однако распараллеливание алгоритма невозможно.

Insertion Sort при большой асимптотической сложности (растущей по квадрату) оказался наиболее эффективным на данных малых размеров, что может быть использовано, например в гибридных методах сортировки: можно в быстрой сортировке для подмассивов малой длины вызывать именно сортировку вставками.

Merge Sort оказался медленнее Heap Sort и Quick Sort, однако хорошо сочетается с распараллеливанием и имеет постоянную вычислительную сложность.

Quick Sort в тестах на больших массивах случайных данных показал наибольшую скорость, и также при желании алгоритм можно распараллелить, однако скорость работы не гарантирована, и при неудачно подобранных данных алгоритм может деградировать.

Список литературы

[1] Левитин А. В. Глава 3. Метод грубой силы: Пузырьковая сортировка // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006. — С. 144—146. — 576 с.

[2] Левитин А. В. Глава 6. Метод преобразования: Пирамиды и пирамидальная сортировка // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006. — С. 275—284. — 576 с.

[3] Кнут Д. Э. 5.2.1 Сортировка путём вставок // Искусство программирования. Том 3. Сортировка и поиск = The Art of Computer Programming. Volume 3. Sorting and Searching / под ред. В. Т. Тертышного (гл. 5) и И. В. Красикова (гл. 6). — 2-е изд. — Москва: Вильямс, 2007. — Т. 3. — 832 с.

[4] Левитин А. В. Глава 4. Метод декомпозиции: Сортировка слиянием // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006. — С. 169—172. — 576 с.

[5] Hoare C. A. R. Algorithm 64: Quicksort (англ.) // Communications of the ACM — New York City: Association for Computing Machinery, 1961. — Vol. 4, Iss. 7. — P. 321

Приложение

Файл comparator.h

```
#pragma once


template<typename Element>
using Comparator = bool(*)(const Element &, const Element &);


template<typename Element>
bool is_greater(const Element &A, const Element &B)
{
    return A > B;
}


template<typename Element>
bool is_less(const Element &A, const Element &B)
{
    return A < B;
}
```

Файл bubble_sort.h

```
#pragma once

#include <vector>

#include <cstddef>

#include "comparator.h"



template<typename Element>

void bubble_sort(std::vector<Element> &array, const Comparator<Element>comp = is_greater)

{

    size_t itr_num = array.size() - 1;  // iterations number


    for(size_t itr = 0; itr < itr_num; itr++) {

        bool is_sorted = true;


        for(size_t idx = 0; idx < itr_num; idx++) {

            if(comp(array[idx], array[idx + 1])) {

                std::swap(array[idx], array[idx + 1]);

                is_sorted = false;

            }

        }


        if(is_sorted) break;

    }

}
```

Файл heap_sort.h

```
#pragma once
#include <vector>
#include <cstddef>
#include "comparator.h"



template<typename Element>
void restore_heap(std::vector<Element> &array, size_t root_idx, const size_t heap_size, const Comparator<Element>comp)
{
    while(true)
    {
        size_t child_left_idx = 2 * root_idx + 1;
        size_t child_right_idx = 2 * root_idx + 2;


        size_t largest_idx = root_idx;
                if(child_left_idx < heap_size && comp(array[child_left_idx], array[largest_idx])) {
            largest_idx = child_left_idx;
        }
               if(child_right_idx < heap_size && comp(array[child_right_idx], array[largest_idx])) {
            largest_idx = child_right_idx;
        }


        if(largest_idx != root_idx) {
            std::swap(array[largest_idx], array[root_idx]);
            root_idx = largest_idx;
        }
        else break;
    }
}
```

```
template<typename Element>
void build_heap(std::vector<Element> &array, const Comparator<Element>comp)
{
    size_t heap_size = array.size();

    for(size_t idx = heap_size / 2; idx > 0; idx--) {
        restore_heap(array, idx - 1, heap_size, comp);
    }
}



template<typename Element>
void heap_sort(std::vector<Element> &array, const Comparator<Element>comp = is_greater)
{
    build_heap(array, comp);

    for(size_t idx = array.size(); idx > 0; idx--) {
        std::swap(array[0], array[idx - 1]);
        restore_heap(array, 0, idx - 1, comp);
    }
}
```

Файл insertion_sort.h

```
#pragma once

#include <vector>

#include <cstddef>

#include "comparator.h"



template<typename Element>

void insertion_sort(std::vector<Element> &array, const Comparator<Element>comp =
is_greater)

{

    size_t idx_max = array.size();


    for(size_t idx = 1; idx < idx_max; idx++) {

        Element elem = array[idx];


        size_t insert_idx = idx;

         for(;insert_idx > 0 && comp(array[insert_idx - 1], elem); insert_idx--)
{

            array[insert_idx] = array[insert_idx - 1];

        }

        array[insert_idx] = elem;

    }

}
```

Файл merge_sort.h

```
#pragma once

#include <vector>

#include <cstddef>

#include "comparator.h"



template<typename Element>

std::vector<Element>&      merge_sort_internal(std::vector<Element>      &array,
std::vector<Element>  &buffer,  const  size_t  idx_start,  const  size_t  idx_end,
const Comparator<Element>comp)

{

    if(idx_start == idx_end) {

        return array;

    }



    size_t idx_middle = idx_start + (idx_end - idx_start) / 2;



       std::vector<Element>  &left_sorted  =  merge_sort_internal(array,  buffer,
idx_start, idx_middle, comp);

      std::vector<Element>  &right_sorted  =  merge_sort_internal(array,  buffer,
idx_middle + 1, idx_end, comp);



    std::vector<Element> &sorted = (left_sorted == array) ? buffer : array;



    size_t idx_left = idx_start;

    size_t idx_right = idx_middle + 1;

    for(size_t idx = idx_start; idx <= idx_end; idx++) {

        if(idx_left <= idx_middle && idx_right <= idx_end) {

            if(comp(left_sorted[idx_left], right_sorted[idx_right])) {

                sorted[idx] = right_sorted[idx_right];

                idx_right++;

            }

            else {

                sorted[idx] = left_sorted[idx_left];

                idx_left++;
```

```
            }
        }
        else if(idx_left <= idx_middle) {
            sorted[idx] = left_sorted[idx_left];
            idx_left++;
        }
        else {
            sorted[idx] = right_sorted[idx_right];
            idx_right++;
        }
    }

    return sorted;
}



template<typename Element>
void merge_sort(std::vector<Element> &array, const Comparator<Element>comp = is_greater)
{
    std::vector<Element>buffer(array.size());

        std::vector<Element> &sorted = merge_sort_internal(array, buffer, 0, array.size() - 1, comp);

    if(sorted == buffer) {
        array = buffer;
    }
}
```

Файл quick_sort.h

```
#pragma once
#include <vector>
#include <cstddef>
#include <cstdlib>
#include "comparator.h"



template<typename Element>
size_t quick_sort_partition(std::vector<Element> &array, const size_t idx_start, const size_t idx_end, const Comparator<Element>comp)
{
       Element ref_element = array[idx_start + (size_t)rand() % (idx_end - idx_start + 1)];

    // Hoare scheme
    size_t idx_low = idx_start;
    size_t idx_high = idx_end;
    while(true) {
        while(comp(ref_element, array[idx_low])) idx_low++;
        while(comp(array[idx_high], ref_element)) idx_high--;

        if(idx_low >= idx_high) break;

        std::swap(array[idx_low], array[idx_high]);
        idx_low++;
        idx_high--;
    }

    return idx_high;
}



template<typename Element>
```

```
void quick_sort_internal(std::vector<Element> &array, const size_t idx_start, const size_t idx_end, const Comparator<Element>comp)
{
    if(idx_start < idx_end) {
            size_t idx_middle = quick_sort_partition(array, idx_start, idx_end, comp);
        quick_sort_internal(array, idx_start, idx_middle, comp);
        quick_sort_internal(array, idx_middle + 1, idx_end, comp);
    }
}



template<typename Element>
void quick_sort(std::vector<Element> &array, const Comparator<Element>comp = is_greater)
{
    quick_sort_internal(array, 0, array.size() - 1, comp);
}
```

Файл main.cpp

```
#include <vector>
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "bubble_sort.h"
#include "heap_sort.h"
#include "insertion_sort.h"
#include "merge_sort.h"
#include "quick_sort.h"


const size_t TEST_LEN[] = {5, 10, 20, 50, 100, 200, 500, 1000, 5000, 10000,
20000};
const int TEST_NUM = sizeof(TEST_LEN) / sizeof(size_t);


int main()
{
    std::cout << "Data length, time (us)" << std::endl;

    for(int num = 0; num < TEST_NUM; num++) {
        size_t len = TEST_LEN[num];
        std::vector<int> arr(len);

        for(size_t idx = 0; idx < len; idx++) {
            arr[idx] = std::rand();
        }

        clock_t clock_start = clock();

        // bubble_sort(arr);
        // heap_sort(arr);
        // insertion_sort(arr);
        // merge_sort(arr);
```

```
        quick_sort(arr);

        clock_t clock_end = clock();

        std::cout << len << ", " << clock_end - clock_start << std::endl;
    }

    return 0;
}
```