**САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО**

**ИНСТИТУТ КОМПЬЮТЕРНЫХ НАУК И КИБЕРБЕЗОПАСНОСТИ**

**ВЫСШАЯ ШКОЛА ПРОГРАММНОЙ ИНЖЕНЕРИИ**

**Индивидуальное задание**

**«Внутренние сортировки»**

**Вариант 11**

**по предмету «Алгоритмы и структуры данных»**

**Выполнил: студент гр. 5130904/30002 Севостьянова А.В.**

**Руководитель: Череповский Д.К.**

**Санкт-Петербург**

**2024 г.**

Оглавление

[1. Общая постановка задачи 3](#_Toc163911368)

[2. Алгоритм решения 3](#_Toc163911369)

[3. Детальные требования и тест план 5](#_Toc163911370)

[Детальные требования: 5](#_Toc163911371)

[Таблица с детальными требованиями и тест планом 5](#_Toc163911372)

[4. Время работы алгоритма 5](#_Toc163911373)

[Вывод 6](#_Toc163911374)

[Приложение 1 7](#_Toc163911375)

[main.cpp 7](#_Toc163911376)

[Introsort.h 7](#_Toc163911377)

[Introsort.cpp 7](#_Toc163911378)

[Приложение 2 11](#_Toc163911379)

1. **Общая постановка задачи**
2. Разработать алгоритм интроспективной сортировки (Introsort). Быстрая сортировка, в которой выполняется переключение на пирамидальную сортировку, когда глубина рекурсии превысит некоторый заранее установленный уровень (например, логарифм от числа сортируемых элементов).
3. Реализовать алгоритм на языке С++

* Написать программу
* Выполнить отладку и тестирование программы

1. Выполнить сравнение результатов экспериментальной оценки временной сложности с теоретическими для массивов, состоящих из 1000, 10000, 100000 и 500000 элементов.

# 2. Алгоритм решения

Интроспективная сортировка включает в себя 3 сортировки:

* Сортировка вставками
* Быстрая сортировка
* Пирамидальная сортировка
  1. Вычисляется максимальная глубина рекурсии равная O(2\*log(n)), где n – количество элементов сортируемого массива. Вызывается рекурсивная функция Introsort.

1. Если количество элементов в сортируемом массиве не превышает 16, то выполняется сортировка вставками, которая работает по следующему алгоритму:
2. Insertionsort проходит по каждому элементу последовательности и сравнивает его со стоящим слева элементом.
3. Если выбранный элемент больше, то они меняются местами, пока элемент слева не станет меньше или равен выбранному.
4. Иначе алгоритм переходит к следующему элементу.
5. Иначе массив сортируется с помощью быстрой сортировки, пока не будет достигнута максимальная глубина рекурсии. QuickSort работает по следующему алгоритму:
6. Выбирается значение, которое называется опорным значением. Опорное значение равно среднему из значений начального, среднего и конечного элементов массива.
7. Массив разделяется на два подмассива:

* Первый содержит значения, которые меньше либо равны опорному значению. Эти значения располагаются слева от опорного значения.
* Второй содержит значения, которые больше опорного значения. Эти значения располагаются справа от опорного значения.

1. Полученные подмассивы передаются в Introsort.
2. Когда будет превышена максимальная глубина рекурсии, массив (подмассив) обрабатывается с помощью пирамидальной сортировки. HeapSort работает по следующему алгоритму:
3. Строится дерево, вершиной которого является максимальный элемент массива.
4. Максимальный элемент изымается из кучи, добавляется на корректное место в массив, элемент, который лежал в массиве по данному индексу становится вершиной дерева
5. Дерево сортируется так, чтобы вершиной опять являлся максимальный элемент. Дерево строится таким образом, что каждый родитель больше своего потомка.

# 3. Детальные требования и тест план

## *Детальные требования:*

Ввод данных не осуществляется. Все данные заданы корректно.

## *Таблица с детальными требованиями и тест планом*

|  |  |  |  |
| --- | --- | --- | --- |
| Требования | Детальные требования | Данные | Ожидаемый результат |
| Данные заданы корректно | - | - | Корректная работа программы |

# 4. Время работы алгоритма

O(n\*log(n)) - ограничение по времени работы сортировки introsort

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | ***Теоритическое время работы (секунды)*** | | | ***Время работы алгоритма***  ***(секунды)*** | | |
| ***Количество элементов массива*** | ***Лучший случай*** | ***Средний случай*** | ***Худший случай*** | ***Лучший случай*** | ***Средний случай*** | ***Худший случай*** |
| **1000** | <=50 | | | 0 | 0.001 | 0.001 |
| **10000** | <=666 | | | 0.037 | 0.041 | 0.048 |
| **100000** | <=8333 | | | 3.858 | 4.2 | 3.033 |
| **500000** | <=47491 | | | 74.389 | 123.97 | 87.828 |

Таблица 1 «Время работы алгоритма Introsort»

**Вывод**

В ходе работы над программой была написана функция Introsort, сортирующая исходный массив целочисленных элементов в порядке возрастания. Данный алгоритм помогает отсортировать массив, состоящий из большого количества элементов за небольшой промежуток времени.

**Приложение 1**

Код программы

***main.cpp***

#include"Introsort.h"

int main()

{

std::cout << "\_\_\_\_\_\_\_\_\_\_\_Introsort for an array of 1000 elements\_\_\_\_\_\_\_\_\_\_\_\n";

countTimeOfIntrosort(1000);

std::cout << "\n\_\_\_\_\_\_\_\_\_\_\_Introsort for an array of 10000 elements\_\_\_\_\_\_\_\_\_\_\_\n";

countTimeOfIntrosort(10000);

std::cout << "\n\_\_\_\_\_\_\_\_\_\_\_Introsort for an array of 100000 elements\_\_\_\_\_\_\_\_\_\_\_\n";

countTimeOfIntrosort(100000);

std::cout << "\n\_\_\_\_\_\_\_\_\_\_\_Introsort for an array of 500000 elements\_\_\_\_\_\_\_\_\_\_\_\n";

countTimeOfIntrosort(500000);

return EXIT\_SUCCESS;

}

***Introsort.h***

#pragma once

#include <iostream>

#include <algorithm>

#include<time.h>

void introsort(int\* arr, const size\_t length);

void introsortRecursive(int\* arr, int start, int end, int deep);

void insertionSort(int\* arr, const size\_t length);

void heapSort(int\* arr, int start, int end);

void heapify(int\* arr, int start, int end, int i);

int partition(int\* arr, int left, int right, int pivot);

void countTimeOfIntrosort(size\_t length);

bool isSorted(int\* arr,const size\_t length);

***Introsort.cpp***

#include "Introsort.h"

void countTimeOfIntrosort(size\_t length)

{

int\* arr = new int [length] {0};

clock\_t start, end;

//лучший случай

for (int i = 0; i < length; ++i)

arr[i] = i;

start = clock();

introsort(arr, length);

end = clock();

std::cout << "the best case: " << (double)(end - start) / CLOCKS\_PER\_SEC << " s\n";

isSorted(arr, length) ? std::cout << "Array is sorted\n" : std::cout << "Array isn't sorted\n";

//средний случай

for (int i = 0; i < length; ++i)

arr[i] = rand();

start = clock();

introsort(arr, length);

end = clock();

std::cout << "the average case: " << (double)(end - start) / CLOCKS\_PER\_SEC << " s\n";

isSorted(arr, length) ? std::cout << "Array is sorted\n" : std::cout << "Array isn't sorted\n";

//худший случай

for (int i = length; i > 0; --i)

arr[length - i] = i;

start = clock();

introsort(arr, length);

end = clock();

std::cout << "the worst case: " << (double)(end - start) / CLOCKS\_PER\_SEC << " s\n";

isSorted(arr, length) ? std::cout << "Array is sorted\n" : std::cout << "Array isn't sorted\n";

delete[] arr;

}

bool isSorted(int\* arr, const size\_t length)

{

for (int i = 1; i < length; ++i)

if (arr[i - 1] > arr[i])

return false;

return true;

}

void introsort(int\* arr, const size\_t length)

{

int deep = std::floor(2 \* log(length));

introsortRecursive(arr, 0, length - 1, deep);

}

void introsortRecursive(int\* arr, int start, int end, int deep)

{

if (end - start <= 16)

{

insertionSort(arr, end);

return;

}

if (deep == 0)

{

heapSort(arr, start, end);

return;

}

int pivot = arr[start] + arr[end] + arr[(end + start) / 2]

- std::min({ arr[start], arr[end], arr[(end + start) / 2] })

- std::max({ arr[start],arr[end], arr[(end + start) / 2] });

int position;

position = partition(arr, start, end, pivot);

introsortRecursive(arr, start, position, deep - 1);

introsortRecursive(arr, position, end, deep - 1);

}

void insertionSort(int\* arr, const size\_t length)

{

for (int i = 1; i <= length; ++i)

{

int value = arr[i];

int j = i;

while (j > 0 && arr[j - 1] > value)

{

arr[j] = arr[j - 1];

j--;

}

arr[j] = value;

}

}

void heapSort(int\* arr, int start, int end)

{

for (int i = end / 2 - 1; i >= start; --i) {

heapify(arr, start, end, i);

}

for (int i = end - 1; i > start; --i) {

std::swap(arr[start], arr[i]);

heapify(arr, start, i, 0);

}

}

void heapify(int\* arr, int start, int end, int i)

{

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

if (left < end && arr[left] > arr[largest]) largest = left;

if (right < end && arr[right] > arr[largest]) largest = right;

if (largest != i)

{

std::swap(arr[i], arr[largest]);

heapify(arr, start, end, largest);

}

}

int partition(int\* arr, int left, int right, int pivot)

{

while (left <= right)

{

while (arr[left] < pivot) left++;

while (arr[right] > pivot) right--;

if (left <= right)

{

std::swap(arr[left], arr[right]);

left++;

right--;

}

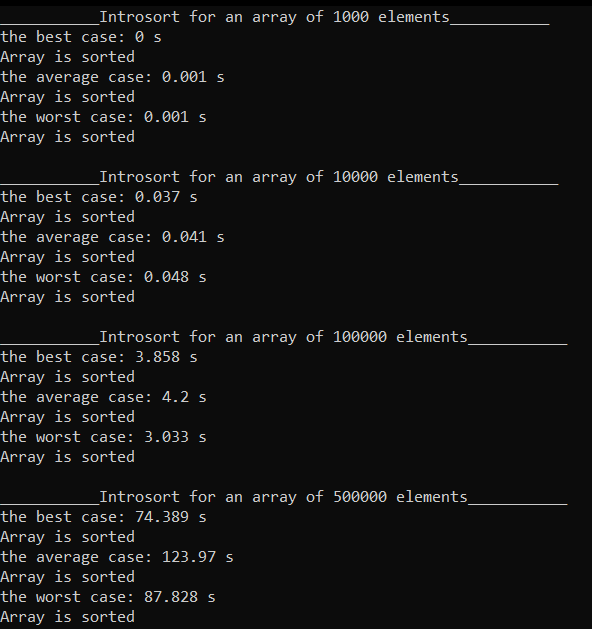
}

return left;

}

**Приложение 2**

Работа программы



*Рис. 1 Время работы алгоритма Introsort на различных массивах данных*