# ЦЕЛЬ РАБОТЫ

Научиться оценивать сложность и количество операций для алгоритмов сортировки.

# ПОСТАНОВКА ЗАДАЧИ

1. Для алгоритмов сортировки (табл. 1) найти оценку для количества шагов и количества требуемой памяти.
2. Создать структуру и реализовать алгоритмы сортировки. Критерий сортировки: по убыванию курса.
3. Реализовать более эффективные алгоритмы сортировки согласно таблице 1.
4. Сравнить производительность различных алгоритмов.

Таблица 1. Алгоритмы сортировки и структура данных.

|  |  |  |  |
| --- | --- | --- | --- |
| Простейший алгоритм №1 | Простейший алгоритм №2 | Быстрый алгоритм | Структура данных |
| Сортировка  посредством  выбора | Модифицированный  метод  «пузырька» | Быстрая  сортировка | Структура студент  содержит ФИО, курс,  факультет |

# ТЕСТ ПРОГРАММЫ

#include <iostream>

#include <vector>

#include <cassert>

#include <ctime>

using namespace std;

const unsigned int MAX\_COURSE = 5;

struct student\_t {

string FIO;

unsigned int course : 3;

string fac;

};

typedef vector<student\_t> vec;

typedef vec::iterator vecIter;

size\_t modBubbleSort(vec &arr) {

size\_t opCounter = 0;

size\_t n = arr.size();

bool flag = false; // + 1

opCounter += 1;

for (size\_t i = n; ((i > 0) && (!flag)); --i) { //1 + 3 \* N

flag = true;

size\_t topBorder = (i - 1); // + 1

opCounter += 1 + 2;

for (size\_t j = 0; j < topBorder; ++j) { // 1 + 3 \* (N/2)

if (arr[j].course < arr[j+1].course) {//+4

swap(arr[j+1], arr[j]);//+3 + 3

flag = false;//+1

opCounter += 7;

}

opCounter += 4;

}

opCounter += 1 + 3 \* (n / 2);

}

opCounter += 1 + 3 \* n;

return opCounter;

}

size\_t selectionSort(vec &arr) {

size\_t opCounter = 0;

size\_t n = arr.size();

for (size\_t i = 0; i < n; ++i) { // 1 + 3 \* N

size\_t maxIndex = i; // + 1

opCounter += 1;

for (size\_t j = (n - 1); j > i; --j) { // 2 + 3 \* (n / 2)

if (arr[j].course > arr[maxIndex].course) { // + 3

maxIndex = j; // + 1

opCounter += 1;

}

opCounter += 3;

}

if (maxIndex != i) {

swap(arr[i], arr[maxIndex]); // 2 + 3

opCounter += 5;

}

opCounter += 1;

opCounter += 2 + 3 \* (n / 2);

}

opCounter += 1 + 3 \* n;

return opCounter;

}

bool cmp(const student\_t & a, const student\_t& b) {

return (a.course > b.course);

}

size\_t quickSort(student\_t &pv, vecIter left, vecIter right) {

size\_t opCounter = 0;

student\_t v = \*(left + ((right - left) / 2)); //+ 4

opCounter += 4;

vecIter f = left;

vecIter l = right;

opCounter += 2;

do {

while ((f < right) && cmp(\*f, v)) {// 3\*n

++f;

opCounter += 4;

}

opCounter += 3;

while ((l > left) && cmp(v, \*l)) {

--l;

opCounter += 4;

}

opCounter += 3;

if (f <= l) {// + 2

std::swap(\*f, \*l);//+ 3

++f; // + 1

--l; // + 1

opCounter += 5;

}

opCounter += 2;

opCounter += 1;

} while (f <= l);//^ + 1

opCounter += 1;

if (l > left) // + 1

opCounter += quickSort(v, left, l);

opCounter += 1;

if (right > f) // +1

opCounter += quickSort(v, f, right);

opCounter += 1;

return opCounter;

}

bool checkOrder(vec &arr) {

for (size\_t i = 1; i < arr.size(); ++i) {

if (arr[i - 1].course < arr[i].course)

return false;

}

return true;

}

void randomFill(vec &arr) {

for (size\_t i = 0; i < arr.size(); ++i) {

arr[i].course = 1 + (rand() % MAX\_COURSE);

}

}

int main() {

size\_t time = 0;

size\_t timeInmSec = 0;

vec mainArr(1000);

randomFill(mainArr);

vec tempArr(mainArr);

//\*

assert(!checkOrder(tempArr));

cout << "Buble sort amount operation: " << modBubbleSort(tempArr) << endl;

time = clock();

timeInmSec = (static\_cast<float>(time)) / 10;

cout << "Sort duration: " << timeInmSec << " ms" << endl;

assert(checkOrder(tempArr));

cout << "Buble sort min amount operation: " << modBubbleSort(tempArr) << endl << endl;

//\*/

/\*

tempArr.clear();

tempArr = mainArr;

assert(!checkOrder(tempArr));

cout << "Line choise sort amount operation: " << selectionSort(tempArr) << endl;

time = clock();

timeInmSec = (static\_cast<float>(time)) / 10;

cout << "Sort duration: " << timeInmSec << " ms" << endl;

assert(checkOrder(tempArr));

cout << "Line choise sort min amount operation: " << selectionSort(tempArr) << endl << endl;

//\*/

/\*

tempArr.clear();

tempArr = mainArr;

assert(!checkOrder(tempArr));

cout << "Quick sort amount operation: " << quickSort(\*(tempArr.begin()),tempArr.begin(),(tempArr.end()-1)) << endl;

time = clock();

timeInmSec = (static\_cast<float>(time)) / 10;

cout << "Sort duration: " << timeInmSec << " ms" << endl;

assert(checkOrder(tempArr));

cout << "Quick sort min amount operation: " << quickSort(\*(tempArr.begin()), tempArr.begin(), (tempArr.end() - 1)) << endl << endl;

//\*/

return 0;

}

# РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММЫ

Тестирование алгоритмов сортировки проводилось на входном наборе из 1000 случайных элементов, а так же на наборе уже отсортированных элементов. На рисунках 1 – 3 отображено количество выполненных элементарных операций, а так же время выполнения сортировки на первом наборе входных данных.

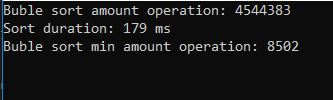


Рисунок 1 – Результаты тестирования алгоритма сортировки модифицированным методом «пузырька»

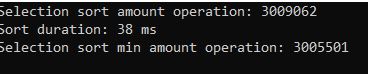


Рисунок 2 – Результат тестирования алгоритма сортировки методом прямого выбора

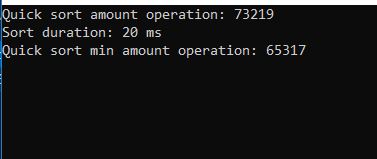


Рисунок 3 – Результат тестирования алгоритма «быстрой сортировки»

# АНАЛИЗ ПОЛУЧЕННЫХ РЕЗУЛЬТАТОВ

Таблица 2. Результаты производительности алгоритмов.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Сортировка модифицированным методом «пузырька» | Сортировка методом прямого выбора | «Быстрая сортировка» |
| Кол-во операций для случайной последовательности (1000 эл). | 4544383 | 3009062 | 73219 |
| Время выполнения сортировки для случайной последовательности (1000 эл.), *мс* | 179 | 38 | 20 |
| Кол-во операций для отсортированной последовательности ( 1000 эл). | 8502 | 3005501 | 65317 |
| Кол-во операций для случайной последовательности (2000 эл.) | 18093799 | 12018219 | 156511 |
| Время выполнения сортировки для случайной последовательности (2000 эл.), *мс* | 666 | 98 | 26 |
| Сложность алгоритма | *O(n2)* | *O(n2)* | *O(n log n)* |

Из данных таблицы 2 видно, что наиболее эффективным алгоритмом для сортировки массива со случайными элементами является метод «быстрой сортировки» предложенный Чарльзом Хоаром, но на отсортированном наборе данных он показывает приблизительно такую же производительность. Так же, в связи с рекурсивной реализацией алгоритм «быстрой сортировки» является наиболее затратным по памяти, так как для его выполнения может потребоваться *O(n)* вложенных рекурсивных вызовов.

Самым долгим по времени является алгоритм сортировки «пузырьком», так как он выполняет полный перебор значений, но при его модификации, данный алгоритм показывает хорошую производительность на уже отсортированном наборе данных.

# ВЫВОДЫ

В ходе лабораторной работы было исследованы три алгоритма сортировки: методом прямого выбора, модифицированным методом «пузырька», методом «быстрой сортировки». На основе разработанной программы был проведен анализ их производительности и быстродействия на наборах данных отличающихся количеством элементов и порядком следования этих элементов. В результате было установлено, что наиболее оптимальным, из данных алгоритмов, является алгоритм «быстрой сортировки» выполняющий поставленную задачу за минимальное время.