МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №4
по дисциплине «Алгоритмы и структуры данных»
Тема: Сортировка данных

Студент гр. 9303 _____ Максимов Е.А.

Преподаватель _____ Филатов Ар. Ю.

Санкт-Петербург
2020

Цель работы.

Познакомиться с алгоритмами сортировки и способами их создания. Реализовать алгоритм сортировки для элементов любого стандартного типа.

Задание.

Вариант №12 лабораторной работы.

Создать алгоритм быстрой сортировки с итеративной реализацией.

Основные теоретические положения.

Алгоритм сортировки — это алгоритм для упорядочивания элементов в списке. Алгоритм сортировки

Основные характеристики алгоритмов сортировки:

1) Вычислительная сложность — основной параметр, характеризующий быстродействие алгоритма.

2) Объём используемой памяти — ряд алгоритмов требует выделения дополнительной памяти под временное хранение промежуточных данных.

3) Устойчивость — устойчивая сортировка не меняет взаимного расположения элементов с одинаковыми ключами.

Описание алгоритма.

«Быстрая сортировка» (quick sort, «qsort») — один из самых быстрых известных алгоритмов сортировки массивов.

Алгоритм использует два вспомогательных вектора (*vector<int>*) - «стеки» верхних и нижних значений индексов исходного массива, индексы определяются во время работы алгоритма. Перед началом работы в стеках лежат индексы последнего и первого элемента соответственно (*size-1* и *0*).

1) Если стеки не пусты, программа достаёт последние значения из стеков и совершает действия, описанные далее.

2) Выбирается элемент из массива, называемый *опорным*. Это может быть любой из элементов массива. В реализации опорным индекс опорного элемента равен среднему арифметическому верхнего и нижнего значений из соответствующих стеков. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях зависит его эффективность (см. ниже). Его значение записывается в отдельную переменную *T pivotElem*.

3) Производится обход по элементам массива с помощью итераторов. Алгоритм проходит от меньшего значения индекса элементов, полученного из предыдущего шага, до индекса, у которого соответствующий элемент не меньше опорного элемента (если на всём пути не встретились такие пути, то им будет сам опорный элемент). Аналогично производится проход от большего индекса к индексу, которому соответствует элемент не больше опорного. Таким образом были найдены либо элементы массива, которые стоят не на своих местах (слева от опорного элемента должны быть значения меньше, справа - больше), либо элементы, равные опорному, либо обходные итераторы встретились в индексе опорного элемента.

Найденные элементы сравниваются и меняются местами, если нужно. Поскольку алгоритм не производит дополнительных проверок, то отсюда можно сделать вывод, что сортировка не является устойчивой. После обходной итератор меньшего индекса увеличивается на 1, большего индекса уменьшается на 1. Процедура повторяется до тех пор, пока обходные итераторы не пройдут по всем входных данным (итераторы должны пересечься).

4) Если меньший итератор меньше верхнего индекса (из пункта 1), то на стеки добавляются значения меньшего итератора и верхнего индекса. Если больший итератор меньше нижнего индекса, то на стеки добавляются

значения нижнего индекса и итератор. После алгоритм возвращается к пункту алгоритма 1.

Таким образом исходные данные упорядочиваются относительно значения опорного элемента: слева расположены те, что меньше, справа - те, что больше (стоит отметить, что позиция опорного элемента могла смениться после прохождения итераторами массива исходных данных). Если условие из пункта 4 не выполняется, то это означает, что обрабатываемых элементов было 3 или меньше.

Достоинства алгоритма:

• работает аналогично алгоритму рекурсивной быстрой сортировки; итеративная форма не создаёт рекурсивно новые функции, стек наполняется медленнее;

• работает быстро для случайно взятой последовательности.

Теоретические недостатки алгоритма:

• можно подобрать такую последовательность входных данных, что алгоритм будет работать наибольшее время; такая последовательность всегда существует, потому что действия алгоритма зависят только от входных данных. Недостаток можно решить при помощи случайного выбора опорного элемента.

Практические недостатки алгоритма:

• функции, реализованные в программном коде, не будут работать с типами данных, для которых не определены операции сравнения (операторы «>», «>=» и.т.д.).

Средняя вычислительная сложность алгоритма составляет $O(n * log(n))$, в худшем случае сложность составляет $O(n^2)$, в лучшем — $O(n * log(n))$.

Сортировка не является устойчивой, т.к. не сохраняет положение

одинаковых элементов относительно друг друга.

Выполнение работы.

В программе реализованы следующие функции:

1. Функция *void printVector(const vector<T>& array)* принимает на вход ссылку на вектор элементов *array* типа *T* (шаблон). Функция печатает последовательно элементы вектора. Если в векторе нет элементов, функция напечатает «No elements!» Функция не имеет возвращаемого значения.

2. Функция *printQSortIterStep(const vector<T>& array, const vector<int>& highIndexVector, const vector<int>& lowIndexVector>)* принимает на вход три ссылки на вектора элементов типов *T* (шаблон) и *int.* Функция печатает на экран сообщения о промежуточных состояниях сортируемого вектора *array.* Функция не имеет возвращаемого значения.

3. Функция *int popVector(vector<int>& array)* принимает на вход ссылку на вектор элементов *array* типа *int*. Функция удаляет последний элемент вектора и возвращает его.

4. Функция *void lineToIntVector(vector<T>& array, string& line)* принимает на вход ссылку на вектор элементов *array* типа *T* (шаблон) и ссылку на строку *line* типа *string.* Функция преобразует строку в вектор чисел типа *int* и заполняет вектор *array.* Функция не имеет возвращаемого значения.

5. Функция *bool isData(char c)* принимает на вход символ *c.* Функция возвращает *true,* если символ является символом английского алфавита или цифрой, и *false* в противном случае.

6. Функция *void qSortIter(vector<T>& array)* принимает на вход ссылку на вектор элементов *array* типа *T* (шаблон). Функция обрабатывает вектор и сортирует его от наименьшего элемента к наибольшему при помощи алгоритма быстрой сортировки. Функция работает с данными любого типа,

для которых существует результат операции сравнения (например, *char, short int,* и.т.д.). В данном исходном коде (см. приложение А) функция совершает операции над данными типа *int.* Функция не имеет возвращаемого значения.

7. Функция *int error(const int n)* принимает на вход переменную *n* типа *int* – номер ошибки, из-за которой не выполнилось одно из условий при проверке в функциях, описанных выше. Функция печатает на экран сообщение об ошибке. Функция всегда возвращает *0.* Функция реализована для сокращения исходного кода программы.

8. Функция *int main(int argc, char *argv[])* обрабатывает входные данные и вызывает функцию *qSortIter*. После запуска программы пользователю предлагается ввести список из целых чисел, и программа напечатает на экран шаги алгоритма и результат работы.

Исходный код программы представлен в приложении А.

Результаты тестирования программы представлены в приложении Б.

Выводы.

Был реализован алгоритм быстрой сортировки произвольных данных итеративным методом. Средняя вычислительная сложность алгоритма составляет *O(n * log(n))*.

Были выполнены следующие требования: возможность ввода данных из консоли, вывод сообщений о промежуточных этапах алгоритма сортировки, вывод информации о работе программы в консоль.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp.

```
#include <iostream>
#include <sstream>
#include <vector>

using namespace std;

template < typename T >
void printVector(const vector<T>& array)
{
    int size = array.size();
    if(size == 0)
        cout << "No elements!";
    else
        for(int i=0; i<size; i++) cout << array[i] << "\t";
    cout << endl;
    return;
}

template < typename T >
void printQSortIterStep(const vector<T>& array, const vector<int>&
 highIndexVector, const vector<int>& lowIndexVector)
{
    cout << endl;
    cout << "Array:\t\t"; printVector(array);
    cout << "LowIndex:\t"; printVector(highIndexVector);
    cout << "HighIndex:\t"; printVector(lowIndexVector);
    return;
}

int popVector(vector<int>& array)
{
    int value = array[array.size()-1];
    array.pop_back();
    return value;
}

int error(const int n){
    cout << "Error!\t";
    switch(n){
        case 0: cout << "Empty file." << endl; break;
        case 1: cout << "Incorrect input arguments. Write file pat
h or list for quick sort." << endl; break;
        case 2: cout << "File not found." << endl; break;
        case 3: cout << "Empty string." << endl; break;
```

```
    };
    return 0;
}

template < typename T >
void qSortIter(vector<T>& array)
{
    const int size = array.size();

    int lowIndexCurr, highIndexCurr, i, j;
    vector<int> lowIndexVector, highIndexVector;
    T pivotElem;

    lowIndexVector.push_back(0);
    highIndexVector.push_back(size-1);

    while (lowIndexVector.size() > 0)
    {
        printQSortIterStep(array, lowIndexVector, highIndexVector);

        lowIndexCurr = popVector(lowIndexVector);
        highIndexCurr = popVector(highIndexVector);
        i = lowIndexCurr;
        j = highIndexCurr;
        pivotElem = array[(highIndexCurr + lowIndexCurr) / 2];
        cout << "Pivot VALUE:\t" << pivotElem << endl;
        do
        {
            while (array[i] < pivotElem)
                i++;
            while (array[j] > pivotElem)
                j--;
            if (i <= j)
            {
                if (array[i] > array[j]){
                    swap(array[i], array[j]);
                    cout << "Swap:\t\ti=" << i << "\tj=" << j << e
ndl;
                    cout << "Array:\t\t";
                    printVector(array);
                }
                i++;
                if (j > 0)
                    j--;
            }
        } while (i <= j);
        if (i < highIndexCurr)
        {
            lowIndexVector.push_back(i);
            highIndexVector.push_back(highIndexCurr);
        }
        if (j > lowIndexCurr)
        {
            lowIndexVector.push_back(lowIndexCurr);
            highIndexVector.push_back(j);
```

```
        }
    }
    printQSortIterStep(array, lowIndexVector, highIndexVector);
    return;
}

void lineToIntVector(vector<int>& array, const string& line)
{
    istringstream lineStream(line);
    int num;
    while(lineStream >> num || !lineStream.eof())
    {
        if(lineStream.fail())
        {
            lineStream.clear();
            string temp;
            lineStream >> temp;
            continue;
        }
        array.push_back(num);
    }
    return;
}

int main(int argc, char *argv[]){
    string qSortString;
    vector<int> array;
    cout << "Write list for quick sort:" << endl;
    getline(cin, qSortString);
    lineToIntVector(array, qSortString);
    if (array.size() > 0)
        qSortIter(array);
    else
        return error(3);
    return 0;
}
```

ПРИЛОЖЕНИЕ Б
ТЕСТИРОВАНИЕ

Таблица 1 - Тестирование программы.

№	Входные данные	Выходные данные	Комментарий
1.		Error! Empty string.	Тест обработки пустой строки.
2.		Error! Empty string.	Тест обработки строки с пробелом.
3.	Qwe rty	Error! Empty string.	Тест обработки строки с некорректным и элементами.
4.	99999 8888 7 666 qwerty 55	Array: 99999 8888 7 666 55 LowIndex: 0 HighIndex: 4 Pivot VALUE: 7 Swap: i=0 j=2 Array: 7 8888 99999 666 55	Тест обработки строки с некорректным и элементами.

		Array: 7 8888 99999 666 55 LowIndex: 1 HighIndex: 4 Pivot VALUE: 99999 Swap: i=2 j=4 Array: 7 8888 55 666 99999 Array: 7 8888 55 666 99999 LowIndex: 1 HighIndex: 3 Pivot VALUE: 55 Swap: i=1 j=2 Array: 7 55 8888 666 99999 Array: 7 55 8888 666 99999 LowIndex: 2 HighIndex: 3 Pivot VALUE: 8888 Swap: i=2 j=3 Array: 7 55 666 8888 99999 Array: 7 55 666 8888 99999	

		LowIndex: No elements! HighIndex: No elements!	
5.	10 0 1 9 2 8 3 7 4 6 5	Array: 10 0 1 9 2 8 3 7 4 6 5 LowIndex: 0 HighIndex: 10 Pivot VALUE: 8 Swap: i=0 j=10 Array: 5 0 1 9 2 8 3 7 4 6 10 Swap: i=3 j=9 Array: 5 0 1 6 2 8 3 7 4 9 10 Swap: i=5 j=8 Array: 5 0 1 6 2 4 3 7 8 9 10 Array: 5 0 1 6 2 4 3 7 8 9 10 LowIndex: 8 0 HighIndex: 10 7 Pivot VALUE: 6 Swap: i=3 j=6	Тест обработки длинной строки.

		Array: 5 0 1 3 2 4 6 7 8 9 10 Array: 5 0 1 3 2 4 6 7 8 9 10 LowIndex: 8 6 0 HighIndex: 10 7 5 Pivot VALUE: 1 Swap: i=0 j=2 Array: 1 0 5 3 2 4 6 7 8 9 10 Array: 1 0 5 3 2 4 6 7 8 9 10 LowIndex: 8 6 2 0 HighIndex: 10 7 5 1 Pivot VALUE: 1 Swap: i=0 j=1 Array: 0 1 5 3 2 4 6 7 8 9 10 Array: 0 1 5 3 2 4 6 7 8 9 10	

		LowIndex: 8 6 2 HighIndex: 10 7 5 Pivot VALUE: 3 Swap: i=2 j=4 Array: 0 1 2 3 5 4 6 7 8 9 10 Array: 0 1 2 3 5 4 6 7 8 9 10 LowIndex: 8 6 4 HighIndex: 10 7 5 Pivot VALUE: 5 Swap: i=4 j=5 Array: 0 1 2 3 4 5 6 7 8 9 10 Array: 0 1 2 3 4 5 6 7 8 9 10 LowIndex: 8 6 HighIndex: 10 7 Pivot VALUE: 6 Array: 0 1 2 3 4 5	

		6 7 8 9 10 LowIndex: 8 HighIndex: 10 Pivot VALUE: 9 Array: 0 1 2 3 4 5 6 7 8 9 10 LowIndex: No elements! HighIndex: No elements!	
6.	1 3 3 3 1 2 2 1 2	Array: 1 3 3 3 1 2 2 1 2 LowIndex: 0 HighIndex: 8 Pivot VALUE: 1 Swap: i=1 j=4 Array: 1 1 3 3 3 2 2 1 2 Array: 1 1 3 3 3 2 2 1 2 LowIndex: 2 0 HighIndex: 8 1 Pivot VALUE: 1	Тест обработки строки с повторяющимися символами.

		Array: 1 1 3 3 3 2 2 1 2 LowIndex: 2 HighIndex: 8 Pivot VALUE: 2 Swap: i=2 j=8 Array: 1 1 2 3 3 2 2 1 3 Swap: i=3 j=7 Array: 1 1 2 1 3 2 2 3 3 Swap: i=4 j=6 Array: 1 1 2 1 2 2 3 3 3 Array: 1 1 2 1 2 2 3 3 3 LowIndex: 6 2 HighIndex: 8 4 Pivot VALUE: 1 Swap: i=2 j=3 Array: 1 1 1 2 2 2	

		3 3 3 Array: 1 1 1 2 2 2 3 3 3 LowIndex: 6 3 HighIndex: 8 4 Pivot VALUE: 2 Array: 1 1 1 2 2 2 3 3 3 LowIndex: 6 HighIndex: 8 Pivot VALUE: 3 Array: 1 1 1 2 2 2 3 3 3 LowIndex: No elements! HighIndex: No elements!	