**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

ОТЧЁТ

**по лабораторной работе №2**

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

**Тема: «Алгоритмы сортировки и поиска»**

**Вариант 2.**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9302 |  | Новокрещенов Д.К. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

## Постановка задачи и описание реализуемых функций.

По заданию мне необходимо реализовать несколько алгоритмов сортировки и поиска в массивах. Для этого мне необходимы следующие функции:

|  |  |  |
| --- | --- | --- |
| **Функция** | **Описание** | **Оценка временной сложности** |
| double average(double\*) | Вычисление среднего арифметического элементов массива | O(n) |
| size\_t BinarySearch(int\*, size\_t, int) | Бинарный поиск элемента в массиве | O(log2n) |
| double bogoSort(int\* array, size\_t size) | Алгоритм глупой сортировки | O(n\*n!) |
| double BubbleSort(int\*, size\_t) | Алгоритм сортировки пузырьком | O(n2) |
| double CountingSort(char\*, size\_t) | Алгоритм сортировки подсчётом | O(n) |
| int\* Get\_Rand\_Array\_Int(size\_t) | Создание массива случайных чисел для типа int | O(n) |
| char\* Get\_Rand\_Array\_Char(size\_t) | Создание массива случайных значений для типа int | O(n) |
| double InsertionSort(int\*, size\_t) | Алгоритм сортировки вставкой | O(n2) |
| bool issorted(int\*, size\_t) | Проверка массива на отсортированность для типа int | O(n) |
| bool issorted\_char(char\*, size\_t) | Проверка массива на отсортированность для типа char | O(n) |
| void Print\_Array(int\*, size\_t) | Вывод массива в консоль | O(n) |
| double QuickSort(int\*, int) | Алгоритм быстрой сортировки Хоара | O(n\* logn) |

Также необходимо сравнить временную сложность алгоритмов быстрой и пузырьковой сортировки для массивов размеров 10, 100, 1000, 10000, 100000. Все значения в таблицах представлены в миллисекундах (ms). Значение 0 свидетельствует о том, что выполнение заняло менее 1ms.

Таблица для алгоритма быстрой сортировки:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Среднее значение** |
| **10** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **100** | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0.1 |
| **1000** | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0.3 |
| **10000** | 3 | 3 | 2 | 2 | 2 | 3 | 2 | 3 | 3 | 3 | 2.6 |
| **100000** | 24 | 23 | 23 | 24 | 24 | 29 | 31 | 25 | 34 | 23 | 26 |

Таблица для алгоритма сортировки пузырьком:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Среднее значение** |
| **10** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **100** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **1000** | 1 | 2 | 1 | 2 | 2 | 1 | 1 | 2 | 1 | 2 | 1.5 |
| **10000** | 211 | 201 | 196 | 168 | 175 | 162 | 169 | 163 | 213 | 208 | 186.6 |
| **100000** | 18191 | 17991 | 18021 | 18113 | 18103 | 18396 | 18177 | 18181 | 18036 | 18071 | 18128 |

## Описание реализованных unit-тестов.

|  |  |
| --- | --- |
| **Наименование теста** | **Описание** |
| issorted\_true\_without\_equal\_elem | Проверка функции issorted на отсортированном массиве без повторяющихся значений |
| issorted\_false | Проверка функции issorted на неотсортированном массиве |
| issorted\_true\_with\_equal\_elem | Проверка функции issorted на отсортированном массиве с повторяющимися значениями |
| issorted\_char\_true | Проверка функции issorted на отсортированном массиве |
| issorted\_char\_false | Проверка функции issorted на неотсортированном массиве |
| BinarySearch\_Normal | Проверка функции BinarySearch при всех корректных значениях |
| BinarySearch\_ErrorKey | Проверка функции BinarySearch при отсутствии искомого элемента |
| BinarySearch\_NotSortedArray | Проверка фунции BinarySearch при неотсортированном массиве |
| BogoSort\_Test | Проверка функции bogoSort на массиве без повторяющихся значений |
| BogoSort\_Test\_with\_equal\_elem | Проверка функции bogoSort на массиве с повторяющимися значениями |
| BubbleSort\_Test | Проверка функции BubbleSort на массиве без повторяющихся значений |
| BubbleSort\_Test\_with\_equal\_elem | Проверка функции BubbleSort на массиве с повторяющимися значениями |
| CountingSort\_Test | Проверка функции CountingSort на массиве без повторяющихся значений |
| InsertionSort\_Test | Проверка функции InsertionSort на массиве без повторяющихся значений |
| InsertionSort\_Test\_with\_equal\_elem | Проверка функции InsertionSort на массиве с повторяющимися значениями |
| QuickSort\_Test | Проверка функции QuickSort на массиве без повторяющихся значений |
| QuickSort\_Test\_with\_equal\_elem | Проверка функции QuickSort на массиве с повторяющимися значениями |

## Листинги программы и тестов

|  |
| --- |
| Lab2.cpp |
| #include <iostream>  #include <iomanip>  #include <time.h>  using namespace std;  int\* Get\_Rand\_Array\_Int(size\_t size)  {  int\* array = new int[size];  int sign;  for (size\_t i = 0; i < size; i++)  {  sign = 1;  if (rand() % 2 == 1)  sign = -1;  array[i] = rand() \* sign;  }  return array;  }  int\* Get\_Rand\_Array\_Char(size\_t size)  {  int\* array = new int[size];  int sign;  for (size\_t i = 0; i < size; i++)  {  sign = 1;  if (rand() % 2 == 1)  sign = -1;  array[i] = rand() \* sign;  }  return array;  }  double CountingSort(char\* array, size\_t size)  {  const size\_t max\_value = 256;  clock\_t time\_test = clock();  int temp\_array[max\_value];  for (size\_t i = 0; i < max\_value; i++)  temp\_array[i] = 0;  for (size\_t i = 0; i < size; i++)  temp\_array[array[i] + 127]+=1;  size\_t control=0;  for (size\_t i = 0; i < max\_value; i++)  {  for (int j = 0; j < temp\_array[i]; j++)  {  array[control] = i - 127;  control++;  }  }  time\_test = clock() - time\_test;  return ((double)time\_test / CLOCKS\_PER\_SEC \* 1000);  }  double InsertionSort(int\* array, size\_t size)  {  clock\_t time\_test = clock();  int key, temp;  for (size\_t i = 1; i < size; i++)  {  key = array[i];  temp = i - 1;  while (temp >= 0 && array[temp] > key)  {  array[temp + 1] = array[temp];  array[temp] = key;  temp--;  }  }  time\_test = clock() - time\_test;  return ((double)time\_test / CLOCKS\_PER\_SEC \* 1000);  }  void Print\_Array(int \*array, size\_t size)  {  for (size\_t i = 0; i < size; i++)  cout << array[i] << ' ';  cout << endl;  }  double BubbleSort(int \*array, size\_t size)  {  clock\_t time\_test = clock();  int control;  for (size\_t i = 0; i < size; i++)  {  control = 0;  for(size\_t j=0; j<size-1; j++)  if (array[j] > array[j + 1])  {  swap(array[j], array[j + 1]);  control++;  }  if (control == 0)  break;  }  time\_test = clock() - time\_test;  return ((double)time\_test\*1000/CLOCKS\_PER\_SEC);  }  bool issorted(int\* array, size\_t size)  {  for(size\_t i=0; i<size-1; i++)  if(array[i]>array[i+1])  return false;  return true;  }  bool issorted\_char(char\* array, size\_t size)  {  for (size\_t i = 0; i < size - 1; i++)  if (array[i] > array[i + 1])  return false;  return true;  }  double bogoSort(int\* array, size\_t size)  {  clock\_t time\_test = clock();  while (!issorted(array, size))  for (size\_t i = 0; i < size; i++)  swap(array[i], array[rand() % size]);  time\_test = clock() - time\_test;  return ((double)time\_test \*1000 / CLOCKS\_PER\_SEC);  }  double QuickSort(int\* array, int size)  {  clock\_t time\_test = clock();  int i = 0, j = size - 1, mid = array[size / 2];  do {  while (array[i] < mid) i++;  while (array[j] > mid) j--;  if (i <= j) {  swap(array[i], array[j]);  i++;  j--;  }  } while (i <= j);  if (j > 0) QuickSort(array, j + 1);  if (i < size) QuickSort(&array[i], size - i);  time\_test = clock() - time\_test;  return ((double)time\_test\*1000 / CLOCKS\_PER\_SEC);  }  size\_t BinarySearch(int\* array, size\_t size, int key)  {  size\_t mid = size / 2, left = 0, right = size - 1;  if (!issorted(array, size))  throw "This array is not sorted";  while (array[mid] != key)  {  mid = (left + right) / 2;  if (key < array[mid])  right = mid-1;  else if (key > array[mid])  left = mid+1;  else return mid;  if (left > right)  throw "Element not found";  }  return mid;  }  double average(double\* array)  {  double result = 0;  for (int i = 0; i < 10; i++)  result += array[i];  return result / 10;  }  int main()  {  srand(time(0));  int size = 10;  int\* test;  cout << "Result for QuickSort: " << endl;  double temp\_array[10];  for (int i = 0; i < 5; i++)  {  for (int j = 0; j < 10; j++)  {  test = Get\_Rand\_Array\_Int(size);  temp\_array[j] = QuickSort(test, size);  cout << temp\_array[j] <<" ";  delete[] test;  }  cout << "for " << size << " elements. " << " Average: " << average(temp\_array) << endl;  size \*= 10;  }  for (int i = 0; i < 5; i++)  {  size = 10;  test = Get\_Rand\_Array\_Char(size);  Print\_Array(test, size);  InsertionSort(test, size);  Print\_Array(test, size);  delete[] test;  }  size = 10;  cout << "Result for BubbleSort: " << endl;  for (int i = 0; i < 5; i++)  {  for (int j = 0; j < 10; j++)  {  test = Get\_Rand\_Array\_Int(size);  temp\_array[j] = BubbleSort(test, size);  cout << temp\_array[j] << " ";  delete[] test;  }  cout << "for " << size << " elements. " << " Average: " << average(temp\_array) << endl;  size \*= 10;  }  } |
| UnitTest.cpp |
| #include "pch.h"  #include "CppUnitTest.h"  #include "..\Lab2\Lab2.cpp"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace UnitTest  {  TEST\_CLASS(UnitTest)  {  public:  TEST\_METHOD(issorted\_true\_without\_equal\_elem)  {  const size\_t size = 5;  int test[size] = { 0,1,2,3,4 };  Assert::AreEqual(issorted(test, size), true);  }  TEST\_METHOD(issorted\_false)  {  const size\_t size = 5;  int test[size] = { 0,1,20,3,4 };  Assert::AreEqual(issorted(test, size), false);  }  TEST\_METHOD(issorted\_true\_with\_equal\_elem)  {  const size\_t size = 5;  int test[size] = { 0,1,2,4,4 };  Assert::AreEqual(issorted(test, size), true);  }  TEST\_METHOD(issorted\_char\_true)  {  const size\_t size = 5;  char test[size] = {1,2,3,4,5};  Assert::AreEqual(issorted\_char(test, size), true);  }  TEST\_METHOD(issorted\_char\_false)  {  const size\_t size = 5;  char test[size] = "leti";  Assert::AreEqual(issorted\_char(test, size), false);  }  TEST\_METHOD(BogoSort\_Test)  {  const size\_t size = 5;  int test[size] = { 0,1,20,3,4 };  bogoSort(test, size);  Assert::AreEqual(issorted(test, size), true);  }  TEST\_METHOD(BubbleSort\_Test)  {  const size\_t size = 5;  int test[size] = { 0,1,20,3,4 };  BubbleSort(test, size);  Assert::AreEqual(issorted(test, size), true);  }  TEST\_METHOD(QuickSort\_Test)  {  const size\_t size = 5;  int test[size] = { 0,1,20,3,4 };  QuickSort(test, size);  Assert::AreEqual(issorted(test, size), true);  }  TEST\_METHOD(InsertionSort\_Test)  {  const size\_t size = 5;  int test[size] = { 0,1,20,3,4 };  InsertionSort(test, size);  Assert::AreEqual(issorted(test, size), true);  }  TEST\_METHOD(CountingSort\_Test)  {  const size\_t size = 10;  char test[size] = {5,4,3,2,1,1,2,3,4,5};  CountingSort(test, size);  Assert::AreEqual(issorted\_char(test,size), true);  }  TEST\_METHOD(BogoSort\_Test\_with\_equal\_elem)  {  const size\_t size = 5;  int test[size] = { 20,1,20,3,4 };  bogoSort(test, size);  Assert::AreEqual(issorted(test, size), true);  }  TEST\_METHOD(BubbleSort\_Test\_with\_equal\_elem)  {  const size\_t size = 5;  int test[size] = { 20,1,20,3,4 };  BubbleSort(test, size);  Assert::AreEqual(issorted(test, size), true);  }  TEST\_METHOD(QuickSort\_Test\_with\_equal\_elem)  {  const size\_t size = 5;  int test[size] = { 20,1,20,3,4 };  QuickSort(test, size);  Assert::AreEqual(issorted(test, size), true);  }  TEST\_METHOD(InsertionSort\_Test\_with\_equal\_elem)  {  const size\_t size = 5;  int test[size] = { 20,1,20,3,4 };  InsertionSort(test, size);  Assert::AreEqual(issorted(test, size), true);  }  TEST\_METHOD(BinarySearch\_NotSortedArray)  {  const size\_t size = 5;  int test[size] = { 0,1,20,3,4 };  try  {  BinarySearch(test, size, 1);  }  catch (const char\* error)  {  Assert::AreEqual(error, "This array is not sorted");  }  }  TEST\_METHOD(BinarySearch\_ErrorKey)  {  const size\_t size = 5;  int test[size] = { 0,1,2,3,4 };  try {  BinarySearch(test, size, 5);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Element not found");  }  }  TEST\_METHOD(BinarySearch\_Normal)  {  const size\_t size = 5, correct\_index = 2;  int test[size] = { 1,3,5,7,9 };  Assert::AreEqual(BinarySearch(test, size, 5), correct\_index);  }  };  } |

## Вывод

В ходе выполнения работы я научился реализовывать различные алгоритмы сортировки, проводить оценку временной сложности алгоритмов, а также создавать и использовать unit-тесты.