**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

отчет

**по лабораторной работе №2**

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

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

**Вариант 1**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 9302 |  | Ленина М.Р. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

Оглавление

[1. Постановка задачи 3](#_Toc56469638)

[2. Описание реализуемых алгоритмов 3](#_Toc56469639)

[3. Оценка временной сложности каждого алгоритма 4](#_Toc56469640)

[4. Сравнение временной сложности алгоритмов 2 и 3 4](#_Toc56469641)

[5. Описание реализованных unit-тестов 4](#_Toc56469642)

[6. Листинг 5](#_Toc56469643)

[7. Вывод 10](#_Toc56469644)

# Постановка задачи

Реализовать следующие алгоритмы:

1. Двоичный поиск (BinarySearch)
2. Быстрая сортировка (QuickSort)
3. Сортировка вставками (InsertionSort)
4. Глупая сортировка (BogoSort)
5. Сортировка подсчётом (CountingSort) для типа ***char***

# Описание реализуемых алгоритмов

1. Двоичный поиск (BinarySearch):

Задача: найти индекс числа N в массиве.

* 1. Определяем границы области поиска: левый (left) и правый (right) элементы
  2. Делим зону поиска пополам, выбирая центральный элемент mid
  3. Сравниваем искомый элемент N c mid:
* eсли N < mid, то сужаем зону поиска, изменяя left = mid – 1
* если N > mid, то сужаем зону поиска, изменяя left = mid + 1
* если N = mid, то это искомый элемент
  1. Возвращаемся к шагу 2

1. Быстрая сортировка (QuickSort):

Алгоритм использует принцип «разделяй и властвуй»: решаемая задача рекурсивно разбивается на 2 или более подзадачи того же типа, но меньшего размера. Разбиения выполняются до тех пор, пока все подзадачи не окажутся элементарными и легко решаемыми.

В алгоритме массив разбивается на подмассивы, и для этих подмассивов рекурсивно вызываются алгоритм quickSort

1. Выбрать элемент из массива. Назовём его опорным.
2. *Разбиение*: перераспределение элементов в массиве таким образом, что элементы меньше опорного помещаются перед ним, а больше или равные после.
3. Рекурсивно применить первые два шага к двум подмассивам слева и справа от опорного элемента. Рекурсия не применяется к массиву, в котором только один элемент или отсутствуют элементы.
4. Сортировка вставками (InsertionSort):
5. Предполагается, что первый эл-т массива отсортирован. Взять второй элемент и сохранить его как key
6. Сравнить key с первым эл-том. Если первый эл-т больше, чем key, то key помещается перед первым элементом
7. Поместить каждый неотсортированный эл-т в правильное место
8. Глупая сортировка (BogoSort):

Алгоритм выполняет случайное перемешивание эл-тов массива. Затем выполняется проверка массива на отсортированность. Если он не отсортирован, то массив перемешивается еще раз, пока не будет отсортирован.

1. Сортировка подсчётом (CountingSort) для типа ***char***:
2. Найти максимальный элемент max массива
3. Создать массив длины (max+1) со нулевыми эл-тами. Этот массив будет использоваться для хранения кол-ва разных эл-тов, встречающихся в изначальном массиве.
4. Создать массив длины (max+1) со нулевыми эл-тами. Этот массив будет использоваться для хранения кол-ва разных эл-тов, встречающихся в изначальном массиве
5. Пройти по массиву count и заполнить изначальный массив эл-тами по правилу: индекс массива count записывается в массив столько раз, какой значение хранится в count

# Оценка временной сложности каждого алгоритма

|  |  |
| --- | --- |
| BinarySearch | O(log(n)) |
| QuickSort | O(n2) |
| InsertionSort | O(n2) |
| BogoSort | O(n\*n!) |
| CountingSort | O(max + n) |

# Сравнение временной сложности алгоритмов 2 и 3

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Quick sort** | | | | | | | | | | | **Insertion sort** | | | | | | | | | | |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | **Ср** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | **Ср** |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 100 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **0** |
| 1000 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | **0.1** | 2 | 1 | 1 | 3 | 2 | 1 | 1 | 0 | 1 | 1 | **1.3** |
| 10 000 | 2 | 2 | 2 | 1 | 2 | 2 | 1 | 2 | 4 | 3 | **2.1** | 116 | 125 | 128 | 107 | 108 | 99 | 118 | 124 | 103 | 123 | **115.1** |
| 100 000 | 26 | 32 | 26 | 24 | 27 | 25 | 24 | 28 | 25 | 29 | **26.6** | 8823 | 9153 | 8709 | 8703 | 8700 | 9706 | 8779 | 8672 | 8627 | 8644 | **8851.6** |

# Описание реализованных unit-тестов

|  |  |
| --- | --- |
| is\_sorted\_not\_sorted\_array | Проверка метода is\_sorted, массив не отсортирован |
| is\_sorted\_sorted\_array | Проверка метода is\_sorted, отсортированный массив |
| is\_sorted\_equal\_elements | Проверка метода is\_sorted, массив не отсортирован и содержит одинаковые элементы |
| is\_sorted\_not\_sorted\_array\_char | Проверка метода is\_sorted для char, массив не отсортирован |
| is\_sorted\_sorted\_array\_char | Проверка метода is\_sorted для char, отсортированный массив |
| is\_sorted\_equal\_elements\_char | Проверка метода is\_sorted, массив не отсортирован и содержит одинаковые элементы |
| BinarySearch\_Array\_is\_not\_sorted | Проверка метода BinarySearch, массив не отсортирован |
| BinarySearch\_key\_exist | Проверка метода BinarySearch, искомый элемент содержится в массиве |
| BinarySearch\_key\_dosent\_exist | Проверка метода BinarySearch, искомый элемент не содержится в массиве |
| QuickSort\_test | Проверка метода QuickSort |
| QuickSort\_equal\_elements | Проверка метода QuickSort, массив содержит одинаковые элементы |
| InsertionSort\_test | Проверка метода InsertionSort |
| InsertionSortt\_equal\_elements | Проверка метода InsertionSort, массив содержит одинаковые элементы |
| BogoSort\_test | Проверка метода BogoSort |
| BogoSort\_equal\_elements | Проверка метода BogoSort, массив содержит одинаковые элементы |
| CountingSort\_test | Проверка метода CountingSort |
| CountingSort\_equal\_elements | Проверка метода CountingSort, массив содержит одинаковые элементы |

# Листинг

|  |
| --- |
| astr\_l2.h |
| #pragma once  int\* creating\_Array\_int(int size);  char\* creating\_Array\_char(int size);  bool is\_sorted(int\* Array, int size);  bool is\_sorted(char\* Array, int size);  int BinarySearch(int\* Array, int size, int N);  void QuickSort(int\* Array, int size);  void InsertionSort(int\* Array, int size);  void BogoSort(int\* Array, int size);  void CountingSort(char\* Array, int size);  float Average(float\* Array); |
| Source.cpp |
| #include <iostream>  #include <ctime>  #include "astr\_l2.h"  using namespace std;  int\* creating\_Array\_int(int size)  {  srand(time(NULL));  int \*Array = new int [size];  for (int index = 0; index < size; index++) {  Array[index] = rand() - rand();  }  return Array;  }  char\* creating\_Array\_char(int size)  {  srand(time(NULL));  char\* Array = new char[size];  for (int index = 0; index < size; index++) {  Array[index] = rand()%128;  }  return Array;  }  bool is\_sorted(int \*Array, int size) {  for (int index = 0; index < (size - 1); index++) {  if (Array[index] > Array[index + 1])  return false;  }  return true;  }  bool is\_sorted(char\* Array, int size) {  for (int index = 0; index < (size - 1); index++) {  if (Array[index] > Array[index + 1])  return false;  }  return true;  }  int BinarySearch(int\* Array, int size, int N)  {  int left = 0, right = size-1, mid = size/2;  if (!is\_sorted(Array, size))  throw out\_of\_range("Array is not sorted");  while (Array[mid] != N) {  mid = left + right;  if (N < Array[mid])  right = mid - 1;  else if (N > Array[mid])  left = mid + 1;  else return mid;  if (left > right) {  throw out\_of\_range("The key doesn't exist");  }  }  return mid;  }  void QuickSort(int\* Array, int size)  {  int i = 0, j = size - 1, pivot = Array[size/2];  int temp = 0;  while (i <= j) {  while (Array[i] < pivot)  i++;  while (Array[j] > pivot)  j--;  if (i <= j) {  temp = Array[i];  Array[i] = Array[j];  Array[j] = temp;  i++;  j--;  }  }  if (j > 0)  QuickSort(Array, j + 1);  if (i < size)  QuickSort(&Array[i], size - i);  }  void InsertionSort(int\* Array, int size) {  int j, key;  for (int i = 1; i < size; i++) {  key = Array[i];  j = i - 1;  while (j >= 0 && Array[j] > key)  {  Array[j + 1] = Array[j];  j = j - 1;  }  Array[j + 1] = key;  }  }  void BogoSort(int\* Array, int size) {  srand(time(NULL));  int temp;  while (!is\_sorted(Array, size))  for (int index = 0; index < size; index++) {  temp = Array[index];  Array[index] = Array[rand() % size];  Array[rand() % size] = temp;  }  }  void CountingSort(char\* Array, int size) {  int Max = 256, count = 0;  int\* result = new int[Max];  for (int index = 0; index < Max; index++)  result[index] = 0;  for (int index = 0; index < size; index++)  result[Array[index] + 127] += 1;  for (int i = 0; i < Max; i++)  for (int j = 0; j < result[i]; j++) {  Array[count] = i - 127;  count++;  }  }  float Average(float\* Array) {  float average = 0;  for (int index = 0; index < 10; index++) {  average += Array[index];  }  average /= 10;  return average;  }  int main()  {  float selection[10];  int\* Array;  int size = 10;  float time = 0;  cout << "For Quick sort:"<<endl;  for (int i = 0; i < 5; i++) {  for (int j = 0; j < 10; j++) {  Array = creating\_Array\_int(size);  time = clock();  QuickSort(Array,size);  time = clock() - time;  selection[j] = ((double)time\*1000/CLOCKS\_PER\_SEC);  cout << selection[j]<<' ';  delete[] Array;  }  cout << "Average: "<<Average(selection)<<endl;  size \*= 10;  }  size = 10;  cout << endl<< "For Insertion sort:"<<endl;  for (int i = 0; i < 5; i++) {  for (int j = 0; j < 10; j++) {  Array = creating\_Array\_int(size);  time = clock();  InsertionSort(Array, size);  time = clock() - time;  selection[j] = ((double)time \* 1000 / CLOCKS\_PER\_SEC);  cout << selection[j] << ' ';  delete[] Array;  }  cout << "Average: " << Average(selection) << endl;  size \*= 10;  }  return 0;  } |
| UnitTest2.cpp |
| #include "CppUnitTest.h"  #include "../astra\_l2/astr\_l2.h"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace UnitTest2  {  TEST\_CLASS(UnitTest2)  {  public:  int\* Array;  // before each test  TEST\_METHOD\_INITIALIZE(setUp)  {  Array = creating\_Array\_int(10);  }  // after each test  TEST\_METHOD\_CLEANUP(cleanUp)  {  delete[] Array;  }  TEST\_METHOD(is\_sorted\_not\_sorted\_array)  {  for (int i = 0; i < 10; i++) {  Array[i] = 1;  }  Array[5] = 60;  Assert::IsFalse(is\_sorted(Array, 10));  }  TEST\_METHOD(is\_sorted\_sorted\_array)  {  for (int i = 0; i < 10; i++) {  Array[i] = i;  }  Assert::IsTrue(is\_sorted(Array,10));  }  TEST\_METHOD(is\_sorted\_equal\_elements)  {  for (int i = 0; i < 10; i++) {  Array[i] = 1;  }  Assert::IsTrue(is\_sorted(Array, 10));  }  TEST\_METHOD(is\_sorted\_not\_sorted\_array\_char)  {  char\* Array\_char = new char [10];  for (int i = 0; i < 10; i++) {  Array\_char[i] = i;  }  Array\_char[5] = 60;  Assert::IsFalse(is\_sorted(Array\_char, 10));  }  TEST\_METHOD(is\_sorted\_sorted\_array\_char)  {  char\* Array\_char = new char[10];  for (int i = 0; i < 10; i++) {  Array\_char[i] = i;  }  Assert::IsTrue(is\_sorted(Array\_char, 10));  }  TEST\_METHOD(is\_sorted\_equal\_elements\_char)  {  char\* Array\_char = new char[10];  for (int i = 0; i < 10; i++) {  Array\_char[i] = 1;  }  Assert::IsTrue(is\_sorted(Array\_char, 10));  }  TEST\_METHOD(BinarySearch\_Array\_is\_not\_sorted)  {  for (int i = 0; i < 10; i++) {  Array[i] = i;  }  Array[5] = 60;  try {  BinarySearch(Array, 10, 5);  }  catch (std::out\_of\_range e) {  Assert::AreEqual("Array is not sorted", e.what());  }  }  TEST\_METHOD(BinarySearch\_key\_exist)  {  for (int i = 0; i < 10; i++) {  Array[i] = i;  }  Assert::AreEqual(BinarySearch(Array,10,5),5);  }  TEST\_METHOD(BinarySearch\_key\_dosent\_exist)  {  for (int i = 0; i < 10; i++) {  Array[i] = i;  }  try {  BinarySearch(Array, 10, 20);  }  catch (std::out\_of\_range e) {  Assert::AreEqual("The key doesn't exist", e.what());  }  }  TEST\_METHOD(QuickSort\_test)  {  for (int i = 9; i >= 0; i--) {  Array[i] = i;  }  QuickSort(Array, 10);  Assert::IsTrue(is\_sorted(Array, 10));  }  TEST\_METHOD(QuickSort\_equal\_elements)  {  for (int i = 9; i >= 0; i--) {  Array[i] = i;  }  Array[8] = 7;  QuickSort(Array, 10);  Assert::IsTrue(is\_sorted(Array, 10));  }  TEST\_METHOD(InsertionSort\_test)  {  for (int i = 9; i >= 0; i--) {  Array[i] = i;  }  InsertionSort(Array, 10);  Assert::IsTrue(is\_sorted(Array, 10));  }  TEST\_METHOD(InsertionSortt\_equal\_elements)  {  for (int i = 9; i >= 0; i--) {  Array[i] = i;  }  Array[8] = 7;  InsertionSort(Array, 10);  Assert::IsTrue(is\_sorted(Array, 10));  }  TEST\_METHOD(BogoSort\_test)  {  for (int i = 9; i >= 0; i--) {  Array[i] = i;  }  BogoSort(Array, 10);  Assert::IsTrue(is\_sorted(Array, 10));  }  TEST\_METHOD(BogoSort\_equal\_elements)  {  for (int i = 9; i >= 0; i--) {  Array[i] = i;  }  Array[8] = 7;  BogoSort(Array, 10);  Assert::IsTrue(is\_sorted(Array, 10));  }  TEST\_METHOD(CountingSort\_test)  {  char\* Array\_char = new char[10];  for (int i = 9; i >= 0; i--) {  Array\_char[i] = i;  }  CountingSort(Array\_char, 10);  Assert::IsTrue(is\_sorted(Array\_char, 10));  }  TEST\_METHOD(CountingSort\_equal\_elements)  {  char\* Array\_char = new char[10];  for (int i = 9; i >= 0; i--) {  Array\_char[i] = i;  }  Array[8] = 3;  CountingSort(Array\_char, 10);  Assert::IsTrue(is\_sorted(Array\_char, 10));  }  };  } |

# Вывод

Я изучила и освоила алгоритмы сортировки и поиска.