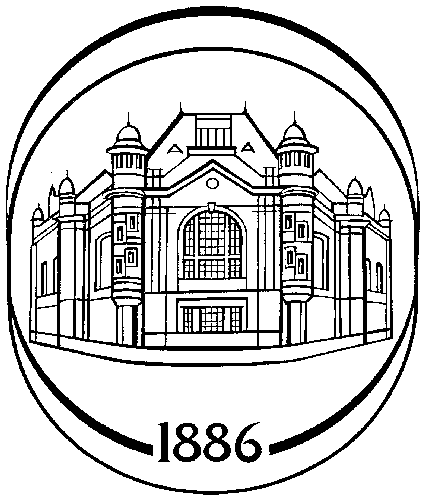
МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное учреждение высшего образования

**Санкт-Петербургский государственный электротехнический университет “ЛЭТИ” им. В. И. Ульянова (Ленина) (СПбГЭТУ «ЛЭТИ»)**



Кафедра САПР

Лабораторная работа № 2

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

Тема: «Алгоритмы сортировки и поиска»

Вариант № 23 (1)

|  |  |  |
| --- | --- | --- |
| Студентка гр. 9301, ФКТИ |  | Синицкая В. А. |
| Преподаватель |  | Тутуева А. В. |

Санкт-Петербург

2020 г.

Постановка задачи

Задача для 1 варианта:

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

Реализовать указанные алгоритмы для типа int, последний – для типа char, оценить временную сложность для каждого, сравнить по времени быструю сортировку и сортировку вставками, выведя данные для 10, 100, 1 000, 10 000, 100 000 элементов.

Описание реализуемых алгоритмов

1. Двоичный поиск (int BinarySearch(int\* array, int size, int element))

По правилам алгоритма ищет элемент, который передается параметром, возвращает его индекс в данном массиве, индекс считается с нуля. Если элементов одинаковых несколько, то возвращает индекс любого из них, если искомого элемента нет, то алгоритм возвращает индекс, на котором стоял бы данный элемент, если бы он был, а размер массива был на 1 больше.

1. Быстрая сортировка (void QuickSort(int\* array, int size))

По правилам алгоритма, выбирает опорный элемент (здесь последний в данном массиве), находит больший элемент, затем идет дальше по массиву, находит меньший элемент, меняет их местами. Хранит индекс для того места, куда в итоге встанет опорный элемнет, изначально считает его равным индексу найденного большего элемента, в итоге еще прибавляя 1, прибавляет 1 с каждой новой парой меняющихся местами элементов. Таким образом, дойдя до конца массива и поставив опорный элемент на его место, массив оказывается поделен на два: слева элементы меньше опорного, справа – больше либо равны. Получается, опорный элемент оказывается на своем месте. Затем быстрая сортировка вызывается для неотсортированных левого и правого подмассивов, не включающих в себя опорный элемент. При размере <= 1 массив считается отсортированным.

1. Сортировка вставками (void InsertionSort(int\* array, int size))

По правилам алгоритма первый элемент массива отсортирован. Следующий ставится слева или справа от первого, чтобы теперь подмассив из двух был отсортирован. Можно сказать, что левый подмассив отсортирован, а правый нет, и на каждом шаге размер левого увеличивается посредством добавления в него элемента из правого. Элемент, который собираются переносить, называют ключом. Его место в левом массиве находится двоичным поиском. Все элементы, которые в левом подмассиве стоят правее полученного индекса, сдвигаются вправо на 1 по индексам всего массива, а ключевой становится на свое место. Процесс продолжается, пока в правом подмассиве не закончатся элементы.

1. Глупая сортировка (void BogoSort(int\* array, int size))

Массив проверяется на отсортированность. Если он не отсортирован, то его элементы перемешиваются путем перемены местами двух случайных элементов, до тех пор, пока массив не окажется отсортированным. Очень долгий на практике.

1. Сортировка подсчётом (CountingSort) для типа ***char***

В данном массиве находится максимальный элемент, создается дополнительный массив, размером на 1 больше максимального элемента, чтобы под каждым индексом подразумевать возможный элемент массива, а в самой ячейке массива хранить количество появлений элемента в данном массиве. Используется тот факт, что char является и буквенным, и численным типом. Используется unsigned char, так как зависит от компьютера, может ли операция int(char) вернуть отрицательное значение. Поскольку таким образом определяется индекс, данный момент важен и требует пояснения. В общем, это нужно для заведомо однозначного перевода в ту и в другую сторону.

1. class Timer

нужен для подсчета времени, в момент создания переменной типа Timer счетчик времени обнуляется, в момент вызова функции elapsed() возвращает количество пройденного с момента создания объекта времени в секундах. Можно обнулить таймер, тогда время будет отсчитываться с момента обнуления.

1. int\* RandomArray(int size) (unsigned char\* RandomCharArray(int))

создает массив из случайных чисел размера size со значениями от -size до size. Для массива чаров – выбор из всех чаров с кодом от 0 до 255.

1. template <typename array\_elements\_type >

bool IsSorted(const array\_elements\_type\* array, int size)

Проверяет на отсортированность переданный массив, сделан шаблоном, чтобы работал и для типа int, и для типа char.

Оценка временной сложности

1. Двоичный поиск (BinarySearch)

O(log(n)) в среднем, где n – количество элементов (столько раз массив можно разделять на 2, выбирая одну из частей).

1. Быстрая сортировка (QuickSort)

При одном проходе, будет сделано n сравнений, где n – количество элементов массива (подмассива). Помимо одного прохода есть и другие – которые делаются от частей. В лучшем случае массив будет делиться примерно поровну и тогда количество проходов можно описать как log(n). В худшем случае массив будет делиться на подмассивы по 1 и n-2 элемента, или будет просто одним, состоящим из n-1 элемента. Тогда количество проходов (то есть рекурсивных вызовов, можно сказать), будет = n-1 ~ n. Средняя сложность, как и лучшая, составит O(n\*log(n)). В худшем случае сложность будет = O(n^2).

1. Сортировка вставками (InsertionSort)

Для каждого из n-1 элементов (исключая первый) (внешний цикл), выполняется поиск его места в отсортированном левом массиве, который выполняется за O(log(k)), где k – количество элементов в отсортированном массиве. Также внутри цикла есть цикл для сдвига элементов, в худшем случае сдвиг выполнится за O(k). Итого, внутри внешнего цикла сложность = O(n) в худшем случае, O(log(n-1)) в лучшем случае. В общем случае сложность = O(n\*(n-1)) ~ O(n^2).

1. Глупая сортировка (BogoSort)

Двойной цикл по всем элементам для перемешивания массива и его проверки на отсортированность влечет сложность O(n^2), но перемешиваний может понадобиться до n!, так как столько существует перестановок, плюс они еще и могут повторяться, так что примерная сложность = O(n\*n!), хотя на деле без проверки на отсутствие повторений можно получать уже полученные ранее неподходящие массивы.

1. Сортировка подсчётом (CountingSort) для типа char

Поиск максимального элемента делается за n-1 ~ n, заполнение вспомогательного массива делается за O(n) (O(max), если считать заполнение нулями). Заполнение массива по вспомогательному: для тех элементов, что во вспомогательном не = 0, по количеству встреч элемента, в сумме встреч всего будет n, это известно, а в вспомогательном массиве max элементов, потому тут суммарная сложность = O(max+n). Итоговая сложность алгоритма = O(max+n).

Сравнение сложностей QuickSort и InsertionSort

Таблица – сравнение временной сложности по таймеру

|  |  |  |
| --- | --- | --- |
| Size | Time of QuickSort | Time of InsertionSort |
| 10 | 4.3e-06 | 1.1e-06 |
| 10 | 3.4e-06 | 7e-07 |
| 10 | 3.2e-06 | 1.1e-06 |
| 10 | 3.3e-06 | 6e-07 |
| 10 | 6e-07 | 6e-07 |
| 100 | 9.54e-05 | 5.9e-06 |
| 100 | 4.43e-05 | 3.7e-06 |
| 100 | 4.29e-05 | 2.6e-06 |
| 100 | 4.25e-05 | 2.6e-06 |
| 100 | 4.33e-05 | 2.7e-06 |
| 1000 | 0.0009184 | 3.37e-05 |
| 1000 | 0.0009601 | 2.8e-05 |
| 1000 | 0.0009716 | 4.4e-05 |
| 1000 | 0.0007096 | 2.93e-05 |
| 1000 | 0.0007299 | 2.83e-05 |
| 10000 | 0.0120042 | 0.0003067 |
| 10000 | 0.0142592 | 0.0003288 |
| 10000 | 0.0143969 | 0.0002778 |
| 10000 | 0.0128927 | 0.0003578 |
| 10000 | 0.0111724 | 0.0002638 |
| 100000 | 0.121677 | 0.0033366 |
| 100000 | 0.124095 | 0.0029718 |
| 100000 | 0.138069 | 0.0037678 |
| 100000 | 0.122299 | 0.0035253 |
| 100000 | 0.144163 | 0.0048804 |

Описание реализованных unit-тестов

Всего был реализован 19 unit-test-ов:

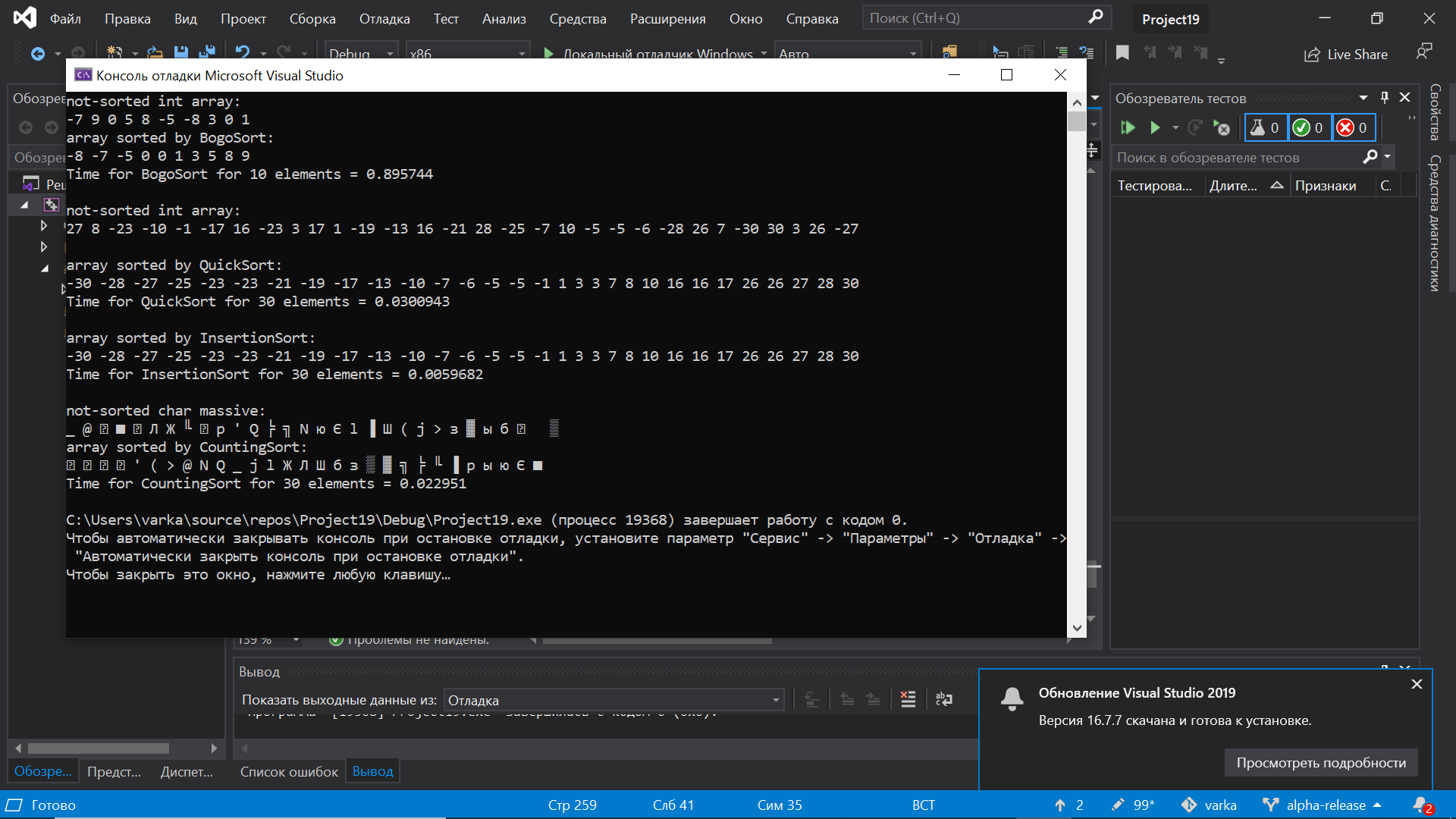
Проверялась работа всех сортировок на 1 элементе, и на разных значениях размера случайно сгенерированного массива. Бинарный поиск проверялся на 1, последний, средний, единственный и не существующий элемент, чтобы он выполнялся так, как от него ожидается. При тестировании сортировок проверка осуществляется путем выяснения, отсортирован ли случайный массив.

Пример работы

При функции main():

1. int main()
2. {
3. int size = 10;
4. int\* array = RandomArray(size);
5. cout << "not-sorted int array:" << endl;
6. for (int i = 0; i < size; i++)
7. cout << array[i] << " ";
8. Timer bogosort;
9. BogoSort(array, size);
10. cout << endl << "array sorted by BogoSort:" << endl;
11. for (int i = 0; i < size; i++)
12. cout << array[i] << " ";
13. cout << endl << "Time for BogoSort for " << size << " elements = " << bogosort.elapsed() << endl;
14. for (int i = 30; i <= 30; i += 10)
15. {
16. int size = i;
17. int\* array = RandomArray(size);
18. unsigned char\* char\_array = RandomCharMas(size);
19. cout << endl << "not-sorted int array:" << endl;
20. for (int i = 0; i < size; i++)
21. cout << array[i] << " ";
22. cout << endl;
23. int\* array\_copy1 = array, \*array\_copy2 = array;
24. Timer sort\_time;
25. QuickSort(array, size);
26. cout << endl << "array sorted by QuickSort:" << endl;
27. for (int i = 0; i < size; i++)
28. cout << array[i] << " ";
29. cout << endl << "Time for QuickSort for " << size << " elements = " << sort\_time.elapsed() << endl;
30. sort\_time.reset();
31. InsertionSort(array\_copy1, size);
32. cout << endl << "array sorted by InsertionSort:" << endl;
33. for (int i = 0; i < size; i++)
34. cout << array[i] << " ";
35. cout << endl << "Time for InsertionSort for " << size << " elements = " << sort\_time.elapsed() << endl;
36. sort\_time.reset();
37. cout << endl << "not-sorted char massive:" << endl;
38. for (int i = 0; i < size; i++)
39. cout << char\_array[i] << " ";
40. CountingSort(char\_array, size);
41. cout << endl << "array sorted by CountingSort:" << endl;
42. for (int i = 0; i < size; i++)
43. cout << char\_array[i] << " ";
44. cout << endl << "Time for CountingSort for " << size << " elements = " << sort\_time.elapsed() << endl;
45. }
46. return 0;
47. }

Результат работы программы:



Листинг

Sorts.h:

1. #pragma once
2. #include <iostream>
3. #include <stdexcept>
4. #include <cstdlib> // is it needed?
5. #include <ctime>
6. #include <chrono>
7. using namespace std;
8. class Timer
9. {
10. private:
11. using clock\_time = std::chrono::high\_resolution\_clock;
12. using second\_time = std::chrono::duration<double, std::ratio<1>>;
13. std::chrono::time\_point<clock\_time> begining;
14. public:
15. Timer();
16. void reset();
17. double elapsed() const;
18. ~Timer();
19. };
20. int\* RandomArray(int size);
21. template <typename array\_elements\_type >
22. bool IsSorted(const array\_elements\_type\* array, int size)
23. {
24. if (size <= 1)
25. return true;
26. for (int i = 0; i < size - 1; i++)
27. {
28. if (array[i] > array[i + 1])
29. return false;
30. }
31. return true;
32. }
33. int BinarySearch(int\*, int, int);
34. void InsertionSort(int\*, int);
35. void BogoSort(int\*, int);
36. unsigned char\* RandomCharArray(int);
37. void CountingSort(unsigned char\*, int);
38. void QuickSort(int\*, int);

Sorts.cpp:

1. #include "/Users/varka/source/repos/BinSearchAndSorts/Sorts.h"
2. Timer::Timer() {begining = clock\_time::now();}
3. void Timer::reset() {begining = clock\_time::now();}
4. double Timer::elapsed() const
5. {return std::chrono::duration\_cast<second\_time>(clock\_time::now() - begining).count();}
6. Timer::~Timer() {}
7. int\* RandomArray(int size)
8. {
9. int\* random\_array = new int[size];
10. srand(time(NULL));
11. for (int i = 0; i < size; i++)
12. random\_array[i] = rand() % (size\*2+1) - size; // -20 ... 20
13. return random\_array;
14. }
15. unsigned char\* RandomCharArray(int size)
16. {// use unsigned char to be sure in operation char(int)
17. unsigned char\* random\_char\_array = new unsigned char[size];
18. srand(time(NULL));
19. for (int i = 0; i < size; i++)
20. {
21. random\_char\_array[i] = char(rand() % 256); // char type from -128 to 127
22. cout << (random\_char\_array[i]) << endl;
23. }
24. return random\_char\_array;
25. }
26. int BinarySearch(int\* array, int size, int element)
27. {
28. int left\_index = 0, right\_index = size - 1, middle\_index = (left\_index + right\_index) / 2;
29. if (element > array[right\_index])
30. return (right\_index + 1); // returns an index of a new element that doesn't exist now
31. while (array[middle\_index] != element)
32. {
33. if (left\_index == right\_index)
34. // returns a right-index place for a new not-existing element
35. break;
36. if (element < array[middle\_index])
37. right\_index = middle\_index;
38. else
39. {
40. if (right\_index - left\_index == 1)
41. left\_index = right\_index;
42. else
43. left\_index = middle\_index;
44. }
45. middle\_index = (left\_index + right\_index) / 2;
46. }
47. return middle\_index; // returns any index if exists several same numbers
48. }
49. void CountingSort(unsigned char\* array, int size)
50. {
51. unsigned char max = array[0];
52. for (int i = 1; i < size; i++) // find max char
53. {
54. if (array[i] > max)
55. max = array[i];
56. }
57. int\* countMas = new int[int(max) + 1]{0}; // mas to count times we meet char = it's index
58. for (int i = 0; i < size; i++)
59. countMas[int(array[i])]++;
60. int now\_index = 0; // to fill the data mas
61. for (int i = 0; i < int(max) + 1; i++) // on elements of count mas
62. {
63. for (int j = 0; j < countMas[i]; j++) // how many times for one char
64. {
65. array[now\_index] = unsigned char(i);
66. now\_index++;
67. }
68. }
69. }
70. void InsertionSort(int\* array, int size)
71. {
72. if (size <= 1)
73. return;
74. for (int i = 1; i < size; i++)
75. {
76. int key = array[i], key\_index = BinarySearch(array, i, key);
77. // move elements to the right to insert a new element
78. for (int j = i; j > key\_index; j--)
79. array[j] = array[j - 1];
80. array[key\_index] = key; // insert new element
81. }
82. }
83. void BogoSort(int\* array, int size)
84. {
85. bool sorted = IsSorted(array, size);
86. while (!sorted)
87. {
88. srand(time(NULL));
89. for (int i = 0; i < size - 1; i++)
90. {
91. int random\_index = rand() % (size-i);
92. if (array[i] > array[random\_index + i])
93. {
94. swap(array[i], array[random\_index + i]);
95. i = 0;
96. }
97. if (IsSorted(array, size))
98. {
99. sorted = true;
100. break;
101. }
102. }
103. }
104. }
105. void QuickSort(int\* array, int size)
106. {
107. if (size <= 1)
108. return;
109. int pivot = array[size - 1];
110. int bigger\_index = 0, pivot\_index = size - 1; // bigger\_index is the index of the last met bigger than pivot element
111. for (int i = 0; i < size - 1; i++)
112. {
113. if (array[i] > pivot)
114. {
115. bigger\_index = pivot\_index = i; // if we won't find element less than pivot, it'll be true
116. for (++i; i < size; i++) // go forward on elements
117. {
118. if (array[i] <= pivot)
119. {
120. swap(array[bigger\_index], array[i]);
121. if (i != size - 1) // than we already shared the massive on two parts with pivot on it's place
122. pivot\_index = ++bigger\_index; // because all less elements went to the left regarding to pivot\_index
123. }
124. }
125. }
126. }
127. QuickSort(array, pivot\_index); // sort left part
128. if (size > pivot\_index + 1)
129. QuickSort(&array[pivot\_index + 1], size - pivot\_index - 1); // sort right part
130. }
131. int main()
132. {
133. for (int i = 10; i <= 100000; i \*= 10)
134. {
135. int size = i;
136. int\* array = RandomArray(size);
137. int\* array\_copy = array;
138. Timer sort\_time;
139. QuickSort(array, size);
140. cout << "Time for QuickSort for " << size << " elements = " << sort\_time.elapsed() << endl;
141. sort\_time.reset();
142. InsertionSort(array\_copy, size);
143. cout << "Time for InsertionSort for " << size << " elements = " << sort\_time.elapsed() << endl;
144. cout << endl;
145. }
146. return 0;
147. }

UnitTestBinSearchAndSorts.cpp:

1. #include "pch.h"
2. #include "CppUnitTest.h"
3. #include "/Users/varka/source/repos/BinSearchAndSorts/Sorts.h"
4. using namespace Microsoft::VisualStudio::CppUnitTestFramework;
5. namespace UnitTestBinSearchAndSorts
6. {
7. TEST\_CLASS(UnitTestBinSearchAndSorts)
8. {
9. public:
11. TEST\_METHOD(BinarySearchTestFirstElement)
12. {
13. int array[3];
14. array[0] = 1; array[1] = 3; array[2] = 5;
15. Assert::AreEqual(BinarySearch(array, 3, 1), 0);
16. }
17. TEST\_METHOD(BinarySearchTestLastElement)
18. {
19. int array[3];
20. array[0] = 1; array[1] = 3; array[2] = 5;
21. Assert::AreEqual(BinarySearch(array, 3, 5), 2);
22. }
23. TEST\_METHOD(BinarySearchTestMiddleElement)
24. {
25. int array[3];
26. array[0] = 1; array[1] = 3; array[2] = 5;
27. Assert::AreEqual(BinarySearch(array, 3, 3), 1);
28. }
29. TEST\_METHOD(BinarySearchTestNotExistingElement)
30. {
31. int array[3];
32. array[0] = 1; array[1] = 3; array[2] = 5;
33. Assert::AreEqual(BinarySearch(array, 3, 6), 3);
34. }
35. TEST\_METHOD(BinarySearchTestOneElement)
36. {
37. int array[1];
38. array[0] = 1;
39. Assert::AreEqual(BinarySearch(array, 1, 1), 0);
40. }
41. TEST\_METHOD(InsertionSortTestOneElements)
42. {
43. int array[1];
44. array[0] = 1;
45. InsertionSort(array, 1);
46. Assert::AreEqual(IsSorted(array, 1), true);
47. }
48. TEST\_METHOD(InsertionSortTestTenElement)
49. {
50. int\* array = RandomArray(10);
51. InsertionSort(array, 10);
52. Assert::AreEqual(IsSorted(array, 10), true);
53. }
54. TEST\_METHOD(InsertionSortTestOneHundredElement)
55. {
56. int\* array = RandomArray(100);
57. Timer sort\_time;
58. InsertionSort(array, 100);
59. double time\_InsertionSort = sort\_time.elapsed();
60. Assert::AreEqual(IsSorted(array, 100), true);
61. }
62. TEST\_METHOD(InsertionSortTestOneThousandElement)
63. {
64. int\* array = RandomArray(1000);
65. InsertionSort(array, 1000);
66. Assert::AreEqual(IsSorted(array, 1000), true);
67. }
68. TEST\_METHOD(InsertionSortTestTenThousandsElement)
69. {
70. int\* array = RandomArray(10000);
71. InsertionSort(array, 10000);
72. Assert::AreEqual(IsSorted(array, 10000), true);
73. }
74. TEST\_METHOD(InsertionSortTestHundredThousandsElement)
75. {
76. int\* array = RandomArray(100000);
77. InsertionSort(array, 100000);
78. Assert::AreEqual(IsSorted(array, 100000), true);
79. }
80. TEST\_METHOD(BogoSortTestElements)
81. {
82. int\* array = RandomArray(10);
83. BogoSort(array, 10);
84. Assert::AreEqual(IsSorted(array, 10), true);
85. }
86. TEST\_METHOD(BogoSortTestOneElements)
87. {
88. int\* array = RandomArray(1);
89. BogoSort(array, 1);
90. Assert::AreEqual(IsSorted(array, 1), true);
91. }
92. TEST\_METHOD(CountingSortTestOneElement)
93. {
94. unsigned char\* char\_array = RandomCharArray(1);
95. CountingSort(char\_array, 1);
96. Assert::AreEqual(IsSorted(char\_array, 1), true);
97. }
98. TEST\_METHOD(CountingSortTestElements)
99. {
100. unsigned char\* char\_array = RandomCharArray(10);
101. CountingSort(char\_array, 10);
102. Assert::AreEqual(IsSorted(char\_array, 10), true);
103. }
104. TEST\_METHOD(CountingSortTestManyElements)
105. {
106. unsigned char\* char\_array = RandomCharArray(100000);
107. CountingSort(char\_array, 100000);
108. Assert::AreEqual(IsSorted(char\_array, 100000), true);
109. }
110. TEST\_METHOD(QuickSortTestOneElement)
111. {
112. int\* array = RandomArray(1);
113. QuickSort(array, 1);
114. Assert::AreEqual(IsSorted(array, 1), true);
115. }
116. TEST\_METHOD(QuickSortTestElements)
117. {
118. int\* array = RandomArray(1000);
119. QuickSort(array, 1000);
120. Assert::AreEqual(IsSorted(array, 1000), true);
121. }
122. TEST\_METHOD(QuickSortTestManyElements)
123. {
124. int\* array = RandomArray(100000);
125. QuickSort(array, 100000);
126. Assert::AreEqual(IsSorted(array, 100000), true);
127. }
128. };
129. }