Государственное образовательное учреждение высшего профессионального

образования

«Московский государственный технический университет

имени Н. Э. Баумана»

(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ЛАБОРАТОРНАЯ РАБОТА №4

Отчет на тему:

«Изучение алгоритмов сортировки»

Монахов Д.И.

ИУ7-54

Москва 2018

Оглавление

[Введение 1](#_Toc527857811)

[1. Аналитическая часть 1](#_Toc527857812)

[1.1 Описание алгоритмов 1](#_Toc527857813)

[2. Конструкторская часть 3](#_Toc527857814)

[2.1 Разработка алгоритмов 3](#_Toc527857815)

[2.2 Сравнительный анализ алгоритмов 5](#_Toc527857816)

[3. Технологическая часть 6](#_Toc527857817)

[3.1 Требования к программному обеспечению 6](#_Toc527857818)

[3.2 Средства реализации 6](#_Toc527857819)

[3.3 Листинг кода 6](#_Toc527857820)

[4. Экспериментальная часть 9](#_Toc527857821)

[4.1 Примеры работы 9](#_Toc527857822)

[4.2 Постановка эксперимента 10](#_Toc527857823)

[4.3 Сравнительный анализ на материале экспериментальных данных 10](#_Toc527857824)

[Заключение 15](#_Toc527857825)

# Введение

В данной лабораторной работе будут реализованы и исследованы алгоритмы, выполняющие сортировку массивов: простые вставки с барьером, пирамидальная сортировка и сортировка расческой.

**1. Аналитическая часть**

В этой части будет дано описание алгоритмов.

**1.1 Описание алгоритмов**

В этой части будут описаны по порядку все исследуемые алгоритмы сортировки.

I) Сортировка слиянием

Сортировка слиянием (англ. merge sort) — алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например — потоки) в определённом порядке. Эта сортировка — хороший пример использования принципа «разделяй и властвуй». Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова или непосредственно, если их размер достаточно мал. Наконец, их решения комбинируются, и получается решение исходной задачи.

II) Сортировка с помощью двоичного дерева

Сортировка с помощью двоичного дерева (сортировка двоичным деревом, сортировка деревом, древесная сортировка, сортировка с помощью бинарного дерева, англ. tree sort) — универсальный алгоритм сортировки, заключающийся в построении двоичного дерева поиска по ключам массива (списка), с последующей сборкой результирующего массива путём обхода узлов построенного дерева в необходимом порядке следования ключей. Данная сортировка является оптимальной при получении данных путём непосредственного чтения с потока (например, из файла, сокета или консоли).

III) Поразрядная сортировка

Исходно предназначен для сортировки целых чисел, записанных цифрами. Но так как в памяти компьютеров любая информация записывается целыми числами, алгоритм пригоден для сортировки любых объектов, запись которых можно поделить на «разряды», содержащие сравнимые значения. Например, так сортировать можно не только числа, записанные в виде набора цифр, но и строки, являющиеся набором символов, и вообще произвольные значения в памяти, представленные в виде набора байт.

Сравнение производится поразрядно: сначала сравниваются значения одного крайнего разряда, и элементы группируются по результатам этого сравнения, затем сравниваются значения следующего разряда, соседнего, и элементы либо упорядочиваются по результатам сравнения значений этого разряда внутри образованных на предыдущем проходе групп, либо переупорядочиваются в целом, но сохраняя относительный порядок, достигнутый при предыдущей сортировке. Затем аналогично делается для следующего разряда, и так до конца.

**2. Конструкторская часть**

В этой части будут написан псевдокод алгоритмов, посчитана сложность алгоритмов, алгоритмы будут сравнены друг с другом.

**2.1 Разработка алгоритмов**

Эта часть содержит описание алгоритмов на псевдокоде.

1) Псевдокод алгоритма “Сортировка слиянием”

TopDownMergeSort(A[], B[], n)

{

CopyArray(A, 0, n, B);

TopDownSplitMerge(B, 0, n, A); }

TopDownSplitMerge(B[], iBegin, iEnd, A[])

{

if(iEnd - iBegin < 2)

return;

// split the run longer than 1 item into halves

iMiddle = (iEnd + iBegin) / 2;

// recursively sort both runs from array A[] into B[]

TopDownSplitMerge(A, iBegin, iMiddle, B);

TopDownSplitMerge(A, iMiddle, iEnd, B);

// merge the resulting runs from array B[] into A[]

TopDownMerge(B, iBegin, iMiddle, iEnd, A);

}

TopDownMerge(A[], iBegin, iMiddle, iEnd, B[])

{

i = iBegin, j = iMiddle;

for (k = iBegin; k < iEnd; k++) {

if (i < iMiddle && (j >= iEnd || A[i] <= A[j])) {

B[k] = A[i];

i = i + 1;

} else {

B[k] = A[j];

j = j + 1;

}

}

}

CopyArray(A[], iBegin, iEnd, B[])

{

for(k = iBegin; k < iEnd; k++)

B[k] = A[k];

}

2) Псевдокод алгоритма “сортировка двоичным деревом”

STRUCTURE BinaryTree

BinaryTree:LeftSubTree

Object:Node

BinaryTree:RightSubTree

PROCEDURE Insert(BinaryTree:searchTree, Object:item)

IF searchTree.Node IS NULL THEN

SET searchTree.Node TO item

ELSE

IF item IS LESS THAN searchTree.Node THEN

Insert(searchTree.LeftSubTree, item)

ELSE

Insert(searchTree.RightSubTree, item)

PROCEDURE InOrder(BinaryTree:searchTree)

IF searchTree.Node IS NULL THEN

EXIT PROCEDURE

ELSE

InOrder(searchTree.LeftSubTree)

EMIT searchTree.Node

InOrder(searchTree.RightSubTree)

PROCEDURE TreeSort(Collection:items)

BinaryTree:searchTree

FOR EACH individualItem IN items

Insert(searchTree, individualItem)

InOrder(searchTree)

3) Псевдокод алгоритма “поразрядная сортировка”

Radix-Sort(A, d)

//It works same as counting sort for d number of passes.

//Each key in A[1..n] is a d-digit integer.

//(Digits are numbered 1 to d from right to left.)

    for j = 1 to d do

        //A[]-- Initial Array to Sort

        int count[10] = {0};

        //Store the count of "keys" in count[]

        //key- it is number at digit place j

        for i = 0 to n do

         count[key of(A[i]) in pass j]++

        for k = 1 to 10 do

         count[k] = count[k] + count[k-1]

        //Build the resulting array by checking

        //new position of A[i] from count[k]

        for i = n-1 downto 0 do

         result[ count[key of(A[i])] ] = A[j]

         count[key of(A[i])]--

        //Now main array A[] contains sorted numbers

        //according to current digit place

        for i=0 to n do

          A[i] = result[i]

    end for(j)

end func

**2.2 Сравнительный анализ алгоритмов**

В этой части будут приведены сложности алгоритмов.

Оценка сложности для алгоритма сортировки деревом:

Лучший случай (дерево сбалансировано): O(N\*Log(N)) [1]

Худший случай (дерево разбалансировано): O(N^2) [1]

Поразрядная сортировка в лучшем, среднем и худшем случае имеет сложность O(wn), где w – размер элемента сортировки. [2]

Расчет сложности лучшего и худшего случаев алгоритма сортировки слиянием:

Путь имеется массив А длиной n. Сортировка слиянием разбивает массив на две одинаковых части и сортирует их по отдельности. Поэтому каждый узел соответствует участку исходного массива. Возьмем, к примеру, участок A[L, R]. Он делится на два участка: A[L, M] и A[M + 1, R], где M = (L + R) / 2. Разбиение этого узла на два займет время равное R – L + 1. Затем слияние двух частей узла в исходный также займет R – L + 1 времени. Следовательно, для каждого участка массива количество операций равно удвоенному размеру участка. Это правило действует для каждого уровня разбиения. Для массива длиной k имеем k + 2 \* k / 2 = 2k операций. Согласно алгоритму, разбиение продолжается до тех пор, пока размер участка не станет равным единице. Построим дерево рекурсии (рисунок 1):

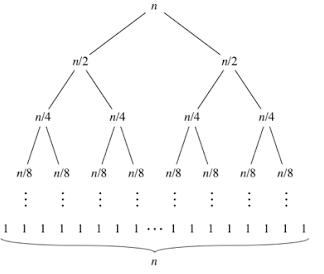


Рисунок . Дерево рекурсии алгоритма сортировки слиянием

Время вычисления для каждого узла дерева равно удвоенному размеру участка массива, который соответствует этому узлу. Следовательно, можно записать выражение 1.

Выражение получено следующим образом – имеется 2^i узлов размера n / 2^i, а время, затрачиваемое на массив размера k равно 2k.

Заметим, что i = k, n / (2 ^ k) = 1, следовательно, n = 2^k => k = log(n).

S обращается в выражение 2.

# 3. Технологическая часть

В этой части обосновывается выбор языка программирования, необходимых библиотек и будет приведен листинг кодов каждого из алгоритмов.

**3.1 Требования к программному обеспечению**

Для того, чтобы запустить программу, необходимо иметь:

ОС windows 10, объем оперативной памяти не менее 512 мб, среда visual studio 2017.

**3.2 Средства реализации**

В качестве языка программирования был выбран C#. Его выбор основан на следующих причинах:

1. язык программирования C# является частью платформы .NET Framework, которая хорошо документирована, обладает хорошей стабильностью работы, и имеет богатый набор библиотечных решений;
2. C# позволяет программисту уделять больше времени написанию кода и не тратить время на побочные действия;
3. C# просто отлаживать, он не обладает уязвимостями языка C++, например, проблемами с указателями;
4. C# по скорости работы почти не уступает языку C++.

Время выполнения замеряется средством .NET под названием StopWatch;

**3.3 Листинг кода**

В этой части будет приведен листинг кода.

Листинг сортировки слияением

protected override void Process(int[] colletion)

{

int[] b = new int[colletion.Length];

TopDownMergeSort(colletion, b);

}

void TopDownMergeSort(int[] a, int[] b)

{

a.CopyTo(b, 0);

TopDownSplitMerge(b, 0, a.Length, a);

}

void TopDownSplitMerge(int[] b, int iBegin, int iEnd, int[] a)

{

if(iEnd - iBegin < 2)

return;

int iMiddle = (iEnd + iBegin) / 2;

TopDownSplitMerge(a, iBegin, iMiddle, b);

TopDownSplitMerge(a, iMiddle, iEnd, b);

TopDownMerge(b, iBegin, iMiddle, iEnd, a);

}

void TopDownMerge(int[] a, int iBegin, int iMiddle, int iEnd, int[] b)

{

int i = iBegin;

int j = iMiddle;

for (int k = iBegin; k < iEnd; k++)

{

if (i < iMiddle && (j >= iEnd || a[i] <= a[j]))

{

b[k] = a[i];

i++;

}

else

{

b[k] = a[j];

j++;

}

}

}

Листинг сортировки двоичным деревом

protected override void Process(int[] collection)

{

result = new int[collection.Length];

index = 0;

result = collection;

TreeSort(collection);

}

private int[] result;

private int index = 0;

class BinaryTree

{

public object Node;

public BinaryTree LeftSubTree;

public BinaryTree RightSubTree;

}

void Insert(BinaryTree searchTree, int item)

{

if(searchTree.LeftSubTree == null)

searchTree.LeftSubTree = new BinaryTree();

if(searchTree.RightSubTree == null)

searchTree.RightSubTree = new BinaryTree();

if (searchTree.Node == null)

{

searchTree.Node = item;

}

else

{

Insert(item < (int) searchTree.Node ?

searchTree.LeftSubTree :

searchTree.RightSubTree, item);

}

}

void Emit(int item)

{

result[index] = item;

index++;

}

void InOrder(BinaryTree searchTree)

{

if (searchTree.Node == null)

{

return;

}

InOrder(searchTree.LeftSubTree);

Emit((int)searchTree.Node);

InOrder(searchTree.RightSubTree);

}

void TreeSort(int[] arr)

{

BinaryTree searchTree = new BinaryTree();

for(int i = 0; i < arr.Length; i++)

{

Insert(searchTree, arr[i]);

}

InOrder(searchTree);

}

Листинг поразрядной сортировки

private void RadixSort(int[] arr)

{

int i = 0, digitPlace = 1;

int[] result = new int[arr.Length];

int largestNum = GetMax(arr);

while (largestNum / digitPlace > 0)

{

int[] count = new int[10];

for (i = 0; i < arr.Length; i++)

count[(arr[i] / digitPlace) % 10]++;

for (i = 1; i < 10; i++)

count[i] += count[i - 1];

for (i = arr.Length - 1; i >= 0; i--)

{

result[count[(arr[i] / digitPlace) % 10] - 1] = arr[i];

count[(arr[i] / digitPlace) % 10]--;

}

for (i = 0; i < arr.Length; i++)

arr[i] = result[i];

digitPlace \*= 10;

}

}

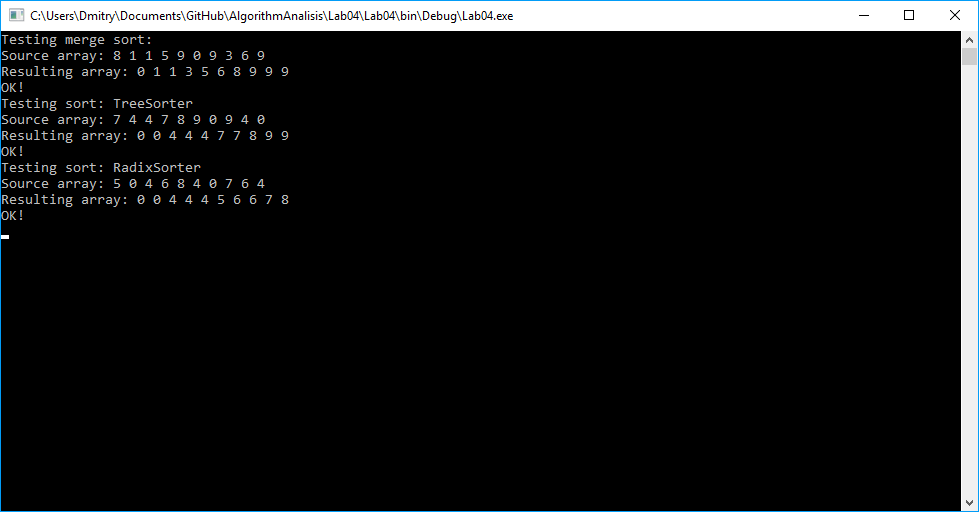
# 4. Экспериментальная часть

В этой части будет показан пример работы программы, проведены эксперименты и сделан вывод о результатах эксперимента.

**4.1 Примеры работы**

В этой части будут приведены примеры работы программы.

Пример, демонстрирующий правильность сортировки всеми алгоритмами.



**4.2 Постановка эксперимента**

Будут проведены 2 эксперимента. В первом эксперименте будут сортироваться массивы размером 100, 200, …, 1000, заполненные случайными числами всеми алгоритмами, а во втором – массивы размером 100, 200, …, 1000, упорядоченные в обратном порядке. Будет замерено время работы алгоритмов, и по результатам этих замеров получены графики, наглядно демонстрирующие скорость работы каждого из алгоритмов.

**4.3 Сравнительный анализ на материале экспериментальных данных**

Рисунки 1-4 подтверждают, что сортировка слиянием и деревом имеют сложность порядка N\*Log(N), а Radix-sort N. Так же видно, что порядок расположения элементов в исходном массиве влияют на время работы алгоритмов сортировки.

# 

Рисунок . Сортировка слиянием и поразрядная сортировка на случайных массивах

# 

Рисунок . Сортировка деревом на случайном массиве

# 

Рисунок . Сортировка слиянием и поразрядная сортировка на массивах, отсортированных по убыванию

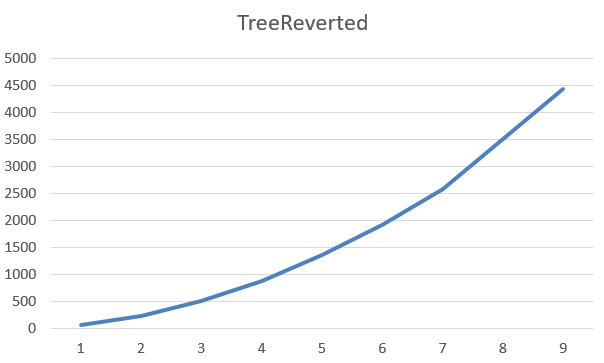


Рисунок . Сортировка деревом на массиве, отсортированном по убыванию

# Заключение

Экспериментально показано, что на скорость работы алгоритмов сортировки влияет изначальное положение элементов в массиве. Все три рассмотренных алгоритма относятся к быстрым алгоритмам сортировки и имеют сложность порядка N\*Log(N).

# Литература

1. 1. https://en.wikipedia.org/wiki/Tree\_sort

2. https://www.codingeek.com/algorithms/radix-sort-explanation-pseudocode-and-implementation/

3. http://www.intuit.ru/studies/courses/12181/1174/lecture/25255?page=3