Содержание

[1 Постановка задачи 3](#_Toc131520301)

[1.1 Начальная постановка задачи 3](#_Toc131520302)

[1.2 Конечная постановка 3](#_Toc131520303)

[2 Методика решения 4](#_Toc131520304)

[2.1 Пирамидальная сортировка 4](#_Toc131520305)

[2.2 Сортировка вставками 4](#_Toc131520306)

[3 Описание алгоритмов решения задачи 5](#_Toc131520307)

[4 Структура данных 7](#_Toc131520308)

[5 Схема алгоритма решения задачи по ГОСТ 19.701-90 10](#_Toc131520309)

[5.1 Схема алгоритма GenerateArray 12](#_Toc131520310)

[5.2 Схема алгоритма outputArray 13](#_Toc131520311)

[5.3 Схема алгоритма siftDown 14](#_Toc131520312)

[5.4 Схема алгоритма insertionSort 16](#_Toc131520313)

[5.5 Схема алгоритма heapsort 18](#_Toc131520314)

[5.6 Схема алгоритма swap 20](#_Toc131520315)

[6 Результаты расчетов 21](#_Toc131520316)

[Приложение А 22](#_Toc131520317)

[Приложение Б 29](#_Toc131520318)

# Постановка задачи

## Начальная постановка задачи

Провести сравнительный анализ быстрой сортировки и сортировки методом “плавающего пузырька” по числу перестановок.

Размерности массивов соответственно: 100, 250, 500, 1000, 2000, 3000.

Типы массивов:

1. Случайный.
2. Сортированный.
3. Перевернутый.

Разработать структуру данных для хранения результатов расчетов.

## Конечная постановка

Провести сравнительный анализ быстрой сортировки и сортировки методом “плавающего пузырька” по числу перестановок.

Размерность массива и его тип вводит пользователь.

Разработать структуру данных для хранения результатов расчетов.

# Методика решения

## Быстрая сортировка

Для теоретического расчета количества перестановок элементов массивов использовались формулы:

1. Случайный массив .
2. Сортированный массив .
3. Перевернутый массив .

Где N – число перестановок, Size – число элементов в массиве.

## Сортировка методом “плавающего пузырька”

Для теоретического расчета количества сравнений элементов массивов использовались формулы:

1. Случайный массив
2. Сортированный массив .
3. Перевернутый массив

Где N – число перестановок, Size – число элементов в массиве.

# Описание алгоритмов решения задачи

Таблица – Описание алгоритмов решения задачи

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| №  п.п. | Наименование алгоритма | Назначение алгоритма | Формальные  параметры | Предпола–гаемый тип реализации |
| 1 | Основной алгоритм | Вызов следующих подпрограмм:  GenerateArray,  QuickSort, BubleSort |  |  |
| 2 | GenerateArray  (x,  i,  Method) | Выполняет генерацию массива х длиной i. Method определяется пользователем и задает способ генерации массива: 1 – случайный, 2 – сортированный, 3 – перевернутый | x– получает от фактического параметра адрес;  i – получает от фактического параметра адрес c защитой;  Method – получает от фактического параметра адрес с защитой | Процедура |
| 3 | outputArray(x, i) | Выводит массив x длиной i | x – получает от фактического параметра адрес c защитой;  i – получает от фактического параметра адрес c защитой | Процедура |
| 4 | siftDown(Tree, nodeStart, nodeLast,  compareCount) | Просеивает двоичное дерево Tree по начальному индексу nodeStart. NodeLast – последний элемент дерева  compareCount – счетчик операций сравнения | Tree – получает от фактического параметра адрес;  nodeStart –  получает от фактического параметра адрес c защитой;  nodeLast –  получает от фактического параметра адрес c защитой; | Процедура |

Продолжение Таблицы 1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  | compareCount – получает от фактического параметра адрес |  |
| 5 | insertionSort(arr, n, flag,  compareCount) | Сортировка вставками массива arr длиной n.  flag – сохраняет выбор пользователя о выводе получаемого массива  compareCount – счетчик операций сравнения | arr – получает от фактического параметра копию значения;  n – получает от фактического параметра адрес c защитой;  flag – получает от фактического параметра адрес c защитой;  compareCount – получает от фактического параметра адрес | Процедура |
| 6 | heapsort(arr, count, flag,  compareCount) | Сортировка кучей массива arr длиной count.  flag – сохраняет выбор пользователя о выводе получаемого массива  compareCount – счетчик операций сравнения | arr – получает от фактического параметра копию значения;  count – получает от фактического параметра адрес c защитой;  flag – получает от фактического параметра адрес c защитой;  compareCount – получает от фактического параметра адрес | Процедура |
| 7 | swap(x, y) | Переставляет элементы x и y местами | x – получает от фактического параметра адрес;  y – получает от фактического параметра адрес | Процедура |

# Структура данных

Таблица – Структура данных основной программы

|  |  |  |
| --- | --- | --- |
| Элементы данных | Рекомендуемый тип | Назначение |
| Table | Тип пользователя «запись» со следующим набором полей:  Record  FillingMethod: byte;  count, NExpIns,  NExpHeap : Integer;  NTeorIns,  NTeorHeap: real;  end; | Запись содержит поля:  FillingMethod – способ заполнения массива;  count – количество элементов в массиве;  NExpIns – экспериментальное количество сравнений в сортировке вставками;  NExpHeap – экспериментальное количество сравнений в сортировке кучей;  NTeorIns – теоретическое количество сравнений в сортировке вставками;  NTeoHeap – теоретическое количество сравнений в сортировке кучей |
| Res | Array of Table | Массив для построения таблицы |
| J | Integer | Счетчик цикла |
| Info | Table | Запись с информацией о сортировках |
| Flag | Byte | Переменная для осуществления вывода массива |
| MainArray | array of Integer | Массив для передачи в процедуры сортировки |

Таблица – Структура данных алгоритма GenerateArray(x, i, Method)

|  |  |  |  |
| --- | --- | --- | --- |
| Элементы данных | Рекомендуемый тип | Назначение | Тип параметра |
| J | Integer | Счетчик цикла | Локальный |
| X | Array of Integer | Генерируемый массив | Формальный |
| I | Integer | Длина массива | Формальный |
| Method | Byte | Метод заполнения массива | Формальный |

Таблица – Структура данных алгоритма outputArray(x, i)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Элементы данных | Рекомендуемый тип | Назначение | | Тип параметра |
| J | Integer | Счетчики цикла | | Локальный |
| X | Array of Integer | Массив для вывода | Формальный | |
| I | Integer | Длина массива | Формальный | |

Таблица – Структура данных алгоритма siftDown(Tree, nodeStart, nodeLast,

compareCount)

|  |  |  |  |
| --- | --- | --- | --- |
| Элементы данных | Рекомендуемый тип | Назначение | Тип параметра |
| isSifted | Boolean | Хранит информацию о том, является ли дерево Max Heap | Локальный |
| child | Integer | Индекс рассматриваемого "ребенка" | Локальный |
| node | Integer | Индекс рассматриваемого "родителя" | Локальный |
| Tree | Array of Integer | Матрица с результатами расчета | Формальный |
| nodeStart | Integer | Индекс начального элемента для просеивания | Формальный |
| nodeLast | Integer | Индекс последнего элемента | Формальный |
| compareCount | Integer | Счетчик операций сравнения | Формальный |

Таблица – Структура данных алгоритма insertionSort(arr, n, flag,

compareCount)

|  |  |  |  |
| --- | --- | --- | --- |
| Элементы данных | Рекомендуемый тип | Назначение | Тип параметра |
| i, j | Integer | Счетчики циклов | Локальный |

Продолжение Таблицы 6

|  |  |  |  |
| --- | --- | --- | --- |
| Tmp | Integer | Переставляемый элемент | Локальный |
| Arr | Array of Integer | Сортируемый массив | Локальный |
| CompareCount | Integer | Счетчик операций сравнения | Формальный |
| N | Integer | Длина массива | Формальный |
| Flag | byte | Переменная для осуществления вывода массива | Формальный |

Таблица – Структура данных алгоритма heapsort(arr, count, flag,

compareCount)

|  |  |  |  |
| --- | --- | --- | --- |
| Элементы данных | Рекомендуемый тип | Назначение | Тип параметра |
| nodeCurr | Integer | Рассматриваемый родитель в дереве | Локальный |
| nodeLast | Integer | Последний родитель в дереве | Локальный |
| Arr | Array of Integer | Сортируемый массив | Локальный |
| compareCount | Integer | Счетчик операций сравнения | Формальный |
| Count | Integer | Длина массива | Формальный |
| Flag | Byte | Переменная для осуществления вывода массива | Формальный |

Таблица – Структура данных алгоритма swap(x, y)

|  |  |  |  |
| --- | --- | --- | --- |
| Элементы данных | Рекомендуемый тип | Назначение | Тип параметра |
| Tmp | Integer | Переменная для перестановки двух элементов | Локальный |
| x, y | Integer | Переставляемые элементы | Формальный |

# Схема алгоритма решения задачи по ГОСТ 19.701-90



Рисунок – Схема алгоритма в соответствии с ГОСТ 19.701-90 (часть 1)



Рисунок – Схема алгоритма в соответствии с ГОСТ 19.701-90 (часть 2)

## Схема алгоритма GenerateArray



Рисунок – Схема алгоритма GenerateArray (часть 2)

## Схема алгоритма outputArray



Рисунок – Схема алгоритма outputArray

## Схема алгоритма siftDown



Рисунок – Схема алгоритма siftDown (часть 1)



Рисунок – Схема алгоритма InputMatrix (часть 2)

## Схема алгоритма insertionSort

## 

Рисунок – Схема алгоритма insertionSort (часть 1)



Рисунок – Схема алгоритма insertionSort (часть 2)

## Схема алгоритма heapsort



Рисунок – Схема алгоритма heapSort (часть 1)



Рисунок – Схема алгоритма heapSort (часть 2)

## Схема алгоритма swap



Рисунок – Схема алгоритма swap

# Результаты расчетов

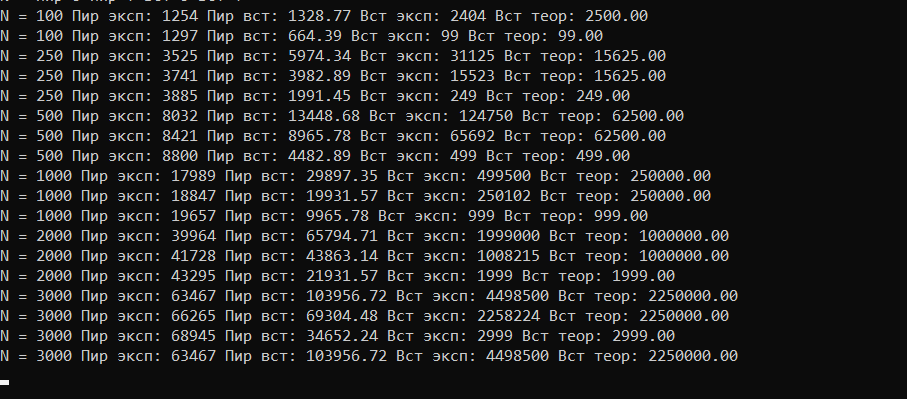


Рисунок – Результаты расчётов

Приложение А

(обязательное)

Исходный код программы

program lab2;

{

Conduct a comparative analysis of pyramidal sorting

and sorting by inserts according to the number

of comparisons.

The dimension of the array and its type are entered by

the user.

Develop a data structure for storing calculation

results.

}

// Determining the Console Program Type

{$APPTYPE CONSOLE}

uses

System.SysUtils;

type

Table = Record

FillingMethod: byte;

count, NExpIns, NExpHeap : Integer;

NTeorIns, NTeorHeap: real;

end;

Res = Array[1..18] of Table;

var

Flag: byte;

Arr: array of Integer;

Info: Res;

j: integer;

{

Flag - a variable for the user to select the output of

the array

Arr1 - array for pyramid sorting

Arr2 - array for sorting by inserts

Info - information about sorting

}

// Procedure for swapping two elements using the third

procedure swap(var x, y:integer);

var

tmp: integer;

// tmp - saving the value

begin

tmp := x;

x := y;

y := tmp;

end;

// Procedure for creating an array

procedure GenerateArray(var x: array of integer; const

i: integer; const Method: byte);

var

j: integer;

// j - counter for the loop

begin

// Depending on the user's choice, the array is filled

// with the necessary elements

for j := 0 to i-1 do

case Method of

1:

x[j] := random(10000);

2:

x[j] := j;

3:

x[j] := i-1-j;

end;

end;

// Output of array values

procedure outputArray(const x: array of integer;

const i: integer);

var

j: integer;

// j - counter for output of values

begin

for j := 0 to i-1 do

write(x[j], ' ');

end;

// Procedure for sifting

procedure siftDown(var Tree: array of integer;

const nodeStart, nodeLast:integer;

var compareCount: integer);

var

isSifted: boolean;

child, node: integer;

{

isSifted - a logical variable for completing the

algorithm

child - index of the considered "child"

node - index of the considered "parent"

}

begin

isSifted := false;

node := nodeStart;

// Cycle for moving an element to its place in the tree

while (not isSifted) and (2\*node <= nodeLast) do

begin

// Getting the index of one child for a parent

child := 2\*node;

// Checking the existence of a second child

if (child + 1 <= nodeLast) then

begin

// Increasing the comparison operation counter

Inc(compareCount);

// Selecting the index of the child with the

// highest values

if (Tree[child + 1] > Tree[child]) then

child := child + 1;

end;

Inc(compareCount);

// Permutation of two elements if they do not satisfy

// Max Heap

if Tree[child] > Tree[node] then

begin

swap(Tree[node], Tree[child]);

node := child;

end

else

isSifted := true;

end;

end;

// Procedure for sorting inserts

procedure insertionSort (arr: array of integer; const n:

integer; const flag: byte; var

compareCount: integer);

var

i, j, tmp: integer;

{

i, j - cycle counters

tmp - saving the rearranged element

}

begin

// Initializing the initial values

i := 1;

compareCount := 0;

// Loop to pass through all elements of the array

while i <= n-1 do

begin

// Initializing variables for the internal loop

tmp := arr[i];

j := i - 1;

// Loop to select the location of the current element

while (j >= 0) and (arr[j] > tmp) do

begin

arr[j+1] := arr[j];

Dec(j);

Inc(compareCount);

end;

// Inserting an element to the desired position

arr[j+1] := tmp;

Inc(i);

// Increase the value of the comparison operations

// counter if the element does not come first

// (when the element comes first, j is equal to -1

// and the comparison of elements does not occur

if j >= 0 then

Inc(compareCount);

end;

// Output the resulting array if the user wants it

if Flag = 1 then

outputArray(arr, n);

end;

// Procedure for performing pyramid sorting

Procedure heapSort(arr: array of integer; const

count: integer; const flag:byte;

var compareCount: integer);

var

nodeCurr, nodeLast:integer;

{

nodeCurr is the current element in the tree

nodeLast - the last element of the tree

}

begin

// Initializing initial values

compareCount := 0;

nodeCurr := count div 2;

// Loop for the initial bringing of the tree to the

// Max Heap state

while nodeCurr >= 0 do

begin

siftDown(arr, nodeCurr, count, compareCount);

dec(nodeCurr);

end;

nodeLast := count;

// Cycle for for final sorting

// 1. We put the largest element in the last place

// 2. Reduce the number of unsorted elements

// 3. Sift the resulting tree by the zero element

while nodeLast > 0 do

begin

swap(arr[0], arr[nodeLast]);

dec(nodeLast);

siftDown(arr, 0, nodeLast, compareCount);

end;

// Output the resulting array if the user wants it

if Flag = 1 then

outputArray(arr, count);

end;

begin

for j := 1 to 18 do

begin

Arr := 0;

case j mod 3 of

0:

info[j].FillingMethod := 3;

1:

info[j].FillingMethod := 1;

2:

info[j].FillingMethod := 2;

end;

case j div 3 of

0:

info[j].count := 100;

1:

info[j].count := 250;

2:

info[j].count := 500;

3:

info[j].count := 1000;

4:

info[j].count := 2000;

5:

info[j].count := 3000;

6:

info[j].count := 3000;

end;

// Setting the array length and generating it

SetLength(Arr, Info[j].count);

GenerateArray(Arr, Info[j].count,

Info[j].FillingMethod);

// Calculation of theoretical values of comparisons

case Info[j].FillingMethod of

1:

begin

Info[j].NTeorHeap := 2\*Info[j].count

\*ln(Info[j].count)/ln(2);

Info[j].NTeorIns := sqr(Info[j].count)/4;

end;

2:

begin

Info[j].NTeorHeap := Info[j].count

\*ln(Info[j].count)/ln(2);

Info[j].NTeorIns := Info[j].count - 1;

end;

3:

begin

Info[j].NTeorHeap := 3\*Info[j].count

\*ln(Info[j].count)/ln(2);

Info[j].NTeorIns := sqr(Info[j].count)/4;

end;

end;

// Implementation of sorting

heapSort(Arr, Info[j].count, Flag,

info[j].NExpHeap);

insertionSort(Arr, Info[j].count, Flag,

Info[j].NExpIns);

WriteLn('N = ', Info[j].count,' Пир эксп: ',

Info[j].NExpHeap,' Пир вст: ',

Info[j].NTeorHeap:5:2,

' Вст эксп: ' , Info[j].NExpIns, ' Вст теор: ',

Info[j].NTeorIns:5:2);

end;

Readln;

end.

Приложение Б

(обязательное)

Тестовые наборы

Таблица – расчеты

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Размерность массива | Тип массива | Быстрая сортировка | | Сортировка пузырьком | |
| Экспериментальное | Теоретическое | Экспериментальное | Теоретическое |
| N=100 | случайный | 49 | 657 | 4950 | 4950 |
| N=100 | сортированный | 0 | 0 | 0 | 0 |
| N=100 | перевернутый | 105 | 657 | 2305 | 2305 |
| N=250 | случайный | 124 | 1983 | 31125 | 31125 |
| N=250 | сортированный | 0 | 0 | 0 | 0 |
| N=250 | перевернутый | 313 | 1983 | 15281 | 15625 |
| N=500 | случайный | 8450 | 8965 | 61759 | 62500 |
| N=500 | сортированный | 0 | 0 | 0 | 0 |
| N=500 | перевернутый | 8032 | 13449 | 124750 | 62500 |
| N=1000 | случайный | 18950 | 19932 | 249060 | 250000 |
| N=1000 | сортированный | 19657 | 9965 | 999 | 999 |
| N=1000 | перевернутый | 17989 | 29897 | 499500 | 250000 |
| N=2000 | случайный | 41725 | 43863 | 987523 | 1000000 |
| N=2000 | сортированный | 43295 | 21932 | 1999 | 1999 |
| N=2000 | перевернутый | 39964 | 65795 | 1999000 | 1000000 |
| N=3000 | случайный | 66296 | 69304 | 2231949 | 2250000 |
| N=3000 | сортированный | 68945 | 34652 | 2999 | 2999 |
| N=3000 | перевернутый | 63467 | 103957 | 4498500 | 2250000 |