Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра программного обеспечения информационных технологий

Дисциплина: Языки программирования

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовой работе

на тему:

«АНАЛИЗ И СРАВНЕНИЕ АЛГОРИТМОВ СОРТИРОВКИ»

БГУИР КП 1-40 01 01  ПЗ

Студент: Киселев Н.И.

Руководитель: !

Минск 2019

Учреждение образования

«Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

УТВЕРЖДАЮ

Заведующий кафедрой ПОИТ

––––––––––––––––––––––––

(подпись)

Лапицкая Н.В. 2019 г.

ЗАДАНИЕ

по курсовому проектированию

Студенту    –––––––––––––

1. Тема работы    ––––

2. Срок сдачи студентом законченной работы 23.05.2016

3. Исходные данные к работе   .

4. Содержание расчётно-пояснительной записки (перечень вопросов, которые подлежат разработке)

*Введение.*

*1.Аналитический обзор литературы и существующих аналогов;*

*2.Разработка алгоритма;*

*3. Разработка программного средства;*

*4. Обоснование технических приемов программирования;*

*5. Тестирование, экспериментальные исследования и анализ полученных результатов;*

*6. Руководство пользователя программы;*

*Заключение, список литературы, ведомость, приложения.*

5. Перечень графического материала (с точным обозначением обязательных чертежей и графиков)

1. Схема программы

6. Консультант по курсовому проекту Болтак С.В.

7. Дата выдачи задания 15.02.2016 г.––   –

8. Календарный график работы над проектом на весь период проектирования (с обозначением сроков выполнения и процентом от общего объёма работы):

раздел 1, введение к 02.03.2016 – 10 % готовности работы;

разделы 2 к 15.03.2016 – 30 % готовности работы;

разделы 3,4 к 15.04.2016 – 60 % готовности работы;

раздел 5, 6 к 05.05.2016 – 90 % готовности работы;

оформление пояснительной записки и графического материала к 20.05.2016 – 100 % готовности работы.

Защита курсового проекта с 23.05 по 12.06 2016 г.

РУКОВОДИТЕЛЬ С.В. Болтак

(подпись)

Задание принял к исполнению –––\_\_\_\_––     15.02.2016 г.

(дата и подпись студента)

**СОДЕРЖАНИЕ**

Введение ........................................................................................................ 5

1 Анализ алгоритмов сортировки........................... 6

1.1 Задача сортировки

1.2 Классификация алгоритмов сортировки .........................................

1.3 Оценка алгоритма сортировки............

2 Основные алгоритмы сортировки и их реализация ….......................................

2.1 Обменные

2.1.1 Сортировка пузырьком

2.1.2 Сортировка перемешиванием

2.1.3 Сортировка расчёской

2.1.4 Быстрая сортировка, сортировка Хоара

2.2 Выбором

2.2.1 Сортировка выбором

2.2.2 Пирамидальная сортировка

2.3 Вставками

2.3.1 Сортировка вставками

2.3.2 Сортировка Шелла

2.3.3 Сортировка деревом

2.4 Сортировка слиянием

3 Разработка программного средства для исследования алгоритмов сортировки..................................................................

3.1 Архитектура классов сортировки

3.2 Пользовательский интерфейс программы ....................................................

3.3 Визуализация алгоритмов.................................................................................

4 Тестирование и анализ полученных результатов………

4.1 Подраздел первый .................................................................................

4.2 Подраздел второй ...........................................................................…..

4.3 Подраздел третий ..................................................................................

5 Руководство пользователя программы......................................................

Заключение ...............................................................................................

Список использованных источников ......................................................

Приложение А.................................................................

Приложение Б …….

**ВВЕДЕНИЕ**

Рекомендуется следующее содержание введения:

* краткий анализ достижений в той области, которой посвящена тема курсового проекта или курсовой работы;
* цель курсового проектирования;
* принципы, положенные в основу проектирования, научного исследования, поиска технического решения;
* краткое изложение содержания разделов пояснительной записки с обязательным указанием задач, решению которых они посвящены.

**1 АНАЛИЗ АЛГОРИТМОВ СОРТИРОВКИ**

* 1. Задача сортировки

# Классификация алгоритмов сортировки

Устойчивость (англ. stability) — устойчивая сортировка не меняет взаимного расположения элементов с одинаковыми ключами[13].

Естественность поведения — эффективность метода при обработке уже упорядоченных или частично упорядоченных данных. Алгоритм ведёт себя естественно, если учитывает эту характеристику входной последовательности и работает лучше.

Использование операции сравнения. Алгоритмы, использующие для сортировки сравнение элементов между собой, называются основанными на сравнениях. Минимальная трудоемкость худшего случая для этих алгоритмов составляет {\displaystyle O} O( {\displaystyle n\cdot \log n} n\cdot \log n), но они отличаются гибкостью применения. Для специальных случаев (типов данных) существуют более эффективные алгоритмы.

Ещё одним важным свойством алгоритма является его сфера применения. Здесь основных типов упорядочения два:

Внутренняя сортировка оперирует массивами, целиком помещающимися в оперативной памяти с произвольным доступом к любой ячейке. Данные обычно упорядочиваются на том же месте без дополнительных затрат.

В современных архитектурах персональных компьютеров широко применяется подкачка и кэширование памяти. Алгоритм сортировки должен хорошо сочетаться с применяемыми алгоритмами кэширования и подкачки.

Внешняя сортировка оперирует запоминающими устройствами большого объёма, но не с произвольным доступом, а последовательным (упорядочение файлов), то есть в данный момент «виден» только один элемент, а затраты на перемотку по сравнению с памятью неоправданно велики. Это накладывает некоторые дополнительные ограничения на алгоритм и приводит к специальным методам упорядочения, обычно использующим дополнительное дисковое пространство. Кроме того, доступ к данным во внешней памяти производится намного медленнее, чем операции с оперативной памятью.

Доступ к носителю осуществляется последовательным образом: в каждый момент времени можно считать или записать только элемент, следующий за текущим.

Объём данных не позволяет им разместиться в ОЗУ.

Также алгоритмы классифицируются по:

потребности в дополнительной памяти или её отсутствию

потребности в знаниях о структуре данных, выходящих за рамки операции сравнения, или отсутствию таковой

* 1. Оценка алгоритма сортировки

Алгоритмы сортировки оцениваются по скорости выполнения и эффективности использования памяти:

Время — основной параметр, характеризующий быстродействие алгоритма. Называется также вычислительной сложностью. Для упорядочения важны худшее, среднее и лучшее поведение алгоритма в терминах мощности входного множества A. Если на вход алгоритму подаётся множество A, то обозначим n = |A|. Для типичного алгоритма хорошее поведение — это O(n log n) и плохое поведение — это O(n2). Идеальное поведение для упорядочения — O(n). Алгоритмы сортировки, использующие только абстрактную операцию сравнения ключей всегда нуждаются по меньшей мере в сравнениях. Тем не менее, существует алгоритм сортировки Хана (Yijie Han) с вычислительной сложностью O(n log log n log log log n), использующий тот факт, что пространство ключей ограничено (он чрезвычайно сложен, а за О-обозначением скрывается весьма большой коэффициент, что делает невозможным его применение в повседневной практике). Также существует понятие сортирующих сетей. Предполагая, что можно одновременно (например, при параллельном вычислении) проводить несколько сравнений, можно отсортировать n чисел за O(log2 n) операций. При этом число n должно быть заранее известно;

Память — ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных. Как правило, эти алгоритмы требуют O(log n) памяти. При оценке не учитывается место, которое занимает исходный массив и независящие от входной последовательности затраты, например, на хранение кода программы (так как всё это потребляет O(1)). Алгоритмы сортировки, не потребляющие дополнительной памяти, относят к сортировкам на месте.

**1.2.1** Связь с распределенными системами

Компьютерные сети относятся к распределенным (или децентрализованным) вы­числительным системам. Поскольку основным признаком распределенной вычис­лительной системы является наличие нескольких центров обработки данных, то наряду с компьютерными сетями к распределенным системам относят также муль­типроцессорные компьютеры и многомашинные вычислительные комплексы

**1.2.2** Мультипроцессорные компьютеры

**1.3** Понятие «открытая система» и проблемы стандартизации

**Аналитический обзор литературы** должен содержать сравнительный анализ существующих по данной тематике технических научных решений (алгоритмов, систем, проектов, примеры возможных аналогов).

***Раздел должен заканчиваться четкой постановкой общей задачи на курсовое проектирование и формулировкой конкретных детализированных задач, решаемых в курсовой работе.***

Пример оформления таблицы:

Таблица 1.1 – Схема маршрутизации маршрутизатора

|  |  |  |  |
| --- | --- | --- | --- |
| Номер сети  назначения | Сетевой адрес следующего маршрутиза-тора | Сетевой адрес выходного порта | Расстояние до сети назначения |
| 51 | М1(2) | М4(1) | 1 |
| 52 | - | М4(1) | 0 (подсоединена) |
| 53 | М1(2) | М4(1) | 1 |
| 54 | М2(1) | М4(1) | 1 |

Пример оформления рисунка:

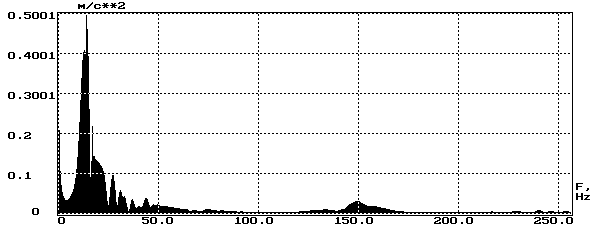
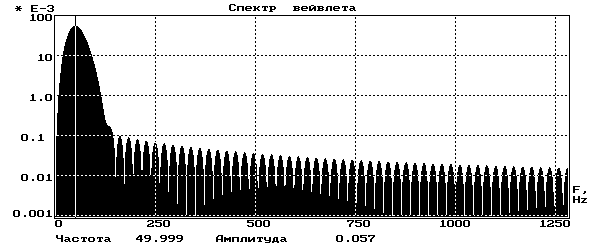
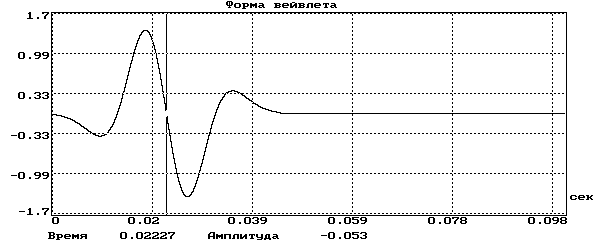


Рисунок 1.1 − Спектр виброударного импульса



а) частотная характеристика вейвлета



б) форма вейвлета

Рисунок 1.2 − Частотная характеристика и форма

гауссова вейвлета 3-го порядка с центральной частотой 50 Гц

# 

# 

**2 ОСНОВНЫЕ АЛГОРИТМЫ СОРТИРОВКИ И ИХ РЕАЛИЗАЦИЯ**

**2.1** Обменные

**2.1.1** Сортировка пузырьком

Алгоритм:

На каждой итерации последовательно сравниваются соседние элементы, и, если порядок в паре неверный, то элементы меняют местами.

За каждый проход по массиву как минимум один элемент встает на свое место, поэтому необходимо совершить не более n−1 проходов, где n размер массива, чтобы отсортировать массив.

После i-ой итерации внешнего цикла i последних элементов уже находятся на своих местах в отсортированном порядке, поэтому нет необходимости производить их сравнения друг с другом. Следовательно, внутренний цикл можно выполнять не до n−2, а до n−i−2.

Если после выполнения внутреннего цикла не произошло ни одного обмена, то массив уже отсортирован. Поэтому внутренний цикл можно выполнять не n−1 раз, а до тех пор, пока во внутреннем цикле происходят обмены.

Ниже приведен псевдокод сортировки пузырьком, на вход которой подается массив a[0..n−1].

function bubbleSort(array a):

i = 0

swapped = true

while swapped

swapped = false

for j = 0 to n - i - 2

if a[j] > a[j + 1]

swap(a[j], a[j + 1])

swapped = true

i = i + 1

В данной сортировке выполняются всего два различных вида операции: сравнение элементов и их обмен. Поэтому время всего алгоритма T=T1+T2T=T1+T2, где T1T1 — время, затрачиваемое на сравнение элементов, а T2T2 — время, за которое мы производим все необходимые обмены элементов.

Так как в алгоритме меняться местами могут только соседние элементы, то каждый обмен уменьшает количество [инверсий](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A2%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0_%D0%B8%D0%BD%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D0%B9) на единицу. Следовательно, количество обменов равно количеству инверсий в исходном массиве вне зависимости от реализации сортировки. Максимальное количество инверсий содержится в массиве, элементы которого отсортированы по убыванию. Несложно посчитать, что количество инверсий в таком массиве n(n−1)2n(n−1)2. Получаем, что T2=O(n2)T2=O(n2).

В неоптимизированной реализации на каждой итерации внутреннего цикла производятся n−1n−1 сравнений, а так как внутренний цикл запускается также n−1n−1 раз, то за весь алгоритм сортировки производятся (n−1)2(n−1)2 сравнений.

В оптимизированной версии точное количество сравнений зависит от исходного массива. Известно, что худший случай равен n(n−1)2n(n−1)2, а лучший — n−1n−1. Следовательно, T1=O(n2)T1=O(n2).

В итоге получаем T=T1+T2=O(n2)+O(n2)=O(n2)T=T1+T2=O(n2)+O(n2)=O(n2).

|  |  |  |  |
| --- | --- | --- | --- |
| **Сложность** | Наилучший случай | В среднем | Наихудший случай |
| Время | O(n) | O(n2) | O(n2) |
| Память | O(1) | O(1) | O(1) |

**2.1.2** Сортировка перемешиванием

**Сортировка перемешиванием** (англ. *cocktail sort*), также известная как **Шейкерная сортировка** — разновидность пузырьковой сортировки, сортирующая массив в двух направлениях на каждой итерации.

function shakerSort(array a):

begin = -1

end = n - 2

while swapped

swapped = false

begin++

for i = begin to end

if a[i] > a[i + 1]

swap(a[i], a[i + 1])

swapped = true

if !swapped

break

swapped = false

end--

for i = end downto begin

if a[i] > a[i + 1]

swap(a[i], a[i + 1])

swapped = true

В среднем, сортировка перемешиванием работает в два раза быстрее пузырька. Сложность — O(n2)O(n2), но стремится она к O(k⋅n)O(k⋅n), где kk — максимальное расстояние элемента в неотсортированном массиве от его позиции в отсортированном массиве.

**2.1.3** Сортировка расчёской

Сортировка расческой (англ. comb sort) — модификация пузырьковой сортировки, основанной на сравнении элементов на расстоянии.

Производятся неоднократные прогоны по массиву, при которых сравниваются пары элементов. Если они неотсортированы друг относительно друга - то производится обмен. В результате крупные элементы мигрируют в конец массива, а небольшие по значению - в начало.

В пузырьковой сортировке при каждом прогоне по массиву сравниваются соседние элементы. Здесь же сравниваются элементы, между которыми некоторое фиксированное расстояние. При каждом следующем прохождении расстояние уменьшается, пока не достигнет минимальной величины.

Уменьшающееся расстояние между сравниваемыми элементами рассчитывается с помощью специальной величины, называемой фактором уменьшения. Длина массива делится на этот фактор, это и есть разрыв между индексами. После каждого прохода расстояние делится на фактор уменьшения и таким образом получается новое значение. В конце концов оно сужается до минимального значения - единицы, и массив просто досортировывается обычным "пузырьком".

В результате практических тестов и теоретических исследований оптимальное значение для фактора уменьшения определено следующее:Фактор уменьшения

function combSort(array a):

k = 1.3

jump = n

bool swapped = true

while jump > 1 and swapped

if jump > 1

jump /= k

swapped = false

for i = 0 to size - jump - 1

if a[i + jump] < a[i]

swap(a[i], a[i + jump])

swapped = true

Сложность — O(n2), но стремится к O(nlogn). Является самой быстрой квадратичной сортировкой. Недостаток — она неустойчива.

**2.1.4** Быстрая сортировка, сортировка Хоара

Выбрать опорный элемент p - середину массива

Разделить массив по этому элементу

Если подмассив слева от p содержит более одного элемента,

вызвать quickSort для него.

Если подмассив справа от p содержит более одного элемента,

вызвать quickSort для него.

function quicksort(array a, low, high):

if low < high then

p := partition(a, low, high)

quicksort(a, low, p - 1)

quicksort(a, p + 1, high)

Разбиение Хоара

Данная схема использует два индекса (один в начале массива, другой в конце), которые приближаются друг к другу, пока не найдётся пара элементов, где один больше опорного и расположен перед ним, а второй меньше и расположен после. Эти элементы меняются местами. Обмен происходит до тех пор, пока индексы не пересекутся. Алгоритм возвращает последний индекс.[5] Схема Хоара эффективнее схемы Ломуто, так как происходит в среднем в три раза меньше обменов (swap) элементов, и разбиение эффективнее, даже когда все элементы равны.[6] Подобно схеме Ломуто, данная схема также показывает эффективность в O(n2), когда входной массив уже отсортирован. Сортировка с использованием данной схемы нестабильна. Следует заметить, что конечная позиция опорного элемента необязательно совпадает с возвращённым индексом. Псевдокод

function partition(array a, low, high):

pivot := a[(low + high) / 2]

i := low - 1

j := high + 1

while true

do

i := i + 1

while a[i] < pivot

do

j := j - 1

while a[j] > pivot

if i >= j then

return j

swap a[i] with a[j]

В ранних реализациях, как правило, опорным выбирался первый элемент, что снижало производительности на отсортированных массивах. Для улучшения эффективности может выбираться средний, случайный элемент или (для больших массивов) [медиана](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D0%B4%D0%B8%D0%B0%D0%BD%D0%B0_(%D1%81%D1%82%D0%B0%D1%82%D0%B8%D1%81%D1%82%D0%B8%D0%BA%D0%B0)) первого, среднего и последнего элементов.[[2]](https://ru.wikipedia.org/wiki/%D0%91%D1%8B%D1%81%D1%82%D1%80%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0#cite_note-sedgewickBook-2) Медиана всей последовательности является оптимальным опорным элементом, но её вычисление слишком трудоёмко для использования в сортировке.

Каждое разделение требует, очевидно, Theta(n) операций. Количество шагов деления(глубина рекурсии) составляет приблизительно log n, если массив делится на более-менее равные части. Таким образом, общее быстродействие: O(n log n), что и имеет место на практике.

Однако, возможен случай таких входных данных, на которых алгоритм будет работать за O(n2) операций. Такое происходит, если каждый раз в качестве центрального элемента выбирается максимум или минимум входной последовательности. Если данные взяты случайно, вероятность этого равна 2/n. И эта вероятность должна реализовываться на каждом шаге... Вообще говоря, малореальная ситуация.

Метод неустойчив. Поведение довольно естественно, если учесть, что при частичной упорядоченности повышаются шансы разделения массива на более равные части.

Сортировка использует дополнительную память, так как приблизительная глубина рекурсии составляет O(log n), а данные о рекурсивных подвызовах каждый раз добавляются в стек.

**2.2** Выбором

**2.2.1** Сортировка выбором

1. Пусть имеется массив A размером N. Берем первый элемент последовательности A[i], здесь i – номер элемента, для первого i равен 1;
2. находим минимальный (максимальный) элемент последовательности и запоминаем его номер в переменную key;
3. если номер первого элемента и номер найденного элемента не совпадают, т. е. если key≠1, тогда два этих элемента обмениваются значениями, иначе никаких манипуляций не происходит;
4. увеличиваем i на 1 и продолжаем сортировку оставшейся части массива, а именно с элемента с номером 2 по N, так как элемент A[1] уже занимает свою позицию;

С каждым последующим шагом размер подмассива, с которым работает алгоритм, уменьшается на 1, но на способ сортировки это не влияет, он одинаков для каждого шага

function selectionSort(array a):

for i = 0 to n - 2

min = i

for j = i + 1 to n - 1

if a[j] < a[min]

min = j

swap(a[i], a[min])

**2.2.2** Пирамидальная сортировка

Сортировка кучей, пирамидальная сортировка (англ. Heapsort) — алгоритм сортировки, использующий структуру данных двоичная куча.

Куча - это специальная древовидная структура данных. Говорят, что двоичное дерево следует структуре данных кучи, если

это полное бинарное дерево;

все узлы в дереве следуют тому свойству, что они больше своих потомков, то есть самый большой элемент находится в корне, и оба его потомка меньше, чем корень, и так далее. Такая куча называется убывающая куча (Max-Heap). Если вместо этого все узлы меньше своих потомков, это называется возрастающая куча (Min-Heap).

Удобная структура данных для сортирующего дерева — такой массив Array, что Array[0] — элемент в корне, а потомки элемента Array[i] являются Array[2i+1] и Array[2i+2].

Основная идея - ищем максимальный элемент в неотсортированной части массива и ставим его в конец этого подмассива. В поисках максимума подмассив перестраивается в так называемое **сортирующее дерево** (она же **двоичная куча**, она же **пирамида**), в результате чего максимум сам "всплывает" в начало массива. После этого перемещаем максимум в конец подмассива. Затем над оставшейся частью массива снова осуществляется процедура перестройки в сортирующее дерево с последующим перемещением максимума в конец подмассива.

Необходимо отсортировать массив AA, размером nn.

Построим на базе этого массива за O(n)O(n) кучу для максимума.

Так как максимальный элемент находится в корне, то если поменять его местами с A[n−1]A[n−1], он встанет на своё место.

Далее вызовем процедуру siftDown(0)siftDown(0), предварительно уменьшив heapSizeheapSize на 11. Она за O(logn)O(log⁡n) просеет A[0]A[0] на нужное место и сформирует новую кучу (так как мы уменьшили её размер, то куча располагается с A[0]A[0] по A[n−2]A[n−2], а элемент A[n−1]A[n−1]находится на своём месте). Повторим эту процедуру для новой кучи, только корень будет менять местами не с A[n−1]A[n−1], а с A[n−2]A[n−2]. Делая аналогичные действия, пока heapSizeheapSize не станет равен 11, мы будем ставить наибольшее из оставшихся чисел в конец не отсортированной части. Очевидно, что таким образом, мы получим отсортированный массив.

Hачать построение пирамиды можно с a[k]...a[n], k = [size/2]. Эта часть массива удовлетворяет свойству пирамиды, так как не существует индексов i,j: i = 2i+1 ( или j = 2i+2 )... Просто потому, что такие i,j находятся за границей массива.

Следует заметить, что неправильно говорить о том, что a[k]..a[n] является пирамидой как самостоятельный массив. Это, вообще говоря, не верно: его элементы могут быть любыми. Свойство пирамиды сохраняется лишь в рамках исходного, основного массива a[0]...a[n].

Далее будем расширять часть массива, обладающую столь полезным свойством, добавляя по одному элементу за шаг. Следующий элемент на каждом шаге добавления - тот, который стоит перед уже готовой частью.

Чтобы при добавлении элемента сохранялась пирамидальность, будем использовать следующую процедуру расширения пирамиды a[i+1]..a[n] на элемент a[i] влево:

1. Смотрим на сыновей слева и справа - в массиве это a[2i+1] и a[2i+2] и выбираем наибольшего из них.
2. Если этот элемент больше a[i] - меняем его с a[i] местами и идем к шагу 2, имея в виду новое положение a[i] в массиве. Иначе конец процедуры.

Итак, задача построения пирамиды из массива успешно решена. Как видно из свойств пирамиды, в корне всегда находится максимальный элемент. Отсюда вытекает алгоритм фазы 2:

Берем верхний элемент пирамиды a[0]...a[n] (первый в массиве) и меняем с последним местами. Теперь "забываем" об этом элементе и далее рассматриваем массив a[0]...a[n-1]. Для превращения его в пирамиду достаточно просеять лишь новый первый элемент.

Повторяем шаг 1, пока обрабатываемая часть массива не уменьшится до одного элемента.

function Heapsort(array A):

BuildHeap(A)

for i = n to 1

swap(A[1], A[i])

n = n - 1

Heapify(A, 1)

function BuildHeap(array A):

n = elements\_in(A)

for i = floor(n/2) to 1

Heapify(A,i,n)

function Heapify(array A, int i, int n):

left = 2i

right = 2i+1

if (left <= n) and (A[left] > A[i])

max = left

else

max = i

if (right<=n) and (A[right] > A[max])

max = right

if (max != i)

swap(A[i], A[max])

Heapify(A, max)

У алгоритма нет благоприятных и вырожденных случаев. При любом входящем массиве (даже если он почти отсортирован) сортировка имеет одну и ту же временную сложность - O(n log n).

Это неустойчивый алгоритм сортировки с временем работы O(nlogn) , где n — количество элементов для сортировки, и использующий O(1) дополнительной памяти.

Операция siftDownsiftDown работает за O(logn)O(log⁡n). Всего цикл выполняется (n−1)(n−1) раз. Таким образом сложность сортировки кучей является O(nlogn)O(nlog⁡n).

Достоинства:

* худшее время работы — O(nlogn)O(nlog⁡n),
* требует O(1)O(1) дополнительной памяти.

Недостатки:

* неустойчивая,
* на почти отсортированных данных работает столь же долго, как и на хаотических данных.

Из-за сложности алгоритма выигрыш получается только на больших *n*. На небольших *n* (до нескольких тысяч) быстрее [сортировка Шелла](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%A8%D0%B5%D0%BB%D0%BB%D0%B0).

**2.3** Вставками

**2.3.1** Сортировка вставками

На вход алгоритма подаётся последовательность {\displaystyle n} чисел: {\displaystyle a\_{1},a\_{2},...,a\_{n}}. Сортируемые числа также называют *ключами*. Входная последовательность на практике представляется в виде массива с {\displaystyle n} элементами. На выходе алгоритм должен вернуть перестановку исходной последовательности {\displaystyle a\_{1}^{'},a\_{2}^{'},...,a\_{n}^{'}}, чтобы выполнялось следующее соотношение {\displaystyle a\_{1}^{'}\leqslant a\_{2}^{'}\leqslant ...\leqslant a\_{n}^{'}}[[2]](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%B2%D1%81%D1%82%D0%B0%D0%B2%D0%BA%D0%B0%D0%BC%D0%B8#cite_note-_2a5a5239f7db967b-2).

В начальный момент отсортированная последовательность пуста. На каждом шаге алгоритма выбирается один из элементов входных данных и помещается на нужную позицию в уже отсортированной последовательности до тех пор, пока набор входных данных не будет исчерпан. В любой момент времени в отсортированной последовательности элементы удовлетворяют требованиям к выходным данным алгоритма[[3]](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%B2%D1%81%D1%82%D0%B0%D0%B2%D0%BA%D0%B0%D0%BC%D0%B8#cite_note-_2a5a5239f7db967a-3).

Данный алгоритм можно ускорить при помощи использования бинарного поиска для нахождения места текущему элементу в отсортированной части. Проблема с долгим сдвигом массива вправо решается при помощи смены указателей[[4]](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%B2%D1%81%D1%82%D0%B0%D0%B2%D0%BA%D0%B0%D0%BC%D0%B8#cite_note-4).

Однако в сортировке пузырьком или выбором можно было четко заявить, что на i-м шаге элементы a[0]...a[i] стоят на правильных местах и никуда более не переместятся. Здесь же подобное утверждение будет более слабым: последовательность a[0]...a[i] упорядочена. При этом по ходу алгоритма в нее будут вставляться (см. название метода) все новые элементы.

Будем разбирать алгоритм, рассматривая его действия на i-м шаге. Как говорилось выше, последовательность к этому моменту разделена на две части: готовую a[0]...a[i] и неупорядоченную a[i+1]...a[n].

На следующем, (i+1)-м каждом шаге алгоритма берем a[i+1] и вставляем на нужное место в готовую часть массива.

Поиск подходящего места для очередного элемента входной последовательности осуществляется путем последовательных сравнений с элементом, стоящим перед ним.

В зависимости от результата сравнения элемент либо остается на текущем месте(вставка завершена), либо они меняются местами и процесс повторяется.

Таким образом, в процессе вставки мы "просеиваем" элемент x к началу массива, останавливаясь в случае, когда

Hайден элемент, меньший x или

Достигнуто начало последовательности.

function insertionSort(a):

for i = 1 to n - 1

j = i - 1

while j ⩾ 0 and a[j] > a[j + 1]

swap(a[j], a[j + 1])

j—

Сортировка вставками наиболее эффективна когда массив уже частично отсортирован и когда элементов массива не много. Если же элементов меньше 10 то данный алгоритм является лучшим

Время выполнения алгоритма зависит от входных данных: чем большее множество нужно отсортировать, тем большее время потребуется для выполнения сортировки. Также на время выполнения влияет исходная упорядоченность массива. Время работы алгоритма для различных входных данных одинакового размера зависит от элементарных операций, или шагов, которые потребуется выполнить

|  |  |  |  |
| --- | --- | --- | --- |
| **Сложность** | Наилучший случай | В среднем | Наихудший случай |
| Время | O(n) | O(n2) | O(n2) |
| Память | O(1) | O(1) | O(1) |

**2.3.2** Сортировка Шелла

При сортировке Шелла сначала сравниваются и сортируются между собой значения, стоящие один от другого на некотором расстоянии {\displaystyle d} d. После этого процедура повторяется для некоторых меньших значений {\displaystyle d}d, а завершается сортировка Шелла упорядочиванием элементов при {\displaystyle d=1}d=1 (то есть обычной [сортировкой вставками](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%B2%D1%81%D1%82%D0%B0%D0%B2%D0%BA%D0%BE%D0%B9))

Каждый проход в алгоритме характеризуется смещением hihi, таким, что сортируются элементы отстающие друг от друга на hihi позиций. Шелл предлагал использовать ht=N/2ht=N/2, ht−1=ht/2ht−1=ht/2, …… , h0=1h0=1. Возможны и другие смещения, но h0=1h0=1всегда.

* Начало.
* **Шаг 0.** i=ti=t.
* **Шаг 1.** Разобьем массив на списки элементов, отстающих друг от друга на hihi. Таких списков будет hihi.
* **Шаг 2.** Отсортируем элементы каждого списка [сортировкой вставками](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%B2%D1%81%D1%82%D0%B0%D0%B2%D0%BA%D0%B0%D0%BC%D0%B8).
* **Шаг 3.** Объединим списки обратно в массив. Уменьшим ii. Если ii неотрицательно — вернемся к шагу 1
* Конец.

function shellSort(array a):

while interval < a.length /3 do:

interval = interval \* 3 + 1

while interval > 0 do:

for outer = interval; outer < a.length; outer ++ do:

valueToInsert = a[outer]

inner = outer;

while inner > interval -1 && a[inner - interval] >= valueToInsert do:

a[inner] = a[inner - interval]

inner = inner - interval

end while

a[inner] = valueToInsert

interval = (interval -1) /3;

Такой алгоритм отсортирует массив в худшем случае (в случае если входной массив отсортирован в обратном порядке) со сложностью O(n²), а в лучшем — O(n log² n).

Известно, что при удачном раскладе этот способ сортирует за O(n). Но, в целом, сортировка работает существенно медленнее чем, к примеру быстрая сортировка или сортировка слиянием. Средняя временная сложность зависит от того, какую последовательность брать для циклических итераций.Первоначально автор сортировки, Дональд Шелл, предложил ряд[n/4], [n/2], [n/8], ..., который давал скорость O(n2).

В течении последующих 50 лет, многие исследователи (среди которых - легендарный Роберт Седжвик) подбирали различные зависимости, постепенно улучшая результат. На данный момент таковым является ряд, предложенный в 2001 году Марсином Сиурой:701, 301, 132, 57, 23, 10, 4, 1.

Худшее время

O(*n*2)

Лучшее время

O(*n* log2 *n*)

Среднее время

зависит от выбранных шагов

Затраты памяти

О(n) всего, O(1) дополнительно

|  |  |  |
| --- | --- | --- |
| Сортировка вставками | Сортировка Шелла | Быстрая сортировка |
| Худший случай | O(*n*2) | O(*n*2) | O(*n*2) |
| Лучший случай | O(*n*) | O(*n* log *n*) | O(*n* log *n*) |
| Средний случай | O(*n*2) | Зависит от расстояния между элементами | O(*n* log *n*) |

**2.3.3** Сортировка деревом

Сортировка с помощью бинарного дерева  — 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, с файла, сокета или консоли) [6].

1. Построение двоичного дерева.

2. Сборка результирующего массива путём обхода узлов в необходимом порядке следования ключей.

инарное (двоичное) дерево поиска – это бинарное дерево, для которого выполняются следующие дополнительные условия (свойства дерева поиска):

оба поддерева – левое и правое, являются двоичными деревьями поиска;

у всех узлов левого поддерева произвольного узла X значения ключей данных меньше, чем значение ключа данных самого узла X;

у всех узлов правого поддерева произвольного узла X значения ключей данных не меньше, чем значение ключа данных узла X.

Данные в каждом узле должны обладать ключами, на которых определена операция сравнения меньше.

Для сортировки с помощью дерева исходная сортируемая последовательность представляется в виде структуры данных «дерево».

STRUCTURE BinaryTree

BinaryTree:LeftSubTree

Object:Node

BinaryTree:RightSubTree

PROCEDURE Insert(BinaryTree:searchTree, Object:item)

IF searchTree.Node IS NULL THEN

searchTree.Node =item

ELSE

IF item <= 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)

bintree binTreeConst(int [] t, int n)

int i

bintree bint

insert(bint,T[1])

for i = 2 to n do

insert(bint,T[i])

end for

return bint

The procedure insert will insert the elements as they are read from the array t. If the element to be inserted

is word and t represents the binary tree where to insert, the insertion is specified as follows:

Insert(bintree t, int word)

if t = nil then

t= construct(word,nil,nil)

else

if word < t.label then

Insert(t.lt,word)

else

Insert(t.rt,word)

end if

end if

construct(r,t1,t2) constructs a binary tree with root labelled r and with t1 as a left subtree and t2 as a right

subtree.

The pseudo-code of symmetric (infix) traversal is:

Infix(bintree t)

if t 6= nil then

Infix(t.lt)

display(T.label)

Infix(t.rt)

end if

Процедура добавления объекта в бинарное дерево имеет среднюю алгоритмическую сложность порядка O(log(n)). Соответственно, для n объектов сложность будет составлять O(n log(n)), что относит сортировку с помощью двоичного дерева к группе «быстрых сортировок». Однако, сложность добавления объекта в разбалансированное дерево может достигать O(n), что может привести к общей сложности порядка O(n²).

При физическом развёртывании древовидной структуры в памяти требуется не менее чем 4n ячеек дополнительной памяти (каждый узел должен содержать ссылки на элемент исходного массива, на родительский элемент, на левый и правый лист), однако, существуют способы уменьшить требуемую дополнительную память.

|  |  |  |
| --- | --- | --- |
| **Сложность по времени** | **Худшая** | **O**(**n2**) |
| **Средняя** | **O**(**n** log **n**) |
| **Лучшая** | **O**(**n** log **n**) |
| **Сложность по памяти** | **Дополнительная** | **Θ**(**n**) |

**2.4** Сортировка слиянием

Алгоритм сортировки слиянием был предложен праотцом современных компьютеров – Джоном фон Нейманом. Сам метод является устойчивым, т. е. он не меняет одинаковые по значению элементы в списке.

В основе сортировки слиянием лежит принцип «разделяй и властвуй». Список разделяется на равные или практически равные части, каждая из которых сортируется отдельно. После чего уже упорядоченные части сливаются воедино.

Несколько детально этот процесс можно расписать так:

массив рекурсивно разбивается пополам, и каждая из половин делиться до тех пор, пока размер очередного подмассива не станет равным единице;

далее выполняется операция алгоритма, называемая слиянием. Два единичных массива сливаются в общий результирующий массив, при этом из каждого выбирается меньший элемент (сортировка по возрастанию) и записывается в свободную левую ячейку результирующего массива. После чего из двух результирующих массивов собирается третий общий отсортированный массив, и так далее. В случае если один из массивов закончиться, элементы другого дописываются в собираемый массив;

в конце операции слияния, элементы перезаписываются из результирующего массива в исходный.

function mergesort(array a):

if length(a) ≤ 1

return a

else

middle = length(m) / 2

for each x in m up to middle

add x to left

for each x in m after middle

add x to right

left = mergesort(left)

right = mergesort(right)

result = merge(left, right)

return result

**Худшее время**‎: ‎O(n log2 n)

**Среднее время**‎: ‎O(n log2 n)

**Лучшее время**‎: ‎O(n log2 n)

**Затраты памяти**‎: ‎O(n) вспомогательных

Достоинства:

Работает даже на структурах данных последовательного доступа.

Хорошо сочетается с подкачкой и кэшированием памяти.

Неплохо работает в параллельном варианте: легко разбить задачи между процессорами поровну, но трудно сделать так, чтобы другие процессоры взяли на себя работу, в случае если один процессор задержится.

Не имеет «трудных» входных данных.

Устойчивая - сохраняет порядок равных элементов (принадлежащих одному классу эквивалентности по сравнению).

Недостатки:

На «почти отсортированных» массивах работает столь же долго, как на хаотичных. Существует вариант сортировки слиянием, который работает быстрее на частично отсортированных данных, но он требует дополнительной памяти, в дополнении ко временному буферу, который используется непосредственно для сортировки.

Требует дополнительной памяти по размеру исходного массива.

**3 РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА ДЛЯ ИССЛЕДОВАНИЯ АЛГОРИТМОВ СОРТИРОВКИ**

**3.1** Архитектура классов сортировки

Включает в себя (согласно ГОСТ 19.701-90) разработку:

- алгоритма;

- схемы и структуры программы;

- схемы работы системы;

- схемы данных;

- алгоритмов отдельных программных модулей или функций;

- спецификаций программной системы.

Пример оформления перечисления.

Целями обработки сигналов могут служить:

* извлечение информации о сигнале (амплитуда, фаза, частота, спектральные составляющие, временные соотношения);
* преобразование формата сигнала;
* сжатие данных;
* формирование сигналов обратной связи (управление промышленными процессами);
* выделение сигнала из шума (фильтрация, автокорреляция, свертка);
* выделение и сохранение сигнала в цифровом виде для последующей обработки.

Состав выполняемых функций:

а) получение сигнальных данных с микрофона мобильного устройства в режиме реального времени;

б) графическое отображение сигнала;

в) расчёт и отображение частотного спектра сигнала в реальном времени;

г) расчёт и отображение информации об уровне сигнала в реальном времени:

− извлечение информации о сигнале (амплитуда, фаза, частота, спектральные составляющие, временные соотношения);

− преобразование формата сигнала;

− сжатие данных;

д) расчёт и отображение сонограммы;

е) расчёт и отображение статистической информации;

ж) сохранение сигнала в файл;

и) реализация пользовательского интерфейса.

Приведенные алгоритмы будут реализованы с помощью нижеприведенных процедур и функций.

**3.2** Пользовательский интерфейс программы

Рис. 1 – Интерфейс программы

Реализация пользовательского интерфейса с использованием данных компонентов приведена на рисунке 1.

**3.3** Визуализация алгоритмов

**4 ТЕСТИРОВАНИЕ И АНАЛИЗ ПОЛУЧЕННЫХ РЕЗУЛЬТАТОВ**

Должны быть разработаны способы проверки правильности и работоспособности отдельных функций и программной системы в целом. Спланированы и проведены экспериментальные исследования, отражающие функционирование разработанной программной системы.

Таблица 3.1 – Результаты тестирования №1

|  |  |
| --- | --- |
| Входные данные | Результат обработки |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ**

Руководство пользователя системы должно содержать последовательность действий, выполняемых пользователем для успешной работы системы.

**ЗАКЛЮЧЕНИЕ**

В этом разделе необходимо перечислить основные результаты, характеризующие полноту и подытоживающие содержание курсовой работы.

Результаты следует излагать в форме констатации фактов, используя слова: «изучены», «исследованы», «сформулированы», «показано», «разработана», «предложена», «подготовлены», «изготовлена», «испытана» и т.п.

Текст перечислений должен быть кратким, ясным и содержать конкретные данные.

В данной курсовой работе рассмотрена

Также успешно реализован

Проведенное тестирование программы показывает корректность её работы. По мнению автора программы, основные задачи курсового проектирования выполнены

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

[1] Гук, М. Процессоры Pentium II, Pentium Pro и просто Pentium / М. Гук. – СПб. : Питер Ком, 1999. – 288 с.

[2] Кузелин, М. О. Современные семейства ПЛИС фирмы Xilinx : справ. пособие / М. О. Кузелин, Д. А. Кнышев, В. Ю. Зотов. – М. : Горячая линия-Телеком, 2004. – 440 с.

[3] Технические средства диагностирования : справочник / В.В. Клюев [и др.]. – М. : Машиностроение, 1989. – 672 с.

[4] Embedded Microcontrollers : Databook / Intel Corporation. – Santa Clara, Ca, 1994.

[5] Проектирование самотестируемых СБИС : монография. В 2 т. / В. Н. Ярмолик и др. – Минск : БГУИР, 2001.

[6] Микропроцессоры и микропроцессорные комплекты интегральных микросхем : справочник. В 2 т. / под ред. В. А. Шахнова. – М. : Радио и связь, 1988. – Т. 1. – 368 с.

[7] Берски, Д. Набор ЭСЛ-микросхем для быстродействующего RISC-процессора / Д. Берски // Электроника. – 1989. – №12. – С. 21 – 25.

[8] Аксенов, О. Ю. Методика формирования обучающих выборок для распознающей системы / О. Ю. Аксенов // VI Всероссийская науч.-техн. конференция «Нейроинформатика–2004» : сб. науч. тр. В 2 ч. / отв. ред. О. А. Мишулина. – М. : МИФИ, 2004. – С. 215 – 222. – (Научная сессия МИФИ–2004).

[9] Xilinx [Электронный ресурс]. – Электронные данные. – Режим доступа: <http://www.plis.ru/>.

[10] Mobile Intel® Pentium® Processor-M [Электронный ресурс] : Datasheet / Intel Corpocation. – Электронные данные. – Режим доступа: 25068604.pdf.

[11] Nokia+Компьютер [Электронный ресурс] : инструкции, программы, драйверы, игры, мелодии, картинки для Nokia. – М., 2004. – 1 компакт-диск (CD-R).

[12] Клепиков, В. Использование QNX Neutrino в системах автоматического управления для ответственных применений / В. Клепиков, Д. Подхватилин, Г. Шарапов // Средства и системы промышленной автоматизации [Электронный ресурс]. - 2008. - Режим доступа: http://www.swd.ru/index.php3?pid=794 Дата доступа: 15.04.08

[13] LabVIEW Real-Time LabVIEW реального времени // Средства и системы промышленной автоматизации [Электронный ресурс]. - 2008. - Режим доступа: http://www.asutp.ru/?p=600045&PHPSESSID=7ca23f6181d2e70b9b64b3263b d28d46 Дата доступа: 15.04.08

[14] Wale, К. VME помогает строить DSP-системы реального времени / Wale, К. // Real-Time Magazine - 2007. - №1. - С. 67-70.

[15] Малинецкий, Г.Г. Современные проблемы нелинейной динамики / Г.Г. Малинецкий, А.Б. Потапов – Москва: Эдиториал УРСС, 2000. – 336 с.

[16] DDK – Windows Driver Development Kit / Microsoft Corporation [Electronic resource]. – 2008 – Mode of access: [http://www.microsoft.com/whdc/ devtools/ddk/default.mspx](http://www.microsoft.com/whdc/%20devtools/ddk/default.mspx) – Date of access: 17.03.2008.

[17] Жданов, А. Замечания о выборе операционных систем при построении систем реального времени / А. Жданов, А. Латыев // PCWeek. – 2001. – №(271)1. – С. 22–29.

[18] Бранцевич, П.Ю. Модель системы диагностики технических объектов по вибрационным параметрам / П.Ю. Бранцевич // Наука и технологии на рубеже XXI века Материалы международной научно–технической конференции / Редкол. И.П. Филонова [и др.]. – Минск.: УП Технопринт, 2000. – С. 259–264.

[19] Бранцевич, П.Ю. Измерительно–вычислительная система распределенного сбора и централизованной обработки виброметрических данных / П.Ю. Бранцевич // Датчики и преобразователи информации систем измерения, контроля и управления. Сборник материалов 12–ой научно–технической конференции с участием зарубежных специалистов / Редкол. В.Н. Азарова. [ и др.]. Москва: МГИЭМ, 2000. – С. 170–171.

[20] Kramer, M.A. A rule–based approach to fault diagnosis using the signed directed graph / M.A. Kramer, B.L. Palowitch // AIChE Journal. – 2004. – Vol. 33, № 7, P. 1067–1078.

[21] Wenzel, H. Ambient Vibration Monitoring / H. Wenzel, D. Pichler. – John Wiley & Sons, 2005. – 295 p.

[22] Бранцевич, П.Ю. Система контроля и анализа технического состояния агрегатов и механизмов по вибрационным параметрам / П.Ю. Бранцевич, Э.И. Кульков, С.Ф. Костюк // Проблемы вибрации, виброналадки, вибромониторинга и диагностики оборудования электрических станций / Сборник докладов. Редкол.: Салимона А.В. [и др.]. – Москва, 2001. – с. 60–64.

[23] Mehr, A.F. A new approach to probabilistic risk analysis in concurrent and distributed design of aerospace systems. / A.F. Mehr, I.Y. Tumer. // ASME 2005 Internation Design Engineering Technical Conferences, sep. 24–28, Long Beach, California, USA. / 2005. – р. 167–189.

[24] ГОСТ Р 517721–2001.Аппаратура радиоэлектронная бытовая. Входные и выходные параметры и типы соединений. Технические требования [Текст]. – Введ. 2002–01–01. – М. : Изд-во стандартов, 2001. –27 с.

[25] ГОСТ 7. 53–2001.Издания. Международная стандартная нумерация книг [Текст]. – Взамен ГОСТ 7.53–86 ; введ. 2002–07–01. – Минск: Межгос. совет по стандартизации, метрологии и сертификации ; М.: Изд-во стандартов, cop. 2002. – 3 с.

**ПРИЛОЖЕНИЕ А**

**(обязательное)**

**Текст программного модуля**

using System;

using System.Text;

using System.Windows.Forms;

using System.Runtime.InteropServices;

using System.Diagnostics;

namespace Timbre

{

// Объявление класса делегатов на основе которых определены события для возврата порции собранных данных и, соответственно, событие окончания сбора

public delegate void DataReadetHandler(ushort NumberOfChannel, float[]

**ПРИЛОЖЕНИЕ Б**

**(обязательное)**

**Текст программного модуля**

public unsafe string SetWorkParameters()

{

csDriverLibrary.WADC\_PAR\_0 InitParameters = new csDriverLibrary.WADC\_PAR\_0();

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Обозначение | | | | Наименование | | | | Дополнительные сведения | | | |
|  | | | | Текстовые документы | | | |  | | | |
|  | | | |  | | | |  | | | |
| БГУИР КП 1–40 01 01 ПЗ | | | | Пояснительная записка | | | | 32 с. | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  |  |  |  |  | БГУИР КП 1-40 01 01  Д1 | | | | | | |
|  |  |  |  |  |
|  |  |  |  |  |
| Изм. | Л. | № докум. | Подп. | Дата | Ваша тема  Ведомость курсового  проекта |  | | | | Лист | Листов |
| Разраб. | | Киселев Н.И. |  | 08.05.15 | Т |  | |  | 32 | 32 |
| Пров. | | Болтак С.В. |  | 10.05.09 | Кафедра ПОИТ | | | | | |
|  | |  |  |  |