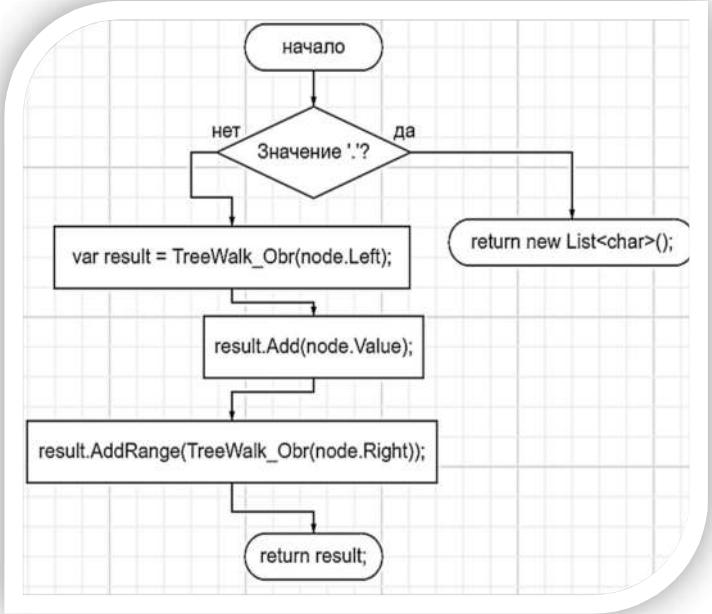
ИТМО

С.Е. Иванов

ПРИКЛАДНЫЕ АЛГОРИТМЫ НА ЯЗЫКЕ ООП C#



Санкт-Петербург
2022

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

УНИВЕРСИТЕТ ИТМО

С.Е. Иванов

ПРИКЛАДНЫЕ АЛГОРИТМЫ НА ЯЗЫКЕ ООП C#

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

РЕКОМЕНДОВАНО К ИСПОЛЬЗОВАНИЮ В УНИВЕРСИТЕТЕ ИТМО
по направлению подготовки 09.03.03 Прикладная информатика
в качестве Учебно-методическое пособие для реализации основных профессиональных образовательных программ высшего образования бакалавриата

Санкт-Петербург
2022

Иванов С.Е., Прикладные алгоритмы на языке ООП C#– СПб: Университет ИТМО, 2022. – 52 с.

Рецензент(ы):
Мельников Виталий Геннадьевич, доктор технических наук, профессор, Федеральное государственное бюджетное образовательное учреждение высшего образования «Санкт-Петербургский горный университет»;

Пособие включает лабораторные работы, выполняемые в рамках курса «Объектно-ориентированное программирование». Представлено описание, блок-схемы и реализация различных прикладных алгоритмов на языке ООП C#.

Пособие предназначено для бакалавров по направлению подготовки 09.03.03 Прикладная информатика по программе «Мобильные и сетевые технологии» и содержит материалы лабораторных работ по дисциплине «Объектно-ориентированное программирование».

Университет ИТМО – ведущий вуз России в области информационных и фотонных технологий, один из немногих российских вузов, получивших в 2009 году статус национального исследовательского университета. С 2013 года Университет ИТМО – участник программы повышения конкурентоспособности российских университетов среди ведущих мировых научно-образовательных центров, известной как проект «5 в 100». Цель Университета ИТМО – становление исследовательского университета мирового уровня, предпринимательского по типу, ориентированного на интернационализацию всех направлений деятельности.

Содержание

Введение

В методическом пособии представлены задачи для реализации средствами ООП C# различных прикладных алгоритмов, таких как алгоритм хеширования, генетический алгоритм, алгоритмы обхода деревьев, поисковые алгоритмы на графах, нахождения кратчайших путей и минимального остова на графах, метод «разделяй и властвуй», алгоритмы сортировки и поиска. Также рассмотрена задача оценки сложности алгоритмов. При выполнении работ обучающийся приобретает соответствующие компетенции для понимания схем работы алгоритмов, навыки программирования и умения реализовывать алгоритмы средствами ООП.

В предлагаемом пособии представлены лабораторные работы, выполняемые в рамках курса «Объектно-ориентированное программирование» для профессиональных образовательных программ высшего образования бакалавриата по направлению подготовки 09.03.03 Прикладная информатика.

Перед выполнением лабораторных работ обучающемуся необходимо изучить теоретический материал по рассматриваемому курсу.

Пособие содержит восемь лабораторных работ, включающих постановку задачи, состав отчета по работе, пример реализации алгоритма посредством ООП C# и контрольные вопросы. Также при выполнении лабораторной работы восемь обучающийся приобретает навыки оценки сложности алгоритмов.

После выполнения работы обучающийся предоставляет отчет, который защищает в семестре. В ходе защиты обучающийся поясняет свой отчет и отвечает на вопросы преподавателя. В конце пособия представлен список рекомендуемой литературы по рассмотренным темам.

Методические рекомендации для выполнения представленных в пособии лабораторных работ:

1. Ознакомиться с алгоритмом метода.
2. Разработать схему работы алгоритма.
3. Реализовать алгоритм средствами ООП C#.
4. Выполнить расчет примера с помощью разработанной программы.
5. Проверить результаты работы программы на примерах.

Шаблон отчета по лабораторной работе №____

«_____»

(название лабораторной работы)

1. Цель и задачи лабораторной работы:
2. Задание на лабораторную работу:
3. Результаты лабораторной работы:
4. Выводы:

Отчет по лабораторной работе представляется в электронном виде в формате, предусмотренном шаблоном отчета по лабораторной работе. Защита отчета проходит в форме доклада обучающегося с демонстрацией результатов и ответов на вопросы преподавателя.

Если оформление отчета и доклад обучающегося во время защиты соответствуют указанным требованиям, обучающийся получает максимальное количество баллов.

Основаниями для снижения количества баллов являются:

- небрежное выполнение,
- низкое качество оформления,
- замечания при ответе на контрольные вопросы преподавателя.

Отчет не может быть принят и подлежит доработке в случае:

- отсутствия необходимых разделов,
- отсутствия результатов выполнения.

При выполнении работ обучающийся закрепляет теоретические знания и получает практические навыки по реализации алгоритмов.

В результате выполнения работ обучающийся приобретает:

Знания: принципов построения алгоритмов; основных этапов решения задачи на компьютере; основ языка программирования высокого уровня C#; технологий работы с информационными системами, техническими средствами хранения информации, ее кодирование; типовых структур данных и способов их записи средствами языка высокого уровня.

Умения: принимать участие в разработке задания на программный продукт; выбирать способ представления данных, определять спецификации на отдельные классы, модули и подпрограммы; по математическому описанию задачи разрабатывать алгоритм работы программы; выбирать способ организации программы, на языке высокого уровня.

Навыки: выбирать язык программирования высокого уровня для реализации поставленной задачи; анализировать и алгоритмизировать задачи в различных областях знаний; выбирать технологии разработки для решения поставленных задач; выбирать способ представления данных и алгоритм решения задачи.

Для самостоятельной работы обучающимся рекомендуется предоставить в соответствии с планом СРО 91.2 часа в семестре на подготовку и выполнение лабораторных работ.

Лабораторная работа №1. Алгоритмы хеширования

ЦЕЛЬ РАБОТЫ

Изучить алгоритмы хеширования, реализовать алгоритм Рабина-Карпа поиска подстроки в строке с применением хеширования.

ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Алгоритмы хеширования применяют хеш-функцию для преобразования входных данных в выходную битовую строку заданной длины. В результате преобразования получается хеш или хеш-код. Хеш-функции применяют для поиска дублирующих данных, вычисления контрольных сумм, создания уникальных идентификаторов, хранения паролей, для электронной подписи и многих других задачах. В соответствии с правилом Дирихле нет строгого однозначного соответствия между хешем и исходными данными, поэтому возникают коллизии. Любая хорошая хеш-функция должна быстро выполняться и иметь минимальное число коллизий. Для разрешения коллизий разработаны различные методы: цепочек, метод открытой адресации. Хеширование применяется в алгоритме поиска подстроки в тексте, разработанном Майклом Рабином и Ричардом Карпом. Алгоритм Рабина-Карпа применяют для поиска плагиата и совпадений шаблонов одинаковой длины. Этот алгоритм основывается на хешировании строк. Дана строка S и текст T, состоящие из маленьких латинских букв. Требуется найти все вхождения строки S в текст T.

Алгоритм состоит из следующих шагов. Посчитаем хеш для строки S. Посчитаем значения хешей для всех префиксов строки T. Далее переберём все подстроки T длины |S| и каждую сравним с |S|.

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

1.Реализовать поиск одинаковых строк.
Дан список строк S[1..n], каждая длиной не более m символов. Требуется найти все повторяющиеся строки и разделить их на группы, чтобы в каждой группе были только одинаковые строки.

2. Реализовать алгоритм Рабина-Карпа поиска подстроки в строке за O (n).

ПРИМЕР ВЫПОЛНЕНИЯ РАБОТЫ

Этапы работы:

1. Реализовать поиск одинаковых строк

Дан список строк S[1..N], каждая длиной не более M символов. Необходимо найти все повторяющиеся строки и разделить их на группы, чтобы в каждой группе были только одинаковые строки.
Ниже представлен листинг для считывания строк

```
static void Main(string[] args)
{
    int m = 50;
    int n = 10;
    string[] stringArray = new string[n];
    for(int i = 0; i < 10; i++)
    {
        stringArray[i] = Console.ReadLine();
    }
```

Полиномиальным хешем этой строки называется число h = hash(s0..n-1) = s0p + p2s1 + p3s2 +… + pnsn-1, где p - некоторое натуральное число, a si - код i-ого символа строки s (он записывается s[i]).
Ниже представлен листинг для вычисления степеней p:

```
int p = 31;
int[] p_pow = new int[m];
p_pow[0] = 1;
for (int i = 1; i < m; i++)
{
    p_pow[i] = p_pow[i - 1] * p;
}
```

Ниже представлен листинг для вычисления хешей строк:

```
var hashs = new Dictionary<int, int>();
for(int i =0; i < n; i++)
{
    hashs[i] = 0;
    for (int j = 0; j < stringArray[i].Length; j++)
    {
        hashs[i] += (stringArray[i][j] - 'a'+1)*p_pow[j];
    }
```

Ниже представлен листинг для сортировки по хешам и вывод ответа:

```
hashs = hashs.OrderBy(pair => pair.Value).ToDictionary(pair => pair.Key, pair => pair.Value);
foreach (var item in hashs)
    Console.WriteLine(item.Key + " " + item.Value);
```

```
int n_groups = 0;
var previous_hash = hashs.First();
Console.WriteLine(previous_hash);
foreach(var hash in hashs)
{
    if (hash.Equals(hashs.First()) || hash.Value != previous_hash.Value)
    {
        n_groups++;
        Console.WriteLine();
        Console.Write("Group {0}: ",n_groups);
    }
    Console.Write(hash.Key+" ");
    previous_hash = hash;
}
```

Ниже представлен вывод результатов: хеши и список групп:

```
love
love
sun
sun
love
love
dive
wind
dive
dive
```

```
2 14124
3 14124
7 132920
6 170380
8 170380
9 170380
0 170574
1 170574
4 170574
5 170574
[2, 14124]
```

```
Group 1: 2 3
Group 2: 7
Group 3: 6 8 9
Group 4: 0 1 4 5
```

2. Реализовать алгоритм Рабина-Карпа поиска подстроки в строке за O(N).

Дана строка S и текст T, состоящие из маленьких латинских букв.
Требуется найти все вхождения строки S в текст T за время O (|S| + |T|).
Сначала так же считаем степени P. Затем хеши подстрок текста и хеш строки.

Ниже представлен листинг для вычисления степеней и хешей:

```
string s = Console.ReadLine();
string t = Console.ReadLine();
int p = 31;
int n = s.Length;
int m = t.Length;
int[] p_pow = new int[m];
p_pow[0] = 1;
for (int i = 1; i < m; i++)
{
    p_pow[i] = p_pow[i - 1] * p;
}
```

```
int[] hashs = new int[m];
for(int i = 0; i < m; i++)
{
    hashs[i] = (t[i] - 'a' + 1) * p_pow[i];
    if (i!=0)
    {
        hashs[i] += hashs[i - 1];
    }
}
int s_hash = 0;
for (int i = 0; i < n; i++)
{
    s_hash+= (s[i] - 'a' + 1) * p_pow[i];
}
```

Используя формулу $hash(s_{0..R}) = hash(s_{0..L-1}) + p^L hash(s_{L..R})$, найдем индексы, где в тексте появляется нужная строка.
Ниже представлен листинг для вывода ответа:

```
int cur_h;
for(int i = 0; (i + n-1) <m; i++)
{
    cur_h = hashs[i + n - 1];
    if (i != 0)
    {
        cur_h -= hashs[i - 1];
    }
    if (cur_h == s_hash * p_pow[i])
    {
        Console.WriteLine(i);
```

Ниже представлены результаты поиска строки в тексте:

```
hi
hinnnnhinnhi hi
0
6
10
13
```

Выводы:
В ходе выполнения работы изучены и реализованы средствами ООП C# алгоритмы хеширования.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие задачи решаются с помощью алгоритма хеширования?
2. Как решается проблема коллизий при хешировании?

3. Какие ограничения и условия на применение алгоритма хеширования?
4. Какие конструкции ООП C# применяются для реализации алгоритма хеширования?
5. Как оценивается сложность алгоритма хеширования?
6. Как можно оценить точность алгоритма хеширования?

Лабораторная работа №2. Алгоритм генетический

ЦЕЛЬ РАБОТЫ

Изучить генетический алгоритм и реализовать алгоритм средствами ООП для решения Диофантова уравнения**.**

ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Генетический алгоритм аналогичен процессу естественного отбора в природе и широко применяется в оптимизационных задачах.

Ниже представлена блок схема генетического алгоритма.

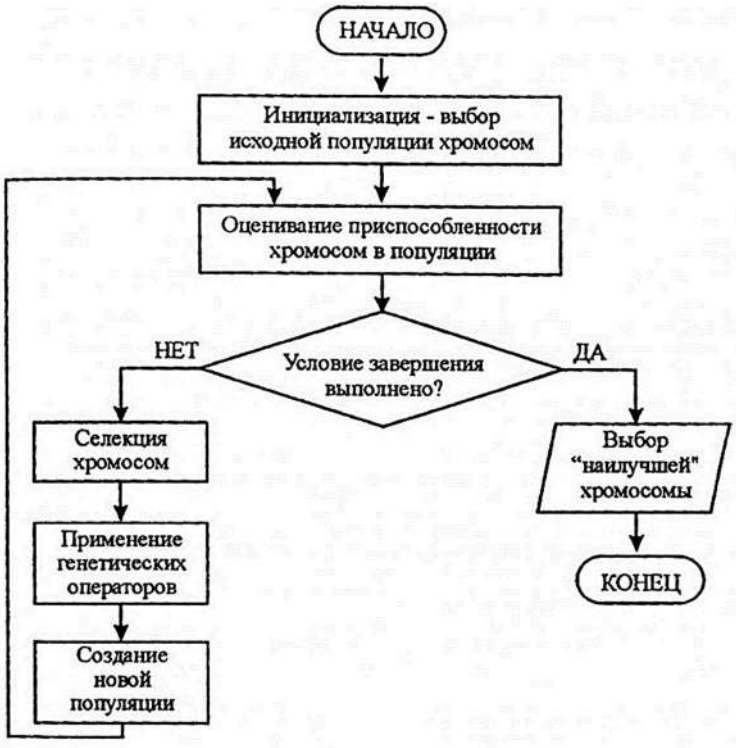


Рисунок 1 – Блок-схема генетического алгоритма

В алгоритме важную роль играют операторы кроссинговера, наследования, мутаций и селекции, которые подбираются под определенную задачу. Решение задачи кодируется вектором генов или генотипом. В алгоритме оценивается множество решений (поколение) посредством функции приспособленности или целевой функции в задаче оптимизации. Алгоритм итерационно повторяется до выполнения критерия остановки – нахождения оптимума, прохождения числа поколений или времени.

Алгоритм состоит из следующих шагов: задания функции приспособленности, генерации начального множества решений, в цикле пока критерий остановки не выполнен выполняются операторы кроссинговера, наследования, мутаций, селекции, для сознания нового множества решений. Алгоритм широко применяется для задач оптимизации.

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Реализовать средствами ООП генетический алгоритм для решения задачи: найти решение Диофантова (только целые решения) уравнения: a+2b+3c+4d=30, где a, b, c и d - некоторые положительные целые.

ПРИМЕР ВЫПОЛНЕНИЯ РАБОТЫ

В данном случае посредством генетического алгоритма реализуется решение Диофантова уравнения. Дано уравнение с целыми коэффициентами, определить целое решение.

Диофантово уравнение имеет вид:
D(a, b, c, d) = a + 2b + 3c + 4d = 30,
где a, b, c и d – некоторые положительные целые.

Должно выполняться условие 1 <= a, b, c, d <= 30.

Создается класс Generation, объектами которого будут поколения, а переменные в уравнении – поля класса.

Ex_res – ожидаемый результат, real_res – получаемый.

Ниже представлен листинг, показывающий поля и конструктор Поколения:

```
class Generation
{
    public int a;
    public int b;
    public int c;
    public int d;
    public static Random r = new Random();
    public int ex_res;
    public Func<int, int, int, int, int> f;
    private int? real_res;
```

```
public int RealResult
{
    get
    {
        if (real_res == null)
        {
            real_res = f(a, b, c, d);
        }
        return (int)real_res;
    }
}
Ссылок: 4
public Generation(Func<int, int, int, int, int> f, int a, int b, int c, int d, int ex_res)
{
    this.f = f;
    this.a = a;
    this.b = b;
    this.c = c;
    this.d = d;
    this.ex_res = ex_res;
}
```

Далее реализуется механизм наследования – в метод передаются два представителя поколения, после чего случайным образом определяется то, как их «ребенок» (новый объект Generation) унаследует хромосомы.

Ниже представлен листинг, реализующий наследование:

```
public static Generation NewGen(Generation p1, Generation p2)
{
    int x = r.Next(1, 4);
    switch (x)
    {
        case 1: return new Generation(p1.f, p1.a, p1.b, p2.c, p2.d, p2.ex_res);
        case 2: return new Generation(p1.f, p1.a, p2.b, p2.c, p2.d, p2.ex_res);
        case 3: return new Generation(p1.f, p1.a, p1.b, p1.c, p2.d, p2.ex_res);
        default: return null;
    }
}
```

Реализуется метод мутаций. Какая хромосома мутирует, определяется также случайно.

Ниже представлен листинг, реализующий мутацию:

```
public void Change()
{
    int imposter = r.Next(1, 5);
    switch (imposter)
    {
        case 1:
            a = r.Next(1, 30);
            break;
```

```
            case 2:
                b = r.Next(1, 30);
                break;
            case 3:
                c = r.Next(1, 30);
                break;
            case 4:
                d = r.Next(1, 30);
                break;
            default:
                break;
        }
    }
```

Также реализуется метод GetAc() для вычисления, насколько реальный результат отклонился от ожидаемого. Далее с помощью GetAc() в методе SrKoef() определяется средний коэффициент выживания поколения.
Ниже представлен листинг для расчета коэффициента выживания:

```
public int GetAc()
{
    return Math.Abs((int)(ex_res - RealResult));
}
Ссылок: 2
public static double SrKoef(List<Generation> generations)
{
    double sum = 0;
    foreach (Generation item in generations)
    {
        sum += item.GetAc();
    }
    return sum / generations.Count();
}
```

В методе Main создается поколение, в которое добавляется 5 объектов, после чего происходит поиск решения в цикле. Каждая итерация поколения выводится на экран.
Ниже представлен листинг метода Main:

```
class Program
{
    public static int res = 30;
    public static Func<int, int, int, int, int> f = (a, b, c, d) => a + 2 * b + 3 * c + 4 * d;
    public static Random r = new Random();
```

```
static void Main(string[] args)
{
    List<Generation> generations = new List<Generation>() ;
    for (int i = 0; i < 5; i++)
    {
        generations.Add(new Generation(f, r.Next(1, 30), r.Next(1, 30), r.Next(1, 30), r.Next(1, 30), res));
    }
    for (int i = 0; ; i++)
    {
        Console.WriteLine("Итерация " + (i + 1));
        foreach (var item in generations)
        {
            Console.Write("a = "+item.a+ " b = " + item.b+ " c = " + item.c+ " d = " + item.d+"\n");
        }
        Console.WriteLine("---");
```

Посредством сравнения коэффициентов выживания, поколения добавляются в список best с последующей проверкой соответствия необходимому результату.

Ниже представлен листинг для продолжения Main:

```
double koef1 = Generation.SrKoef(generations);
var best = generations
    .OrderBy(g => g.GetAc())
    .ToList();
if (best.Any(g => g.RealResult == res))
{
    Solution(best[0]);
    break;
}

generations.Clear();
generations.Add(Generation.NewGen(best[0], best[1]));
generations.Add(Generation.NewGen(best[1], best[0]));
generations.Add(Generation.NewGen(best[0], best[2]));
generations.Add(Generation.NewGen(best[2], best[0]));
generations.Add(Generation.NewGen(best[1], best[2]));
double koef2 = Generation.SrKoef(generations);
if (koef2 <= koef1)
{
    Random ran = new Random();
    int x = ran.Next(1, 5);
    for (int j = 0; j < x; j++)
    {
        generations[ran.Next(0, 5)].Change();
    }
}
```

Ниже представлен листинг для вывода решения:

```
public static void Solution(Generation g)
{
    Console.WriteLine("Решение: ");
    Console.Write("a = " + g.a + " b = " + g.b + " c = " + g.c + " d = " + g.d + "\n");
}
```

Ниже представлены результаты расчета примера:

```
Итерация 152
a = 2 b = 10 c = 5 d = 1
a = 2 b = 17 c = 2 d = 1
a = 2 b = 10 c = 2 d = 1
a = 18 b = 10 c = 2 d = 1
a = 2 b = 10 c = 2 d = 1
---
Итерация 153
a = 2 b = 10 c = 2 d = 1
a = 2 b = 9 c = 2 d = 1
a = 2 b = 10 c = 5 d = 19
a = 2 b = 6 c = 2 d = 1
a = 2 b = 10 c = 5 d = 1
---
Итерация 154
a = 2 b = 10 c = 2 d = 1
a = 2 b = 10 c = 2 d = 1
a = 2 b = 10 c = 2 d = 1
a = 6 b = 6 c = 2 d = 1
a = 10 b = 9 c = 2 d = 1
---
Решение:
a = 10 b = 9 c = 2 d = 1
```

Выводы:

В ходе выполнения работы реализован генетический алгоритм на языке C#, позволяющий решить Диофантово уравнение, выбрав наилучшее решение.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие конструкции ООП C# применяются для реализации генетического алгоритма?
2. Как оценивается сложность генетического алгоритма?
3. Какие задачи решаются с помощью генетического алгоритма?
4. Какие ограничения и условия на применение алгоритма?
5. Какие определяются операторы кроссинговера, наследования, мутаций и селекции?
6. Как можно оценить точность генетического алгоритма?

Лабораторная работа №3. Алгоритмы с бинарными деревьями

ЦЕЛЬ РАБОТЫ

Изучить алгоритмы работы с деревьями. Реализовать средствами ООП дерево и рекурсивные функции, выполняющие обход дерева в прямом, обратном, концевом порядке. Вычислить значение выражения, заданного деревом.

ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Алгоритмы поиска по дереву или обхода дерева широко применяются к двоичным деревьям. Различают алгоритмы обхода дерева в глубину и ширину. Для обхода в глубину применяют прямой порядок, центрированный и обратный.
Для поиска в глубину двоичного дерева обработка вершин выполняется рекурсивно в различном порядке: вначале рекурсивный обход левого поддерева (L), потом правого (R), далее просматривается корень (N). Представленный порядок обхода слева – направо можно менять: прямой (NLR), центрированный (LNR), обратный обход (LRN). Центрированный обход двоичного дерева получает данные в отсортированном порядке.

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

1. Реализовать средствами ООП дерево и рекурсивные функции, выполняющие обход дерева в прямом, обратном, концевом порядке. Выполнить расчет примера.
   Обход дерева в прямом порядке a b d e c f.
   Обход дерева в обратном порядке d b e a c f.
   Обход дерева в концевом порядке d e b f c a.

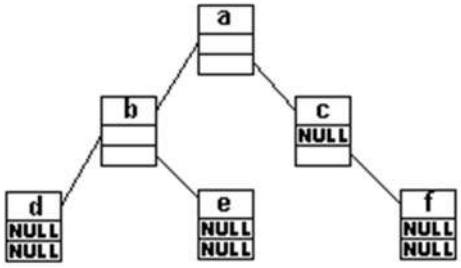


Рисунок 2 – Пример дерева

2. Вычислить значение выражения, заданного деревом. Вычисление выполнять в порядке концевого обхода.

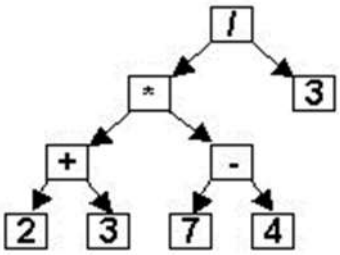


Рисунок 3 – Пример выражения в дереве

ПРИМЕР ВЫПОЛНЕНИЯ РАБОТЫ

Реализуется класс Tree – узел дерева. Его полями являются левый и правый узлы-потомки, значение, хранимое в узле, булева переменная flag (для последующего использования в функции добавления узла и сообщающая о том, что узел добавлен в нужное место), список значений узлов для обхода и отдельно корень дерева.
Ниже представлен листинг для класса Tree:

```
public class Tree
{
    public Tree Left;
    public Tree Right;
    public char Value;
    public bool flag;
    public static List<char> list = new List<char>();
    public static Tree Node;
    ссылка: 1
    public Tree()
    {
        Node = null;
    }
    Ссылок: 3
    public Tree(Tree left, Tree right, char val)
    {
        Value = val;
        Left = left;
        Right = right;
    }
```

В классе также реализуется метод Add, добавляющий узлы таким образом, что, пока не встретится значение узла ‘.’, добавление будет происходить слева, и только потом – справа. Когда и там, и там дошли до точек, поднимаемся на уровень выше и так до конца.

Ниже представлен листинг для добавления узлов:

```
public void Add(ref Tree node, char value)
{
    if (node == null)
    {
        node = new Tree(null, null, value);
        flag = false;
    }
    else if (node.Value != '.' && flag == false)
    {
        if (node.Left != null)
        {
            Add(ref node.Left, value);
            if (flag == false)
            {
                if (node.Right != null)
                {
                    Add(ref node.Right, value);
                }
                else
                {
                    node.Right = new Tree(null, null, value);
                    flag = true;
                }
            }
        }
        else
        {
            node.Left = new Tree(null, null, value);
            flag = true;
        }
    }
}
```

В методе Main заполнение дерева выглядит следующим образом.
Ниже представлен листинг для заполнения дерева:

```
Tree t = new Tree();
List<char> chars = new List<char>() { 'a', 'b', 'd', '.','.','e','.','.','c','.','f','.','.'};
for (int i = 0; i < chars.Count; i++)
{
    t.flag = false;
    t.Add(ref Tree.Node, chars[i]);
}
```

Дополнительный метод PrintTree служит для более наглядного убеждения в том, что дерево построено верно.
Ниже представлен листинг для метода PrintTree:

```
public void PrintTree(Tree node)
{
    if (node == null) return;
    if (node.Value != '.')
    {
        Console.WriteLine("значение " + node.Value + " левый " + node.Left.Value + " правый " + node.Right.Value);
    }
    PrintTree(node.Left);
    PrintTree(node.Right);
}
```

Для реализации прямого обхода, а именно: корень – левое поддерево – правое поддерево создан метод TreeWalk_Pr, работающий рекурсивно. Значения в нужном порядке добавляются в список list.

Ниже представлен листинг для прямого обхода:

```
public static void TreeWalk_Pr(Tree node)
{
    if (node.Value != '.')
    {
        list.Add(node.Value);
        TreeWalk_Pr(node.Left);
        TreeWalk_Pr(node.Right);
    }
}
```

TreeWalk_Obr реализует обход: левое поддерево – корень – правое поддерево.

Ниже представлен листинг для обратного обхода:

```
public static List<char> TreeWalk_Obr(Tree node)
{
    if (node.Value == '.') return new List<char>();
    var result = TreeWalk_Obr(node.Left);
    result.Add(node.Value);
    result.AddRange(TreeWalk_Obr(node.Right));
    return result;
}
```

Обход в концевом порядке осуществляет метод TreeWalk_K.

Ниже представлен листинг для концевого обхода: левое поддерево – правое поддерево – корень.

```
public static void TreeWalk_K(Tree node)
{
    if (node.Value != '.')
    {
        TreeWalk_K(node.Left);
        TreeWalk_K(node.Right);
        list.Add(node.Value);
    }
```

Ниже представлены блок-схемы прямого, обратного и концевого обходов.

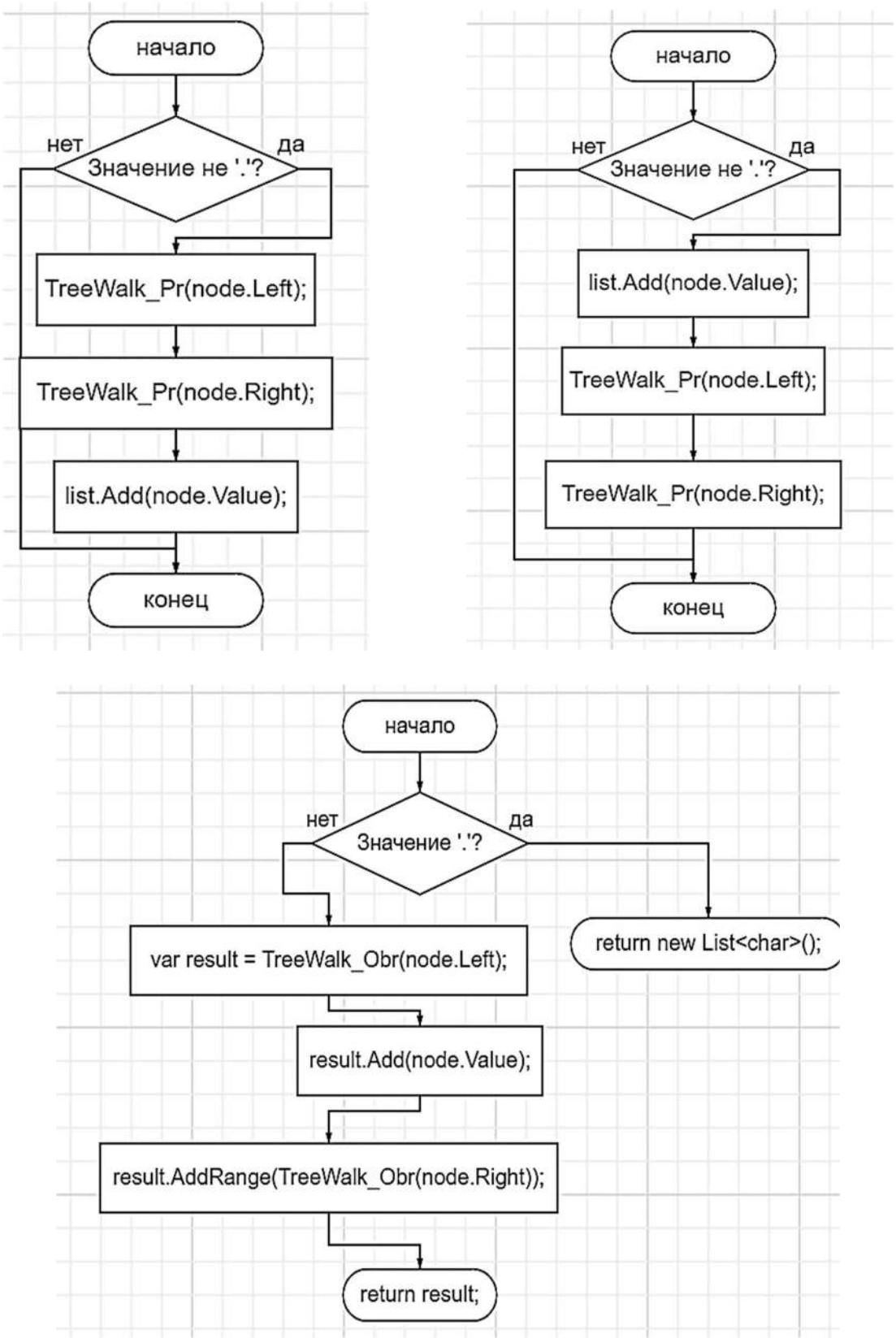


Рисунок 4 – Блок-схемы прямого, обратного и концевого обходов

Ниже представлен листинг для вызова методов:

```
t.PrintTree(Tree.Node);
Console.WriteLine("Прямой порядок");
Tree.TreeWalk_Pr(Tree.Node);
foreach (var item in Tree.list)
{
    Console.Write(item + " ");
}
Tree.list.Clear();
Console.WriteLine() ;
Console.WriteLine("Обратный порядок");
Tree.list = Tree.TreeWalk_Obr(Tree.Node);
foreach (var item in Tree.list)
{
    Console.Write(item+" ");
}
Tree.list.Clear();
Console.WriteLine();
Console.WriteLine("Концевой порядок");
Tree.TreeWalk_K(Tree.Node);
foreach (var item in Tree.list)
{
    Console.Write(item + " ");
}
```

Ниже представлен результат расчета примера:

```
значение a левый b правый c
значение b левый d правый e
значение d левый . правый .
значение e левый . правый .
значение c левый . правый f
значение f левый . правый .
Прямой порядок
a b d e c f
Обратный порядок
d b e a c f
Концевой порядок
d e b f c a
```

Далее реализуется расчет заданного деревом выражения.
Сначала дерево обходится в концевом порядке и создается упорядоченный список значений узлов. Далее вызывается метод CalcTree.

Ниже представлен листинг для вычисления выражения:

```
public static int CalcTree(Tree node)
{
    if (char.IsDigit(node.Value)) return node.Value;
    int num1, num2;
    num1 = CalcTree(node.Left);
    num2 = CalcTree(node.Right);
    switch (node.Value)
    {
        case '+': return num1 + num2;
        case '-': return num1 - num2;
        case '*': return num1 * num2;
        case '/': return num1 / num2;
        default: return 0;
    }
}
```

Если рассматриваемое значение узла – число, то оно возвращается. Для символов-операций рекурсивно вычисляются операнды-поддеревья, после чего возвращается необходимый для данной операции результат.
Ниже представлен листинг для метода Main:

```
Tree t = new Tree();
List<char> chars = new List<char>() { '/', '*', '+', '2', '.', '.', '3', '.', '.', '-', '7',
for (int i = 0; i < chars.Count; i++)
{
    t.flag = false;
    t.Add(ref Tree.Node, chars[i]);
}
Console.WriteLine("Концевой порядок");
Tree.TreeWalk_K(Tree.Node);
foreach (var item in Tree.list)
{
    Console.Write(item + " ");
}
Console.Write("Выражение равно ");
Console.Write(Tree.CalcTree(Tree.Node));
```

Ниже представлен результат расчет примера:

```
Концевой порядок
2 3 + 7 4 - * 3 / Выражение равно 5
```

Ниже представлена блок-схема метода вычисления выражения по дереву:

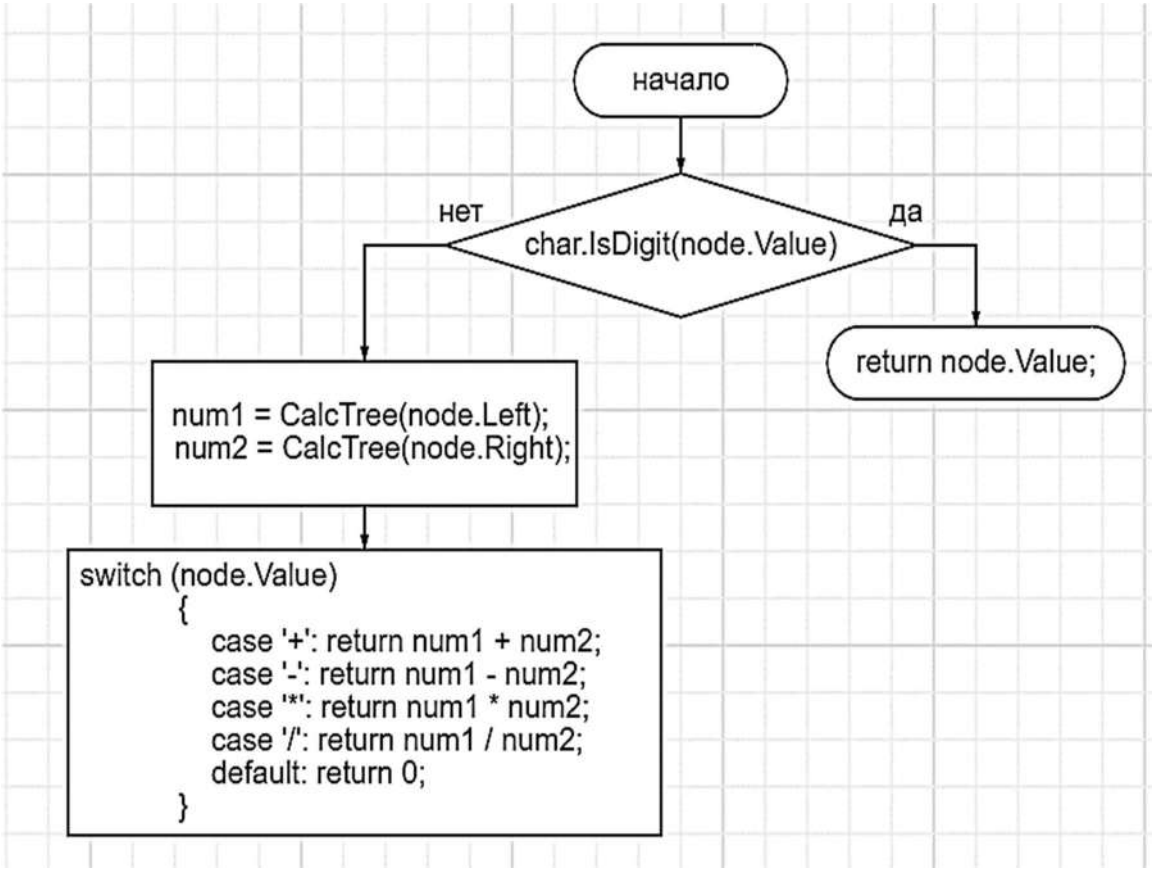


Рисунок 5 – Блок-схема метода вычисления выражения по дереву

Выводы:

В ходе выполнения работы изучены и реализованы такие алгоритмы работы с деревьями, как прямой, обратный и концевой обход, а также алгоритм расчета выражения, заданного деревом.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие ограничения и условия на применения алгоритма обхода?
2. К какому классу сложности относится алгоритм обхода?
3. Какие конструкции ООП C# применяются для реализации прямого, обратного и концевого обхода?
4. Как оценивается сложность алгоритма обхода?
5. Какие задачи решаются с помощью алгоритма обхода?
6. Как можно оценить точность алгоритма обхода?
7. Какой обход дерева применяется для вычисления выражения?

Лабораторная работа №4. Алгоритмы на графах

ЦЕЛЬ РАБОТЫ

Научиться реализовывать алгоритмы на графах. Реализовать средствами ООП поиск в глубину и ширину на графе. Реализовать средствами ООП алгоритм нахождение кратчайших путей (Алгоритм Дейкстры). Реализовать алгоритм Крускала.

ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Алгоритмы поиска кратчайшего пути на графе выполняют минимизацию суммы весов рёбер всего пути. Данный алгоритм имеет прикладное значение в GPS-навигаторах.

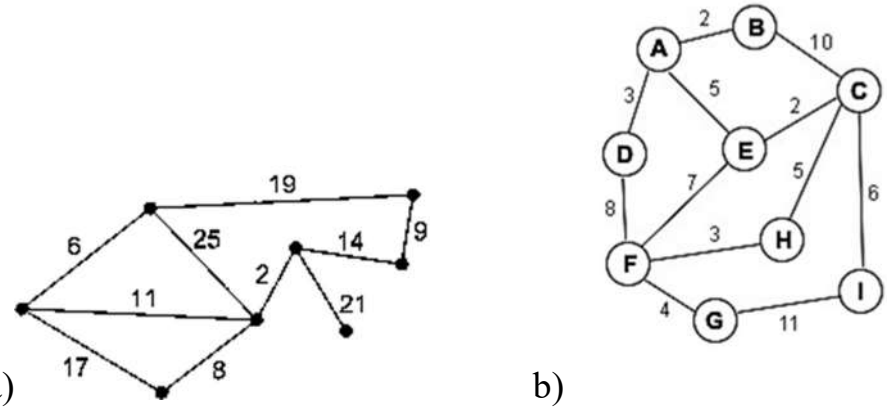
Алгоритм Дейкстры определяет кратчайшие пути от одной из вершин графа до всех остальных и работает только с положительными весами ребер. Алгоритм широко применяется в протоколах маршрутизации OSPF и IS-IS.

Алгоритм Дейкстры завершается, когда все вершины посещены. До тех пор выбирается вершина с минимальной меткой и рассматриваются всевозможные маршруты, для которых выбранная вершина будет предпоследним пунктом. Для каждого не посещённого соседа вершины определяется новая длина пути как сумма значений текущей метки и длины ребра, соединяющего вершину с этим соседом. Значение метки заменяется значением новой длины, если оно меньше. После рассмотрения вершина отмечается как посещенная и шаги алгоритма повторяются.

Для построения минимального остовного дерева неориентированного графа применяют алгоритм Крускала. При построении остовного дерева минимального веса сначала удаляются все ребра и добавляются ребра с минимальным весом при условии не образования замкнутого цикла.

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

1. Реализовать средствами ООП поиск в глубину и ширину на графе. Выполнить расчет примера.
2. Реализовать средствами ООП алгоритм нахождения кратчайших путей из одного источника (Алгоритм Дейкстры). Выполнить расчет примера.
3. Найти минимальный остов неориентированного взвешенного графа (Алгоритм Крускала), представленного ниже.



a) b)

Рисунок 7 – Неориентированный взвешенный графа для алгоритма Крускала.

ПРИМЕР ВЫПОЛНЕНИЯ РАБОТЫ

1.Реализуется поиск в глубину и ширину на графе.
Идея поиска в глубину заключается в том, что мы двигаемся от начальной вершины (точки, места) в определенном направлении (по определенному пути) до тех пор, пока не достигнем конца пути или пункта назначения (искомой вершины). Если мы достигли конца пути, но он не является пунктом назначения, то мы возвращаемся назад (к точке разветвления или расхождения путей) и идем по другому маршруту.
Создается программа, реализующая данный алгоритм. В полях класса n – число вершин графа, список used – список пройденных вершин, массив массивов g – матрица связей между вершинами.
Ниже представлен листинг для полей класса:

```
class Program
{
    static Random rand = new Random();
    static int n = 6;
    static List<int> used = new List<int>();
    static int[][] g = new int[6][];
```

В методе Main заполняется матрица связей. Алгоритм организован так, что между элементами g[i][j] и g[j][i] будет одинаковая связь 0 или 1 – то есть по условию граф является неориентированным.

Ниже представлен листинг создания матрицы связей:

```
for (int i = 0; i < n; i++)
{
    g[i] = new int[n];
}
for (int i = 0; i < n; i++)
{
    Console.Write("\n(" + (i) + ") вершина -->[");
    for (int j = i; j < n; j++)
    {
        g[i][j] = rand.Next(0, 2);
        g[j][i] = g[i][j];
    }
    foreach (var item in g[i])
    {
        Console.Write(item);
    }
    Console.Write("]\n");
}
```

Далее в методе Main реализуется вызов метода Method, который и реализует поиск в глубину. Принимая номер вершины, он начинает обходить смежные с ней вершины. Для каждой из них сразу же вызывается свой Method, чтобы поиск был именно в глубину. Когда все вершины пройдены, на экран выводится порядок их прохождения – список used.
Ниже представлен листинг для метода обхода в глубину:

```
Method(0);
Console.WriteLine("Вершины были пройдены в следующем порядке:");
foreach (var item in used)
{
    Console.Write(item + " ");
}
public static void Method(int i)
{
    used.Add(i);
    for (int j = 0; j < n; j++)
    {
        if (g[i][j] == 1 && !used.Contains(j))
        {
            Method(j);
        }
    }
}
```

Для алгоритма обхода в ширину в Method происходит добавление каждой вершины, смежной с данной, в очередь. Вызывается Method последовательно для

вершин в очереди. Вершина, для которой подошла очередь, из очереди удаляется. Также осуществляется вывод на экран текущего вида очереди.
Ниже представлен листинг для метода обхода в ширину:

```
public static void Method(int i)
{
    used.Add(i);
    for (int j = 0; j < g[i].Length; j++)
    {
        if (g[i][j] == 1 && !used.Contains(j) && !queue.Contains(j))
        {
            queue.Enqueue(j);
        }
    }
    foreach (var item in queue)
    {
        Console.Write(item+" ");
    }
    Console.WriteLine();

    if (queue.Count != 0)
    {
        Method(queue.Dequeue());
    }
}
```

Ниже представлена блок-схема метода обхода в глубину и в ширину:

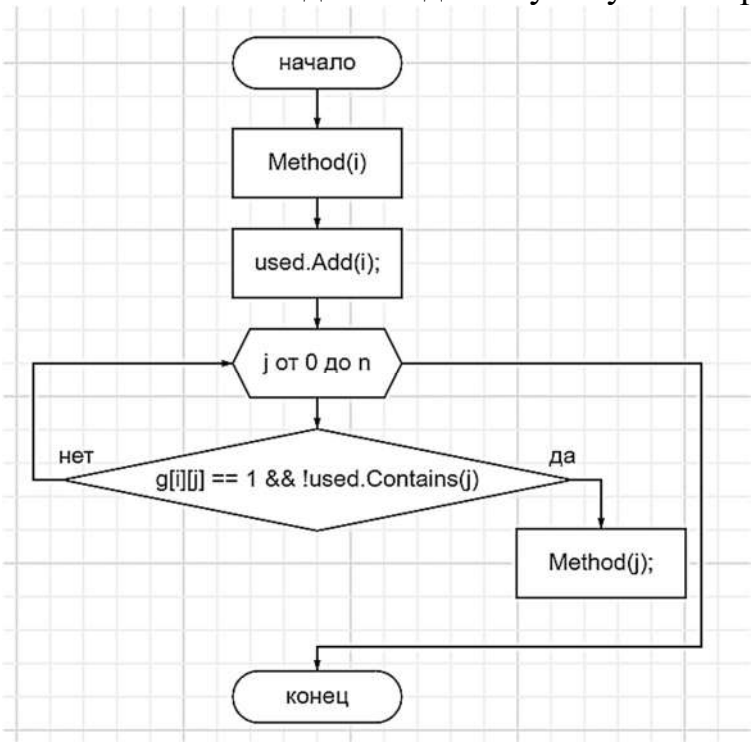


Рисунок 8 – Блок-схема метода обхода в глубину

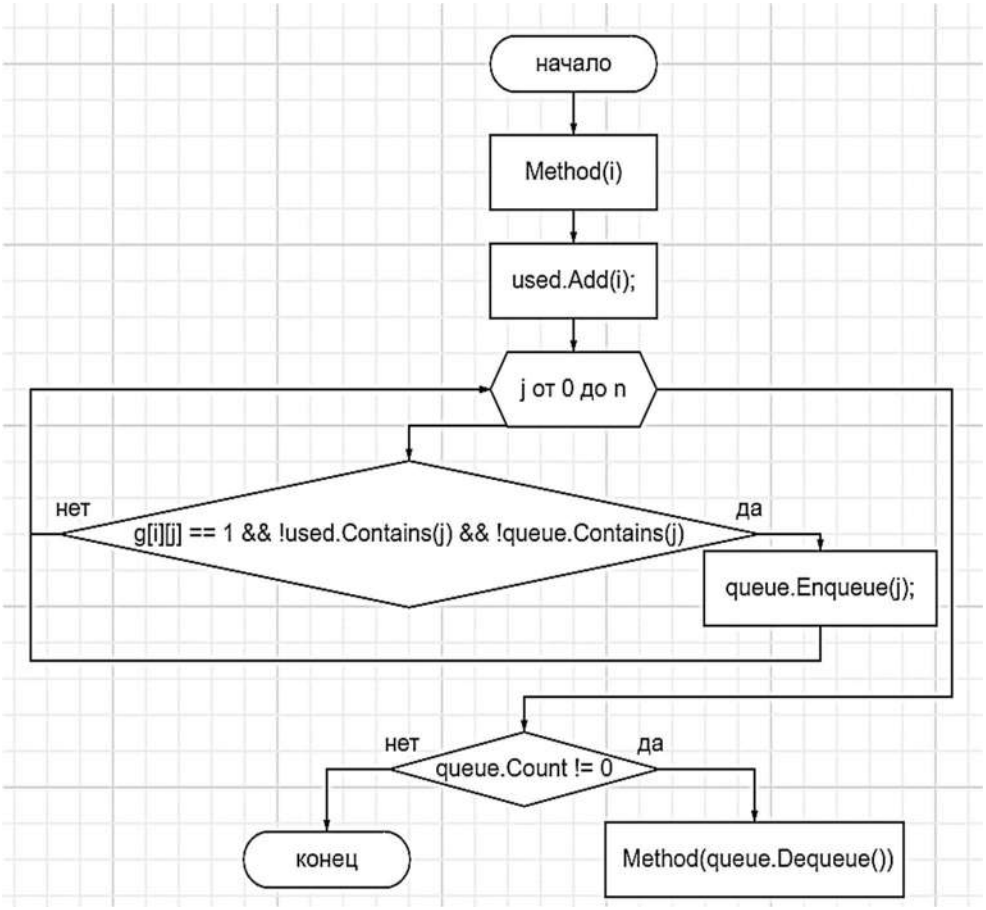


Рисунок 9 – Блок-схема метода обхода в ширину

Ниже представлен результат расчета примера алгоритма обхода в глубину:

```
(0) вершина -->[100111]

(1) вершина -->[011100]

(2) вершина -->[010101]

(3) вершина -->[111100]

(4) вершина -->[100011]

(5) вершина -->[101011]
Вершины были пройдены в следующем порядке:
0 3 1 2 5 4
```

Ниже представлен результат расчета примера алгоритма обхода в ширину:

```
(0) вершина -->[000101]

(1) вершина -->[011111]

(2) вершина -->[010101]

(3) вершина -->[111000]

(4) вершина -->[010000]

(5) вершина -->[111000]
Вид очереди:
3 5
5 1 2
1 2
2 4
4

Вершины были пройдены в следующем порядке:
0 3 5 1 2 4
```

Реализуется алгоритм нахождения кратчайших путей из одного источника (Алгоритм Дейкстры).

Для данного алгоритма создается три массива:
1) массив a, такой что a[i]=1, если вершина уже рассмотрена, и a[i]=0, если нет.
2) массив b, такой что b[i] – длина текущего кратчайшего пути из заданной вершины x в вершину i;
3) массив c, такой что c[i] – номер вершины, из которой нужно идти в вершину i в текущем кратчайшем пути.
W – массив массивов, представляющий собой матрицу связей, заполняется вручную.

Алгоритм реализует метод Method.

Ниже представлен листинг для метода Main:

```
for (int i = 0; i < n; i++)
{
    a[i] = 0;
    c[i] = 0;
    b[i] = 10000;
}
b[0] = 0;
```

```
Method(0);
for (int i = 1; i < n; i++)
{
    Console.WriteLine("Путь до вершины "+i+" от вершины 0:");
    Console.Write(i);
    int z = i;
    do
    {
        Console.Write(" <- " + c[z]);
        z = c[z];
    }
    while (z != 0);
    Console.WriteLine();
}
```

Ниже представлен листинг для алгоритма Дейкстры:

```
public static void Method(int i)
{
    int min = 1000000;
    int min_j = 0;
    for (int j = 0; j < n; j++)
    {
        if (a[j] == 0 && W[i, j] != 0)
        {
            if (b[j]> (b[i] + W[i, j]))
            {
                c[j] = i;
                b[j] = b[i] + W[i, j];
            }
            if (b[j] < min)
            {
                min = b[j];
                min_j = j;
            }
        }
    }
    a[i] = 1;
    if (min_j != 0)
    {
        Method(min_j);
    }
```

Ниже представлена блок-схема алгоритма Дейкстры:

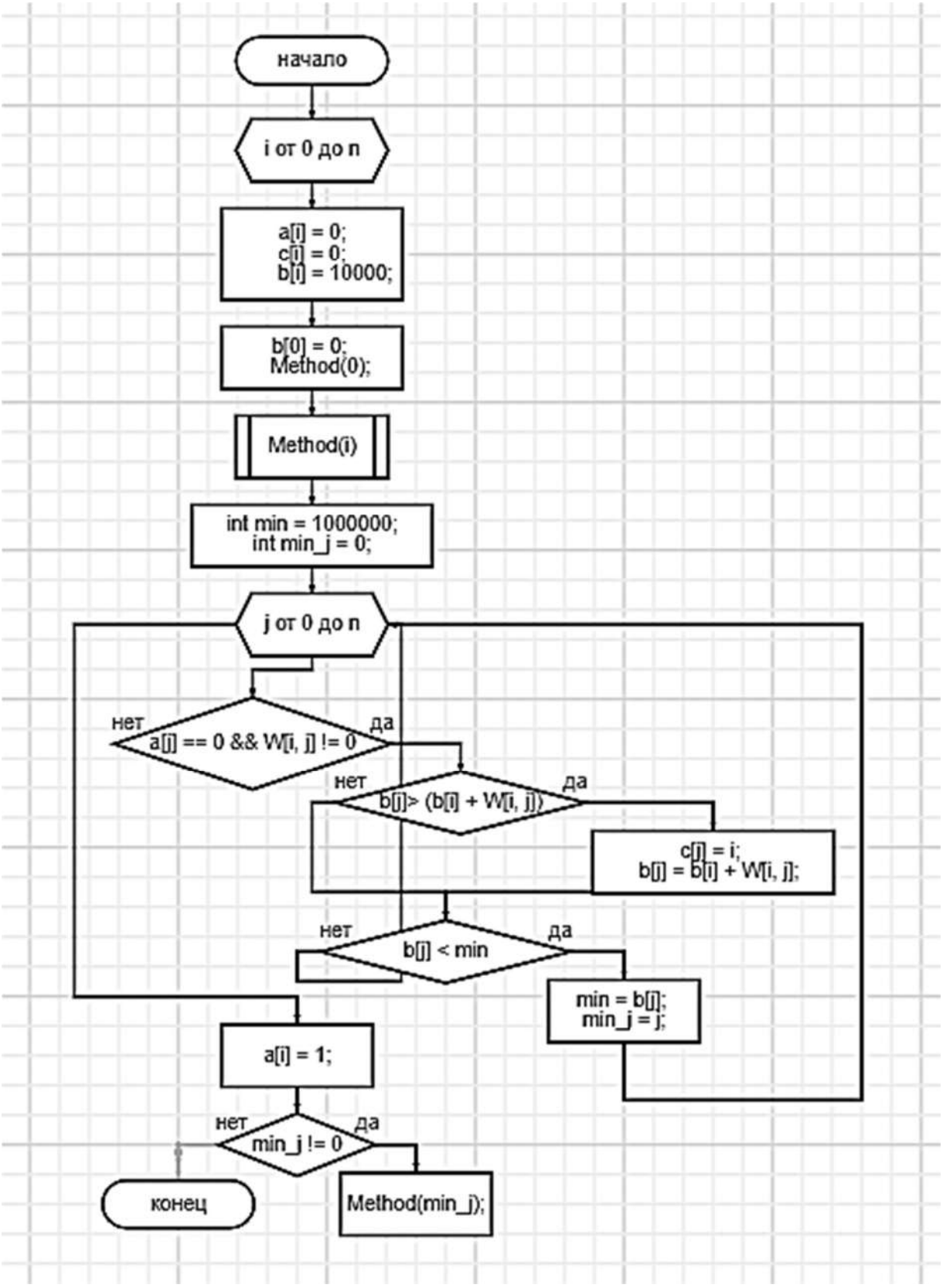


Рисунок 6 – Блок-схема алгоритма Дейкстры

Ниже представлен результат расчет примера алгоритма Дейкстры:

```
Путь до вершины 1 от вершины 0:
1 <- 0
Путь до вершины 2 от вершины 0:
2 <- 0
Путь до вершины 3 от вершины 0:
3 <- 2 <- 0
Путь до вершины 4 от вершины 0:
4 <- 5 <- 2 <- 0
Путь до вершины 5 от вершины 0:
5 <- 2 <- 0
```

3. Реализация алгоритма нахождения минимального остова неориентированного взвешенного графа (Алгоритм Крускала).
Заполняется матрица связей, вместе с этим создаются объекты класса Edge – ребра между вершинами, которые добавляется в список ребер.
Ниже представлен листинг для создания матрицы связи и ребер:

```
for (int i = 0; i < n; i++)
{
    Console.Write("\n(" + (i) + ") вершина -->[");
    for (int j = i; j < n; j++)
    {
        {
            g[i][j] = rand.Next(0, 2);
            if (g[i][j] == 1)
            {
                g[i][j] = rand.Next(1, 21);
                Edge edge = new Edge(i, j, g[i][j]);
                Edge.reb.Add(edge);
            }
            g[j][i] = g[i][j];
        }
    }
    foreach (var item in g[i])
    {
        Console.Write(item +"; ");
    }
    Console.Write("]\n");
}
```

Ниже представлен листинг для класса Edge:

```
public class Edge
{
    Ссылок: 8
    public int I { get; set; }
    Ссылок: 8
    public int J { get; set; }
    Ссылок: 8
    public int Weight { get; set; }
    public static List<Edge> reb = new List<Edge>();
    public static List<HashSet<int>> toplist = new List<HashSet<int>>();
    ссылка: 1
    public Edge(int i, int j, int weight)
    {
        I = i;
        J = j;
        Weight = weight;
    }
```

Класс ребер содержит следующие свойства: первая вершина, вторая и вес. Также имеется список ребер и список подмножеств графа, не соединенных друг с другом. В методе Main далее ребра сортируются по весу, после чего последовательно для каждого ребра вызывается метод Edge_Check.

Ниже представлен листинг для сортировки ребер, вызова метода, вывода результатов:

```
Console.WriteLine("Отсортированный список ребер:");
Edge.reb.Sort((a, b) => a.Weight.CompareTo(b.Weight));
foreach (var item in Edge.reb)
{
    Console.Write(item.Weight + " ");
}
foreach (var item in Edge.reb)
{
    item.Edge_Check();
}
Console.WriteLine();
foreach (var item in Edge.reb)
{
    if (item.Weight != -1)
    {
        Console.WriteLine("ребро между вершинами "+item.I+" и "+item.J+" вес"+item.Weight);
    }
}
```

Метод Edge_Check() работает следующим образом: Последовательно в цикле for проходятся все подмножества графа с целью поиска в них вершин данного ребра. Если найдена вершина I или J, устанавливаются true значения булевых переменных

flag_i или flag_j соответственно. Однако подмножество не может содержать в себе сразу две вершины – в таком случае значение веса ребра станет равным -1, то есть его не будет в остове.

Переменная n_top хранит в себе номер подмножества, в котором находится первая найденная вершина, чтобы, в случае нахождения второй вершины в другом подмножестве, они могли объединиться в одно и цикл бы прекратился.
Если цикл закончен, а найдена только одна вершина – вторая добавляется в подмножество первой. Если ни одна не найдена – создается новое подмножество.

Ниже представлен листинг для метода Edge_Check:

```
public void Edge_Check()
{
    bool flag_i = false;
    bool flag_j = false;
    int n_top = -1;
    for (int n = 0; n < toplist.Count;n++)
    {
        if (toplist[n].Contains(this.I) && !toplist[n].Contains(this.J))
        {
            flag_i = true;
            if (n_top != -1)
            {
                toplist[n].UnionWith(toplist[n_top]);
                toplist[n_top].Clear();
                break;
            }
            else
            {
                n_top = n;
            }
        }
else if (!toplist[n].Contains(this.I) && toplist[n].Contains(this.J))
{
    flag_j = true;
    if (n_top != -1)
    {
        toplist[n].UnionWith(toplist[n_top]);
        toplist[n_top].Clear();
        break;
    }
```

```
    else
    {
        n_top = n;
    }
}
else if (toplist[n].Contains(this.I) && toplist[n].Contains(this.J))
{
    flag_i = true;
    flag_j = true;
    this.Weight = -1;
    break;
}
```

Ниже представлен листинг для продолжения метода:

```
if (flag_i == false && flag_j == false)
{
    if (this.I != this.J)
    {
        toplist.Add(new HashSet<int>() { this.I, this.J });
    }
    else
    {
        this.Weight = -1;
    }
}
else if (flag_i == true && flag_j == false)
{
    toplist[n_top].Add(this.J);
}
else if (flag_i == false && flag_j == true)
{
    toplist[n_top].Add(this.I);
}
```

Когда цикл закончен, все вершины объединены в одно подмножество.
Ниже представлена блок-схема алгоритма Крускала:

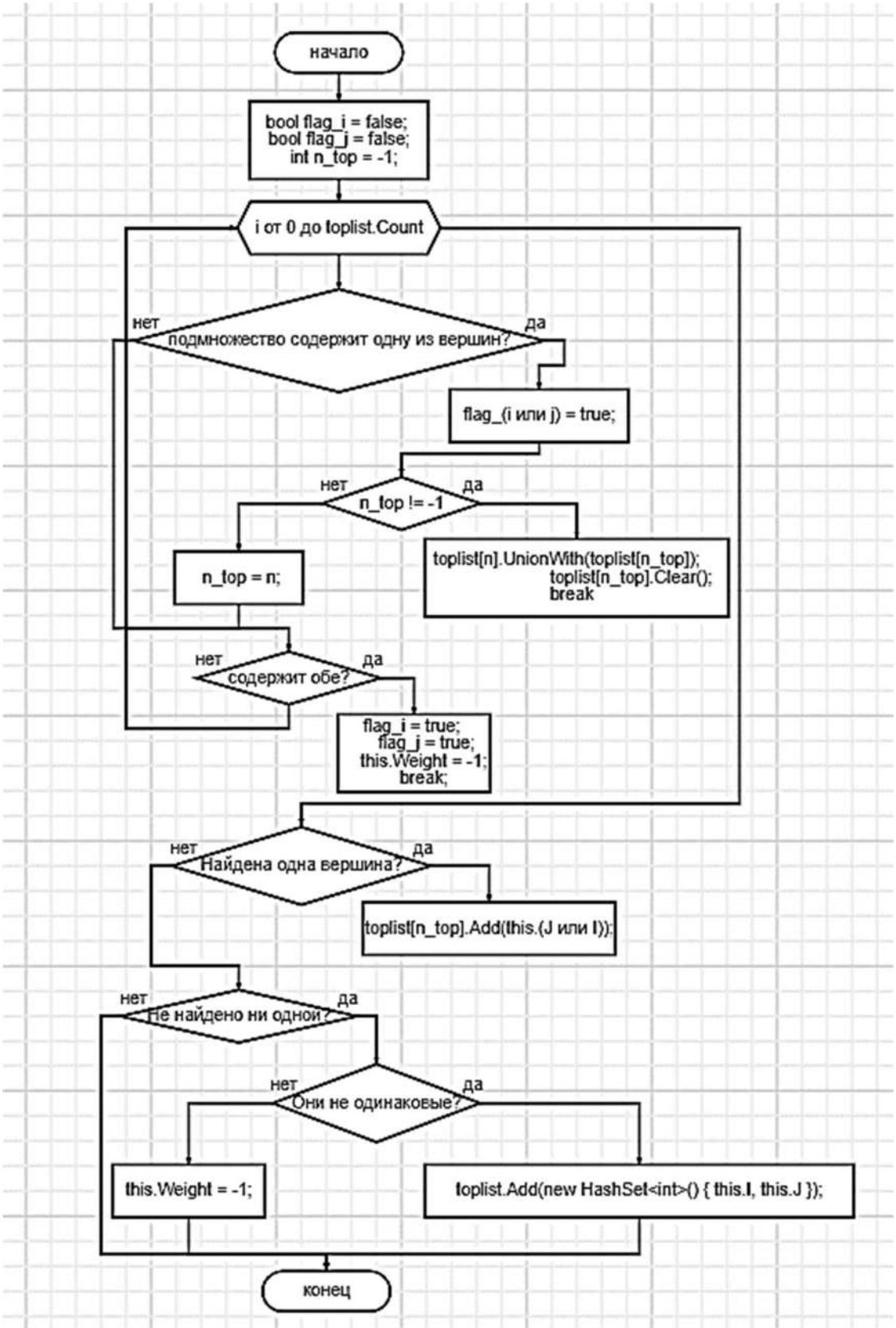


Рисунок 10 – Блок-схема алгоритма Крускала

Ниже представлен результат расчета примера:

```
(0) вершина -->[7; 12; 10; 0; 11; 12; ]

(1) вершина -->[12; 12; 0; 14; 0; 18; ]

(2) вершина -->[10; 0; 9; 0; 0; 5; ]

(3) вершина -->[0; 14; 0; 0; 0; 0; ]

(4) вершина -->[11; 0; 0; 0; 1; 12; ]

(5) вершина -->[12; 18; 5; 0; 12; 12; ]
Отсортированный список ребер:
1 5 7 9 10 11 12 12 12 12 12 14 18
ребро между вершинами 2 и 5 вес5
ребро между вершинами 0 и 2 вес10
ребро между вершинами 0 и 4 вес11
ребро между вершинами 0 и 1 вес12
ребро между вершинами 1 и 3 вес14
```

Выводы:

В ходе выполнения работы изучены и реализованы на языке C# основные алгоритмы на графах – алгоритм поиска в ширину и в глубину, алгоритм Дейкстры и алгоритм Крускала.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие задачи решаются с помощью алгоритма Дейкстры и алгоритма Крускала?
2. Какие ограничения и условия на применение алгоритма Дейкстры?
3. Какие конструкции ООП C# применяются для реализации алгоритма поиска в ширину и в глубину?
4. Как оценивается сложность алгоритма Дейкстры?
5. К какому классу сложности относится алгоритм Дейкстры?
6. Как можно оценить точность алгоритма Дейкстры?

Лабораторная работа №5. Алгоритм метода «разделяй и властвуй»

ЦЕЛЬ РАБОТЫ

Изучить метод «разделяй и властвуй». Реализовать средствами ООП C# быструю сортировку.

ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

В методе «разделяй и властвуй» задача рекурсивно разбивается на элементарные подзадачи и комбинируется их решение. Метод «разделяй и властвуй» применяется при поиске оптимального решения и приводит каждую задачу к одной подзадаче, как в алгоритме бинарного поиска.
Метод «разделяй и властвуй» включает:

1. **Разделение** – деление на несколько задач, которые являются меньшими экземплярами той же задачи
2. **Властвование** – решение подзадач с помощью рекурсии
3. **Комбинирование** – решение подзадач в решении исходной задачи.

Метод «разделяй и властвуй» представлен в алгоритме сортировки. Блок-схема алгоритма сортировки показана ниже.

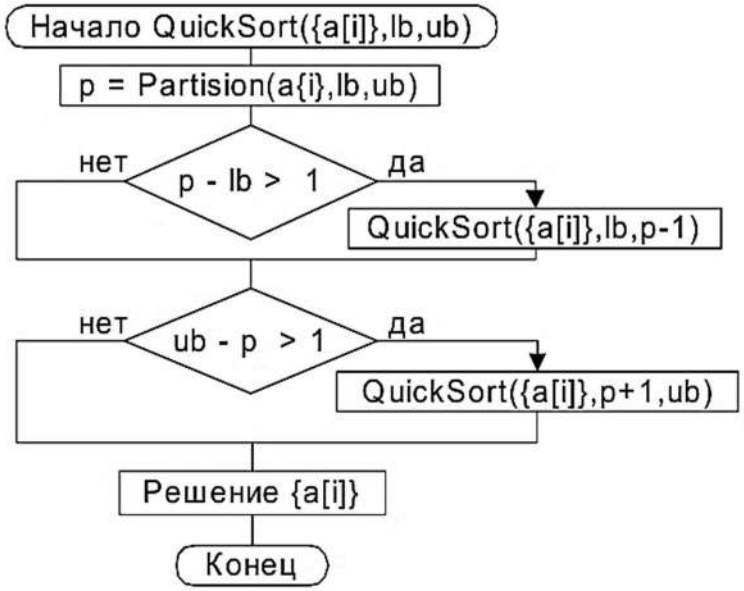


Рисунок 11 – Блок-схема алгоритма сортировки

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Реализовать средствами ООП C# быструю сортировку методом «разделяй и властвуй». Выполнить расчет примера.

ПРИМЕР ВЫПОЛНЕНИЯ РАБОТЫ

Создадим функцию, которая будет делить массив на левую и правую часть и возвращать максимальный массив и выгоду. После разделения функция рекурсивно получает максимальный массив и выгоды из левой и правой части. Также считается выгода для случая, если максимальный массив начинается в левой

и заканчивается в правой части, после чего выбирается наибольшая выгода из трех и возвращается соответствующий максимальный массив.
Ниже представлен листинг сортировки:

```
public int[] SortA1(int[] array, int IndexL, int IndexR)
{
        var i = IndexL;
        var j = IndexR;
        var pivot = array[IndexL];
while (i <= j)
{
        while (array[i] < pivot)
{
        i++;
}
while (array[j] > pivot)
{
        j--;
}
if (i <= j)
{
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
        i++;
        j--;
}
}
if (IndexL < j)
        SortA1(array, IndexL, j);
if (i < IndexR)
        SortA1(array, i, IndexR);
return array;
}
```

Выводы:

В ходе выполнения работы изучен метод «разделяй и властвуй» на примере алгоритма сортировки.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие ограничения и условия на применение алгоритма метода разделяй и властвуй?
2. Какие конструкции ООП C# применяются для реализации алгоритма быстрой сортировки?

3. Как оценивается сложность алгоритма быстрой сортировки?
4. Какие задачи решаются с помощью алгоритма метода разделяй и властвуй?
5. К какому классу сложности относится алгоритм быстрой сортировки?
6. Как можно оценить точность алгоритма метода разделяй и властвуй?
7. В каких алгоритмах применяется метод разделяй и властвуй?

Лабораторная работа №6. Алгоритмы сортировки

ЦЕЛЬ РАБОТЫ

Научиться реализовывать на ООП алгоритмы сортировки: вставками, выбором, пузырьком, слиянием, Шелла.

ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Рассмотрим классические алгоритмы сортировки.

Алгоритм сортировки пузырьком состоит из повторяющихся проходов по массиву в которых сравниваются соседние элементы и выполняется перестановка при необходимости.

Блок-схема алгоритма сортировки пузырьком представлена ниже:

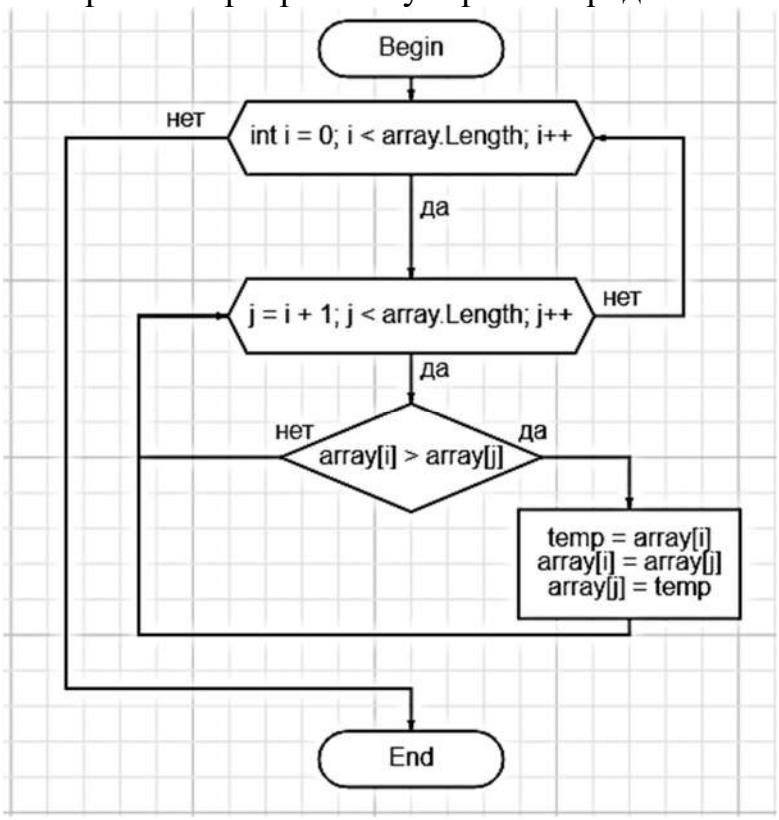


Рисунок 12 – Блок-схема пузырьковой сортировки

Алгоритм сортировки вставками выполняет просмотр элементов входной последовательности и размещение в соответствующее место упорядоченного массива элементов.

Алгоритм сортировки выбором находит минимальный элемент в массиве и ставит его на первую позицию. Далее повторяется обмен только для неотсортированной части массива.

Блок-схема алгоритма сортировки выбором представлена ниже:

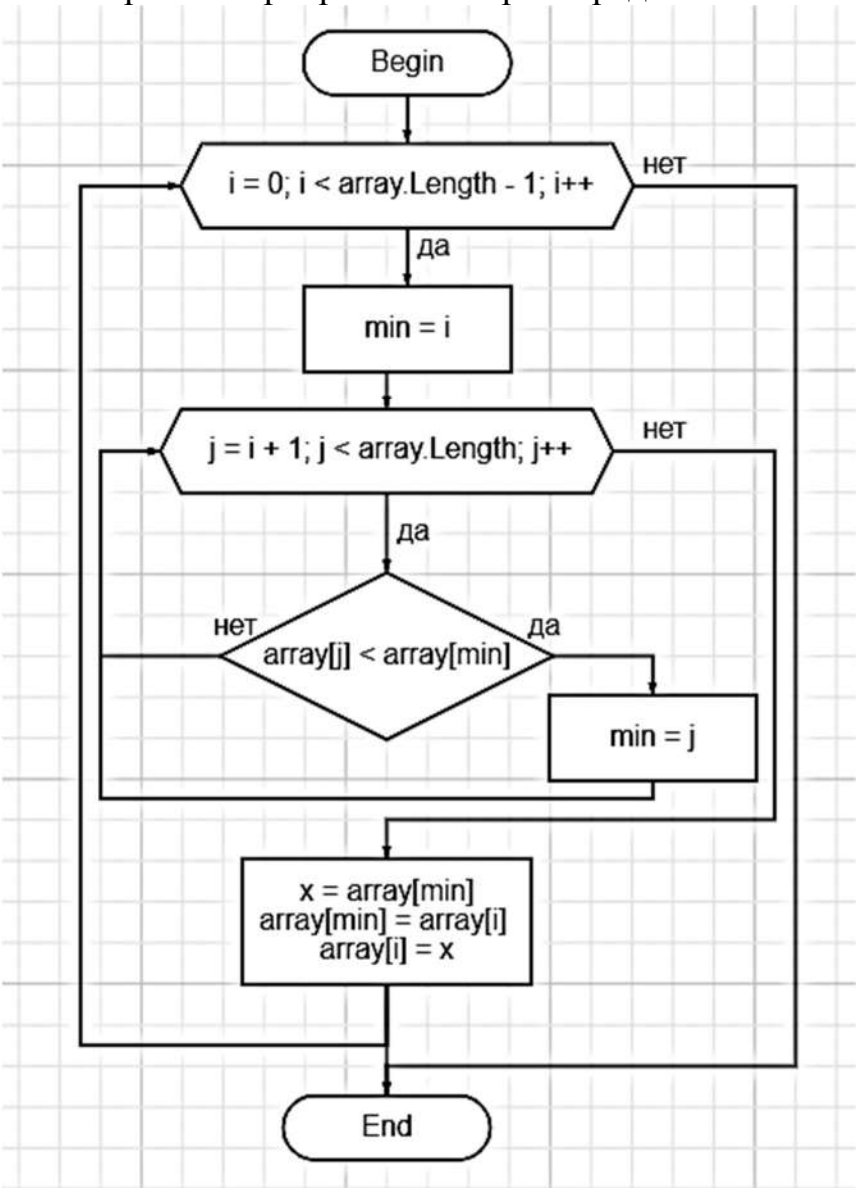


Рисунок 13 – Блок-схема алгоритма сортировки выбором

Алгоритм сортировки слиянием применяет метод «разделяй и властвуй». Массив разбивается на части, для которых рекурсивно решается задача сортировки и результаты решения комбинируются.

Алгоритм сортировки Шелла усовершенствует сортировку вставками. Здесь выполняется сравнение не соседних элементов массива, а элементов на заданном расстоянии.

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

1. Реализовать средствами ООП C# два алгоритма сортировки массива данных.
2. Выполнить расчет примера.

ПРИМЕР ВЫПОЛНЕНИЯ РАБОТЫ

Для реализации были выбраны алгоритмы сортировки выбором, пузырьковая сортировка. Каждый из алгоритмов был реализован на языке ООП C#.

Рассмотрим реализацию выборочной сортировки. Это алгоритм сортировки, при котором выбирается минимальное значение, производится обмен этого значения со значением на первой позиции в массиве. Эти операции повторяются с хвостом списка: исключаются уже отсортированные элементы – до тех пор, пока весь список не будет отсортирован.

В данном случае сортировка реализуется с помощью метода ViborSort, принимающего на вход целочисленный массив, а также вспомогательного метода Swap, принимающего элементы массива, которые нужно поменять местами.

Ниже представлен листинг для сортировки выбором:

```
static void Swap(ref int a, ref int b)
{
    int x = a;
    a = b;
    b = x;
}
static int[] ViborSort(int[] array)
{
    for (int i = 0; i < array.Length - 1; i++)
    {
        int min = i;
        for (int j = i + 1; j < array.Length; j++)
        {
            if (array[j] < array[min])
            {
                min = j;
            }
        }
        Swap(ref array[min], ref array[i]);
    }
    return array;
}
```

Ниже представлен листинг для метода Main:

```
static void Main(string[] args)
{
    Console.WriteLine("Ведите количество чисел во входном массиве");
    int n = Convert.ToInt32(Console.ReadLine());
    int[] array = new int[n];
    Console.WriteLine("Введите числа:");
    for (int i = 0; i < array.Length; i++)
    {
        Console.Write("array[{0}] = ", i);
        array[i] = Convert.ToInt32(Console.ReadLine());
    }
    ViborSort(array);
    Console.WriteLine("Отсортированный массив:");
    for (int i = 0; i < array.Length; i++)
    {
        Console.WriteLine(array[i]);
    }
    Console.ReadLine();
}
```

Ниже представлен результат расчета примера:

```
Ведите количество чисел во входном массиве
5
Введите числа:
array[0] = 4
array[1] = 0
array[2] = 9
array[3] = 2
array[4] = 10
Отсортированный массив:
0
2
4
9
10
```

Далее рассмотрим пузырьковую сортировку. Алгоритм состоит из повторяющихся проходов по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов. Проходы по массиву повторяются N-1 раз или до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает — массив отсортирован.

В данном случае сортировка реализуется с помощью метода BubbleSort, принимающего на вход целочисленный массив.

Ниже представлен листинг для пузырьковой сортировки:

```
static int[] BubbleSort(int[] array)
{
    int temp;
    for (int i = 0; i < array.Length; i++)
    {
        for (int j = i + 1; j < array.Length; j++)
        {
            if (array[i] > array[j])
            {
                temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
    }
    return array;
}
```

Ниже представлен листинг для метода Main:

```
static void Main(string[] args)
{
    Console.WriteLine("Введите количество элементов входного массива");
    int n = Convert.ToInt32(Console.ReadLine());
    int[] array = new int[n];
    Console.WriteLine("Введите числа:");
    for (int i = 0; i < array.Length; i++)
    {
        Console.Write("a[{0}] = ", i);
        array[i] = Convert.ToInt32(Console.ReadLine());
    }

    BubbleSort(array);
    Console.WriteLine("Отсортированный массив:");
    for (int i = 0; i < array.Length; i++)
    {
        Console.WriteLine(array[i]);
    }
}
```

Ниже представлен результат расчета примера

```
Введите количество элементов входного массива
5
Введите числа:
a[0] = 0
a[1] = 3
a[2] = 8
a[3] = 4
a[4] = 10
Отсортированный массив:
0
3
4
8
10
```

Выводы:
В ходе выполнения работы изучены и реализованы средствами ООП C# алгоритмы сортировки.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие ограничения и условия на применение алгоритма сортировки: вставками, выбором, пузырьком, слиянием?
2. Какие конструкции ООП C# применяются для реализации алгоритма сортировки?
3. Как оценивается сложность алгоритма сортировки: вставками, выбором, пузырьком, слиянием?
4. Какие задачи решаются с помощью алгоритма Шелла?
5. К какому классу сложности относится алгоритм сортировки?
6. Как можно оценить временную сложность алгоритма сортировки?
7. Какой из алгоритмов сортировки является наиболее быстрым.

Лабораторная работа №7. Алгоритмы поиска

ЦЕЛЬ РАБОТЫ

Изучить алгоритмы поиска. Реализовать средствами ООП C# алгоритм поиска.

ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Алгоритм линейного поиска выполняется последовательным сравнением всех элементов с искомым. Алгоритм эффективно применяется только для небольших массивов данных.

Алгоритм бинарного поиска требует предварительную сортировку массива и применяет метод деления пополам. В алгоритме элемент в середине сравнивается с искомым, и далее поиск итерационно выполняется для левой или правой части пока элемент не найден.
Ниже представлена блок-схема линейного поиска:

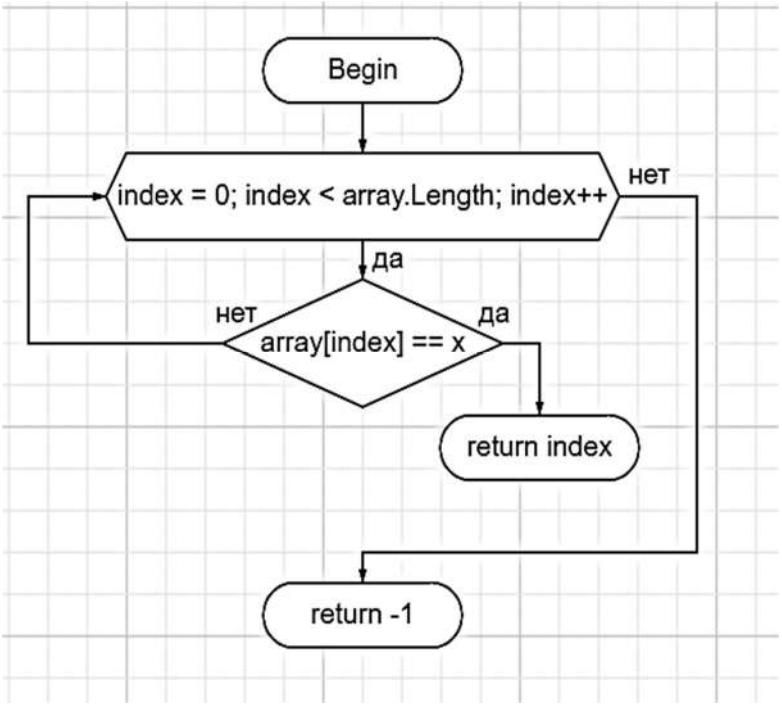


Рисунок 14 – Блок-схема линейного поиска

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Реализовать средствами ООП C# один алгоритм поиска. Выполнить расчет примера.

ПРИМЕР ВЫПОЛНЕНИЯ РАБОТЫ

Рассмотрим алгоритм линейного поиска. Поиск осуществляется на заданном множестве (массиве) путем последовательного сравнения очередного рассматриваемого значения с искомым до тех пор, пока эти значения не совпадут.

В данном случае поиск осуществляется посредством метода LinearSearch(), который принимает на вход целочисленный массив и искомое значение. Метод возвращает либо индекс искомого элемента, если он найден, либо -1, если не найден.

Ниже представлен листинг для линейного поиска:

```
static int LinearSearch(int[] array, int x)
{
    for (int index = 0; index < array.Length; index++)
    {
        if (array[index] == x)
        {
            return index;
        }
    }

    return -1;
}
```

Ниже представлен листинг для метода Main

```
static void Main(string[] args)
{
    Console.WriteLine("Введите количество элементов входного массива");
    int n = Convert.ToInt32(Console.ReadLine());
    int[] array = new int[n];
    Console.WriteLine("Введите числа:");
    for (int i = 0; i < array.Length; i++)
    {
        Console.Write("a[{0}] = ", i);
        array[i] = Convert.ToInt32(Console.ReadLine());
    }
    Console.WriteLine("Введите искомое значение");
    int x = Convert.ToInt32(Console.ReadLine());
    int index = LinearSearch(array,x);
    if (index == -1)
    {
        Console.WriteLine("Элемент не найден");
    }
    else
    {
        Console.WriteLine("Индекс искомого элемента - " + index);
    }
}
```

Ниже представлен результаты расчета примера (элемент найден и не найден

```
Введите количество элементов входного массива
5
Введите числа:
a[0] = 6
a[1] = 7
a[2] = 1
a[3] = 3
a[4] = 4
Введите искомое значение
4
Индекс искомого элемента - 4
```

```
Введите количество элементов входного массива
5
Введите числа:
a[0] = 3
a[1] = 6
a[2] = 8
a[3] = 1
a[4] = 2
Введите искомое значение
5
Элемент не найден
```

Выводы:

В ходе выполнения работы изучены и реализованы средствами ООП C# алгоритмы поиска.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие задачи решаются с помощью алгоритма поиска?
2. Какие конструкции ООП C# применяются для реализации алгоритма поиска?
3. Как оценивается сложность алгоритма поиска?
4. Какие ограничения и условия на применение алгоритма поиска?
5. К какому классу сложности относится алгоритм поиска?
6. Как можно оценить время выполнения алгоритма поиска?
7. Какую необходимо выполнить предобработку при реализации алгоритма бинарного поиска?

Лабораторная работа №8. Анализ сложности алгоритмов

ЦЕЛЬ РАБОТЫ

Изучить методику анализа сложности алгоритмов наилучшего, наихудшего и среднего случаев.

ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Сложность алгоритма –количественная мера ресурсов алгоритма для успешного решения поставленной задачи.
Рассмотри основные ресурсы:

- время определяет временную сложность
- объем памяти определяет ёмкостную сложность

Рассмотрим метрики для анализа сложности наилучшего, наихудшего и среднего случаев.
Для наихудшего случая - функция, определяемая максимальным количеством шагов для обработки любого входного экземпляра размером n;
Для наилучшего случая - функция, определяемая минимальным количеством шагов для обработки любого входного экземпляра размером n;
Для среднего случая - функция, определяемая средним количеством шагов для обработки всех экземпляров размером n;
Введем асимптотические обозначения: (O, Θ, Ω) верхние и нижние пределы функции.
g(n) = O(f(n)) обозначает, что C × f(n) - верхняя граница функции g(n)
g(n) = Ω(f(n)) обозначает, что C × f(n) - нижняя граница функции g(n).
g(n) = Θ(f(n)) обозначает, что C1 × f(n) выше функции g(n) и C2 × f(n) ниже функции g(n).
Здесь константы C, C1, и C2 не зависят от n.

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

1. Реализовать средствами ООП C# на выбор два алгоритма сортировки массива данных (например, быстрая сортировка, сортировка вставкой) и один на выбор алгоритм поиска (бинарный поиск). Выполнить расчет примера.
2. Выполнить анализ сложности наилучшего, наихудшего и среднего случаев для трех реализованных алгоритмов, привести вывод расчета сложности.

ПРИМЕР ВЫПОЛНЕНИЯ РАБОТЫ

Рассмотрим сортировку выбором, реализованную в предыдущей лабораторной. Максимальное количество перестановок на некотором массиве длины n будет равно n-1, при этом число сравнений будет одним и тем же. Тогда имеем T(n) = n-1+n-2+n-3+…+1 = n(n-1)/2 количество сравнений всего. Сюда добавляется количество присваиваний 3(n-1), но эта функция растет значительно медленней, поэтому ее при оценке асимптотической сложности учитывать не

будем. Получаем, что O(n) возрастает приблизительно как N^2 – проще говоря, мы проходим по n-ому массиву n раз. Это верно для наилучшего, для наихудшего и для среднего случаев.

Далее рассмотрим пузырьковую сортировку, реализованную в предыдущей лабораторной. При пузырьковой сортировке массива, состоящего из n элементов, необходимо просмотреть его n-1 раз, каждый раз уменьшая диапазон просмотра на один элемент. Получаем количество сравнений n-1+n-2+n-3+…+1 = n(n-1)/2 и количество присваиваний 3(n-1+n-2+n-3+…+1) = 3n(n-1)/2. Итого T(n) = n(n-1)/2 + 3n(n-1)/2. В наилучшем случае алгоритм сработает в один проход (например, при массиве {0, 1, 4, 7, 10}), тогда сложность алгоритма будет O(n) = n. В остальных же случаях, когда на входе мы имеем неотсортированный массив, сложность будет O(n) = n^2, так как мы опять же проходим по n-ому массиву n раз.

В ходе линейного поиска, реализованного в предыдущей лабораторной, массив размером n проходится один раз либо до последнего элемента, либо до искомого. Таким образом, мы имеем сложность O(n) = n как для наилучшего, так и для наихудшего и среднего случаев.

Выводы:

В ходе выполнения лабораторной работы рассмотрена сложность реализованных алгоритмов сортировки выбором, пузырьковой сортировки, а также линейного поиска. Проведен анализ сложности каждого алгоритма.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как оценивается сложность алгоритма поиска?
2. Как определить к какому классу сложности относится алгоритм?
3. Как оценивается сложность алгоритма сортировки?
4. Как определить какие ограничения и условия на применение алгоритма?
5. Как можно оценить временную сложность алгоритма?
6. Какие существуют классы сложности алгоритмов?

Литература

1. Рутковская Д., Пилиньский М., Рутковский Л. Нейронные сети, генетические алгоритмы и нечеткие системы: Пер.с польск. И.Д.Рудинского. М.: Горячая линия-Телеком, 2013. - 384 с. - Режим доступа: http://e.lanbook.com/books/element.php?pl1_cid=25&pl1_id=11843
2. Окулов, С.М. Алгоритмы обработки строк [Электронный ресурс] : учебное пособие / С.М. Окулов. — Электрон. дан. — Москва : Издательство "Лаборатория знаний", 2015. — 258 с. — Режим доступа: https://e.lanbook.com/book/66113. — Загл. с экрана.

3. Костюкова Н.И. Графы и их применение. Комбинаторные алгоритмы для программистов. М., 2007.
4. Клейнберг Дж., Тардос Е. Алгоритмы: разработка и применение. Классика Computers Science / Пер. с англ. Е. Матвеева. — СПб.: Питер, 2016. — 800 с.
5. Курейчик В. М. Генетические алгоритмы и их применение. – 2002.
6. Снитюк В. Е. Прогнозирование. Модели, методы, алгоритмы //К.: Маклаут. – 2008.
7. Кормен Т. и др. Алгоритмы. Построение и анализ:[пер. с англ.]. – Издательский дом Вильямс, 2009.
8. Спринджук М. В. и др. Современные алгоритмы обработки данных транскриптомов: обзор методов и результаты апробации //Системный анализ и прикладная информатика. – 2021. – №. 2. – С. 54-62.
9. Кныш Д. С., Курейчик В. М. Параллельные генетические алгоритмы: обзор и состояние проблемы //Известия Российской академии наук. Теория и системы управления. – 2010. – №. 4. – С. 72-82.
10. Литвинов В. Г. и др. Разработка и применение типовых решений для распараллеливания алгоритмов численного моделирования : дис. – Сам. гос. аэрокосм. ун-т им. СП Королева, 2015.
11. Яцко В. А. Алгоритмы и программы автоматической обработки текста //Вестник Иркутского государственного лингвистического университета. – 2012. – №. 1 (17). – С. 150-160.
12. Матренин П. В. и др. Разработка адаптивных алгоритмов роевого интеллекта в проектировании и управлении техническими системами //Новосибирский государственный технический университет. – 2018.
13. Гриффитс И. Программирование на C# 5.0. – Litres, 2022.
14. Троелсен Э., Джепикс Ф. Язык программирования C# 7 и платформы. NET и. NET Core. – Litres, 2022.
15. Кириченко А. А. Объектно-ориентированное программирование на алгоритмическом языке C#: учебное пособие //М.: Издательство «Высшая школа экономики. – 2015.
16. Додобоев Н. Н., Кукарцева О. И., Тынченко Я. А. Современные языки программирования //Современные технологии: актуальные вопросы, достижения и инновации. – 2019. – С. 81-85.

Иванов Сергей Евгеньевич

Прикладные алгоритмы на языке ООП C#

Учебно-методическое пособие

В авторской редакции
Редакционно-издательский отдел Университета ИТМО
Зав. РИО Н.Ф. Гусарова
Подписано к печати
Заказ №
Тираж
Отпечатано на ризографе

Редакционно-издательский отдел
Университета ИТМО
197101, Санкт-Петербург, Кронверкский пр., 49, литер А