

Министерство образования и науки Российской Федерации
ФГАО УВО “Севастопольский государственный университет”

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к выполнению лабораторных работ по дисциплине “ Основы
теории алгоритмов” студентами всех форм обучения для
направлений:

09.03.01 – “Информатика и вычислительная техника”,
09.03.02 – “Информационные системы и технологии”,
09.03.04 – “Управление в технических системах”

Севастополь
2018

УДК 004.423 + 453

Методические указания к выполнению лабораторных и контрольных работ по дисциплине “Основы теории алгоритмов” студентами всех форм обучения для направлений: 09.03.01 – “Информатика и вычислительная техника”, 09.03.02 – “Информационные системы и технологии”, 09.03.04 – “Управление в технических системах” [Текст] / Разраб. П.П. Кижее. – Севастополь: Изд-во СевГУ, 2018. – 44 с.

Цель методических указаний: Обеспечение студентов дидактическим материалом для освоения методов анализа алгоритмов.

Методические указания рассмотрены и утверждены на заседании кафедры Информационных систем, протокол № 13 от 26 января 2018 г.

Допущено учебно-методическим центром СевГУ в качестве методических указаний.

Рецензент Кожеев Е.А., кандидат техн. наук, доцент кафедры Информатики и вычислительной техники.

СОДЕРЖАНИЕ

Ведение	4
1. Лабораторная работа № 1. “Асимптотический анализ количественно-зависимых алгоритмов”	4
2. Лабораторная работа № 2. “Теоретическая оценка среднего количества операций переприсваивания в алгоритме поиска минимума”	12
3. Лабораторная работа № 3. “Экспериментальное определение количества элементарных операций языка высокого уровня в программной реализации алгоритма”	16
4. Лабораторная работа № 4. “Определение среднего времени выполнения обобщенной элементарной операции методом тестовых прогонов”	23
5. Лабораторная работа № 5. “Поиск кратчайших путей на графе”	27
6. Лабораторная работа № 6. “Анализ алгоритмов сортировки”	36
7. Лабораторная работа № 7. “Изучение алгоритмов для нахождения НОД целых чисел”	45
8. Лабораторная работа № 8. “Построение кратчайших остовых деревьев графа”	49
Заключение	
Библиографический список	

ВВЕДЕНИЕ

ЛАБОРАТОРНАЯ РАБОТА № 1. “АСИМПТОТИЧЕСКИЙ АНАЛИЗ КОЛИЧЕСТВЕННО-ЗАВИСИМЫХ АЛГОРИТМОВ”

1. ЦЕЛЬ РАБОТЫ

Изучить поведение функций трудоемкости количественно-зависимых алгоритмов в реальных интервалах значений мощности множества исходных данных. На основании этого сделать предпочтительный выбор того или иного алгоритма. Для сравнения функций трудоемкости использовать аппарат интервального анализа, реализованный в виде программы на языке C++.

2. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

2.1 Основные определения асимптотического анализа алгоритмов

При использовании алгоритмов для решения практических задач мы сталкиваемся с проблемой рационального выбора алгоритма решения задачи. Решение проблемы выбора связано с построением системы сравнительных оценок, которая в свою очередь существенно опирается на формальную модель алгоритма.

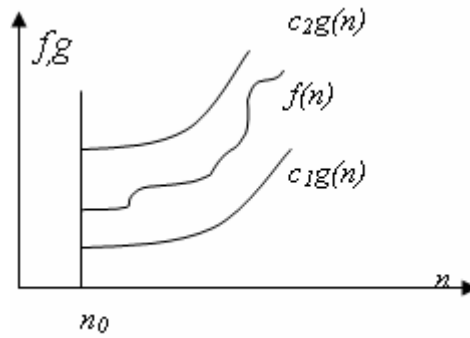
Под *трудоемкостью* алгоритма для данного конкретного входа – $F_a(N)$, будем понимать количество «элементарных» операций, совершаемых алгоритмом для решения конкретной проблемы в данной формальной системе. При анализе поведения функции трудоемкости алгоритма часто используют принятые в математике асимптотические обозначения, позволяющие показать *скорость роста функции*, маскируя при этом конкретные коэффициенты.

Такая оценка функции трудоемкости алгоритма называется *сложностью алгоритма* и позволяет определить предпочтения в использовании того или иного алгоритма для больших значений размерности исходных данных.

В асимптотическом анализе приняты следующие обозначения:

1) Оценка Θ (тетта)

Пусть $f(n)$ и $g(n)$ – положительные функции положительного аргумента, $n \geq 1$ (количество объектов на входе и количество операций – положительные числа), тогда: $f(n) = \Theta(g(n))$, если существуют положительные c_1, c_2, n_0 , такие, что: $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$, при $n > n_0$

Рисунок 2.1 – Оценка Θ

Обычно говорят, что при этом функция $g(n)$ является асимптотически точной оценкой функции $f(n)$, т.к. по определению функция $f(n)$ не отличается от функции $g(n)$ с точностью до постоянного множителя.

Отметим, что из $f(n) = \Theta(g(n))$ следует, что $g(n) = \Theta(f(n))$.

Примеры:

1) $f(n) = 4n^2 + n \ln N + 174$ – $f(n) = \Theta(n^2)$;

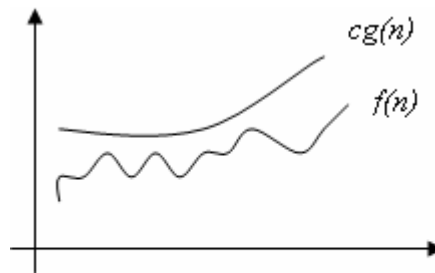
2) $f(n) = \Theta(1)$ – запись означает, что $f(n)$ или равна константе, не равной нулю, или $f(n)$ ограничена константой на ∞ : $f(n) = 7 + 1/n = \Theta(1)$.

2) Оценка O (O большое)

В отличие от оценки Θ , оценка O требует, чтобы функция $f(n)$ не превышала $g(n)$ начиная с $n > n_0$, с точностью до постоянного множителя:

$$\exists c > 0, n_0 > 0 :$$

$$0 \leq f(n) \leq c * g(n), \quad \forall n > n_0$$

Рисунок 2.2 – Оценка O (O большое)

Выбор того или иного вычислительного алгоритма для решения некоторой задачи требует исследования претендующих алгоритмов не только для оценки сложности, но и детального исследования для определения размерности и иных характеристик множества реальных исходных данных задачи. Это существенно влияет на рациональный выбор алгоритма.

Проблема состоит в том, что не всегда алгоритм, имеющий асимптотически оптимальную трудоемкость (в смысле оценок O или Θ) имеет лучшие показатели для реального множества исходных данных из-за

существенного различия констант, скрывающихся за O или Θ асимптотическими оценками.

Это требует практического сравнительного анализа алгоритмов, для чего осуществляется:

- детальный анализ трудоемкости алгоритмов, получение в явном виде функции трудоемкости;
- сравнительный анализ функций трудоемкости претендующих алгоритмов для выбора рационального алгоритма решения данной задачи при реальных ограничениях множества исходных данных.

Реализация второго этапа – сравнительного анализа функций трудоемкости – предполагает определение характеристических значений реального множества исходных данных задачи (Da), при которых предпочтение может быть отдано одному из анализируемых алгоритмов. Можно считать, что таким характеристическим значением является размерность множества исходных данных - n , а трудоемкость алгоритма в основном определяется количеством исходных данных - $f(Da) = f(n)$.

Таким образом, сравнительный анализ функций трудоемкости предполагает *определение тех интервалов аргумента функций трудоемкости, при которых данный алгоритм может быть выбран в качестве рационального*. В связи с этим предлагаемый ниже метод назван *интервальным анализом функций* (трудоемкости алгоритмов) по аналогии с асимптотическим анализом - определением сложности алгоритмов.

Исходными данными для интервального анализа функций являются *известные функции трудоемкости алгоритмов*, а *результатами* - *предпочтительные интервалы значений множества исходных данных задачи* для рационального применения того или иного алгоритма.

Реально выбор данного алгоритма как предпочтительного, может быть произведен даже тогда, когда его трудоемкость на данном интервале несколько хуже, не более, чем на порог φ , чем трудоемкость других алгоритмов. Или несколько алгоритмов могут быть использованы на этом интервале эквивалентно, с расхождением на более чем на порог φ . Это приводит к необходимости введения *специальных обозначений* в аппарате интервального анализа.

2.2. Система обозначений в интервальном анализе функций

Пусть трудоемкости двух алгоритмов заданы в явном виде функциями $f(n)$ и $g(n)$ соответственно, пусть задан также порог φ допустимого расхождения значений функций на интервале, мера $\pi(f(n), g(n))$ расхождения функций при данном n и интервал (a, b) значений n . Тогда считаем, что:

$$a) \quad f(n) = \Delta_{\varphi}(g(n)), \text{ если } \min_{n \in (a, b)} \{\pi(f(n), g(n))\} \geq \varphi;$$

$$\text{б)} \quad f(n) = \Theta_{\varphi}(g(n)), \quad \text{если} \quad \max_{n \in (a,b)} |\pi(f(n), g(n))| < \varphi;$$

$$\text{в)} \quad f(n) = O_{\varphi}(g(n)), \quad \text{если} \quad \max_{n \in (a,b)} \{\pi(f(n), g(n))\} \leq -\varphi.$$

2.3. Метод интервального анализа

На основании введенной системы обозначений может быть предложен метод интервального анализа функций трудоемкости алгоритмов, включающий этапы:

- Определение для данной конкретной задачи допустимого интервала изменения размерности множества исходных данных (a, b) ;
- Получение в явном виде функций трудоемкости анализируемых алгоритмов $f(n)$ и $g(n)$;
- Разбиение интервала (a, b) на подынтервалы, в каждой целочисленной точке которых явно выполняется одно из следующих соотношений для принятой меры $\pi(f(n), g(n))$:

$$1) \quad \text{если} \quad \varphi - \pi(f, g) < 0, \quad \text{то} \quad f(n) = \Delta_{\varphi}(g(n)) \quad \text{и}$$

предпочтительным на данном подынтервале является алгоритм, имеющий функцию трудоемкости $g(n)$.

$$2) \quad \text{если} \quad |\pi(f, g)| - \varphi < 0, \quad \text{то} \quad f(n) = \Theta_{\varphi}(g(n)) \quad \text{и оба алгоритма}$$

с точностью до порога φ могут быть использованы на этом подынтервале.

$$3) \quad \text{если} \quad \pi(f, g) + \varphi < 0, \quad \text{то} \quad f(n) = O_{\varphi}(g(n)) \quad \text{и}$$

предпочтительным на данном подынтервале является алгоритм, имеющий функцию трудоемкости $f(n)$.

2.4. Мера $\pi(f(n), g(n))$

Мера расхождения значений функций $f(n)$ и $g(n)$ при фиксированном n должна удовлетворять следующим требованиям:

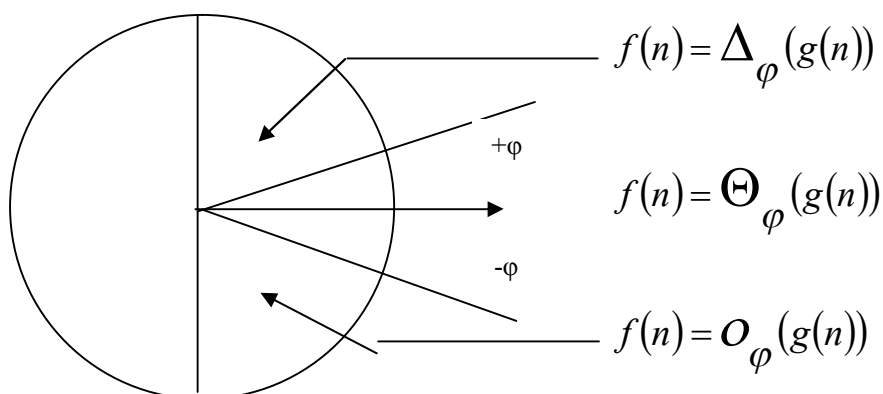
- $\pi(f(n), g(n)) = -\pi(g(n), f(n))$;
- $\pi(f(n), g(n)) = 0$, если $f(n) = g(n)$.

Может быть определено несколько различных мер, удовлетворяющих данным требованиям. В работе предлагается использовать следующую функцию меры, которая может быть интерпретирована, как угловое расхождение значений аргументов:

$$\pi(f(n), g(n)) = \arctg \frac{f(n)}{g(n)} - \arctg \frac{g(n)}{f(n)};$$

Поскольку значения функций трудоемкости положительны, то при таком определении меры значения $\pi(f, g)$ ограничены между $\pi/2$ и $-\pi/2$. Пороговое значение φ может быть интерпретировано, как максимальное значение угла расхождения, для которого алгоритмы могут быть признаны равноприменимыми на интервале.

Может быть предложена следующая графическая интерпретация метода интервального анализа функций трудоемкости:



3. ХОД РАБОТЫ

1. Ознакомиться с теоретическим разделом настоящих методических указаний и повторить соответствующий лекционный материал.

2. Для указанной в варианте задания пары функций трудоемкости, целочисленных интервалов $\{(20;50), (100;120), (500, 540)\}$ и значений $\varphi = \{\frac{\pi}{32}, \frac{\pi}{24}, \frac{\pi}{18}\}$, определить, каково соотношение между функциями трудоемкости на заданном интервале. Для вычисления значений функций и определения соотношений написать программу на языке С. Пример программы приведен в приложении. Результаты работы программы сохранять в текстовые файлы. Сведения об основных алгебраических функциях и основных конструкциях языка С можно получить либо из системы помощи интегрированной среды разработки, либо из методических указаний по АиП, посвященных программированию на языке С.

3. Путем подбора значений аргумента определить интервалы, на которых выполняется соотношение: $f(n) = \Theta_{\varphi}(g(n))$

4. Построить графики заданных функций на указанном интервале (вручную или с помощью любого программного обеспечения).

5. Сделать выводы по работе, оформить отчет, подготовить ответы на контрольные вопросы.

4. ВАРИАНТЫ ЗАДАНИЙ

	$f(n)$	$g(n)$
1	$2,5n^2 + 3n \log n + 18n$	$17n \log n + 38n$
2	$8n^3 + 4n^2 + 4n$	$48n^{\log_2 7}$
3	$\ln n^{\ln n}$	$2^{\sqrt{n}}$
4	$2,5n^2 + 6n$	$7,5n\sqrt{n} + 22n$
5	$14n \ln n + 22n$	$4n\sqrt{n} - 11n$
6	$n^{\frac{\ln n}{2}}$	$e^{1,5\sqrt{n}}$
7	$6 \cdot 2^{n-3}$	$18n^4 + 48n^3 + 324n^2$
8	$44 \ln^3 n + 24 \ln^2 n$	$1,5n + 8$
9	$19\sqrt{n} + 54$	$28 \log_2^2 n$
10	$11n \log_2^2 n + 7n \log_2 n$	$14n\sqrt{n} + 89\sqrt{n}$
11	$\sqrt[n]{\frac{\sqrt{n}}{8}}$	$n^{\ln \ln n}$
12	$\ln \ln n^{\sqrt{n}}$	$\sqrt{n}^{\ln \ln n}$
13	$22n^2\sqrt{n} + 7n \ln n$	$\frac{8n^3}{\ln n} + 16\sqrt{n}^3$
14	$\frac{2^{n-4}}{\sqrt{n}}$	$1,618^n$
15	$\ln \ln n^{\sqrt{n}}$	$e^{\sqrt{\ln n \cdot \ln \ln n}}$
16	$17n^3 + 19n^2 \ln n$	$3n^4 - 24n^2\sqrt{n}$
17	$27n^{\log_2 5}$	$37n^2 \ln n$
18	$29\sqrt{n} \ln n$	$7 \log_2^7 n$

	$f(n)$	$g(n)$
19	$\frac{59n}{\ln n}$	$14\sqrt{n}^{\log_{10} 11}$
20	$11\ln(2^{n-1} + 3^{n-2})$	$19(n - \ln n)$

5. ПРИМЕР ПРОГРАММЫ ИНТЕРВАЛЬНОГО АНАЛИЗА

Пример программы определения интервалов предпочтения алгоритмов для функций трудоемкости $F(n) = 1.75 * n^2$ и $G(n) = 18 * n * \ln(n)$

/* подключение основных используемых в программе библиотек*/

#include <conio.h>

#include <math.h>

#include <stdio.h>

#include <stdlib.h>

#include <iomanip.h>

main()

{

double

Fn, //F(n)

Gn, //G(n)

ATg_FG,

ATg_GF,

pi,

Nbegin, //Левая граница интервала

Nend, //Правая граница интервала

step, //Шаг изменения аргумента

phi, //Значение угла

k, //Коэффициент кратности

Delta, //Оценка «Дельта»

Theta, //Оценка «Тетта»

O_large, //Оценка «О-большое»

ii; //Значение аргумента функций трудоемкости

FILE *stream;

//Указатель на файл, в который осуществляется ввод-вывод расчетов

```

stream = fopen("Example_TA.TXT", "w+"); // открытие файла для
записи
/*Ввод значений границ интервалов, шага изменения
аргумента внутри интервала, коэффициента кратности*/

cout<<"Input Nbegin ";   cin>>Nbegin; //Левая граница, ввод
значения
cout<<"Input Nend ";   cin>>Nend;   //Правая граница, ввод
значения
cout<<"Input step ";   cin>>step;    //Шаг изменения аргумента
cout<<"Input koefficient"; cin>>k;    //Коэффициент кратности

phi = M_PI/k;           /*Определили угол изменения как  $\frac{\pi}{k}$ , M_Pi
– встроенная константа яз.С, число  $\pi = 3.1415...^*$ /

ii=Nbegin;              //Аргумент функций равен левой границе
интервала
while (ii<=Nend)
{   Fn   = 1.75*ii*ii; //Расчет значения функции F(n)
    Gn   = 18*ii*log(ii); //Расчет значения функции G(n)
    ATg_FG   = atan(Fn/Gn);
    ATg_GF   = atan(Gn/Fn);
    pi      = ATg_FG - ATg_GF;
    Delta   = phi - pi;
    Theta= fabs(pi) - phi;
    O_large = pi + phi;
    fprintf(stream, "%f %f %f %f %f %f %f %f %f\n", ii, Fn, Gn,
ATg_FG, ATg_GF, pi, Delta, Theta, O_large);
    //Запись расчетов в файл
    ii=ii+step;      //Получение следующего значения
аргумента
} //end while
fclose(stream);      //Закрыли файл
} // end main

```

6. СОДЕРЖАНИЕ ОТЧЁТА

1. Цель работы.
2. Вариант задания.
3. Текст программы, реализующей расчеты по соответствующему варианту.
4. Анализ результатов работы программы в виде таблицы результатов и графиков.
5. Развернутый вывод по работе.

7. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Определение алгоритма по Маркову, Колмогорову.
2. Основные требования, предъявляемые к алгоритму.
3. Понятие о трудоемкости алгоритма.
4. Асимптотические обозначения в анализе функции трудоемкости алгоритма.
5. Определение сложности алгоритма.
6. Основные оценки в асимптотическом анализе алгоритмов.
7. Метод интервального анализа функций трудоемкости алгоритмов, его графическая интерпретация.

ЛАБОРАТОРНАЯ РАБОТА № 2. “ТЕОРЕТИЧЕСКАЯ ОЦЕНКА СРЕДНЕГО КОЛИЧЕСТВА ОПЕРАЦИЙ ПЕРЕПРИСВАИВАНИЯ В АЛГОРИТМЕ ПОИСКА МИНИМУМА”

1. ЦЕЛЬ РАБОТЫ

Лабораторная работа посвящена экспериментальной проверке теоретической оценки трудоемкости алгоритма поиска минимума и включает ознакомление с принципами использования генератора случайных чисел для создания наборов исходных данных.

2. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

2.1. Теоретическая оценка среднего количества операций переприсваивания в алгоритме поиска минимума

Алгоритм поиска минимума последовательно перебирает элементы массива, сравнивая текущий элемент массива с текущим значением минимума. На очередном шаге, когда просматривается k -ый элемент массива, переприсваивание минимума произойдет, если в подмассиве из первых k элементов минимальным элементом является последний. В случае равномерного распределения исходных данных вероятность того, что максимальный из k элементов расположен в определенной (последней) позиции равна $1/k$.

Тогда в массиве из N элементов среднее количество операций переприсваивания максимума определяется как:

$$\sum_{i=1}^N 1/i = H_n \approx \ln(N) + \gamma, \quad \gamma = 0,57$$

Величина H_n называется n -ым гармоническим числом. Таким образом среднее значение (математическое ожидание) среднего количества операций присваивания в алгоритме поиска минимума в массиве из N элементов определяется величиной H_n (для бесконечно большого количества испытаний).

2.2. Понятие о генераторе псевдослучайных чисел

Для тестирования программ часто необходимо использовать наборы случайных чисел. Такие числа могут быть использованы и для реализации тех или иных алгоритмов. Для создания таких чисел используется генератор случайных чисел. В общем случае алгоритм называется рандомизированным (randomized), если его поведение определяется не только набором входных величин, но и значениями, которые выдает генератор случайных чисел.

На практике большинство сред программирования предоставляют в распоряжение программиста генератор псевдослучайных чисел, т.е. детерминированный алгоритм, который возвращающий числа, которые ведут себя при статистическом анализе как случайные.

3. ХОД РАБОТЫ

1. Ознакомиться с теоретическим разделом настоящих методических указаний и повторить соответствующий лекционный материал.

2. Изучить функции языка C++, использующиеся для генерации псевдослучайных чисел, привести их описание в отчете (rand, random, randomize и др.).

3. Составить структурную схему и написать программу поиска минимума в массиве сгенерированных псевдослучайных чисел.

4. Написать программу подсчета n -го гармонического числа.

5. Подсчитать количество операций переприсваивания для программной реализации поиска минимума в массиве случайных чисел. Внести изменения в соответствующую программу. Длину массива и максимальное случайное число в последовательности взять в соответствии с вариантом.

6. Сравнить практически полученное значение с теоретическим n -м гармоническим числом. Примеры основных функций приведены в приложении.

7. Сделать выводы по работе, оформить отчет, подготовить ответы на контрольные вопросы.

4. ВАРИАНТЫ ЗАДАНИЙ

Вариант	Наибольшее случайное число в последовательности	Количество элементов в массиве случайных чисел
1	100	100, 1000, 5000
2	250	150, 550, 1000
3	150	100, 500, 1000
4	950	250, 300, 400
5	200	200, 300, 500
6	800	100, 1000, 2000
7	250	300, 400, 1400
8	850	300, 450, 1000
9	200	150, 200, 1000
10	350	100, 500, 1500
11	100	300, 900, 3000
12	400	100, 200, 500
13	500	100, 1000, 1500
14	950	250, 500, 750
15	150	100, 1000, 5000
16	200	150, 550, 1000
17	750	100, 500, 1000
18	550	250, 300, 400
19	150	200, 300, 500
20	200	100, 1000, 2000

5. ПРИМЕРЫ ОСНОВНЫХ ФУНКЦИЙ, ИСПОЛЬЗУЕМЫХ В ПРОГРАММЕ

```
#include <conio.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <iomanip.h>
#include <time.h>
```

```
int vector[10];
// Создание массива из 10 псевдослучайных целых чисел величиной
от 0 до 100
// массив записывается в файл Example_TA2.TXT, на экран выводим
максимальное // целое
```

```
void create_array(int Nmax) {
    int i;
```

```

FILE *stream;
//Nmax = 10; соответствует размерности массива
stream = fopen("Example_TA2.TXT", "w+");
randomize();
cout<<"Maximal integer ";   cout<<RAND_MAX;
printf("\n%d%s\n", Nmax, " random numbers from 0 to 100");
for(i=0; i<Nmax; i++){
    vector[i]= rand() % 100;
    printf("%d\n", vector[i]);
    fprintf(stream,"%d\n", vector[i]);
}
fclose(stream);
}

```

```

void main(){

    int      i,
            N,
            min, // значение минимума
            cnt; //счетчик операций переприсваивания

    double result;

    cout<<"Input amount of numbers"; cin>>N;
    result = harmonic(N);
                // harmonic(N) – функция подсчета n-го гармонического
числа
    cout<<result;
    create_array(N);
                //генерация массива псевдослучайных чисел
    min = vector[0];
    cnt = 1;
    for (i=1;i<N;i++){
        if(vector[i]<min) {min = vector[i];      cnt++;}
    }
    printf("%s%d%s%d\n", "Minimal ", min, " Num oper ", cnt);
}

```

6. СОДЕРЖАНИЕ ОТЧЁТА

1. Цель работы.
2. Вариант задания.
3. Тексты программы, реализующих расчеты по соответствующему варианту, описания функций, используемых для генерации наборов исходных данных, структурные схемы основных функций, используемых в программе.
4. Результаты работы программы.
5. Развернутый вывод по работе.

7. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Понятие о трудоемкости алгоритма.
2. Понятие о сложности алгоритма.
3. Определение детерминированного и рандомизированного алгоритмов.
4. Определение генератора псевдослучайных чисел.
5. Описать основные функции и константы, использующиеся для создания последовательности псевдослучайных чисел в языке C++.

ЛАБОРАТОРНАЯ РАБОТА № 3.

“Экспериментальное определение количества элементарных операций языка высокого уровня в программной реализации алгоритма”

1. ЦЕЛЬ РАБОТЫ

Экспериментальная проверка теоретически полученной функции трудоемкости для алгоритма точного решения задачи о сумме методом полного перебора. Получение практических навыков расстановки счетчика операций в программе на языке высокого уровня.

2. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

2. 1. Понятие элементарных операций в формальной системе языка высокого уровня

Будем понимать под **элементарными** операциями языка высокого уровня (на примере языка C++) те, которые могут быть представлены в виде **элементарных конструкций** данного языка (но не обязательно в виде одной

машинной команды). Так, за одну элементарную операцию можно считать следующие:

- 1) операцию присваивания $a \leftarrow b$
- 2) операцию индексации массива $a[i]$
- 3) арифметические операции $*, /, -, +$
- 4) операции сравнения $a < b$
- 5) логические операции or, and, not

Неявно в операцию сравнения входит машинная команда перехода в конструкции if-then-else.

Цикл не является элементарной операцией, т.к. может быть представлен в виде:

for $i \leftarrow 1$ to N		$i \leftarrow 1$	\leftarrow
тело	\Leftrightarrow	тело	
end;		$i \leftarrow i+1$	\rightarrow
		if $i \leq N$	

Таким образом конструкция цикла требует $1 + 3 * N$ элементарных операций.

В качестве примера построения функции трудоемкости и расстановки элементарных операций рассмотрим задачу суммирования двумерного массива на псевдоязыке:

SumM(A,N;Sum);	элементарных операций в строке
Sum $\leftarrow 0$	1
for $i \leftarrow 1$ to N	$1 + 3 * N$
for $j \leftarrow 1$ to N	$1 + 3 * N$
Sum \leftarrow Sum + A[i,j];	$1(\leftarrow) + 1(+) + 1(i) + 1(j)$
end for j	
end for i	
Return (Sum);	
End;	

Таким образом $f_A(N) = 1 + 1 + N * (3 + 1 + N * (3 + 4)) = 7N^2 + 4 * N + 2 = \Theta(N^2)$.

2.2. Формулировка задачи о сумме

Словесно задача о сумме формулируется как задача нахождения таких чисел из данной совокупности, которые в сумме дают заданное число.

В терминах языка высокого уровня задача формулируется, как задача определения таких элементов исходного массива из N чисел, которые в сумме дают число V (задача относится к классу NPC).

Дано: Массив $S[i]$, $i=1, N$ и число V .

Требуется: определить S_j : $\sum S_j = V$

Получим **асимптотическую** оценку сложности решения данной задачи при использовании прямого перебора вариантов:

Поскольку исходный массив содержит N чисел, то проверке на V подлежат варианты:

- V содержит 1 слагаемое $\Rightarrow C_N^1 = N$;
- V содержит 2 слагаемых $\Rightarrow C_N^2 = (N*(N-1))/(1*2)$;
- V содержит 3 слагаемых $\Rightarrow C_N^3 = (N*(N-1)*(N-2))/(1*2*3)$;
- и т.д. до N слагаемых.

Т.е. необходимо рассмотреть сумму всех возможных **сочетаний** элементов массива из N чисел.

Любое подмножество из k элементов множества, содержащего n элементов, называется **сочетанием** из n элементов по k .

Число всех различных сочетаний из n элементов по k равно

$$C_n^k = \frac{n!}{k!(n-k)!}.$$

Для всех действительных чисел a , b , отличных от нуля, и для всех натуральных чисел n справедливо следующее соотношение

$$(a+b)^n = \sum_{k=0}^n C_n^k a^{n-k} b^k = C_n^0 a^n b^0 + C_n^1 a^{n-1} b^1 + \dots + C_n^n a^0 b^n.$$

Если в формуле (1.5) заменить b на $-b$, то получим

$$(a-b)^n = \sum_{k=0}^n (-1)^k C_n^k a^{n-k} b^k.$$

Биномиальные коэффициенты формулы C_n^k составляют в **треугольнике Паскаля** строку с номером n .

Каждый коэффициент образуется путем сложения двух стоящих над ним (слева и справа). Крайние значения известны для любого n : $C_n^0 = C_n^n = 1$. В строке с номером n слева направо стоят значения $C_n^0, C_n^1, C_n^2, \dots, C_n^n$.

Свойства биномиальных коэффициентов. Для целых $n \geq 0, k \geq 0$ справедливо свойство симметрии:

$$C_n^k = C_n^{n-k}.$$

n	C_n^k												
0	1												
1	1		1										
2	1			2	1								
3	1				3	3		1					
4	1					4	6	4	1				
5	1						5	10	10	5	1		
6	1							6	15	15	6	1	
7	1								7	21	21	7	1
\dots												

При $a = b = 1$ получаем

$$\sum_{k=0}^n C_n^k = 2^n.$$

При $a = 1, b = -1$ получаем

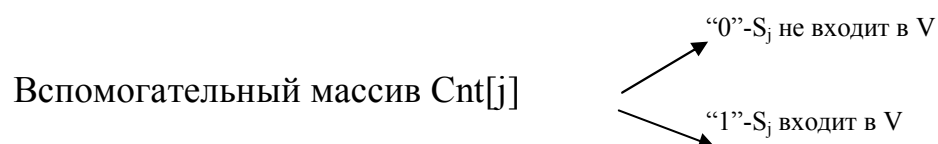
$$\sum_{k=0}^n (-1)^k C_n^k = 0.$$

Поскольку сумма биномиальных коэффициентов равна $(1+1)^n = \sum C_n^k = 2^n$ и для каждого варианта необходимо выполнить суммирование, то оценка сложности алгоритма решения задачи о сумме имеет вид:

$$f_A(N, V) = O(N \cdot 2^N).$$

2.3. Точное решение задачи о сумме

Определим вспомогательный массив Counter, хранящий текущее сочетание исходных чисел, подлежащее проверке на V



Решение получено, если $V = \sum_{j=1}^N S[j] \cdot Cnt[j]$

Условия существования решения имеют вид: $\min(S_i) \leq V \leq \sum S_i$

Могут быть предложены следующие две реализации полного перебора:

1. Перебор по $C_N^K \sum_{k=1}^N C_N^K = 2^N - 1$

2. Перебор по двоичному счётчику реализованному в массиве Cnt:

00...01

00...10

Вторая идея алгоритмически более проста и сводится к задаче о увеличении счётчика в массиве Cnt на «1».

при 00...0111 увеличение на «1» приводит к сбросу всех правых «1»

при 00...1000 увеличение на «1» приводит к переустановке «0» в «1»

Таким образом процедура перебора на псевдоязыке имеет вид:

TASKSUM(S,N,V;CNT,FL)

FL ← false

i ← 1

repeat

 Cnt[i] ← 0

 i ← i + 1

Until i > N

Cnt[N] ← 1

Repeat

 Sum ← 0

 i ← 1

 Repeat

 Sum ← Sum + S[i] * Cnt[i]

 i ← i + 1

 Until i > N

if Sum = V

 FL ← true

 Return (Cnt, FL)

j ← N

While Cnt[j] = 1

```

    Cnt[j]=0
    j←j-1
  Cnt[j] ← 1
Until Cnt[0]=1
Return(Cnt,FL)

```

2.4 Функция трудоемкости процедуры TaskSum

- а) В лучшем случае $f^*(S, N, V) = \Theta(N)$, если $V = S[N]$;
 б) В худшем случае $f^*(S, N, V) = \Theta(N \cdot 2^N)$, если решения вообще нет

$$f_A(N) = 8 \cdot N \cdot 2^N + 16 \cdot 2^N - 3 \cdot N - 12,$$

что согласуется с асимптотической оценкой.

3. ХОД РАБОТЫ

1. Изучить теоретический раздел настоящих методических указаний и повторить соответствующий лекционный материал.

2. Ознакомится с программой, реализующей алгоритм точного решения задачи о сумме – Приложение 1. Составить блок-схему алгоритма точного решения задачи о сумме.

3. Включить в функцию TaskSum строки счетчика элементарных операций в соответствии с принятой методикой.

4. Для заданных значений по варианту задания заполнить таблицу (см. Приложение 2). Рассмотреть два варианта массива – 1) заполненный вручную; 2) заполненный с помощью генератора случайных чисел (см. лабораторную работу №2).

5. Сделать выводы по работе, оформить отчет, подготовить ответы на контрольные вопросы.

4. ВАРИАНТЫ ЗАДАНИЙ

Вариант	Размерность вектора случайных чисел	Максимальное случайное число в векторе	Значение суммы (V)
1	10	100	10
2	11	80	15
3	12	70	20
4	13	90	25
5	9	25	60
6	8	35	55
7	10	40	50
8	15	60	45
9	14	50	40
10	13	45	30
11	12	55	35
12	11	100	58
13	9	65	48
14	7	85	38
15	9	75	68
16	8	100	17
17	12	90	29
18	11	50	37
19	10	40	41
20	14	30	43

6. СОДЕРЖАНИЕ ОТЧЁТА

1. Цель работы.
2. Вариант задания.
3. Тексты программы, реализующих расчеты по соответствующему варианту, описания функций, используемых для генерации наборов исходных данных, структурные схемы основных функций, используемых в программе.
4. Результаты работы программы.
5. Развернутый вывод по работе.

7. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Определение трудоемкости алгоритма.
2. Обозначения лучшего, худшего, среднего случая в анализе трудоемкости алгоритма.
3. Классификация алгоритмов по виду функции трудоемкости.
4. Перечень элементарных операций языка высокого уровня.
5. Формулировка задачи о сумме.
6. Основные комбинаторные формулы, используемые в задаче о сумме.

ЛАБОРАТОРНАЯ РАБОТА № 4.

“ОПРЕДЕЛЕНИЕ СРЕДНЕГО ВРЕМЕНИ ВЫПОЛНЕНИЯ ОБОБЩЕННОЙ ЭЛЕМЕНТАРНОЙ ОПЕРАЦИИ МЕТОДОМ ТЕСТОВЫХ ПРОГОНОВ”

1. ЦЕЛЬ РАБОТЫ

Экспериментальное определение среднего времени выполнения обобщенной элементарной операции в языке высокого уровня и получению на этой основе прогноза времени выполнения программы для больших размерностей множества исходных данных в зависимости от типов данных (целые, длинные целые, действительные).

2. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

2. 1. Понятие элементарных операций в формальной системе языка высокого уровня

Анализ трудоемкости алгоритмов позволяет получить функциональную зависимость между параметрами исходной задачи (мощностью множества исходных данных) и количеством выполненных элементарных операций в заданной среде реализации.

При этом переход от функции трудоемкости к времени выполнения программы для алгоритма с известной трудоемкостью зависит от:

- 1) времени выполнения элементарных операций в среде языка реализации алгоритма.
- 2) специфики реализации алгоритма (квалификация программиста, структуры данных и др.) на данном языке программирования.

Таким образом, имея аналитическую оценку трудоемкости данного алгоритма - $f_a(N)$ и ее экспериментальное подтверждение и зная среднее время выполнения обобщенной элементарной операции для данного языка и

данного процессора t_{cp} можно получить оценку времени выполнения программы:

$$t_a(N) = t_{cp} * f_a(N); \quad (1)$$

Построение (определение) оценки t_{cp} связано с решением следующих трех задач:

1. Определение времени выполнения элементарных операций в среде языка реализации для различных типов данных (типы операций - целые, действительные).

2. Построение средневзвешенной оценки для типов операций, на основе предположений или экспериментальных оценок по частотной встречаемости операции в рамках данного типа.

3. Построение на этой основе оценки t_{cp} с использованием весов, характеризующих особенности данного класса задач

При решении задачи 1 можно воспользоваться техническими данными процессора, только в том случае, когда языком реализации является Ассемблер.

2.2. Формулировка задачи о сумме

Словесно задача о сумме формулируется как задача нахождения таких чисел из данной совокупности, которые в сумме дают заданное число.

В терминах языка высокого уровня задача формулируется, как задача определения таких элементов исходного массива из N чисел, которые в сумме дают число V (отметим, что задача относится к классу NPC).

Формулировка:

Дано: Массив $S[i]$, $i=1, N$ и число V .

Требуется: определить S_j : $\sum S_j = V$

Получим асимптотическую оценку сложности решения данной задачи при использовании прямого перебора вариантов:

Поскольку исходный массив содержит N чисел, то проверке на V подлежат варианты:

- V содержит 1 слагаемое $\Rightarrow C_N^1 = N$;
- V содержит 2 слагаемых $\Rightarrow C_N^2 = (N*(N-1))/(1*2)$;
- V содержит 3 слагаемых $\Rightarrow C_N^3 = (N*(N-1)*(N-2))/(1*2*3)$;
- и т.д. до N слагаемых.

Поскольку сумма биномиальных коэффициентов равна $(1+1)^n = \sum_{k=0}^n C_n^k = 2^n$

и для каждого варианта необходимо выполнить суммирование, то оценка сложности имеет вид:

$$f_A(N, V) = O(N * 2^N)$$

3. ХОД РАБОТЫ

1. Ознакомиться с теоретическим разделом настоящих методических указаний и повторить соответствующий лекционный материал.

2. Изучить функции языка C++, использующиеся для генерации псевдослучайных чисел, привести их описание в отчете (rand, random, randomize и др.).

3. Составить структурную схему и написать программу поиска минимума в массиве сгенерированных псевдослучайных чисел.

4. Написать программу подсчета n -го гармонического числа.

5. Подсчитать количество операций переприсваивания для программной реализации поиска минимума в массиве случайных чисел. Внести изменения в соответствующую программу. Длину массива и максимальное случайное число в последовательности взять в соответствии с вариантом.

6. Сравнить практически полученное значение с теоретическим n -м гармоническим числом. Примеры основных функций приведены в приложении.

7. Сделать выводы по работе, оформить отчет, подготовить ответы на контрольные вопросы.

4. ВАРИАНТЫ ЗАДАНИЙ

Вариант	Наибольшее случайное число в последовательности	Количество элементов в массиве случайных чисел
1	100	100, 1000, 5000
2	250	150, 550, 1000
3	150	100, 500, 1000
4	950	250, 300, 400
5	200	200, 300, 500
6	800	100, 1000, 2000
7	250	300, 400, 1400
8	850	300, 450, 1000
9	200	150, 200, 1000
10	350	100, 500, 1500
11	100	300, 900, 3000
12	400	100, 200, 500
13	500	100, 1000, 1500
14	950	250, 500, 750
15	150	100, 1000, 5000
16	200	150, 550, 1000
17	750	100, 500, 1000
18	550	250, 300, 400
19	150	200, 300, 500
20	200	100, 1000, 2000

5. ПРИМЕРЫ ОСНОВНЫХ ФУНКЦИЙ, ИСПОЛЬЗУЕМЫХ В ПРОГРАММЕ

```

#include <conio.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <iomanip.h>
#include <time.h>

int vector[10];
// Создание массива из 10 псевдослучайных целых чисел величиной
// от 0 до 100
// массив записывается в файл Example_TA2.TXT, на экран выводим
// максимальное // целое

void create_array(int Nmax) {
    int i;
    FILE *stream;
    //Nmax = 10; соответствует размерности массива
    stream = fopen("Example_TA2.TXT", "w+");
    randomize();
    cout<<"Maximal integer ";    cout<<RAND_MAX;
    printf("\n%d%s\n", Nmax, " random numbers from 0 to 100");
    for(i=0; i<Nmax; i++){
        vector[i]= rand() % 100;
        printf("%d\n", vector[i]);
        fprintf(stream,"%d\n", vector[i]);
    }
    fclose(stream);
}

void main(){
    int    i,
           N,
           min, // значение минимума
           cnt; //счетчик операций переприсваивания

    double result;

    cout<<"Input amount of numbers"; cin>>N;
    result = harmonic(N);

```

```

        // harmonic(N) – функция подсчета n-го гармонического
числа
        cout<<result;
        create_array(N);
        //генерация массива псевдослучайных чисел
        min = vector[0];
        cnt = 1;
        for (i=1;i<N;i++){
            if(vector[i]<min) {min = vector[i];        cnt++;}
        }
        printf("%s%d%s%d\n", "Minimal ", min, " Num oper ", cnt);
    }

```

6. СОДЕРЖАНИЕ ОТЧЁТА

1. Цель работы.
2. Вариант задания.
3. Тексты программы, реализующих расчеты по соответствующему варианту, описания функций, используемых для генерации наборов исходных данных, структурные схемы основных функций, используемых в программе.
4. Результаты работы программы.
5. Развернутый вывод по работе.

ЛАБОРАТОРНАЯ РАБОТА № 5. “ПОИСК КРАТЧАЙШИХ ПУТЕЙ НА ГРАФАХ”

1 ЦЕЛЬ РАБОТЫ

Целью работы является изучение алгоритмов поиска кратчайших путей на графах на примере метода динамического программирования

2 ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

2.1 Основные определения

Путем (или ориентированным маршрутом) ориентированного графа называется последовательность дуг, в которой конечная вершина всякой дуги, отличной от последней, является начальной вершиной следующей. Так на рисунке 5 последовательности дуг $\mu_1=\{a_6, a_5, a_9, a_8, a_4\}$, $\mu_2=\{a_1, a_6, a_5, a_9\}$, $\mu_3=\{a_1, a_6, a_5, a_9, a_{10}, a_6, a_4\}$ являются путями.

Ориентированной цепью (орцепью) называется такой путь, в котором каждая дуга используется не больше одного раза. Так, например, приведенные выше пути μ_1 и μ_2 являются орцепями, а путь μ_3 не является таким, поскольку дуга a_6 в нем используется дважды.

Маршрут есть неориентированный "двойник" пути, и это понятие рассматривается в тех случаях, когда можно пренебречь направленностью дуг в графе.

Таким образом, маршрут есть последовательность ребер a_1, a_2, \dots, a_q , в которой каждое ребро a_i , за исключением, возможно, первого и последнего ребер, связано с ребрами a_{i-1} и a_{i+1} своими двумя концевыми вершинами.

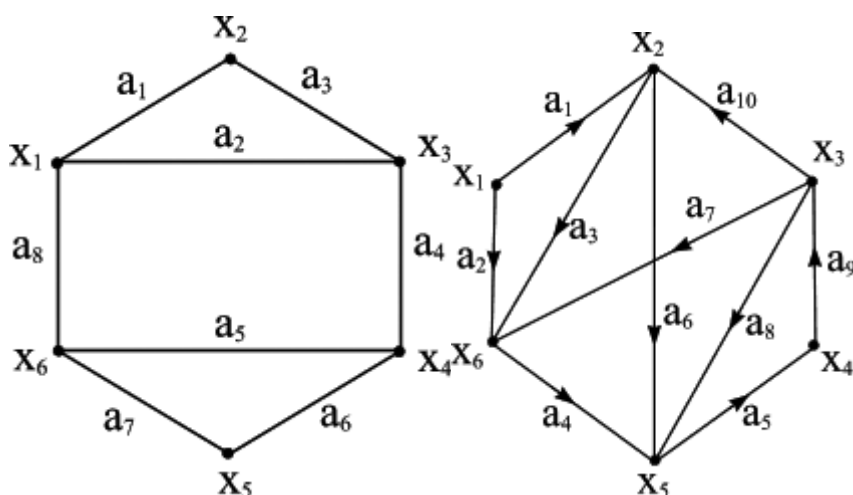


Рисунок 1 – Пример графа

Рисунок 2 – Пример графа

Последовательности дуг на рисунке 2

$\mu_4 = \{a_2, a_4, a_8, a_{10}\}$, $\mu_5 = \{a_2, a_7, a_8, a_4, a_3\}$ и $\mu_6 = \{a_{10}, a_7, a_4, a_8, a_7, a_2\}$

являются маршрутами.

Контуром (простой цепью) называется такой путь (маршрут), в котором каждая вершина используется не более одного раза. Например, путь μ_2 является контуром, а пути μ_1 и μ_3 - нет. Очевидно, что контур является также цепью, но обратное утверждение неверно. Например, путь μ_1 является цепью, но не контуром, путь μ_2 является цепью и контуром, а путь μ_3 не является ни цепью, ни контуром.

Аналогично определяется простая цепь в неориентированных графах. Так, например, маршрут μ_4 есть простая цепь, маршрут μ_5 - цепь, а маршрут μ_6 не является цепью.

Путь или маршрут можно изображать также последовательностью вершин. Например, путь μ_1 можно представить также: $\mu_1 = \{x_2, x_5, x_4, x_3, x_5, x_6\}$ и такое представление часто оказывается более полезным в тех случаях, когда осуществляется поиск контуров или простых цепей.

Иногда дугам графа G сопоставляются (приписываются) числа - дуге (x_i, x_j) ставится в соответствие некоторое число c_{ij} , называемое *весом*, или

длиной, или *стоимостью* (ценой) дуги. Тогда граф G называется *взвешенным*. Иногда веса (числа v_i) приписываются вершинам x_i графа.

При рассмотрении пути μ , представленного последовательностью дуг (a_1, a_2, \dots, a_q) , за его *вес* (или *длину*, или *стоимость*) принимается число $L(\mu)$, равное сумме весов всех дуг, входящих в μ , т. е.

$$L(\mu) = \sum_{(x_i, x_j) \in \mu} c_{ij} \quad (1)$$

Таким образом, когда слова "длина", "стоимость", "цена" и "вес" применяются к дугам, то они эквивалентны по содержанию, и в каждом конкретном случае выбирается такое слово, которое ближе подходит по смыслу.

Длиной (или *мощностью*) пути и называется число дуг, входящих в него.

2.2 Задача о кратчайшем пути

Пусть дан граф $G=(X, \Gamma)$, дугам которого приписаны веса (стоимости), задаваемые матрицей $C=[c_{ij}]$. *Задача о кратчайшем пути* состоит в нахождении кратчайшего пути от заданной начальной вершины (*источка*) s до заданной конечной вершины (*стока*) t , при условии, что такой путь существует:

Найти $\mu(s, t)$ при $L(\mu) \rightarrow \min$, $s, t \in X$, $t \in R(s)$,

где $R(s)$ - множество, достижимое из вершины s .

В общем случае элементы c_{ij} матрицы весов C могут быть положительными, отрицательными или нулями. Единственное ограничение состоит в том, чтобы в G не было циклов с суммарным отрицательным весом. Отсюда следует, что дуги (ребра) графа G не должны иметь отрицательные веса.

Почти все методы, позволяющие решить задачу о кратчайшем $(s-t)$ -пути, дают также (в процессе решения) и все кратчайшие пути от s к $x_i (\forall x_i \in X)$. Таким образом, они позволяют решить задачу с небольшими дополнительными вычислительными затратами.

Допускается, что матрица весов C не удовлетворяет условию треугольника, т. е. не обязательно $c_{ij} \leq c_{ik} + c_{kj}$ для всех i, j и k .

Если в графе G дуга (x_i, x_j) отсутствует, то ее вес полагается равным ∞ .

Ряд задач, например, задачи нахождения в графах путей с максимальной надежностью и с максимальной пропускной способностью, связаны с задачей о кратчайшем пути, хотя в них характеристика пути (скажем, вес) является не суммой, а некоторой другой функцией характеристик (весов) дуг, образующих путь. Такие задачи можно переформулировать как задачи о кратчайшем пути и решать их соответствующим образом.

Существует множество методов решения данной задачи, отличающиеся областью применимости и трудоемкостью (Дейкстры, Флойда, динамического программирования). Среди них большое распространение получили частные алгоритмы, применяющиеся при решении частных задач, и имеющие меньшую трудоемкость. Эти частные случаи встречаются на практике довольно часто (например, когда C_{ij} являются расстояниями), так что рассмотрение этих специальных алгоритмов оправдано.

На практике задачу кратчайшего пути часто требуется решать для класса ориентированных ациклических графов. Такая задача успешно решается с помощью метода динамического программирования.

2.3.1 Алгоритм Дейкстры

Данный алгоритм предложил датский исследователь *Дейкстра* в 1959 году.

Пусть требуется найти кратчайшее расстояние из вершины s в вершину f . Изначально необходимо создать массив $d[]$, в котором будет храниться для каждой вершины кратчайшее расстояние, за которое можно попасть в нее из исходной. При создании $d[s] = 0$, а все остальные элементы равны бесконечности (на практике выбирают очень большое число). Кроме того, потребуется массив $used[]$, чтобы хранить бит для каждой вершины, определяющий помечена она или еще нет. Изначально все вершины являются непомеченными.

Сам алгоритм состоит из n (n – количество вершин) итераций, на очередной итерации выбирается такая непомеченная вершина, что кратчайшее расстояние от исходной до нее минимально. То есть вершина v такая, что $d[v] = \min$ из всех вершин, которые еще не помечены (на первой итерации будет выбрана вершина s , что логично). А далее просматриваются все ребра, выходящие из вершины v , и производятся так называемые *релаксации*. Таким образом, если есть ребро v to s весом len , то будет предпринята попытка улучшить значение $d[to] = \min(d[to], d[v] + len)$. Действительно, выбирается минимальное значение между тем, что уже было определено и новым способом добраться до вершины to . На этом текущая итерация заканчивается. После окончания n итераций в массиве d будут лежать ответы - расстояния для каждой вершины. То есть в вершину f мы сможем добраться за $d[f]$ единиц расстояния.

Стоит заметить, что если до некоторых вершин добраться невозможно, то значение d для них не изменится, то есть останется равным бесконечности (тому числу, которым мы инициализировали массив).

```
void solve() {
    // n - количество вершин
    // g[n][n] - матрица смежности, g[i][j] = 0, если текущего ребра нет
    // d[n] - массив ответов
```

```

// s - стартовая вершина
// used - массив для пометок вершин
int i,j, // i-я итерация, j - для поиска минимальной
        v, // минимальная вершина
        to, // ребро из вершины v в to
        len; // длины len
d[s] = 0;
for(i = 0; i < n; i++) {
    v = -1;
    for(j = 0; j < n; j++) // поиск вершины с минимальным d[j]
        if(!used[j] && (v == -1 || d[j] < d[v]))
            v = j;
    used[v] = true; // пометка вершины
    for(to = 0; to < n; to++) {
        if(g[v][to]) {
            len = g[v][to];
            if(d[v] + len < d[to]) {
                d[to] = d[v] + len;
            }
        }
    }
}
}

```

2.3.2 Метод динамического программирования

Прямая итерация. Пусть вершины пронумерованы так, что дуга (x_i, x_j) всегда ориентирована от вершины x_i к вершине x_j , имеющей больший номер. Для ациклического графа такая нумерация всегда возможна и производится очень легко. При этом начальная вершина s получает номер 1, а конечная t - номер n .

Пусть $\lambda(x_i)$ – пометка вершины x_i , равная длине кратчайшего пути от 1 до x_j , s – начальная вершина (источник), t – конечная вершина (сток).

Шаг 1. Положить $\lambda(s) = 0$, $\lambda(x_i) = \infty$ для всех вершин $x_i \in X/s; i=1$;

Шаг 2. $i=i+1$. Присвоим вершине x_j пометку $\lambda(x_j)$, – равную длине кратчайшего пути от 1 до x_j , используя для этого соотношение

$$\lambda(x_j) = \min (\lambda(x_i) + c_{ij}) \quad (2)$$

Шаг 3. Повторить п.2. до тех пор, пока последняя вершина n не получит пометку $\lambda(t)$.

Необходимо отметить, что если вершина x_j помечена, то пометки $\lambda(x_i)$ известны для всех вершин $x_i \in \Gamma^{-1}(x_j)$, так как в соответствии со способом нумерации это означает, что $x_i < x_j$ и, следовательно, вершины x_i уже помечены в процессе применения алгоритма.

Пометка $\lambda(t)$ равна длине самого короткого пути от s до t . Сами дуги, образующие путь, могут быть найдены способом последовательного возвращения. А именно дуга (x_i, x_j) , согласно (2), принадлежит пути тогда и только тогда, когда

$$\lambda(x_j) = \lambda(x_i) + c_{ij} \quad (3)$$

Обратная итерация: начиная с вершины t , имеющей номер n , полагаем на каждом шаге x_j равной такой вершине (скажем, x_j^*), для которой выполняется соотношение (3), и так продолжаем до тех пор, пока не будет достигнута начальная вершина (т.е. пока не будет $x_j^* \equiv s$).

Совершенно очевидно, что пометка $\lambda(x_j)$ вершины x_j дает длину кратчайшего пути μ от s до x_j .

2.4 Алгоритм топологической сортировки

В некоторых случаях исходный граф является ациклическим, но имеет неправильную нумерацию – содержит дуги (x_j, x_i) , ориентированные от вершины x_j к вершине x_i , имеющей меньший номер ($j > i$). Для успешного нахождения кратчайшего пути с помощью метода динамического программирования к такому графу сначала применяется алгоритм топологической сортировки вершин.

Алгоритм топологической сортировки вершин очень простой. Он позволяет не только правильно перенумеровать вершины графа, но и определить его ацикличность.

Шаг 1. Положить $i=n$, где n – число вершин графа G .

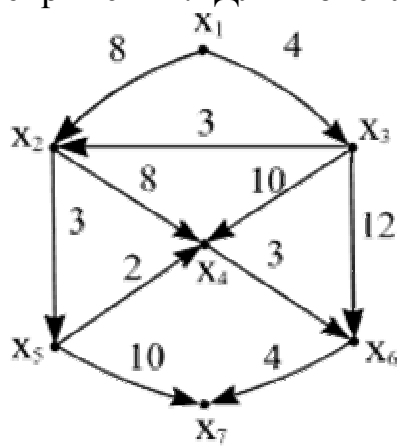
Шаг 2. В графе определяется вершина x_k , для которой выполняется условие $|\Gamma(x_k)| = \emptyset$ (т.е., вершина, из которой не выходит ни одна дуга). Вершина x_k получает порядковый номер i (перенумеруется) и исключается из дальнейшего рассмотрения вместе со всеми входящими в нее инцидентными дугами. $i=i-1$.

Шаг 3. Повторять п.2. до тех пор, пока не будет выполнено одно из условий:

1) $i=1$ – достигнута начальная вершина. Вершины графа получили правильную нумерацию.

2) Невозможно определить вершину, для которой выполнялось бы условие $|\Gamma(x_k)| = \emptyset$. В графе имеется цикл.

В последнем случае алгоритм динамического программирования неприменим. Для поиска кратчайших путей на таком графе необходимо использовать более эффективные методы, например, алгоритм Дейкстры.



2.5 Контрольный пример

Для графа, изображенного на рисунке 3,

определим кратчайший путь между вершинами x_1 и x_7 , используя метод динамического программирования.

Так как граф содержит дуги (x_3, x_2) и (x_5, x_4) , имеющие неправильную нумерацию (от большего к меньшему), необходимо перенумеровать вершины графа, применяя алгоритм топологической сортировки вершин.

В нашем случае из графа будут последовательно исключаться вершины $x_7, x_6, x_4, x_5, x_2, x_3, x_1$. Соответственно, вершины графа получают новую нумерацию, и граф будет иметь вид, представленный на рисунке 8 (старая нумерация вершин сохранена в скобках).

Рисунок 3 –
Начальный вид
графа

вершине
только
оценка для
(2), будет
вершине
 $\Gamma^1(x_3) = \{x_1,$

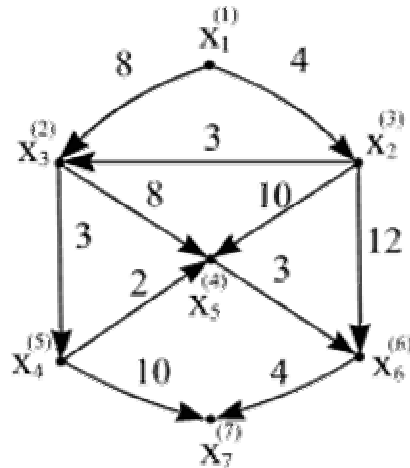


Рисунок 4 –
Итог перенумерации
перехода к
сразу три входящих дуги $(x_2, x_5), (x_3, x_5), (x_4, x_5)$. По формуле (2) определяется оценка для вершины x_5 :

$$\lambda(x_5) = \min\{4+10, 7+8, 10+2\} = 12.$$

Далее, аналогичным образом вершина x_6 получает оценку $\lambda(x_6) = \min\{4+12, 12+3\} = 15$ и, наконец, вершина x_7 получает оценку $\lambda(x_7) = \min\{10+10, 15+4\} = 19$.

Таким образом, конечная вершина x_7 пути $\mu(x_1, x_7)$ достигнута, длина пути равна $L(\mu) = 19$.

С помощью выражения (3) последовательно определяются вершины, которые входят в кратчайший путь. Перемещаясь от конечной вершины x_7 , выбирается последовательность вершин, для которой выражение (3) принимает значение «истинно»: $x_6, x_5, x_4, x_3, x_2, x_1$, т.е. кратчайший путь проходит последовательно через все вершины графа.

На первом шаге оценка $\lambda(x_1) = 0$. Осуществляется переход к x_2 . Множество $\Gamma^1(x_2)$ включает одну вершину x_1 . Следовательно, вершины x_2 определяемая по формуле $\lambda(x_2) = \min\{0+4\} = 4$. Переходим к x_3 . Для вершины x_3 множество $\Gamma^1(x_3)$ включает вершину x_1 . В этом случае, оценка будет выбираться как минимальная из двух возможных:

$\lambda(x_3) = \min\{0+8, 4+3\} = 7$. Для вершины x_4 оценка определяется снова однозначно: $\lambda(x_4) = \min\{7+3\} = 10$. Однако после вершине x_5 необходимо рассматривать

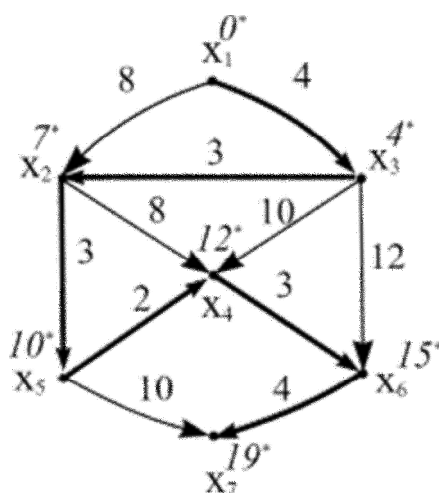


Рисунок 5 - Результат

Действительно, легко убедиться в истинности выражений:

$$\lambda(x_7) = \lambda(x_6) + c_{67}, \quad (19=15+4);$$

$$\lambda(x_6) = \lambda(x_5) + c_{56}, \quad (15=12+3);$$

$$\lambda(x_5) = \lambda(x_4) + c_{45}, \quad (12=10+2);$$

$$\lambda(x_4) = \lambda(x_3) + c_{34}, \quad (10=7+3);$$

$$\lambda(x_3) = \lambda(x_2) + c_{23}, \quad (7=4+3);$$

$$\lambda(x_2) = \lambda(x_1) + c_{12}, \quad (4=0+4);$$

После проведения перенумерации вершин графа на исходную, окончательно определяется кратчайший путь:

$$\mu(x_1, x_7) = \{x_1, x_3, x_2, x_5, x_4, x_6, x_7\}$$

Задача решена.

Результат решения в виде выделенного пути изображен на рисунке 5. Курсивом «со звездочкой» отмечены значения пометок вершин $\lambda(x_i)$.

2.6. Пример программы

```
constint INF = 1000000000; // бесконечность
bool used[100] = {0}; // массив для пометок
int top[100] = {0}; // топологический список
int g[100][100] = {0}; // матрица смежности
int n; // количество вершин
int l; // l-я вершина для добавления
int s; // стартовая вершина
int f; // конечная вершина
intd[100] = {0}; // массив ответов

intdfs(intv) {
    if(used[v])
        return 0;
    used[v] = true;
```

```

    for(int to = 0; to < n; to++)
        if(g[v][to])
            dfs(to);
    top[l++] = v; // добавление вершины v в отсортированный список
}

int topSort() {
    l = 0; // номер добавляемой вершины в отсортированный список
    for(int i = 0; i < n; i++)
        dfs(i); // запустить пробежку из всех вершин
    reverse(top, top+l); // развернуть массив
    return 0;
}

int solve() {
    int i, j;
    for(i = 0; i < n; i++)
        d[i] = INF;
    d[s] = 0;
    for(i = 1; i < n; i++)
        for(j = 0; j < i; j++)
            if(g[top[j]][top[i]])
                d[top[i]] = min(d[top[i]], d[top[j]] + g[top[j]][top[i]]);
    return 0;
}

```

3. ХОД РАБОТЫ

1. Получить задание у преподавателя в виде исходного ориентированного графа.
2. Составить структурную схему программы, определяющей кратчайший путь на графе от заданной начальной вершины s до заданной конечной вершины t с помощью метода динамического программирования.
3. Составить структурную схему программы, реализующей алгоритм топологической сортировки с произвольной нумерацией вершин графа.
4. Создать программу, реализующую метод динамического программирования и алгоритм топологической сортировки вершин. Исходный граф задается в виде матрицы смежности, вводимой построчно с помощью консоли. Указание: для определения вершин, входящих во множество $\Gamma^{-1}(x_i)$ используйте j -й столбец матрицы смежности.
5. Создать программу, которая использует приведенный в данной работе алгоритм Дейкстры для заданного графа.
6. Сравнить время выполнения двух алгоритмов.

4. СОДЕРЖАНИЕ ОТЧЁТА

1. Исходное задание и цель работы.
2. Структурная схема программы по п.2.
3. Структурная схема программы по п.3.
3. Распечатка текста программы по п.4.
4. Распечатка текста программы по п.5.
5. Контрольный пример и результаты машинного расчета.
6. Выводы по работе.

5. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Дайте определение пути, маршрута, цепи, контура.
2. Какой граф называется взвешенным?
3. Как определяется длина пути графа?
4. Задача нахождения кратчайшего пути на графе.
5. Реализация метода динамического программирования для нахождения кратчайшего пути на графе.
6. Ограничения применения метода динамического программирования для нахождения кратчайшего пути на графе.
7. Что называется правильной нумерацией вершин графа?
8. Применение алгоритма топологической сортировки для перенумерации вершин графа.
9. Алгоритм Дейкстры. Сравнение с динамическим методом.

ЛАБОРАТОРНАЯ РАБОТА № 6. “АНАЛИЗ АЛГОРИТМОВ СОРТИРОВКИ”

1. ЦЕЛЬ РАБОТЫ

Научиться оценивать сложность и количество операций для алгоритмов сортировки.

2. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

Задача сортировки последовательности богата существующими алгоритмами её решения. В данной работе предлагается рассмотреть и реализовать некоторые из алгоритмов сортировки и сравнить их производительность.

Вначале сформулируем задачу сортировки.

Определение: Задача сортировки заключается в построении алгоритма со следующими входом и выходом:

INPUT: последовательность $A = (a_1, \dots, a_n)$, сравнимых элементов некоторого множества.

OUTPUT: перестановка $A' = (a'_1, \dots, a'_n)$, в которой
 $a'_1 \leq a'_2 \leq \dots \leq a'_n$

2.1 Сортировка вставками

В данном алгоритме главной идеей является следующее: предполагается, что массив уже отсортирован до некоторого элемента с номером i , тогда просто необходимо вставить этот элемент на необходимое место.

```
void insertSort(int *a, int n) {
    int i, j;
    for(i = 2; i <= n; i++) { // установить i-й элемент на свое место
        a[0] = a[i];          // запомнить i-й элемент в нулевой ячейке
        for(j = i-1; a[j] > a[0]; j--) // пока j-ый элемент больше текущего
            a[j+1] = a[j];      // сдвигать его влево
        a[j+1] = a[0];        // на свободное место вернуть исходный
элемент
    }
}
```

2.2 Сортировка прямым выбором

Идея сортировки выбором заключается в поиске максимального элемента в неупорядоченной части последовательности и его последующего исключения путём записи в конец массива.

```
void selectSort(int *a, int n) {
    int i, j, imax;
    for(i = n; i > 1; i--) { // заполнение i-й ячейки
        imax = 1;
        for(j = 1; j <= i; j++) // поиск максимального элемента
            if(a[j] > a[imax])
                imax = j;
        swap(a[i], a[imax]); // вставка максимального элемента на
свое место
    }
}
```

2.3 Сортировка пузырьком

При сортировке пузырьком в последовательности сравниваются соседние элементы и, если они не упорядочены, меняются местами.

```
void bubbleSort(int *a, int n) {
    int i, j;
    for(i = n; i > 1; i--)
        for(j = 1; j < i; j++)    // выталкивается самый тяжелый
            if(a[j] > a[j+1])    // если он больше своего соседа
                swap(a[j], a[j+1]);
}
```

2.4 Сортировка слиянием

Данный алгоритм сортировки относится к классу «разделяй и властвуй». Массив будет делиться на две половины, а затем функция сортировки будет вызывать саму себя для сортировки каждой из половин. После этого нам необходимо совместить два уже отсортированных массива, что можно сделать с помощью двух указателей.

```
void merge(int *a, int l, int r) {
    int m = (l+r) / 2;
    int i, j, k;

    // b - промежуточный массив, объявленный как глобальная
    // переменная

    i = l; // начало первой половины
    j = m+1; // начало второй половины
    k = l;
    while(i <= m || j <= r) { // пока в массив не добавятся все элементы
        if(i > m) { // если в первой половине чисел больше нет
            b[k++] = a[j++]; // добавляется число из второй
        }
        if(j > r) { // если во второй половине чисел больше нет
            b[k++] = a[i++]; // добавляется число из первой
        }
        if(a[i] < a[j]) // если в первой половине минимальное число
            b[k++] = a[i++]; // добавляется минимальное во второй
        else
            b[k++] = a[j++];
    }
    for(i = l; i <= r; i++)
        a[i] = b[i];
}
```

```

        b[k++] = a[i++]; // добавляем число из первой
    else // иначе
        b[k++] = a[j++]; // из второй
    }
    for(i = l; i <= r; i++) // возвращение данных из промежуточного
    массива //в исходный
        a[i] = b[i];
}
void mergeSort(int *a, int l, int r) {
    if(l == r) // если один элемент, то он отсортирован
        return;
    int m = (l + r) / 2; // найти середину
    mergeSort(a, l, m); // отсортировать первую половину
    mergeSort(a, m+1, r); // отсортировать вторую половину
    merge(a, l, r); // совместить две половины, не теряя свойства
    //возрастания
}

```

2.5 Быстрая сортировка

Алгоритм быстрой сортировки заключается в следующем: выбирается некоторый элемент (в примере это будет центральный элемент, но наиболее эффективно было бы выбрать случайный), а затем массив изменяется таким образом, что в первой его части будут все элементы не большие выбранного числа, а во второй все остальные. То есть друг относительно друга две эти части будут отсортированы и нам останется только вызвать рекурсивно алгоритм для каждой из частей.

```

void quickSort(int *a, int n) {
    int i = 0, j = n; // поставить указатели на исходные места
    int temp, p;

    p = a[n/2]; // центральный элемент

    // процедура разделения
    do {
        while (a[i] < p)
            i++;
        while (a[j] > p)
            j--;

        if (i <= j) {
            swap(a[i], a[j]);
            i++;
            j--;
        }
    } while (i < j);
}

```

```

    }
}
while(i <= j);
// рекурсивные вызовы, если есть, что сортировать
if(j > 0)
    quickSort(a, j);
if(n > i)
    quickSort(a+i, n-i);
}

```

2.6 Сортировка Шелла

Данная сортировка является модификацией сортировки вставки, но с использованием шага, то есть рассматриваются не все элементы, а только те, которые находятся на расстоянии шага. Шаг постоянно уменьшается в два раза, что позволяет добиться сложности $O(\log(N))$.

```

void shellSort(int *a, int n) {
    int i, j, temp, step = n / 2; // инициализация шага
    while (step > 0) { // пока шаг не нулевой
        for (i = 0; i < (n - step); i++) {
            j = i;
            // будем идти начиная с i-го элемента
            while (j >= 0 && a[j] > a[j + step]) {
                // пока не достигнуто начало массива
                // и пока рассматриваемый элемент больше
                // чем элемент находящийся на расстоянии шага
                // элементы меняются местами
                swap(a[j], a[j+step]);
                j--;
            }
        }
        step = step / 2; // уменьшение шага
    }
}

```

2.7 Пирамидальная сортировка

Пирамидальная сортировка основывается на построение кучи(пирамиды) - бинарного дерева, в котором элемент каждого узла больше, чем элементы его потомков. После построение такого дерева несложно найти отсортированный массив используя алгоритм слияния.

```

void heapSort(int *a, int n) {
    int i, sh = 0; // смещение
    bool b = false;

```



```

for(;;) {
    b = false;
    for (i = 0; i < n; i++) {
        if( i * 2 + 2 + sh < n ) {
            if( ( a[i + sh] > a[i * 2 + 1 + sh] ) || ( a[i + sh] > a[i * 2
+ 2 + sh] ) ) {
                if ( a[i * 2 + 1 + sh] < a[i * 2 + 2 + sh] ) {
                    swap( a[i + sh], a[i * 2 + 1 + sh] );
                    b = true;
                }
                else
                    if ( a[i * 2 + 2 + sh] < a[ i * 2 + 1 + sh] ) {
                        swap( a[ i + sh], a[i * 2 + 2 + sh]);
                        b = true;
                    }
            }
            //дополнительная проверка для последних двух
элементов
            //с помощью этой проверки можно отсортировать пирамиду
            //состоящую всего лишь из трех элементов
            if( a[i*2 + 2 + sh] < a[i*2 + 1 + sh] ) {
                swap( a[i*2+1+sh], a[i * 2 +2+ sh] );
                b = true;
            }
        }
        else
            if( i * 2 + 1 + sh < n ) {
                if( a[i + sh] > a[ i * 2 + 1 + sh] ) {
                    swap( a[i + sh], a[i * 2 + 1 + sh] );
                    b = true;
                }
            }
    }
    if (!b)
        sh++; //смещение увеличивается, когда на текущем
этапе
        //сортировать больше нечего
        if ( sh + 2 == n ) break;
    } //конец сортировки
}

```

3. ХОД РАБОТЫ

1. Для каждого из приведённых алгоритмов найдите оценку для количества шагов и количества требуемой памяти.

2. Создайте структуру и реализуйте алгоритм сортировки согласно варианту задания. Критерий сортировки также указан в варианте.

3. Реализуйте более эффективные алгоритмы сортировки согласно варианту задания.

4. Сравните производительность различных алгоритмов.

4. ВАРИАНТЫ ЗАДАНИЙ

Таблица 1 – Варианты заданий для выполнения лабораторной работы

	Простейший алгоритм 1	Простейший алгоритм 2	Быстрый алгоритм 3	Структура и критерий сортировки
1	Метод «пузырька»	Сортировка вставками	Сортировка Шелла	Структура дата, содержит день, месяц и год, сортировка по возрастанию дат
2	Модифицированный метод «пузырька»	Сортировка посредством выбора	«Быстрая» сортировка	Структура дата, содержит день, месяц и год, сортировка по убыванию дат
3	Сортировка вставками	Метод «пузырька»	Сортировка слиянием	Структура Студент, содержит ФИО, курс, факультет, сортировка по возрастанию курса
4	Сортировка посредством выбора	Модифицированный метод «пузырька»	Быстрая сортировка	Структура Студент, содержит ФИО, курс, факультет, сортировка по убыванию курса
5	Метод «пузырька»	Сортировка вставками	Пирамидальная сортировка	Структура Время, содержит часы, минуты, секунды, сортировка по возрастанию времени
6	Модифицированный метод «пузырька»	Сортировка посредством выбора	Сортировка Шелла	Структура Время, содержит часы, минуты, секунды, сортировка по убыванию времени

	Простейший алгоритм 1	Простейший алгоритм 2	Быстрый алгоритм 3	Структура и критерий сортировки
7	Сортировка вставками	Метод «пузырька»	«Быстрая» сортировка	Структура Сотрудник, ФИО, ЗП. Сортировка по возрастанию ЗП.
8	Сортировка посредством выбора	Модифицированный метод «пузырька»	Сортировка слиянием	Структура Сотрудник, ФИО, ЗП. Сортировка по убыванию ЗП.
9	Метод «пузырька»	Сортировка вставками	Сортировка Шелла	Структура студент, ФИО, группа. Сортировка по ФИО.
10	Модифицированный метод «пузырька»	Сортировка посредством выбора	Пирамидальная сортировка	Структура дробь, содержит числитель и знаменатель, сортировка по возрастанию дробей.
11	Сортировка вставками	Метод «пузырька»	Сортировка Шелла	Структура дробь, содержит числитель и знаменатель, сортировка по убыванию дробей.
12	Сортировка посредством выбора	Модифицированный метод «пузырька»	«Быстрая» сортировка	Структура Продукт, название и стоимость, сортировка по возрастанию стоимости.
13	Метод «пузырька»	Сортировка вставками	Сортировка слиянием	Структура Продукт, название и стоимость, сортировка по убыванию стоимости.
14	Модифицированный метод «пузырька»	Сортировка посредством выбора	Быстрая сортировка	Структура Продукт, название и стоимость, сортировка по названию.
15	Сортировка вставками	Метод «пузырька»	Пирамидальная сортировка	Структура Товар, название, код, сортировка по возрастанию кода.

	Простейший алгоритм 1	Простейший алгоритм 2	Быстрый алгоритм 3	Структура и критерий сортировки
16	Сортировка посредством выбора	Модифицированный метод «пузырька»	Сортировка Шелла	Структура Товар, содержит название, стоимость и код, сортировка по убыванию кода.
17	Метод «пузырька»	Сортировка вставками	«Быстрая» сортировка	Структура Товар, содержит название, стоимость и код, сортировка по возрастанию стоимости.
18	Модифицированный метод «пузырька»	Сортировка посредством выбора	Сортировка слиянием	Структура Товар, содержит название, стоимость и код, сортировка по убыванию стоимости.
19	Сортировка вставками	Метод «пузырька»	Сортировка Шелла	Структура заказ, содержит день, месяц, год, а также часы и минуты заказа, сортировка по возрастанию времени заказа.
20	Сортировка посредством выбора	Модифицированный метод «пузырька»	Пирамидальная сортировка	Структура заказ, содержит день, месяц, год, а также часы и минуты заказа, сортировка по убыванию времени заказа.

5. СОДЕРЖАНИЕ ОТЧЁТА

1. Цель работы.
2. Вариант задания.
3. Текст программы, реализующей расчеты по соответствующему варианту.
4. Анализ результатов работы программы в виде таблицы результатов и графиков.
5. Развернутый вывод по работе.

6. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Для элементов каких множеств можно корректно поставить задачу сортировки?
2. Что такое стабильная сортировка?
3. Что такое беспорядок?

4. Какова сложность оптимального алгоритма сортировки в худшем случае?
5. Всегда ли существует решение задачи сортировки?
6. Единственно ли решение задачи сортировки?
7. Как работает встроенная сортировка `std::sort`

ЛАБОРАТОРНАЯ РАБОТА № 7.

“ИЗУЧЕНИЕ АЛГОРИТМОВ ДЛЯ НАХОЖДЕНИЯ НОД ЦЕЛЫХ ЧИСЕЛ”

1. ЦЕЛЬ РАБОТЫ

1. Познакомиться с алгоритмами для вычисления НОД;
2. Изучить основные правила построения алгоритмов вычисления НОД чисел;
3. Рассмотреть алгоритм Эвклида для нахождения НОД чисел.

2. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

Все задачи представленные в лабораторной работе необходимо решать на языке программирования C++.

Вариант задания для учащегося соответствует его порядковому номеру в группу (этот номер можно узнать у преподавателя).

Задачи данной лабораторной работы направлены на введение в теорию алгоритмов.

Для решения представленных задач необходимо выполнить следующие требования:

1. Повторить работу в выбранной среде на языке C++ и основные команды операционной среды.
2. Не следует писать операторы сплошным текстом. Их следует располагать структурировано.
3. Рекомендуются любую программу сопровождать комментариями, поясняющими назначение, структуру и другие особенности программы.

В лабораторной работе приводятся основные определения и факты из элементарной теории чисел.

2.1 Алгоритма Евклида

Пусть $a \in \mathbb{Z}$, $b \in \mathbb{N}$, и необходимо найти $d = \text{НОД}(a, b)$. Производится деление a на b с остатком:

$$a = q_0 b + r_0, 0 \leq r_0 \leq b$$

Если $r_0 > 0$, то b делится на r_0 :

$$b = q_1 r_0 + r_1, 0 \leq r_1 \leq r_0$$

Получается последовательность равенств с убывающими остатками r_j :

$$r_{j-2} = q_j r_{j-1} + r_j, j = 0, 1, 2 \dots (1)$$

где $r_{-2} = a$, $r_{-1} = b$. Если r_k —последний ненулевой остаток, то

$$r_{k-1} = q_{k+1} r_k$$

Тогда справедливо равенство $d = \text{НОД}(a, b) = r_k$. Вычисление последовательности (1) и есть алгоритм Евклида.

2.2 Бинарный алгоритм

На входе алгоритма заданы числа $a, b \in \mathbb{N}$, $a \geq b$. На выходе $d = \text{НОД}(a, b)$.

1 шаг. Присвоить r остаток от деления a на b . Затем $a := b$, $b := r$.

2 шаг. Если $b = 0$, то выдать a и закончить работу. Иначе $k := 0$, и затем, пока a и b оба четные, выполнять:

$$\begin{cases} k := k + 1, \\ a := a/2, \\ b := b/2. \end{cases}$$

(после этого 2^k идет в $\text{НОД}(a, b)$, оставшаяся часть НОД будет нечетной).

3 шаг. По крайней мере, одно из чисел a и b нечетное. Если a четное, то делать присвоение $a := a/2$ до тех пор, пока a не станет нечетным. То же для b .

4 шаг. a и b оба нечетные. Присвоить $t := (a - b)/2$. Если $t = 0$, то выдать $2^k a$ и закончить работу.

5 шаг. До тех пор, пока t четное, делать $t := t/2$.

6 шаг. t нечетное, $t \neq 0$. Если $t > 0$, то $a := t$. Если $t < 0$, то $b := -t$. Идти на 4 шаг.

Конец алгоритма

2.3 Примеры решения задач лабораторной работы:

Даны два натуральных числа a и b , не равные нулю одновременно. Вычислить $\text{НОД}(a, b)$ - наибольший общий делитель a и b .

```
intgcd(inta, intb) {      // функция, возвращающая НОД двух чисел
    intk = min(a, b);    // НОД не может быть больше минимума из них
    for(; a % k || b % k; k--); // перебрать все числа, пока не
    найдено//нужное
    return k;
}
```

Рассмотрим решение данной задачи с помощью алгоритма Эвклида:

Будем считать, что $\text{НОД}(0, 0) = 0$. Тогда $\text{НОД}(a, b) = \text{НОД}(a - b, b) = \text{НОД}(a, b - a)$; $\text{НОД}(a, 0) = \text{НОД}(0, a) = a$ для всех $a, b \geq 0$.

```

int gcd(int a, int b) {
    if(b < a)
        swap(a, b); // b > a
    while(a) { // пока a != 0, в этом случае b будет являться ответом
        b -= a; // уменьшить b на a
        if(a > b)
            swap(a, b); // b всегда должно быть больше a
    }
    return b;
}

```

Несложно заметить, что данный алгоритм можно улучшить, заменив $b - a$ на $b \% a$. Действительно, вычитая постоянно a , в итоге получится остаток от деления на него. В этом случае сложность алгоритма будет $O(\log(\max(a, b)))$. Данную функцию рекомендуется реализовать самим.

Кроме того, благодаря алгоритму Эвклида можно за такую же сложность найти наименьшее общее кратное - НОК(a, b). Очевидно, что $\text{НОК}(a, b) = a * b / \text{НОД}(a, b)$. Стоит также упомянуть, что $a * b$ может превысить стандартный тип данных, что необходимо учитывать.

3. ХОД РАБОТЫ

1. Ознакомиться с теоретическим разделом настоящих методических указаний и повторить соответствующий лекционный материал.

2. Разработать алгоритм нахождения НОД или НОК в соответствии с вариантом задания.

3. Реализовать алгоритм, полученный в п.2, на языке высокого уровня C++.

4. Сделать выводы по работе, оформить отчет, подготовить ответы на контрольные вопросы.

4. ВАРИАНТЫ ЗАДАНИЙ

Вариант №1.

Написать алгоритм Евклида, использующий соотношения $\text{НОД}(a, b) = \text{НОД}(a \bmod b, b)$ при $a \geq b$ и $\text{НОД}(a, b) = \text{НОД}(a, b \bmod a)$ при $b \geq a$

Вариант №2.

Написать алгоритм Евклида, использующий соотношения $\text{НОД}(a, b) = \text{НОД}(a, b \bmod a)$ при $b \geq a$

Вариант №3

Даны натуральные a и b , не равные 0 одновременно. Найти $d = \text{НОД}$

(a,b) и такие целые x и y , что $d = a*x + b*y$.

Вариант №4

Даны натуральные a и b , не равные 0 одновременно. Найти $d = \text{НОД}(a,b)$ и такие целые x и y , что $d = a*x + b*y$.

Вариант №5

Даны натуральные числа a и b , не равные 0 одновременно. Найти $\text{НОД}(a, b)$ и $\text{НОК}(a, b)$.

Вариант №6.

Даны натуральные a и b , не равные 0 одновременно. Найти $d = \text{НОД}(a,b)$ и такие целые x и y , что $d = a*x + b/y$.

Вариант №7

Написать вариант алгоритма Евклида, использующий соотношения $\text{НОД}(2*a, 2*b) = 2*\text{НОД}(a,b)$

Вариант №8

Написать вариант алгоритма Евклида, использующий соотношения $\text{НОД}(2*a, b) = \text{НОД}(a,b)$ при нечетном b

Вариант №9

Написать бинарный алгоритм, использующий соотношения $\text{НОД}(a, b) = \text{НОД}(a \bmod b, b)$ при $a \geq b$ $\text{НОД}(a, b) = \text{НОД}(a, b \bmod a)$ при $b \geq a$

Вариант №10

Даны натуральные a и b , не равные 0 одновременно. Найти $d = \text{НОД}(a,b)$ и такие целые x и y , что $d = a*x + b*y$, используя бинарный алгоритм.

5. СОДЕРЖАНИЕ ОТЧЁТА

1. Цель работы.
2. Вариант задания.
3. Текст программы, реализующей расчеты по соответствующему варианту.
4. Анализ результатов работы программы в виде таблицы результатов и графиков.
5. Развернутый вывод по работе.

6. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Определение алгоритма.
2. Алгоритм Эвклида, применение алгоритма Эвклида.
3. Сравнение сложностей алгоритма Эвклида и простого поиска НОД двух чисел.
4. Нахождение НОК(a, b) при помощи алгоритма Эвклида.
5. При каких a, b алгоритм будет работать быстрее всего? Дольше всего?

ЛАБОРАТОРНАЯ РАБОТА № 8. “ПОСТРОЕНИЕ КРАТЧАЙШИХ ОСТОВЫХ ДЕРЕВЬЕВ ГРАФА”

1. ЦЕЛЬ РАБОТЫ

Целью работы является изучение метода построения кратчайших остовых деревьев графа на примере алгоритма Прима-Краскала.

2. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

2.1 Основные определения

Одним из наиболее важных понятий теории графов является *дерево*. *Неориентированным деревом* называется связанный граф, не имеющий циклов.

Суграфом графа G является подграф G_p , содержащий все вершины исходного графа.

Если $G = (X, A)$ - неориентированный граф с n вершинами, то связный суграф G_p , не имеющий циклов, называется *остовым деревом (остовом) графа G* .

Для остового дерева справедливо соотношение:

$$G_p = (X_p, A_p) \subseteq G, \quad \text{где } X_p = X, A_p \subseteq A(1)$$

Легко доказать, что остовое дерево имеет следующие свойства:

1. остовое дерево графа с n вершинами имеет $n-1$ ребро:

$$|X_p| = |A_p| + 1;$$

2. существует единственный путь, соединяющий любые две вершины остова графа:

$$\forall x_i, x_j \in X_p (i \neq j) \rightarrow \exists! \mu(x_i, x_j)$$

Например, если G - граф, показанный на рисунке 1(а), то графы на рисунках 1(б,в) являются остовами графа G . Из сформулированных выше определений вытекает, что остова графа G можно рассматривать как минимальный связанный остовый подграф графа G .

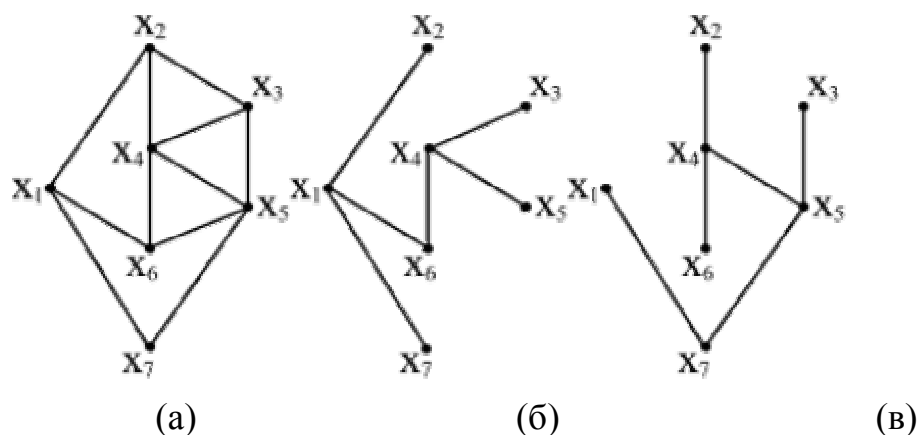


Рисунок 1 – Представление графа в виде остовых деревьев

Понятие дерева как математического объекта было впервые предложено Кирхгофом в связи с определением фундаментальных циклов, применяемых при анализе электрических цепей. Приблизительно десятью годами позже Кэли вновь (независимо от Кирхгофа) ввел понятие дерева и получил большую часть первых результатов в области исследования свойств деревьев. Большую известность получила его знаменитая теорема:

Теорема Кэли. На графе с n вершинами можно построить n^{n-2} остовых деревьев.

Ориентированное дерево представляет собой ориентированный граф без циклов, в котором полустепень захода каждой вершины, за исключением одной (вершины r), равна единице, а полустепень захода вершины r (называемой корнем этого дерева) равна нулю.

На рисунке 2 показан граф, который является ориентированным деревом с корнем в вершине x_1 . Из приведенного определения следует, что ориентированное дерево с n вершинами имеет $n-1$ дуг и связно.

Неориентированное дерево можно преобразовать в ориентированное: надо взять его произвольную вершину в

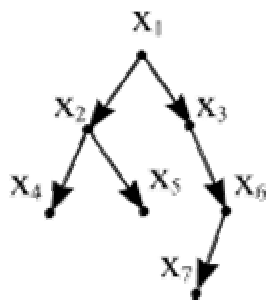


Рисунок 2 –
Ориентированное
дерево

качестве корня и ребрам приписать такую ориентацию, чтобы каждая вершина соединялась с корнем (только одной) простой цепью.

"Генеалогическое дерево", в котором вершины соответствуют лицам мужского пола, а дуги ориентированы от родителей к детям, представляет собой хорошо известный пример ориентированного дерева. Корень в этом дереве соответствует "основателю" рода (лицу, родившемуся раньше остальных).

2.2 Кратчайший остов графа

В лабораторной работе исследуется метод прямого построения *кратчайших остовых деревьев* во взвешенном графе (в котором веса приписаны дугам). Кратчайшее остовое дерево графа находит применение при прокладке дорог (газопроводов, линий электропередач и т. д.), когда необходимо связать n точек некоторой сетью так, чтобы общая длина "линий связи" была минимальной. Если точки лежат на евклидовой плоскости, то их можно считать вершинами полного графа G с весами дуг, равными

соответствующим "прямолинейным" расстояниям между концевыми точками дуг. Если "разветвление" дорог допускается только в заданных n точках, кратчайшее остовое дерево графа G будет как раз требуемой сетью дорог,

имеющей наименьший вес.

Рассмотрим взвешенный связный неориентированный граф $G = (X, A)$;

вес ребра (x_i, x_j) обозначим c_{ij} . Из большого числа остовов графа нужно найти один, у которого сумма весов ребер наименьшая. Такая задача возникает, например, в том случае, когда вершины являются клеммами электрической сети, которые должны быть соединены друг с другом с помощью проводов наименьшей общей длины (для уменьшения уровня наводок). Другой пример: вершины представляют города, которые нужно связать сетью трубопроводов; тогда наименьшая общая длина труб, которая должна быть использована для строительства (при условии, что вне городов "разветвления" трубопроводов не допускаются), определяется кратчайшим остовом соответствующего графа.

Задача построения кратчайшего остова графа является одной из немногих задач теории графов, которые можно считать полностью решенными.

2.3 Алгоритм Прима-Краскала

Этот алгоритм порождает остовое дерево посредством разрастания только одного поддерева, например X_p , содержащего больше одной вершины. Поддерево постепенно разрастается за счет присоединения ребер

(x_i, x_j) , где $x_i \in X_p$ и $x_j \notin X_p$; причем добавляемое ребро должно иметь наименьший вес c_{ij} . Процесс продолжается до тех пор, пока число ребер в A_p не станет равным $n-1$. Тогда поддерево $G_p = (X_p, A_p)$ будет требуемым остовым деревом. Впервые такая операция была предложена Примом и Краскалом (с разницей – в способе построения дерева), поэтому данный алгоритм получил название Прима-Краскала.

Алгоритм начинает работу с включения в поддерево начальной вершины. Поскольку остовое дерево включает все вершины графа G , то выбор начальной вершины не имеет принципиального значения. Будем каждой очередной вершине присваивать пометку $\beta(x_i)=1$, если вершина x_i принадлежит поддереву X_p и $\beta(x_j)=0$ – в противном случае.

Алгоритм имеет вид:

Шаг 1. Пусть $X_p = \{x_1\}$, где x_1 - начальная вершина, и $A_p = \emptyset$ (A_p является множеством ребер, входящих в остовое дерево). Вершине x_1 присвоить пометку $\beta(x_1)=1$. Для каждой вершины $x_i \notin X_p$ присвоить $\beta(x_i)=0$.

Шаг 2. Из всех вершин $x_j \in \Gamma(X_p)$, для которых $\beta(x_j)=0$, найти вершину x_j^* такую, что

$$c(x_i, x_j^*) = \min_{x_j \in \Gamma(X_p)} \{c(x_i, x_j)\}, \text{ где } x_i \in X_p \text{ и } x_j \notin X_p. \quad (2)$$

Шаг 3. Обновить данные: $X_p = X_p \cup \{x_j^*\}$; $A_p = A_p \cup (x_i, x_j^*)$

Присвоить $\beta(x_j^*) = 1$.

Шаг 4. Если $|X_p|=n$, то остановиться. Ребра в A_p образуют кратчайший остов графа.

Если $|X_p| < n$, то перейти к шагу 2.

2.4 Контрольный пример

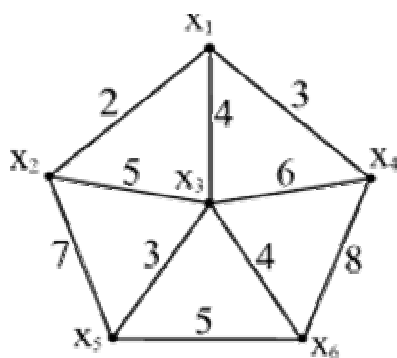


Рисунок 3 –
Пример

Для примера рассмотрим граф, изображенный на рисунке 3. Найдем для него кратчайшее остовое дерево, используя для этой цели рассмотренный выше алгоритм Прима-Краскала. Обозначим множество смежных вершин, не входящих в порожденное поддерево, как $\Gamma^*(X_p)$. Таким образом, для всех вершин, входящих в это множество, оценка $\beta(x_j) = 0$,

$\forall x_j \in \Gamma^*(X_p)$. Вектор B представляет собой множество оценок $\beta(x_i)$ для всех вершин графа G :

$\forall x_i \in X$. Длина порожденного поддерева обозначается как L .

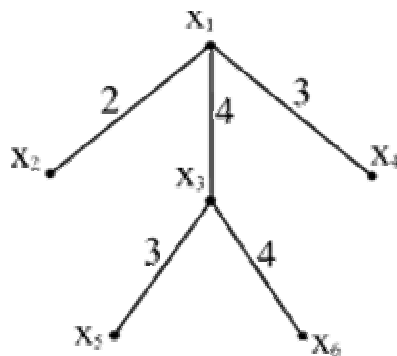
Итерация 1: $X_p = \{x_1\}$; $A_p = \emptyset$; $\Gamma^*(X_p) = \{x_2, x_3, x_4\}$; $c(x_1, x_2^*) = 2$;
 $B = \{1, 1, 0, 0, 0, 0\}$; $L = 2$.

Итерация 2: $X_p = \{x_1, x_2\}$; $A_p = \{(x_1, x_2)\}$; $\Gamma^*(X_p) = \{x_3, x_4, x_5\}$;
 $c(x_1, x_4^*) = 3$; $B = \{1, 1, 0, 10, 0, 0\}$; $L = 2 + 3 = 5$.

Итерация 3: $X_p = \{x_1, x_2, x_4\}$; $A_p = \{(x_1, x_2); (x_1, x_4)\}$; $\Gamma^*(X_p) = \{x_3, x_5, x_6\}$;
 $c(x_1, x_3^*) = 4$; $B = \{1, 1, 1, 1, 0, 0\}$; $L = 5 + 4$.

Итерация 4: $X_p = \{x_1, x_2, x_3, x_4\}$; $A_p = \{(x_1, x_2); (x_1, x_4); (x_1, x_3)\}$;
 $\Gamma^*(X_p) = \{x_5, x_6\}$; $c(x_3, x_5^*) = 2$; $B = \{1, 1, 1, 1, 1, 0\}$; $L = 9 + 3 = 12$.

Итерация 5: $X_p = \{x_1, x_2, x_3, x_4, x_5\}$; $A_p = \{(x_1, x_2); (x_1, x_4); (x_1, x_3); (x_3, x_5)\}$;
 $\Gamma^*(X_p) = \{x_6\}$; $c(x_3, x_6^*) = 4$; $B = \{1, 1, 1, 1, 1, 1\}$; $L = 12 + 4 = 16$.



Составляют кратчайшее остовое дерево:

$X_p = \{x_1, x_2, x_3, x_4, x_5, x_6\}$; $A_p = \{(x_1, x_2); (x_1, x_4); (x_1, x_3); (x_3, x_5)\}$;

Суммарная длина кратчайшего остового дерева $L = 16$.

Результат решения задачи представлен на рисунке 4.

Рисунок 4 - Результат решения

2.5 Пример программы

```

const int INF = 1000000000; // бесконечность
bool used[100] = {0}; // массив для пометок
int g[100][100] = {0}; // матрица смежности
int n; // количество вершин

```

```

int solve() {
    int i, j, v, to;
    vector<int> min_e(n, INF), sel_e(n, -1);
    min_e[0] = 0;

    for(i = 0; i < n; i++) {
        v = -1;
        for(j = 0; j < n; j++)

```

```

        if(!used[j] && (v == -1 || min_e[j] < min_e[v]))
            v = j;
    if (min_e[v] == INF) {
        cout <<"No MST!";
        exit(0);
    }
    used[v] = true;
    if(sel_e[v] != -1)
        cout << v <<" "<< sel_e[v] << endl;
    for(to = 0; to < n; to++)
        if (g[v][to] < min_e[to]) {
            min_e[to] = g[v][to];
            sel_e[to] = v;
        }
    }
    return 0;
}

```

3. ХОД РАБОТЫ

1. Получить задание у преподавателя в виде исходного неориентированного графа.

2. Составить схему алгоритма программы, определяющей кратчайшее остовое дерево графа с помощью алгоритма Прима-Краскала.

3. Создать программу, реализующую алгоритм Прима-Краскала. Исходный граф задается в виде матрицы смежности, вводимой построчно с помощью консоли. Программа должна вывести список ребер, входящих в кратчайшее остовое дерево.

4. СОДЕРЖАНИЕ ОТЧЁТА

1. Исходное задание и цель работы.
2. Структурная схема программы по п.2.
3. Распечатка текста программы по п.3.
4. Контрольный пример и результаты машинного расчета.
5. Выводы по работе.

5. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Дайте определение дерева; ориентированного дерева.
2. Какое дерево называется остовым?
3. Свойства остовых деревьев. Теорема Кэли.
4. Что называется корнем дерева?
5. Как преобразовать неориентированное дерево в ориентированное?
6. Сколько ребер содержит остовое дерево графа?

7. Задача нахождения кратчайшего остова графа.
8. Приведите практические примеры нахождения кратчайшего остова графа.
9. Реализация алгоритма Прима-Краскала для нахождения кратчайшего остова графа.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Агафонов В.Н. Сложность алгоритмов и вычислений. Новосибирск: Изд-во НГУ, 2010. – 70 с.
2. Ахо А.В., Хопкрофт Дж. Е., Ульман Дж. Д. Построение и анализ вычислительных алгоритмов. – М.: Мир, 1979. – 536 с.
3. Глушков В.М., Цейтлин Г.Е., Ющенко Е.Л. Алгебра. Языки, Программирование. – К.: Наукова думка, – 318 с.
4. Грин Д., Кнут Д. Математические методы анализа алгоритмов. – М.: Мир, 1987. – 120 с.
5. Гулд Х., Тобочник Я. Компьютерное моделирование в физике: в 2-х частях. Часть 2: Пер.с англ. - М.: Мир, 1990. –400 с.
6. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. М.: – МЦИМО, 2001. – 960 с.
7. Левитин А.В. Алгоритмы: введение, разработка и анализ. – М.: Вильячс, 2006. – 576 с.
8. Ульянов М.В., Шептунов М.В. Математическая логика и теория алгоритмов, часть 2: Теория алгоритмов. – М.: МГАПИ, 2003. – 80 с.

Заказ №		от “		“		2018		Тираж		экз.
						Изд-во СевГУ				