"Алгоритмы в школьной информатике"

Составитель:

Юрцева Светлана Сергеевна
гимназия 42

СОДЕРЖАНИЕ

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА.

Превращение программирования из удела немногих интеллектуалов в отрасль промышленности современного информационного общества предъявляет школьной информатике требования по обязательному изучению определенных глав фундаментальной информатики. Информатика в современной российской школе состоит из двух локальных составляющих:

- алгоритмизация и программирование;
- офисные и сетевые технологии.

Надо заметить, что изучение первой части с течением времени потеряло свой приоритет, хотя нельзя отрицать тот факт, что изучение программирования полезно не только с точки зрения подготовки некоторых профессиональных навыков, но и как средство развития операционного и системного мышления.

Данное методическое пособие подготовлено по материалам, используемым в курсе информатики классов профиля “Информатика - математика” и занятиях олимпийской сборной по программированию гимназии №42 г.Барнаула, ведущий игрок которой (Гозман Дима) принимал непосредственное участие в отладке текстов приведенных программ.

В пособии большое внимание уделяется оптимальности реализации задачи с точки зрения оптимальности алгоритма ее решения (что фактически не рассматривается в школьном курсе). Примеры задач реализованы на языке Паскаль, который является не только стартовым языком профессионального программирования, но и оптимален при реализации олимпиадных задач.

Разделы пособия достаточно независимы, что позволяет их изучать непоследовательно, при этом темы, рассматриваемые в отдельных разделах, имеют свои сложности. Как показывает опыт, при использовании данных материалов следует соблюдать принцип дидактической спирали: изучать разделы не целиком, а частично – последовательно, т.е. возвращаться к каждому разделу всякий раз на более глубоком уровне.

Строение всех разделов одинаково: небольшие теоретические выкладки, пояснения их на примерах и задачи для самостоятельного решения.

Методическое пособие предназначено как для преподавателей в качестве материалов к уроку или дополнительным занятиям, так и для самостоятельной работы увлекающихся программированием школьников.

§ 1 ВВЕДЕНИЕ В КОМПЬЮТЕРНУЮ АЛГОРИТМИКУ

Алгоритм должен быть определен
настолько четко, чтобы его указаниям
мог следовать даже компьютер.
Дональд Э.Кнут

Предметом обсуждения данного раздела станут алгоритмы, и следует дать формальное определение этого понятия. В современной научной и любой другой литературе, оперирующей понятием "информация", очень много определений алгоритма, но далеко не все они могут быть применимы в области программирования, поэтому за основу возьмем представление об алгоритме как описание некоторого вычислительного процесса и введем некоторые уточнения.

Форма записи алгоритма (формат) и его содержание не произвольны, а подчинены определенным ограничениям. Форматы известны различные: словесное описание, графические (блок-схема или диаграммы Насси-Шнейдермана).

В данном учебном пособии предпочтение отдано "первичной" форме – словесному описанию с нумерацией шагов и дополнительными комментариями.

Определяющей особенностью вычислительного процесса является возможность расчленить его на отдельные, дискретные действия. Второй особенностью является последовательность действий процесса, оформленных как предписание алгоритма. Предписаний должно быть конечное число, каждое из них должно быть точным и не допускать неопределенного толкования. Точное предписание вызывает шаг алгоритма. Весь процесс, включающий все шаги от начала до завершения, должен быть конечен. Процесс должен преследовать конкретную цель, которая, в свою очередь, должна быть достижимой. Анализ алгоритма позволяет оценить, как велико число шагов. Представить алгоритм без входных и/или выходных данных довольно затруднительно: зачем он тогда нужен? Кроме входных и выходных данных, алгоритм, как правило, предусматривает временное формирование промежуточных данных, которые вновь поступят на обработку. Поэтому при конструировании алгоритма необходимо строго определить каждый его шаг, предусмотрев возможные состояния процесса и соответствующие инструкции для их обработки. Только такой алгоритм при неоднократном применении к одинаковым входным данным всегда приведет к одному итогу.

В противоположность детерминированному (зависящему от входных данных), в алгоритме стохастическом заложена некоторая неопределенность в выборе очередной инструкции. Выбор конкретной инструкции происходит на основе вероятностного механизма, т.е. разработчик планирует, что

независимо от выбранного продолжения, конечный результат будет удовлетворять условиям поставленной задачи.

АНАЛИЗ АЛГОРИТМА

> Поразительно, скольким программистам приходится слишком дорогим способом выяснять,что их программа не может обработать входные данные раньше, чем через несколько дней машинного времени. Лучше было бы предугадать такие случаи с помощью карандаша и бумаги.
>
> S.E. Goodman,
> S.T.Hedetniemi

Анализ алгоритма должен дать четкое представление о емкостной и временной сложности алгоритма.

Емкостная сложность - это размеры памяти, в которой предстоит размещать все данные, участвующие в вычислительном процессе. К ним относятся: входные данные, промежуточные и выходная информация. Возможно, что не все перечисленные наборы требуют одновременного хранения (возможно применение динамических структур), что позволяет экономить затраты памяти.

Вопрос о временной трудоемкости алгоритма гораздо сложнее. Пусть поставлена некоторая задача и спроектирован алгоритм ее решения, который описывает вычислительный процесс, завершающийся за конечное число действий - шагов.

Реальное время выполнения каждого отдельного шага зависит от конкретного вычислительного устройства, т.е. непосредственным участником вычислительного процесса (но не алгоритма!) является ПК (быстродействие и т.д.).

А как зависит время выполнения программы от построенного алгоритма?

Как известно, решение одной задачи может быть реализовано минимум несколькими алгоритмами. Нас интересует тот, который в сравнении с конкурентами, нуждается в наименее продолжительном по времени вычислительном процессе.

Скорость реализации выбираемого алгоритма может существенно зависеть от содержания набора входных данных. Быстрый "в среднем" механизм способен давать сбои в отдельных "плохих" случаях. В этом случае предпочтение отдается медленному в "среднем", но надежному в худших ситуациях алгоритму.

Давая оценку быстродействия алгоритма, следует рассмотреть поведение вычислительного процесса в среднем и, отдельно, в экстремальных для него условиях.

Моделирование "плохих" случаев всегда связано с содержанием самого алгоритма:

- проверка поведения алгоритма на "границе" диапазона входных данных;
- проверка на максимально больших по объему входных данных.

Умение предвидеть "нехорошие" ситуации - отличие квалифицированного алгоритмиста от обыкновенного "кодера".

В настоящее время в связи с превращением программирования в промышленную индустрию сформировалась специализация "тестеры программ", которые занимаются составлением таких тестов, чтобы алгоритм прошел через "огонь и воду".

Временная эффективность алгоритма не связана с качествами определенного ПК. Речь идет о количестве шагов алгоритма, каждый из которых реализуется некоторым числом машинных операций. Можно привести такое сравнение: человек, использующий плохой алгоритм, подобен повару, отбивающему мясо отверткой: едва съедобный и малопривлекательный результат достигается ценой больших условий. Поэтому анализ алгоритма требует такой детализации алгоритма, чтобы в отношении отдельного шага не требовалась его дальнейшая алгоритмическая проработка. Возможны две ситуации: либо фиксированное время такого шага определено некоторым набором простых (без циклов) команд языка программирования, либо речь идет об "укрупненном" шаге, в отношении которого соответствующий анализ уже проводился и результаты известны.

ПРИМЕР

Алгоритм обмена двух переменных A и B реализуется за 3 шага, независимо от того, к какому типу простых переменных он применяется. С точки зрения количества машинных операций, две разные ситуации - обмена содержимым между переменными, занимающими одно или два машинных слова - неравноценны. Но оценка алгоритмической трудоемкости это не учитывает.

Временем работы алгоритма называется число элементарных шагов, которые он выполняет. Что считать элементарным шагом? Одна строка псевдокода (простой оператор) требует не более чем фиксированного числа операций.

Существует специальный механизм определения верхней оценки временной трудоемкости алгоритма (O(f(n)), что означает, что число операций алгоритма не более f(n)).

Если алгоритм связан с обработкой N входных элементов и нет формулы для быстрого вычисления результата, то достижение

эффективности O(n) (линейная функция), следует рассматривать как большой успех.

ПРИМЕР

Известная задача о количестве счастливых шестизначных (N- значных) билетов имеет минимум два решения:

1. Для перебора всех шестизначных билетов для каждой цифры можно организовать свой цикл. Тогда временная трудоемкость алгоритма (шесть вложенных циклов) будет порядка O(N6).
2. Единственный цикл перебора всех билетов с определением каждой цифры числа. Определение цифр числа:
   - - число единиц в числе A равно A mod 10,
   - - число десятков в числе A равно (A div 10) mod 10 и т.д.

В числе, состоящем из 6 цифр, количество операций не превысит 6*(6+2), где 6- число операций определения цифр, а 2- операции сравнения сумм цифр.

Для N цифр: - трудоемкость данного алгоритма < O(N).

Уделяя внимание возможности оценки временной эффективности алгоритма, стоит поискать ответ на вопрос: а насколько это нужно практически?

В самом деле, вычислительный процесс должен выполняться на современном ПК, который работает "довольно быстро". Предположим, что в нашем распоряжении процессор, который способен осуществлять десяток миллионов - 107 - микроинструкций за секунду (mips), а спроектированный алгоритм требует, в среднем, по десять mips на каждый свой шаг. Очевидно, что одной секунды машинного времени хватит на миллион - 106 - алгоритмических шагов вычислительного процесса!

ПРИМЕР

В классе N учеников. Рассмотреть все возможные способы рассадки всех N учеников на N местах.

Обычная переборная задача. Для N=10 число вариантов N! > 3.5млн. "Хороший" ПК (очень хороший!) справится, предположим, за 1 секунду. Но для N=15 времени потребуется в 360360 раз больше для этого же ПК или более 4 суток непрерывного счета, другими словами, применять алгоритм с трудоемкостью O(N!) надо очень осторожно!

Одним из подходов к решению переборной задачи заключается в том, что в зависимости от условия, алгоритм предусматривает некоторое сокращение полного перебора, тем самым искусственно понижая его трудоемкость.

Другой подход при конструировании алгоритма состоит в том, чтобы решение исходной, сложной задачи свести к поочередному решению более простых подзадач, т.к. совокупная трудоемкость решения подзадач намного меньше, чем общей задачи. Этот механизм называют методом декомпозиции задачи и композиции решения.

Методы, основанные на применении обоих указанных подходов, относятся к т.н. точным алгоритмам, гарантирующим получение искомого решения. Назвать универсальными эти методы нельзя, т.к. существует много типов задач, имеющих экспоненциальную сложность (порядка O(NN)), что не позволяет "дорешать" их за разумное время.

В таких случаях практический выход состоит в получении неточного решения - т.н. приближенные алгоритмы - когда разница между двумя последовательными приближенными решениями становится меньше требуемой точности. Примером использования таких алгоритмов могут быть задачи вычисления числа Pi или вычисление определенного интеграла.

К эвристическим алгоритмам относятся алгоритмы, позволяющие найти некоторое, заведомо неточное, но удовлетворительное решение задачи. Но в данном случае нельзя говорить о степени "похожести" результата на истинное решение. Эвристический алгоритм дает "усеченное" решение путем отсеивания "малосущественных" условий из постановки задачи. Такое решение и принимается как подходящее.

§ 2 ПРИНЦИПЫ ПРОВЕРКИ УЧЕБНЫХ И ОЛИМПИАДНЫХ ЗАДАЧ ПО ИНФОРМАТИКЕ

К программе, реализующей некоторый алгоритм, можно предъявить разнообразные требования: структурная запись программного кода, наличие комментариев и т.д.

Но, в настоящее время, в программировании для проверки правильности работы алгоритма и четкости его работы с входными/выходными данными составляются тесты, осуществляющие проверку всех его качеств.

Тесты разделяются на группы:

1. Первый тест должен быть максимально прост, т.к. его цель - проверить, работает ли программа вообще. Обычно в качестве первого теста используется тест из условия задачи, который показывает верное понимание учеником условия задачи и соответствие программы форматам входных и выходных данных.
2. Вторая группа тестов должна отслеживать т.н. вырожденные случаи:
   - случаи, когда решение задачи или не существует (в условии задачи должно быть сказано, что должна сообщать программа в такой ситуации);
   - случаи, коренным образом отличающиеся от основного алгоритма решения, например, нулевые значения для числовых входных данных (при допустимости таких значений в условии), для текстовых - пустой входной файл, либо последовательность из пробелов и /или символов перевода строки;
   - нарушение общности входных данных, требующих специальной их обработки (например, для квадратного уравнения равенство

нулю одного из коэффициентов приводит к рассмотрению отдельных случаев решения уравнения);

3. Следующая группа тестов должна проверять граничные случаи, т.е. когда входные и выходные данные принимают граничные значения. Назначение подобных тестов – обнаружить возможные программистские ошибки при реализации даже правильного алгоритма (проверка выбора типов данных, максимальной точности вычислений, выхода за границу массива данных, корректности работы с динамической памятью и т.д.).
4. Группа тестов, проверяющая правильность алгоритма решения задачи в целом:
   - общие тесты, которые проверяют все ветви логической схемы алгоритма ("испытание ветвей") - например, при использовании алгоритмов на графах программа проверяется на данных для связанного и несвязанного графа, графу без циклов, граф с пустым множеством ребер и т.д.;
   - тесты специального вида, которые проверяют работоспособность программы в случае специальной организации входных данных (например, на входе - данные отсортированы, хотя это и не обязательно, а алгоритм решения предусматривает сортировку данных).
5. При применении эвристических алгоритмов необходимы тесты, проверяющие верность/неверность приближенных вычислений.
6. Тесты, проверяющие эффективность используемых алгоритмов. Упрощая алгоритм или его реализацию, учащиеся могут неоправданно увеличивать вычислительную сложность алгоритма, что делает его непригодным для реализации (например, непрохождение по времени алгоритма полного перебора при больших данных).
7. Случайные тесты: максимально допустимый объем входных данных. Данные тесты пишутся с помощью т.н. генератора и для получения выходных данных необходима специальная программа, по сложности не уступающая проверяемой программе.

Пример

Решить квадратное уравнение a*x^2+b*x+c=0, если 0<=a,b,c<=255.

Входной файл: через пробел A,B,C
Выходной файл: “Нет решения”, если их нет, или значения решений через пробел, если они есть.

Тесты:

Input.txt	Output.txt
1 2 1	-1
0 0 0	0
0 1 0	0
0 0 1	Нет решения

127 254 127	-1
10 1 255	Нет решения
2 8 8	-2
5 7 3	Нет решения
1 5 6	-2 -3
13 0 3	Нет решения
78 134 15	-0.028 -1.689

§ 3.ЧИСЛА

Данный раздел рассматривает задачи, требующие от учащихся реализации фрагментов из теории чисел в программировании. Оптимизация таких задач возможна за счет различных возможностей языка программирования, например, стандартных функций, знания некоторых уникальных математических теорем и формул и алгоритмов, определенных на разных диапазонах рассматриваемых чисел.

3.1 ЦИФРЫ ЧИСЛА

Ранее упоминалось об алгоритме определения каждой цифры числа, который можно использовать при решении следующих задач:

- найти сумму цифр числа.
- составить программу, проверяющую, является ли заданное натуральное число палиндромом, т.е. числом, десятичная запись которого читается одинаково слева направо и справа налево.

Программная реализация:

```
uses crt;
type TArr=array[1..10]of byte; {Тип для хранения цифр числа}
var m:TArr;  {Цифры}
   c:byte;  {Их количество}
   x,sum:longint;

procedure make_digits(x:longint);{Записывает в m цифры X,в sum-сумму}
begin
  c:=0;
  sum:=0;
  while x>0 do
   begin
     inc(c);
     m[c]:=x mod 10;  {Очередная цифра числа}
     x:=x div 10;
     sum:=sum+m[c];   {Сумма цифр числа}
```

```
    end;
  end;

  function is_polin(const a:TArr;c:byte):boolean; {проверка числа на
“палиндром”}
  var i:byte;
  begin
    is_polin:=false;
    for i:=1 to (c div 2) do
      if a[i]<>a[c+1-i] then exit;
    is_polin:=true;
  end;

  begin
    clrscr;
    writeln('Введите число:');
    readln(x);
    make_digits(x);
    writeln('Количество цифр ',c);
    writeln('Сумма цифр      ',sum);
    if is_polin(m,c) then
      writeln('Число - палиндром')
    else
      writeln('Число - не палиндром');
    readkey;
  end.
```

ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОГО РЕШЕНИЯ УЧАЩИМИСЯ

1. Дано натуральное N. подсчитать количество цифр данного числа.
2. Найти количество четных цифр числа.
3. Найти самую большую цифру числа.

3.2 ПРОСТЫЕ ЧИСЛА

Определение: натуральное число N (N>1) называется простым, если его делителями являются только оно само и единица.

Для определения «простоты» числа (N в пределах типов BYTE и WORD) можно воспользоваться следующими достаточно простыми и эффективными алгоритмами:

Алгоритм 1

Если N- четное, то оно не простое (составное). Следует проверить, существует ли хотя бы один делитель числа N (числа 3,5,7,.....SQRT(N)-

проверять до N-1 и даже до N/2 нет необходимости). Если хотя бы для одного числа ответ будет положительным, то число N- составное. Заметим при этом, что число 2 - простое.

Программная реализация:

```
Procedure PROST (N: WORD);
var       I: word;
        SIMP: boolean;
begin
if N mod 2=0
      then SIMP:=false                          {число составное}
      else  SIMP:=true;
  I:=3;
  while I<=Int(Sqrt(N)) and SIMP do
      if N mod I=0                               {I- делитель N}
           then SIMP:=False                     {число составное}
           else I:=I+2;
 PROST:=SIMP;
end.
```

Алгоритм 3. **"Решето Эратосфена"**

Из множества чисел от 2 до N отбросим все числа, кратные первому простому, то есть двойке. Наименьшее оставшееся число является вторым простым числом. Отбросим все числа кратные этому простому числу. Наименьшее оставшееся число является третьим простым числом и т.д.

Описание алгоритма:

1) Подготовим к работе необходимые массивы:

A - будем содержать:
   0- если число не простое.
   1- если число простое.

K- будем содержать:
   очередные числа, которые нужно
   вычеркнуть для каждого i ого простого числа.
   Если I- ое число в массиве равно 0
   значит число I не простое

2) Найдем все простые числа из интервала от 1 до корня из N, где N- число до которого нужно найти простые числа.
3) Пусть M+1 – число, с которого начинается очередной массив A. вычеркнем из этого массива все числа кратные простым числам, эти числа мы будем брать из массива K (в ячейках этого массива для каждого простого числа хранятся числа, которые надо вычеркнуть (кратные простому числу)).
4) Распечатаем простые числа.
5) Увеличим переменную M на корень из N.

6) Перейдем на пункт 3.

Программная реализация:

```
uses crt;
const max=32767;
var is:array[1..max]of boolean;
   m,x,i,k,n:longint;
begin
 clrscr;
 writeln('Введите M и N');
 readln(m,n);
 if (m>n)or(n>=max) then
  begin
   writeln('Неправильные числа!');
   halt;
  end;
 writeln('Простые числа от ',m,' до ',n,':');
 while keypressed do readkey;
 readkey;
 fillchar(is,sizeof(is),true);
 is[1]:=false;
 i:=2;
 while (i<=n) do
  if is[i] then
   begin
    if wherex>=75 then writeln;
    if (i>=m) then
     write(i,',');
    x:=(n div i)-1;
    for k:=1 to x do
     is[i+k*i]:=false;
    inc(i);
   end
  else inc(i);
 gotoxy(wherex-1,wherey);
 write(' ');
 while keypressed do readkey;
 readkey;
end.
```

Для чисел типа Longint приведенные выше алгоритмы являются не достаточно эффективными, т.к. требуют выполнения слишком большого количества операций.

Алгоритм 4.

```
{Для каждого нечётного кандидата “на простоту” достаточно проверить делимость на предыдущие простые числа}
uses crt;
{$M 65520,0,0}
var n,i,j,pc:longint;
   p:array[1..10000]of longint;
   is:boolean;
begin
 clrscr;
 writeln('Введите N');
 readln(n);
 writeln('Простые числа до ',n,':');
 readkey;
 pc:=1;
 p[1]:=2;
 i:=3;
 write('2,');
 while (i<=n)do
  begin
    is:=true;
    for j:=1 to pc do
      if (i mod p[j])=0 then begin is:=false; break; end;
    if is then
      begin
        inc(pc);
        p[pc]:=i;
        write(i,',');
      end;
    inc(i,2);
  end;
 gotoxy(wherex-1,wherey);
 write(' ');
 readkey;
end.
```

ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОЙ РЕАЛИЗАЦИИ УЧАЩИМИСЯ

Задача 1.

Реализовать алгоритм определения «простоты» числа N: число является простым, если у него всего два делителя: 1 и N.

Задача 2.

Вывести все простые числа P в указанном интервале (A;B).

Задача 3.

Дано натуральное число N. Выяснить, имеются ли среди N, N+1,....,2N близнецы, т.е. простые числа, разность между которыми равна 2.

3.2.1 ПРОСТЫЕ ДЕЛИТЕЛИ

Рассмотрим две задачи:

- Задано натуральное число N. Найти и напечатать все его простые делители.
- Дано натуральное число n. Получить все натуральные числа, меньшие n, и взаимно простые с ним.

Алгоритм:

Взаимно простыми называются числа, наибольший общий делитель которых равен единицы. Поэтому для решения задачи необходимо рассмотреть все числа m=1, 2, ..., n-1 и определить НОД чисел n и m. Если его величина 1, то число m - взаимно простое с числом n.

Программная реализация:

```
var
     N, M, A, B, R, NOD: word;
begin
   Write(' Введите число N ');
   Readln(n);
   Write(' Числа, взаимно простые с числом N: ');
   for M:=1 to N-1 do
      begin
         { Определяем НОД M и N  по алгоритму Евклида:}
         A:=M;
         B:=N; {используем "аналоги" чисел M и N}
         while B>0 do
            begin
               R:=A mod B;
               A:=B;
               B:=R;
            end;
         NOD:=A;
         if NOD=1 {число m - взаимно простое с n}
            then write(m,' ')
      end;
end.
```

3. 3 ЧИСЛА ФИБОНАЧЧИ

Числа Фибоначчи определяются следующей рекурентной формулой: F(0)=1, F(1)=1, F(n, n>1)=F(n-1)+F(n-2);

В задачах работа с числами Фибоначчи ограничена типом Longint, т.к. последующие числа требуют использования т.н. «длинной арифметики».

Номер последнего числа Фибоначчи в типе Longint равен 45.

Задача1.

Вычислить число Фибоначчи по его номеру.

Программная реализация:

```
uses crt; var i,n:integer;     f:array[0..45]of longint;
begin
  clrscr;
  writeln('Введите N');
  readln(n);
  f[0]:=1;
  f[1]:=1;
  for i:=2 to n do
    f[i]:=f[i-1]+f[i-2];
  writeln('F(',n,')=',f[n]);
  readln;
end.
```

Задача 2.

Вычислить номер заданного числа Фибоначчи.

Программная реализация:

```
uses crt; var i,n:integer;
    f:array[0..45]of longint;
begin
  clrscr;
  writeln('Введите F(n)');
  readln(n);
  f[0]:=1;
  f[1]:=1;
  if n=1 then i:=1 else
    for i:=2 to 45 do
      begin
        f[i]:=f[i-1]+f[i-2];
        if f[i]=n then break;
      end;
  if (i=45)and(n<>f[45]) then writeln('Это не число Фибоначчи!') else
    writeln('n=',i);
  readln;
end.
```

Задача 3.*

Разложить произвольное число N в виде суммы чисел Фибоначчи: N=F(i1) +F(i2)+F(i3)+…+F(iK); iM>i(M-1)+1 для любого M.
(Доказано, что Fm =Fm-2+Fm-1)

Программная реализация:

```
uses crt;
var f:array[0..45]of longint;
```

```
    i,n:longint;
begin
  clrscr;
  f[0]:=1;
  f[1]:=1;
  for i:=2 to 45 do
    f[i]:=f[i-1]+f[i-2];
  i:=45;
  writeln('Введите n');
  readln(n);
  writeln('Его разложение в числа Фибоначчи:');
  write(n,'=');
  while n<>0 do
    begin
      while f[i]>n do
        dec(i);
      n:=n-f[i];
      if n<>0 then
        write(f[i],'+') else
        write(f[i]);
    end;
  readln;
end.
```

ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОЙ РЕАЛИЗАЦИИ УЧАЩИМИСЯ

1. «Фибина заначка»

Жил да был бухгалтер Фиба со стажем работы N лет. Честный был да грамотный. Грамотный – потому что знал, что если не воровать, то заподозрят, что скрывается от честного народа, а сам загребает несчитано-немерено. А честный – потому что воровал немного, каждый год по К условных единиц.

А в той стране, где он жил, была инфляция. Поэтому деньги и измеряли в условных единицах. Ну и Фиба, стало быть, доход свой нелегальный с точки зрения никому не нужного закона так же измерял.

А как пришла к Фибе старость, решил он свою заначку потратить на шестисотый мерс, который продавала фирма за б.е. – осталось только пересчитать в безусловные единицы. Попросил он у парней в фирме разрешения посидеть за плохоньким ПК, да и написал нехилую программку на Паскале, которая и перевела его заначку в б.е. – Фиба–то давно подметил, что инфляция у него на родине растет по тому же закону, что и числа Фибоначчи у математиков: 1,1,2,3,5,8,13,21…..

Переведите-ка и вы Фибину заначку в б.е.

Input.txt: содержит два целых числа K и N . K<2000, N<400.

Output.txt: Целое число – результат программы Фибы.

2. Числа вида 2^P-1, где P - простое число, называются числами Мерсенна. Являются ли числа Мерсенна при значениях P 2,3,5,7,11,13,17,19,23,29,31 простыми?

Программа состоит из двух частей: в первой вычисляется число Мерсенна для значения P (вводится с клавиатуры), во второй проверяется, является ли оно простым.

3. Совершенным называется число, равное сумме всех своих делителей, меньших, чем оно само. Например, 6=1+2+3, 28=1+2+4+7+14.

Составить программу проверки заданного натурального числа на совершенство.

4. Автоаморфным называется число, равное последним цифрам своего квадрата. Например, 5*5=25, 25*25=625. Очевидно, что автоаморфные числа должны оканчиваться либо на 1, либо на 5, либо на 6. Составить программу нахождения аморфных чисел, не превышающих значения 999.

5. Кубические автоаморфные числа равны последним цифрам своих кубов. Например: 6*6*6=216. Составить программу нахождения двузначных и трехзначных кубических автоаморфных чисел.

6. Натуральное число из N цифр является числом Армстронга, если сумма его цифр, возведенных в N-степень, равна самому числу. Например: 153=1*1*1+5*5*5+3*3*3. Получить все числа Армстронга, состоящие из трех и четырех цифр.

7. Найти все тройки пифагоровых чисел, т.е. целых k,l,m таких, что k*k+l*l=m*m, при условии, что все три числа не превышают 32767.

§4 СОРТИРОВКА И ПОИСК

Алгоритмов сортировки настолько много, что им можно посвятить целиком все методическое пособие; при этом следует учитывать, что разные по тематике используемых алгоритмов задачи требуют использования разных алгоритмов сортировки.

В данном разделе рассматриваются некоторые алгоритмы сортировки и поиска с точки зрения их оптимальности и применения. Сравнительный анализ алгоритмов – неотъемлемая часть при обучении программированию.

4.1 ПОНЯТИЕ СОРТИРОВКИ

Сортировка - это упорядочение данных по определенному признаку.

Рассмотрим одномерный массив целых чисел:

```
Const   N=... ;
Type MyArray=Array[l..N] Of Integer;
Var A:MyArray;
```

Алгоритмы сортировки отличаются друг от друга степенью эффективности, под которой понимается количество сравнений и количество обменов, произведенных в процессе сортировки. Эффективность оценивается количеством операций сравнения (порядком этого значения).

Элементы массива можно сортировать:

- *по неубыванию* — каждый следующий элемент не меньше предыдущего, в случае равенства элементов они перечисляются подряд

  A[1] <= A [2] <= ... <= A[N];

- *по невозрастанию* — каждый следующий элемент не больше предыдущего, в случае равенства элементов они перечисляются подряд

  A [1] >= A [2] >= ... >= A [N].

Известно достаточно много алгоритмов сортировки, эффективность которых совпадает и равна $O(N^2)$, где N – число элементов массива.

Рассмотрим в сравнении эффективности два алгоритма сортировки: общеизвестный «Метод пузырька» и «быстрый», быстродействие которых очень сильно различаются, что немаловажно при решении больших олимпиадных задач и прикладных задач практического программирования.

4.1.1 СОРТИРОВКА ПРОСТЫМ ОБМЕНОМ

(метод "пузырька")

Название это происходит от образной интерпретации, при которой в процессе выполнения сортировки более "легкие" элементы (элементы с заданным свойством) всплывают на "поверхность".

Рассмотрим идею метода на примере. Отсортируем по возрастанию массив из 5 элементов: 5 2 8 4 9. Длина текущей (неупорядоченной) части массива — (N — k + 1), где k — номер просмотра,

i — номер первого элемента проверяемой пары,

N — k - номер последней пары .

Первый просмотр: рассматривается весь массив.

i= 1 5 4 8 2 9 (k=1, i=1, N-k=4) > меняем

i= 2 4 5 8 2 9 < не меняем

i= 3 4 5 8 2 9 > меняем

i = 4 4 5 2 8 9 < не меняем 9 находится на своем месте.

Второй просмотр: рассматриваем часть массива с первого до предпоследнего элемента.

i= 1 5 2 8 9 < не меняем

i= 2 4 5 2 8 9 > меняем

i= 3 4 2 5 8 9 < не меняем 8 — на своем месте.

Третий просмотр: рассматриваемая часть массива содержит три первых элемента.

i= 1 4 2 5 8 9 > меняем

i= 2 2 4 5 8 9 < не меняем 5 — на своем месте.

Четвертый просмотр: рассматриваем последнюю пару элементов.

i= 1 2 4 5 8 9 < не меняем 4 — на своем месте.

Наименьший элемент — 2 оказывается на первом месте.

Количество просмотров элементов массива равно N-1.

Программная реализация :

```
Procedure Sort;
Var k, i, t:Integer;        {k — номер просмотра, изменяется от 1 до N-1;
                             i — номер первого элемента рассматриваемой
                             пары; t — рабочая переменная для перестановки
                             местами элементов массива.}
Begin
For k:=l To N-1 Do
{Цикл по номеру просмотра.}
For i:=l To N-k Do
If 'A[i]>A[i+l] Then
{Перестановка элементов.} Begin
t:=A[i];A[i] :=A[i+l] ;A[i+l] :=t;
End;
End;
```

ЭФФЕКТИВНОСТЬ АЛГОРИТМА:

При сортировке методом "пузырька" выполняется N — 1 просмотров, на каждом i-м просмотре производится N — i сравнений. Общее количество С равно N* (N—1)/2,или $O(N^2)$.

4.1.2 БЫСТРАЯ СОРТИРОВКА

Данный метод был предложен Ч.Э.Р. Хоаром в 1962 году. В общем случае его эффективность достаточно высока (O(n*logn)), поэтому автор назвал его "быстрой сортировкой". Такая эффективность достигается за счет отсечения ненужных перестановок для уже отсортированного массива.

АЛГОРИТМ:

В исходном массиве A выбирается некоторый элемент X (его называют "барьерным"). Целью является запись X "на свое место" в массиве, пусть это будет место k, такое, чтобы слева от X были элементы массива, меньшие или равные X, а справа — элементы массива, большие X, т.е. массив A будет иметь вид:

(A[1], A[2], ..., A[k - 1]), A[k] = (X), (A[k + 1], ..., A[n]).

В результате элемент A[k] находится на своем месте и исходный массив A разделен на две неупорядоченные части, барьером между которыми является элемент A[k]. Далее требуется отсортировать полученные части тем же методом до тех пор, пока в каждой из частей массива не останется по одному элементу, то есть пока не будет отсортирован весь массив.

ПРИМЕР.

Исходный массив состоит из 8 элементов: 8 12 3 7 19 11 4 16. В качестве барьерного элемента возьмем средний элемент массива (7). Произведя необходимые перестановки, получим: (4 3) 7 (12 19 11 8 16); теперь элемент 7 находится на своем месте. Продолжаем сортировку.

Левая часть:	Правая часть:
(3) 4 7 (12 19 11 8 16)	3 4 7 (8) 11 (19 12 16)
3 4 7 (12 19 11 8 16)	3 4 7 8 11 (19 12 16)
3 4 7 8 11 12 (19 16)	
3 4 7 8 11 12 (16) 19	
3 4 7 8 11 12 16 19	

Из описания алгоритма видно, что он может быть реализован посредством рекурсивной процедуры, параметрами которой являются нижняя и верхняя границы изменения индексов сортируемой части исходного массива.

Программная реализация :

```
Procedure Quicksort (m, t: Integer);{Быстрая сортировка, первый вызов Quicksort(1,N)}
Var i , j , x , w: Integer;
Begin
i:=m; j:=t; x:=A[ (m+t) Div 2];          {Определение «барьерного элемента»}
Repeat
  While A[i]<x Do Inc(i);
  While A[j]>x Do Dec(j);
  If i<=j Then Begin w:=A[i]; A[i]:=A[j]; A[jl:=w; Inc(i); Dec(j) End {Меняем местами-если больше}
Until i>j;
If m<j Then Quicksort(m, j);    {Рекурсия}
If i<t Then Quicksort(i,t),
End;
```

4.2 ПОИСК ДАННЫХ

Основной вопрос задачи поиска: где в заданной совокупности данных находится элемент, обладающий заданным свойством?

4.2.1 ЛИНЕЙНЫЙ ПОИСК.

Большинство задач поиска сводится к простейшей — к поиску в массиве элемента с заданным значением.

Рассмотрим именно эту задачу. Пусть требуется найти элемент X в массиве A. В данном случае известно только значение разыскиваемого элемента, никакой дополнительной информации о нем или о массиве, в

котором производится поиск, нет. Наверное, единственным способом является последовательный просмотр массива и сравнение значения очередного рассматриваемого элемента массива с X. Напишем фрагмент:

For i:=1 To N Do If A[i]=X Then k:=i;

В этом цикле находится индекс последнего элемента A, равного X, при этом массив просматривается полностью.

Эффективность алгоритма: O(N).

ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОЙ РЕАЛИЗАЦИИ УЧАЩИМИСЯ:

1. Найти первый элемент, равный X.
2. Найти последний элемент, равный X.
3. Написать программу поиска заданного слова в тексте. Слово и текст являются массивами символов заданной длины. Если слово присутствует в тексте, то выдать номер начальной позиции совпадения, в противном случае - 0.
4. Для каждой буквы заданного текста указать, сколько раз она встречается в тексте.

4.2.2 БИНАРНЫЙ ПОИСК

Если заранее известна некоторая информация о данных, среди которых ведется поиск, например, известно, что массив данных отсортирован, то удается сократить время поиска, используя бинарный (двоичный, дихотомический, методом деления пополам — все это другие названия алгоритма, основанного на той же идее) поиск.

Итак, требуется определить место X в отсортированном (например, в порядке неубывания) массиве A. Делим пополам и сравниваем X с элементом, который находится на границе этих половин. Отсортированность массива A позволяет по результату сравнения со средним элементом массива исключить из рассмотрения одну из половин.

Пример.

Пусть X = 6, а массив A состоит из 10 элементов: 3 5 6 8 12 15 17 18 20 25.

1-й шаг. Найдем номер среднего элемента: m =[(1+10)/2] = 5.

Так как 6 < A[5], далее можем рассматривать только элементы, индексы которых меньше 5:

3 5 6 8 ~~**12** 15 17 18 20 25~~.

2-й шаг. Рассматриваем лишь первые 4 элемента массива, находим индекс среднего элемента этой части

m =[(1+4)/2] = 2, 6 > A [2], следовательно, первый и второй элементы ,из рассмотрения исключаются:

~~3 **5** 6 8 12 15 17 18 20 25~~.

3-и шаг. Рассматриваем два элемента, значение m =[(3+4)/2] = 3:

~~3 5~~ **6** 8 ~~12 15 17 18 20 25~~.

A [3]= 6. Элемент найден, его номер — 3.

Программная реализация бинарного поиска:

```
Procedure Search;
Var i,j,m:Integer; f:Boolean;
Begin
i:=1; j:=N;          {На первом шаге рассматривается весь массив.}
f:=False;            {Признак того, что X не найден.}
While (i<=j) And Not f Do Begin
m:=(i+j) Div 2;
{Или m:=i+.(j-i) Div 2; , так как i+(j-i) Div 2= (2*i+(j-i)) Div 2=(i+j) Div 2.}
If A[m]=X Then f:=True          {Элемент найден, поиск прекращается.}
Else If A[m]<X Then i:=m+l      {Исключаем из рассмотрения левую часть A}
Else j:=m-l                     {Правую часть.}
End
End;
```

Рекурсивная реализация бинарного поиска:

Значением переменной t является индекс искомого элемента или ноль, если элемент не найден.

```
Procedure Search (i,j:Integer;Var t:Integer);
{Массив A и переменная X глобальные величины}
Var m:Integer;
Begin
If i>j Then t:=0 Else Begin m:=(i+j) Div 2;
If A[m]<X Then Search(m+1,j,t) Else
If A[m]>X Then Search(i,m-l,t) Else t:=m
End
End;
```

Эффективность алгоритма:

Каждое сравнение уменьшает диапазон поиска приблизительно в два раза. Следовательно, общее количество сравнений имеет порядок O (logN), но не стоит забывать, сколько может «весить» предварительная сортировка массива!

ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОЙ РЕАЛИЗАЦИИ УЧАЩИМИСЯ:

1. Задано N точек своими целыми координатами на плоскости. Отсортировать их по возрастанию абцисс.
2. Задана БД «Аптека», содержащая поля: «Наименование», «Страна изготовитель», «Дата выпуска», «Фасовка». Отсортировать БД по полю «Дата выпуска».

§5. ОСНОВЫ ВЫЧИСЛИТЕЛЬНОЙ ГЕОМЕТРИИ

В олимпиадах по программированию большое место стали занимать задачи вычислительной геометрии, которые требуют не только определенных знаний по геометрии, и аккуратности в использовании структур данных. Геометрические задачи обычно состоят из большого количества маленьких подзадач, которые оформляются в виде подпрограмм, что имеет большое значение при формировании структурного стиля программирования.

В данном разделе сознательно используется тип Extended для достижения оптимальной точности вычислений. Все локальные задачи оформлены в едином стиле оформления данных в виде отдельных подпрограмм, что позволяет осуществлять «сборку» большой программы из маленьких «кирпичиков».

5.1 ПРЯМАЯ ЛИНИЯ И ОТРЕЗОК ПРЯМОЙ

1. Прямая линия на плоскости, проходящая через две точки V и W, заданные своими координатами: $(V_x;V_y)$ и $(W_x;W_y)$, определяется следующим линейным уравнением от двух переменных:

$$(W_x-V_x)*(y-V_y)=(W_y-V_y)*(x-V_x).$$

После преобразований получаем:

$$(W_y-V_y)*x+(W_x-V_x)*y+(W_y-V_y)*V_x-(W_x-V_x)*V_y=0$$

или после соответствующих обозначений:

$A*x+B*y+C=0.$, где $A= V_y-W_y$, $B = W_x-V_x$, $C = (W_y-V_y)*V_x + (V_x-W_x)*V_y$

2. Точка пересечения двух прямых, заданных уравнениями

$$A1*x+B1*y+C1=0 \text{ и } A2*x+B2*y+C2=0,$$

Координаты точки их пересечения, в случае ее существования, определяется по формулам:

$$X= -(C1*B2-C2*B1)/(A1*B2-A2*B1),$$
$$Y=(A2*C1-A1*C2)/(A1*B2-A2*B1).$$

Существование определяется неравенством $(A1*B2-A2*B1)<>0$

3. Принадлежность некоторой точки отрезку $(V_x;V_y)$-$(W_x;W_y)$:

Координаты точек отрезка можно задать двумя параметрическими уравнениями от одной независимой переменной t:

$$x=V_x+(W_x-V_x)*t,$$
$$y=V_x+(W_y-V_y)*t.$$

При $0<=t<=1$ точка (x, y) лежит на отрезке, а при $t<0$ или $t>1$ - вне отрезка на прямой линии, продолжающей отрезок.

4. Взаимное расположение отрезков на плоскости:

Даны два отрезка. Первый отрезок задан точками p1=(x1;y1) и p2=(x2;y2), а второй - точками p3=(x3; y3) и p4=(x4; y4). Точка пересечения (если она есть) обозначена как p=(x;y).

Взаимное расположение отрезков можно проверить с помощью векторных произведений:

V1=p3p4 x p3p1,
V2=p3p4 x p3p2,
V3=p1p2 x p1p3,
V4=p1p2 x p1p4.

Векторное произведение: $V \times W = (Vx. Wy – Vy. Wx)$,
Ориентированный отрезок p1p2 определяется как разность векторов W – V , т.е. $X_{p3p4}=X_{p3}-X_{p4}$, $Y_{p3p4}=Y_{p3}-Y_{p4}$, тогда
$V1 = p3p4 \text{ x } p3p1 = X_{p3p4}*Y_{p3p1} – Y_{p3p4}*X_{p3p1}$.

Известно, что если:
V1V2<0 и V3V4<0, то отрезки пересекаются;
V1V2>0 или V3V4>0, то отрезки не пересекаются;
V1V2<=0,V3=0,V4<>0, то точка p3 расположена на отрезке p1p2;
V1V2<=0,V4=0,V3<>0, то точка p4 расположена на отрезке p1p2;
V3V4<=0,V1=0,V2<>0, то точка p1 расположена на отрезке p3p4;
V3V4<=0,V2=0,V1<>0, то точка p2 расположена на отрезке p3p4;
V1=0,V2=0,v3=0,V4=0, то отрезки p1p2 и p3p4 расположены на одной прямой, необходимы дополнительные проверки для выяснения того, пересекаются они или нет.

5.2 ВЕКТОРНАЯ ГЕОМЕТРИЯ

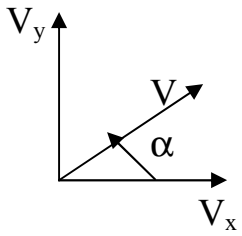
Каждую точку плоскости можно отождествлять с вектором, начало которого находится в точке (0,0).Обозначим координаты точки (вектора) V в декартовой прямоугольной системе координат через (V_x,V_y), точки $W=(W_x,W_y)$, $Q=(Q_x,Q_y)$. Для векторов, лежащих в плоскости XOY, определены следующие операции:

сумма: $G = V + W$	$G_x = V_x + W_x$, $G_y = V_y + W_y$
разность: $G = V – W$	$G_x = V_x – W_x$, $G_y = V_y – W_y$
умножение на число K: $G = K*V$	$G_x = K*V_x$, $G_y = K*V_y$
скалярное произведение:	$V*W = V_x*W_x + V_y*W_y$
векторное произведение:	$V \times W = (V_x*W_y – V_y*W_x)Z$, где Z – единичный вектор, направленный по оси OZ, $\lambda=V_x*W_y– V_y*W_x$ – числовой множитель.

Заметим, что если начальная точка вектора $V=(V_x,V_y)$, а конечная $W=(W_x,W_y)$, то он является разностью векторов W – V или *ориентированным отрезком*.

Вектор V можно задать в полярной системе координат через его длину (модуль) V и угол α относительно оси OX. Координаты (V, α) полярной

системы координат и (V_x,V_y) прямоугольной декартовой связаны соотношениями:



$V_x = V\cos\alpha$, $V_y = V\sin\alpha$,
$V = \mathrm{sqrt}(\mathrm{sqr}(V_x) + \mathrm{sqr}(V_y))$, $\mathrm{tg}\alpha = V_y/V_x$.

Для скалярного произведения векторов (V, α) и (W, β) справедливо соотношение:
$V*W = V_x*W_x + V_y*W_y = V*\cos\alpha*W*\cos\beta + V*\sin\alpha*W*\sin\beta = V*W*\cos(\alpha-\beta)$
для векторного:
$V\times W=(V_x*W_y–V_y*W_x)Z=(V*\cos\alpha*W*\sin\beta - V*\sin\alpha*W*\cos\beta)Z=$
$(V*W*\sin(\beta - \alpha))Z;$

Из этих соотношений следует:

- скалярное произведение ненулевых векторов равно нулю, если векторы перпендикулярны;
- векторное произведение ненулевых векторов равно нулю, если векторы параллельны;
- при общей начальной точке у двух векторов скалярное произведение больше нуля, если угол между векторами острый, и меньше нуля, если угол тупой;
- при общей начальной точке двух векторов их векторное произведение больше нуля, если второй вектор направлен влево от первого, и меньше нуля, если вправо.

Процедуры и функции работы с отрезками и линиями:

```
Type TLine = Record A, B, C:Extended; End; {Описание типа для прямых}
Type TPoint = Record  X,Y:Extended; End;  {Описание типа для точек }

Procedure Point2ToLine(Const A, B:Tpoint; Var L: TLine);
{Определение уравнения прямой по координатам двух точек}
Begin
  L.A:=B.y-A.y; L.B:=A.x-B.x; L.C:=-(A.x*L.A+A.y*L.b)
End;

Function Line2ToPoint(Const fL, sL: TLine; Var P: Tpoint): Boolean;
{Определение координат точки пересечения двух линий. Значение функции
равно true, если точка пересечения есть,и false ,если прямые параллельны }
Var st:Extended;
Begin
st:=fL.A*sL.B-sL.A*fL.B;
If  Not (st<>0) Then Begin
Line2ToPoint:=True; {Координаты точки пересечения двух прямых}
```

```
P.X:=-(fL.C*sL.B-sL.C*fL.B)/st; P.Y:=(sL.A*fL.C-fL.A*sL.C)/st; End
Else Line2ToPoint:=False
End;
```

Векторная геометрия:

```
Type
TvecDec = Record x,y:Extended; End;
TvecPol = Record rst, angl: Extended; End;  {описание векторов }

Procedure AddVecDec(Const a,b: TvecDec; Var c:TvecDec);
{Сумма двух векторов в декартовой системе}
Begin
  C.x:=A.x+B.x;
  C.y:= A.y+B.y;
End;

Procedure SubVecDec(Const a,b: TvecDec;  Var c:TvecDec);
{Разность двух векторов в декартовой системе координат}
Begin
  C.x:=A.x-B.x;
  C.y:=A.y-B.y;
End;

Procedure MultVecDecScalar(Const a: TvecPol;const k:extended; Var b:TvecPol);
{Умножение вектора на число в декартовой системе координат}
Begin  B.x:=A.x*k;
B.y:=A.y*k; End;

Procedure MulkVecPol(Const A: TvecPol; Const K:Extended; Var B:TvecPol);
{Умножение вектора на число в полярной системе координат}
Begin
  If k>=0 then
  begin
    B.Rst:=A.Rst*K;
    B.Angl:= A.Angl;
  End else
  Begin
    B.Rst:=A.Rst*abs(k);
    B.Angl:=A.Angl+Pi;
    If B.Angl>2*Pi then B.Angl:=B.Angl-2*Pi;
  End;
End;
```

Функция определения угла при переводе из декартовой в полярную систему координат:

```
Function GetAngle(Const x,y:Real):Extended;
{Возвращает угол от 0 до 2Pi, начиная от Ox, против часовой стрелки}
Var rs, c:Real;
Begin
  rs:= Sqrt(Sqr(x)+Sqr(y));
  if rs=0 Then GetAngle:=0
  Else Begin
         C:=x/rs;
         If c=0 Then C:=Pi/2    {При определении угла следует учитывать
                                  период     функции  Tan(x) }
         Else C:=Arctan (Sqrt(Abs(1- Sqr(c)))/c);
         If c<0 Then C:=Pi+c;
         If y<0 Then C:=2*Pi-C;
         GetAngle:=C;
       End;
End;

Procedure TurnDecPol(Const a:TvecDec; Var b:TvecPol);
{Перевод из декартовой системы координат в полярную}
Begin
 b.rst:=Sqrt(Sqr(a.x)+Sqr(a.y));
 b.angle:=getangle(a.x,a.y);
End;

Procedure TurnPolDec(Const a:TvecPol; Var b:TVecDec );
{Перевод из полярной системы координат в декартовую}
Begin
 b.x:=a.rst*cos(a.angle);
 b.y:=a.rst*sin(a.angle);
End;

Function VectorMulDec (Const a,b:TvecDec): Extended;
{Скалярное произведение векторов в декартовой системе координат}
Begin
 SkalarMulDec:=a.x*b.x+a.y*b.y;
End;

Function VectorMulDec(Const a,b:TVecDec):Extended;
{Векторное произведение векторов в декартовой системе координат}
Begin
 VectorMulDec:=a.x*b.y-a.y*b.x;
End;

Function SkalarMulPol(Const a,b:TvecPol): Extended;
```

```
{скалярное произведение векторов в полярной системе координат}
Begin
  SkalarMulPol:= cos(a.angle-b.angle)*a.rst*b.rst;
End;

Function VectorMulPol(const  a,b:TvecPol): Extended;
{Векторное произведение векторов в полярной системе координат}
Begin
  VectorMulPol:=sin(b.angl-a.angl)*a.rst*b.rst;
End;
```

5.3 ТРЕУГОЛЬНИК

5.3.1 ПЛОЩАДЬ ЛЮБОГО ТРЕУГОЛЬНИКА

Даны точки p1=(x1,y1), p2=(x2,y2), p3=(x3,y3), образующие некоторый треугольник

$$\text{Определитель} \quad \begin{matrix} x1 & y1 & 1 \\ x2 & y2 & 1 \\ x3 & y3 & 1 \end{matrix}$$

дает удвоенную *ориентированную площадь треугольника* (p1,p2,p3).Значение площади положительно тогда и только тогда, когда обход (p1,p2,p3) ориентирован *против часовой стрелки*.

5.3.2 ЗАМЕЧАТЕЛЬНЫЕ ЛИНИИ И ТОЧКИ ТРЕУГОЛЬНИКА.

Высоту треугольника, опущенную на сторону a, обозначим через b_a. Через три стороны она выражается формулой

b_a = 2*sqrt(p*(p-a)*(p-b)*(p-c))/a, где p=(a+b+c)/2.

Медианы треугольника пересекаются в одной точке (всегда внутри треугольника), являющейся центром тяжести треугольника. Эта точка делит каждую медиану в отношении 2: 1, считая от вершины.

Медиану на сторону A обозначим через m_a. Через три стороны треугольника она выражается формулой

m_a = sqrt(2*b*b+2*c*c-a*a)/2.

Три биссектрисы треугольника пересекаются в одной точке (всегда внутри треугольника), являющейся центром вписанного круга. Его радиус вычисляется по формуле

r = sqrt((p-a)*(p-b)*(p-c)/p).

Биссектрису к стороне a обозначим через l_a, она выражается формулой

l_a = 2*sqrt(b*c*p*(p-a))/(b+c), где p – полупериметр.

5.3.3 СВОЙСТВА ТРЕУГОЛЬНИКОВ.

В РАВНОБЕДРЕННОМ треугольнике высота, медиана и биссектриса, опущенные на основание, а также перпендикуляр, проведенный через середину основания, совпадают друг с другом.

В РАВНОСТОРОНЕМ те же свойства имеют место для всех трех сторон.

Центр тяжести, центр вписанного круга и центр описанного круга совпадают друг с другом.

Процедуры и функции:

```
Procedure Read;   {Ввод координат точек треугольника}
Var  k:integer;
Begin
  For k:=1 to 3 do
  Begin
   Writeln('Ввод координат точки №   ',k);
    Readln (P[k].x,P[k].y)   {Массив P описан в глобальных переменных}
  End;
End;

Procedure Trian (var a,b,c:extended);  {Вычисление длин сторон треугольника}
Begin
  A:=sqrt(sqr(P[1].x-P[2].x)+Sqr(P[1].y-P[2].y));
  B:=sqrt(sqr(P[2].x-P[3].x)+Sqr(P[2].y-P[3].y));
  C:=sqrt(sqr(P[3].x-P[1].x)+Sqr(P[3].y-P[1].y));
End;

 Function IsTriangle(Const a,b,c: Extended): Boolean;
{проверка существования треугольника со сторонами a,b,c}
Begin
  IsTriangle:=(a+b>c) And (a+c>b) And (b+c>a);
End;

Fuction  SquareTrian : extended;
{Вычисление ориентированной площади треугольника, P-глобальный}
Begin
   SquareTrian:=
(P[1].x*(P[2].y-P[3].y)+P[2].x*(P[3].y-P[1].y) + P[3].x(P[1].y-P[2].y))/2;
End;

Function GetHeight(Const A,B,C: Extended):Extended;
{Вычисление длины высоты, проведенной из вершины, противоположной
стороне треугольника с длиной A}
```

```
Var p: Extended;
Begin
P:=(A+B+C)/2;
GetHeight:=2*Sqrt(p*(p-A)*(p-B)*(p-C))/A
End;

Function GetMed(Const A,B,C: extended): extended;
{Вычисление длины медианы, проведенной из вершины, противоположной
стороне треугольника с длиной A}
Begin
GetMed:=Sqrt(2*(b*b+c*c)-a*a)/2
End;

Function GetBiss(Const A,B,C: Extended):Extended;
{Вычисление длины биссектрисы, проведенной из вершины,
противоположной стороне треугольника с длиной A}
Var p:Extended;
Begin
P:=(A+B+C)/2;
GetBiss:=2*Sqrt(B*C*P*(P-A))/(B+C)
End;

Function GetRadInc(Const A,B,C :Extended):Extended
{Вычисление радиуса окружности, вписанной в треугольник с длинами
сторон A, B, C}
Var p:Extetnded;
Begin
P:=(A+B+C)/2;
GetRadInc:=Sqrt((P-A)*(P-B)*(P-C)/P)
End;

Function GetRadOut(Const A,B,C: Extended):Extended;
{Вычисление радиуса окружности, описанной около треугольника с длинами
сторон A, B,C}

Var p:Extended;
Begin
P:=(A+B+C)/2;
GetRadOut:=A*B*C/(4*Sqrt(P*(P-A)*(P-B)*(P-C)))
End;
```

5.4 МНОГОУГОЛЬНИК

Многоугольник называется простым, если никакая пара непоследовательных его ребер не имеет общих точек.

Реализация в программе: если нет ни одного пересечения сторон, включая случай их частичного наложения, то многоугольник простой.

```
Function CheckSimple(Const A:Array Of Tpoint; Const N:Word):Boolean;
{проверка простоты многоугольника}
Var i,j:Integer:
      Rs:Tpoint;
Begin
  CheckSimple:=False;
   For i:=0 To N-2 Do
     For J:=i+1 To N-1 Do
       Case checkMutL(A[i],A{i+1},A[j},A[(j+1 Mod N},rs) Of
{Функция проверки взаимного расположения двух отрезков описана ранее}
0: If (J<>I+1) And (I<>(J+1) Mod N) Then Exit;
2: Exit:
End;
CheckSimple :=True;
End;
```

5.4.1 ОПРЕДЕЛЕНИЕ ВЫПУКЛОСТИ МНОГОУГОЛЬНИКА

1 способ

Многоугольник является выпуклым, если все диагонали лежат внутри него.

Сумма внутренних углов в выпуклом многоугольнике равна 180*(N-2), где N – число сторон многоугольника.

2 способ

Все треугольники, образованные тройками соседних вершин в порядке их обхода, имеют одну ориентацию. (Нахождение тройки вершин на одной прямой не нарушают факта выпуклости).

```
Function CheckPromin (Const A:Array Of Tpoint; Const N:Word):Boolean;
{Функция равна True, если многоугольник выпуклый }
Var bn,nw:Byte;
      I:Integer;  rp:Extended;
Begin
  CheckPromin:=False;
   If n>3 Then Begin
     Bn:=1;
     For i:=0 To N-1 Do Begin
       Rp:= SguareTrian(A[(I+N-1) Mod N], A[i], A[(i+1)Mod N]);
```

```
{Ориентированная площадь треугольника, построенного по трем соседним
вершинам многоугольника}
      If rp=0  Then  nw:=1 {Точки находятся на одной прямой}
                Else If rp<0 Then Nw:=0 Else nw:=2;
      If (bn=1) Then bn:=nw Else
                   If (nw<>1) And (nw<>bn) Then Exit;
{Вершины многоугольника лежат не на одной прямой и ориентация
«соседних» треугольников не совпадает – многоугольник не выпуклый}
                                  End;
                     End;
CheckPromin:=True;
End;
```

5.4.2 ПЛОЩАДЬ ПРОСТОГО ПЛОСКОГО МНОГОУГОЛЬНИКА

Пусть N-угольник задан координатами своих вершин. Его ориентированная площадь (точки перечисляется против часовой стрелки) будет равна:

S:=1/2*(X1Y2-X2Y1+X2Y3-X3Y2+.....+XnY1-X1Yn)

Процедуры и функции:

```
{ Считает определитель 2x2 }
Function Det(Const a,b,c,d : Extended) : Extended;
Begin
  Det:=a*d-b*c;
End;

{ Вычисляет ориентированную площадь треугольника,  }
{ натянутого на отрезок [P1,P2] и начало координат }
Function SSquare(Const P1,P2: TPoint): Extended;
Begin
  SSquare:=Det(P1.x,P2.x,P1.y,P2.y)/2;
End;

{ Вычисляет ориентированную площадь треугольника, }
{ натянутого на точки P1, P2 и P3 }
Function P3Square(Const P1,P2,P3: TPoint) : Extended;
Begin
  P3Square:=Det(P1.x-P3.x,P2.x-P3.x,P1.y-P3.y,P2.y-P3.y)/2;
End;

{ Вычисляет ориентированную площадь многоугольника,}
{ заданного массивом вершин A, N - число вершин}
```

```
Function PolySquare(N: Word; Const A: Array Of Point): Extended;
Var S : Extended;
   i : Word;
Begin
  S:=SSquare(A[N-1],A[0]);
  For i:=0 To N-2 Do S:=S+SSquare(A[i],A[i+1]);
  PolySquare:=S;
End;
```

ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОЙ РЕАЛИЗАЦИИ УЧАЩИМИСЯ

1. На плоскости заданы треугольник и отрезок. Определить их взаимное расположение.

2. На плоскости действительными координатами своих вершин заданы два треугольника. Используя перемещения и повороты, определить, можно ли один из треугольников вложить в другой.

3. На плоскости заданы N точек. Найти ближайшего «соседа» для каждой точки.

4. На плоскости заданы N точек. Найти минимальное количество прямых, из которых можно разместить все точки.

5. Определить радиус и центр окружности, на которой лежит наибольшее число точек заданного на плоскости множества точек.

6. ПЕРЕСЕЧЕНИЕ. Найти площадь пересечения N прямоугольников со сторонами, параллельными осям координат.

Входные данные: число N (2<=N<=1000) и N четверок действительных чиселXi1 Yi1 Xi2 Yi2 , задающих координаты левого нижнего и правого верхнего углов прямоугольника.

Выходные данные: Площадь пересечения (общая часть) всех данных прямоугольников, либо выдать, что они не пересекаются.

7. ОБЪЕДИНЕНИЕ. Найти площадь объединения N прямоугольников со сторонами, параллельными осям координат.

Входные данные: число N (2<=N<=1000) и N четверок действительных чисел (два знака после запятой) Xi1 Yi1 Xi2 Yi2 , задающих координаты левого нижнего и правого верхнего углов прямоугольника.

Выходные данные: Площадь объединения прямоугольников.(два знака после запятой)

8. ТОРТИК. Требуется разрезать торт (выпуклый N-угольник) на К частей равной площади. Разрешается проводить прямые вертикальные разрезы от одной границы торта до другой. Различные разрезы могут иметь общие точки лишь в своих концевых вершинах. Написать программу построения требуемых К-1 разрезов.

Входные данные: Два числа К и N (1<=K, N<=50)/ Далее следуют N пар вещественных чисел (2 знака после запятой) – координаты последовательно расположенных вершин многоугольника.

Выходные данные: Каждый из Л-1 разрезов должен быть представлен четверкой чисел- координатами концов соответствующего разреза.

9. КАРТИННАЯ ГАЛЕРЕЯ. В картинной галерее, имеющей форму N-угольника, расположено М люстр, которые мы будем считать точечными источниками света. Точка стены называется освещенной, если из нее видна хотя бы одна из люстр. Неосвещенным участком называется максимальное связное множество точек стены галереи, ни одна их которых не освещена (участок может содержать углы галереи). Определить все неосвещенные участки.

Входные данные: N и M(1<=N,M<=30)). В каждой из следующих N строк записаны координаты очередного угла галереи. Углы перечислены в порядке обхода стены по часовой стрелке.. Далее идут М строк, которые содержат координаты очередной из люстр.

10. Простой многоугольник задан своими вершинами в порядке обхода. Определить, лежит ли заданная точка внутри многоугольника.

§6. ПЕРЕБОР И МЕТОДЫ ЕГО СОКРАЩЕНИЯ.

Данный раздел рассматривает, вероятно, самые «старые» по способам исследования их решений задачи. Один из локальных методов их решения – метод динамического программирования был найден в 70-е годы XX столетия.

Как показывает опыт, элементы перебора встречаются, фактически, почти во всех решаемых задачах.

Большое значение при изучении данного раздела имеет формирование у учащихся умения «диагностировать» способ решения определенной задачи.

6.1 NP – ПОЛНЫЕ ЗАДАЧИ

Существует достаточно много задач, которые требуют перебора всех комбинаций данных для выбора оптимального решения – т.н. NP – полные задачи (решение которых нельзя найти за полиномиальное время). Такие задачи реализуются только при определенных ограничениях (т.к. очень трудоемки для памяти ПК) и практически не рассматриваются в школьном программировании, поэтому рассмотрим их кратко.

6.1.1 РЕШЕНИЕ NP-ПОЛНЫХ ЗАДАЧ

В соответствии с представлением алгоритма решения NP-задач с помощью алгоритма угадывания и алгоритма проверки NP-полные задачи

требуют полного перебора и решаются рекурсивно, так, что алгоритм поиска решения задачи размера n на каждом шаге рассматривает все возможные варианты решений на глубину 1 и оставшуюся задачу меньшего размера n-1.

6.1.1.1 ТИПЫ РЕКУРСИВНЫХ АЛГОРИТМОВ

Рассмотрим случаи, в которых разработка рекурсивных алгоритмов является наиболее эффективной. Обычно рекурсивный алгоритм целесообразно разрабатывать при наличии одного из следующих условий:

1. При необходимости обработки данных, имеющих рекурсивную структуру. Процедуры анализа рекурсивных структур наиболее эффективны, когда они сами рекурсивны, т.к. эти процедуры отражают особенности построения данных, и в результате строение программы соответствует структуре обрабатываемых данных.

2. Если алгоритм, обрабатывающий набор некоторых данных, можно построить, разбивая эти данные на части и получая из этих частичных решений общее решение на всей совокупности данных. Этот прием, особенно если применять его рекурсивно, часто приводит к эффективному решению задачи, подзадачи которой представляют собой ее меньшие версии. Данный прием получил название "разделяй и властвуй". При этом, как правило, задачу следует разбивать на подзадачи равных размеров. Поддержание равновесия - основной принцип при разработке хорошего алгоритма.

3. Если задача поставлена так, что ее решением является выбор какого-то варианта из некоторого множества возможных решений. Решение задачи определяется после некоторого конечного числа шагов, так что выбирая на каждом шаге вариант решения, мы удаляем часть информации из всей подлежащей обработке информации и пытаемся решить задачу на меньшем объеме данных. Поиск решения завершается в двух случаях:

- либо когда кончаются данные;
- либо находится решение на текущем наборе данных.

Одним из приемов решения олимпиадных задач является выход из перебора решений по истечению времени, отпущенного на тест. И если решение за данное время не найдено на всем множестве решений, то выдается ответ «NO». В частности, таким методом обычно решаются NP-полные задачи.

4. Если имеется рекурсивная схема некоторой функции. Существуют некоторые функции, которые легко можно определить рекурсивно, но которые нельзя определить в терминах обычных алгебраических выражений.

6.1.2 ПРИМЕРЫ NP - ПОЛНЫХ ЗАДАЧ

Задача 1. Расписание для мультипроцессорной системы.

Задано конечное множество A заданий, которые необходимо выполнить в мультипроцессорной системе, состоящей из m процессоров. Мультипроцессорная система - это система, в которой имеется несколько независимых процессоров, на каждом из которых задача решается независимо от загрузки остальных процессоров.

Для каждой задачи $a \in A$ известна длительность $t(a) \in N$ ее решения. Задано время T, в течение которого необходимо решить все задачи множества A.

Вопрос: Можно ли так распределить задачи по процессорам, чтобы при параллельном решении этих задач общее время решения всей совокупности задач не превышало заданное T?

Задача 2. Минимальный набор тестов.

Задано конечное множество A возможных диагнозов заболевания. Известно, что некоторые заболевания имеют частично совпадающие симптомы, поэтому задан набор $C=<C_1,C_2,\ldots,C_n>$; $Ci \in A$, представляющий двоичные тесты. Имеется некоторое натуральное число m, ограничивающее число тестов, которые должен пройти один больной.

Вопрос: существует ли такое множество тестов C длины m, что для любой пары a_i, a_j возможных диагнозов из A имеется некоторый тест, различающий a_i и a_j, т.е. тест $c \in C$ равен 1 только для определенного диагноза.

Задача 3. Сельский почтальон.

Задан граф G=(V,E), L(e) - длина каждого ребра и положительное целое число K. Существует ли в G простой цикл, сумма длин ребер которого не меньше K?

Задача 4. Упаковка в контейнеры.

Задано конечное множество $U=\{u_1, u_2, \ldots, u_m\}$ предметов, для каждого из которых задано натуральное число $s(u_i)$ - линейный размер предмета u_i. Даны также положительное целое число B - вместимость контейнера и число K - количество контейнеров.

Вопрос: Существует ли такое разбиение множества U на K непересекающихся подмножеств $U_1, U_2,\ldots,U_K$, что сумма размеров предметов из каждого подмножества U_i не превосходит B?

Задача 6. Составление кроссворда.

Заданы конечное множество слов W и матрица A из нулей и единиц размером n*n. Вопрос: Можно ли составить кроссворд из слов множества W, чтобы слова вписывались в клетки матрицы A, заполненные нулями?

Существует достаточно много методов ограниченного перебора (оптимизация решения задачи в процессе решения), которые помогают решить как классические задачи, так и реально поставленные задачи.

Надо заметить, что о решении любой задачи нельзя сказать, что оно единственно с точки зрения применения какого-либо алгоритма, т.к.

реализация разных алгоритмов может быть одинаковой с точки зрения их временной трудоемкости. Решение переборных задач часто рассматривается с точки зрения теории графов, содержащей очень много методов оптимизации.

6.2 ПЕРЕБОР ВАРИАНТОВ

Задача о коммивояжере.

Имеется N городов, расстояния между которыми заданы. Коммивояжеру необходимо выйти из какого-либо города, посетить остальные N-1 городов точно по одному разу и вернуться в исходный город. Маршрут коммивояжера должен быть минимальной длины (стоимости).

Задача относится к NP- полным, но даже при простом переборе не обязательно просматривать все варианты.

РЕШЕНИЕ. На каждом шаге будем проверять все возможные следующие города (так сделаем для каждого первого города).

Программная реализация:

```
const max=100; var w:array[1..max,1..max]of integer;  {Матрица
смежности}     c{количество пройденных},n{количество городов},i,j:integer;
        min{минимальная сумма},l{текущая}:longint;
        t,r:array[1..max]of byte; {матрицы ответа}
        is:array[1..max]of boolean; {были-ли мы в городе?}

      procedure run; {основная процедура}
      var i:byte;
      begin
       if c=n then {дошли?}
        begin
         if w[t[c],t[1]]<>-1 then {Если можем вернуться…}
          if (min=-1)or(l+w[t[c],t[1]]<min) then  {меньше…}
           begin
            min:=l+w[t[c],t[1]];
            r:=t;
           end;
         exit;
        end;
       if (l>min)and(min<>-1) then exit; {слишком много-выходим}
       for i:=1 to n do  {по всем новым городам}
        if is[i] then {если не были…}
         if w[t[c],i]<>-1 then  {…и можем добраться - входим}
          begin
           inc(l,w[t[c],i]);
           is[i]:=false;
           inc(c);
```

```
          t[c]:=i;
          run;
          dec(c);
          is[i]:=true;
          dec(l,w[t[c],i]);
        end;
end;

begin
  assign(input,'input.txt');
  reset(input);
  readln(n); {Читаем данные}
  for i:=1 to n do
    begin
      for j:=1 to n do
        read(w[i,j]);
      readln;
    end;
  close(input);
  fillchar(t,sizeof(t),0);
  fillchar(r,sizeof(r),0);
  fillchar(is,sizeof(is),true);
  min:=-1; {Не нашли пока}
  l:=0;
  c:=1;
  for i:=1 to n do {по всем первым городам}
    begin
      t[1]:=i;
      is[1]:=false;
      run;
    end;
  assign(output,'output.txt');
  rewrite(output);
  writeln(min);
  if min<>-1 then  {нашли?}
   begin
    for i:=1 to n do
      write(r[i],' ');
    write(r[1]);
   end;
  close(output);
end.
```

6.3 ПЕРЕБОР С ВОЗВРАТОМ

Данный метод рассмотрим на примере классической задачи о лабиринте: Дано клеточное поле, часть клеток занята препятствиями. Необходимо попасть из некоторой заданной клетки в другую заданную клетку путем последовательного перемещения по клеткам.

Классический перебор осуществляется по правилам, предложенным в 1891 году Э.Люка в "Математических досугах":

- в каждой клетке выбирается еще не исследованный путь;
- если из исследуемой в данный момент клетки нет путей, то возвращаемся на один шаг назад (в предыдущую клетку) и пытаемся выбрать другой путь.

Предлагаем написать самостоятельно!

6.4 ПЕРЕБОР С ОТСЕЧЕНИЕМ ВЕТВЕЙ И СКЛЕИВАНИЕМ ВЕТВЕЙ

Рассмотрим задачу: на шахматной доске N*N требуется расставить N ферзей таким образом, чтобы ни один ферзь не атаковал другой.

Все возможные расстановки ферзей C^{N}_{N*N} (для N=8 это около $4.4*10^4$) возможностей. Анализируя игру ферзя, можно заметить, что по условию задачи каждый столбец может содержать не более одного ферзя, что дает N в степени N расстановок (для N=8 это $1.7*10^7$).

Никакие два ферзя нельзя поставить в одну строку, а поэтому, чтобы вектор (A_1,A_2,...,A_N) был решением, он должен быть перестановкой элементов (1,2,...,N), что дает только N! (при $N=8.4*10^4$) возможностей.

Никакие два ферзя не могут находиться на одной диагонали, что еще сокращает число возможностей (для N=8 в дереве остается 2056 узлов). Итак, с помощью наблюдений можно исключить из рассмотрения большое число расстановок N ферзей на доске размером N*N.

Идея слияния ветвей состоит в том, чтобы избежать выполнения одной и той же работы: если решения для двух или более диапазонов данных совпадают, то можно исследовать только одно из них. В задаче о ферзях можно использовать склеивание, заметив, что если $A_1>N/2$, то найденное решение можно отразить и получить решение, для которого $A_1<=N/2$, т.е. для случаев $A_1=2$ и $A_1=N-1$ решения совпадают.

Программная реализация:

```
const n=4; var a:array[1..2*n,1..2*n]of byte;  {Матрица-степень "битости" клетки}
    t:integer;                {Количество поставленных ферзей}
    vect:array[1..2*n]of byte;    {Сама расстановка}
    count:longint;            {Их количество}
```

```
procedure correct(x,y,t:integer);
{Корректирует степени "битости" всех клеток}
{T=1 - добавляет ферзя, T=-1 - удаляет}
var i,j:integer;
begin
  for i:=1 to 2*n do
    for j:=1 to 2*n do
      if (i=x) then inc(a[i,j],t) else         {Вертикаль}
      if (j=y) then inc(a[i,j],t) else         {Горизонталь}
      if (x-i)=(y-j) then inc(a[i,j],t) else   {Одна диагональ}
      if (x-i)=(j-y) then inc(a[i,j],t);       {Другая диагональ}
end;

procedure print;
{Выводит расстановку и изоморфную ей - симметричную относительно
вертикали}
var i,j:integer;
begin
  inc(count,2);
  writeln('Расстановка № ',count-1,':');
  write('  ');
  for i:=1 to 2*n do
    write(' ',chr(ord('a')+i-1));
  writeln;
  for i:=1 to 2*n do
   begin
    write(i:2);
    for j:=1 to 2*n do
      if vect[i]=j then write(' Ф') else
       if odd(i+j) then write('  ') else write('--');
    writeln;
   end;
  writeln;
  writeln('Расстановка № ',count,':');
  write('  ');
  for i:=1 to 2*n do
    write(' ',chr(ord('a')+i-1));
  writeln;
  for i:=1 to 2*n do
   begin
    write(i:2);
    for j:=1 to 2*n do
      if vect[i]=2*n+1-j then write(' Ф') else
       if odd(i+j) then write('  ') else write('--');
    writeln;
```

```
    end;
  writeln;
end;

procedure run;
var i:byte;
begin
  if t=2*n then  {Расставили всех}
    begin
      print;
      exit;
    end;
  inc(t); {Добавляем}
  for i:=1 to 2*n do
    if a[t,i]=0 then {Если не бьётся...}
      begin
        vect[t]:=i;
        correct(t,i,+1);  {Корректировка}
        run;
        correct(t,i,-1);  {Корректировка}
      end;
  dec(t); {Убираем ферзя}
end;

begin
  assign(output,'output.txt');
  rewrite(output);
  fillchar(a,sizeof(a),0);
  t:=1;
  count:=0;
  for vect[1]:=1 to n do {Первый ферзь-от 1 до 4, остальные в
симметричных}
    begin
      correct(1,vect[1],+1);
      run;
      correct(1,vect[1],-1);
    end;
  writeln('Всего: ',count,' расстановки');
  close(output);
end.
```

ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОЙ РЕАЛИЗАЦИИ УЧАЩИМИСЯ

1. Найти все способы расстановки ферзей на шахматной доске N*N. (Для доски 8*8 - ответ 92.)

2. Задача о шахматном коне. Составить программу подсчета числа способов обхода конем доски. (Общее число способов разметки доски 8*8 равно 64! - без сокращения перебора по логике очередного хода коня).

3. Составить таблицу из числа способов обхода конем шахматной доски для небольших значений N и M. Определить, при каких значениях начинается т.н. "зависание" ПК.

4. ПРО ЛЯГУШКУ. С одного берега болота на другой, между которыми на различном расстоянии друг от друга расположены камни, пытается перепрыгнуть лягушка. Расстояние между камнями оценивается количеством единиц первоначальной скорости лягушки (Первоначальная скорость =1). Составить программу движения лягушки по камням (с минимальным количеством прыжков и/или минимальной длиной пути (эти варианты могут и совпадать)), если, находясь на камне (или на берегу), она может свою скорость:

A) оставить прежней;
B) уменьшить на единицу;
C) увеличить на единицу.

Лягушка может двигаться как вперед, так и назад. Составить тесты для отсутствия решения.

6.5 ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

(т.н. Табличный метод)

6.5.1 АЛГОРИТМ ДИНАМИЧЕСКОГО ПРОГРАМИРОВАНИЯ

Динамическое программирование - метод оптимизации, приспособленный к задачам, в которых требуется построить решение с максимизацией или минимизацией, т.е. оптимальным значением некоторого параметра.

Его алгоритм можно сформулировать так:

1. Выделить и описать подзадачи, через решение которых будет выражаться искомое решение;

2. Выписать рекуррентные соотношения (уравнения), связывающие оптимальные значения параметра для всех подзадач;

3. Вычислить оптимальное значение параметра для всех подзадач;

4. Построить само оптимальное решение.

В задачах на подсчет количеств допустимых вариантов пункт 4 не нужен.

Автор Д.П. Беллман так сформулировал принцип оптимальности:
каково бы ни было начальное состояние на любом шаге и решение, выбранное на этом шаге, последующие решения должны выбираться оптимальными относительно состояния, к которому придет система в конце данного шага.

Использование этого принципа гарантирует, что решение, выбранное на любом шаге, является не локально лучшим, а лучшим с точки зрения задачи в целом.

Данный метод усовершенствует решение задач, решаемых, например, с помощью рекурсий или перебора вариантов.

6.5.2 УСЛОВИЯ ПРИМЕНЕНИЯ ДИНАМИЧЕСКОГО ПРОГРАММИРОВАНИ:

1. Оптимальное решение задачи выражается через оптимальное решение задач меньшей размерности, представимых в виде подзадач (подпрограмм). Улучшая решение подзадач, тем самым, улучшается решение общей задачи;

2. Небольшое число подзадач, что позволяет хранить решения каждой подзадачи в таблице. Уменьшение числа подзадач происходит из-за многократного их повторения (т.н. перекрывающиеся подзадачи).

3. Дискретность (неделимость) величин, рассматриваемых в задаче.

4. Один из главных критериев, когда принцип ДП дает эффект уменьшения временной сложности: если в процессе решения необходимо многократно перебирать одни и те же варианты (за счет увеличения емкостной сложности уменьшается временная сложность).

6.5.3 РЕАЛИЗАЦИЯ АЛГОРИТМА ДП

6.5.3.1 СВЕДЕНИЕ ЗАДАЧИ К ПОДЗАДАЧАМ

Одним из основных способов решения задач является их сведение к решению такого набора подзадач, чтобы, исходя из решений подзадач, было возможно получить решение исходной задачи. При этом для решения исходной задачи может потребоваться решение одной или нескольких подзадач.

ПРИМЕР.

Рассмотрим задачу нахождения суммы N элементов таблицы A.

Пусть функция S(N) соответствует решению нашей исходной задачи. Эта функция имеет один аргумент N - количество суммируемых элементов таблицы A. Понятно, что для поиска суммы N элементов достаточно знать сумму первых N-1 элементов и значение N-го элемента. Поэтому решение исходной задачи можно записать в виде соотношения $S(N) = S(N - 1) + a[N]$.

Следует отметить, что это соотношение справедливо для любого количества элементов $N>1$.

Это соотношение можно переписать в виде $S(i) = S(i - 1) + a[i]$ при $i>1$, $S(0) = 0$.

Последовательное применение первого соотношения при i = 1, 2, ..., N и используется при вычислении суммы N элементов, при этом вычисление функции производится от меньших значений аргументов к большим.

```
S[0]: = 0;
for i:= 1 to N do
 S[i]: = S[i - 1] + a[i];
```

6.5.3.2 ПОНЯТИЕ РЕКУРРЕНТНОГО СООТНОШЕНИЯ

Найденный способ сведения решения исходной задачи к решению некоторых подзадач может быть записан в виде соотношений, в которых значение функции, соответствующей исходной задаче, выражается через значения функций, соответствующих подзадачам. При этом важнейшим условием сведения является тот факт, что значения аргументов у любой из функций в правой части соотношения меньше значения аргументов функции в левой части соотношения. Если аргументов несколько, то достаточно уменьшения одного из них. Соотношения должны быть определены для всех допустимых значений аргументов.

ПРИМЕР.

Вычислить сумму $S=1+1/x+1/x^2+...+1/x^N$ при x, не равном 0. Можно записать следующее соотношение:

$$S(i) = S(i - 1) + a(i),\ i>1,\ \text{где } a(i) = 1/x^i,\qquad S(0) = 1.$$

Возникла новая задача - найти способ вычисления a(i). Для этого можно воспользоваться тем же приемом - попытаться вычислить a(i) через значение a(i - 1). Соотношение между значениями a(i) и a(i - 1) имеет следующий вид:

$$a(i) = a(i - 1)/x,\qquad i>1,\quad a(0) = 1$$

Поставленную задачу можно решить следующим образом.

```
S[0]: = 1;
a[0]: = 1;
for i:=1 to N do
  begin
    a[i]: = a[i - 1]/x;
    S[i]: = S[i - 1] + a[i]
  end;
```

6.5.3.3 ПРАВИЛЬНЫЕ РЕКУРРЕНТНЫЕ СООТНОШЕНИЯ

Правильными рекуррентными соотношениями (уравнениями) будем называть такие рекуррентные соотношения, у которых количество или значения аргументов у функций в правой части соотношения меньше количества или, соответственно, значений аргументов функции в левой части

соотношения. Если аргументов несколько, то достаточно уменьшения одного из аргументов.

Соотношения должны быть определены для всех допустимых значений аргументов. Поэтому должны быть определены значения функций при начальных значениях параметров.

В приведенных примерах соотношения связывали функции только с двумя различными параметрами: S(i) и S(i - 1), а также a(i) и a(i - 1) для любого натурального i. При этом были определены начальные значения S(0) и a(0).

Отметим, что без этих начальных значений рекуррентное соотношение $S(i) = S(i - 1) + a(i), \ i>1,$ было бы неправильным, так как оно не определено при i = 1.

Пример.

Написать рекуррентную формулу для подсчета количества различных укладок плитками размера 1x2 коридора размера 2xN. При N=2 таких укладок две:

1 1 1 2
2 2 2 1

Важнейшим моментом при решении задачи является способ сведения задачи к подзадачам. Но не менее важным вопросом является и способ построения решения исходной задачи из решений подзадач. Одним из наиболее эффективных способов построения решения исходной задачи является

6.5.3.4 ИСПОЛЬЗОВАНИЕ ТАБЛИЦ ДЛЯ ЗАПОМИНАНИЯ РЕШЕНИЙ ПОДЗАДАЧ

В этом и заключается алгоритм решения задач, называемый методом динамического программирования.

Задача может быть формализована в виде функции, которая зависит от одного или нескольких аргументов. Если взять таблицу, у которой количество элементов равно количеству всех возможных различных наборов аргументов функции, то каждому набору аргументов может быть поставлен в соответствие элемент таблицы. Вычислив элементы таблицы (решения подзадач), можно найти и решение исходной задачи.

Одним из способов организации таблиц является такой подход, когда размерность таблицы определяется количеством аргументов у функции, соответствующей подзадаче.

Пример.

В заданной числовой последовательности A[1.. N] определить максимальную длину последовательности подряд идущих одинаковых элементов.

Пусть L(i) обозначает максимальную длину последовательности, последним элементом которой является элемент с номером i. Тогда значение L(i+1) может быть либо на 1 больше L(i), если элементы A(i+1) и A(i) равны, либо L(i+1) будет равно 1, так как перед элементом с номером i+1 стоит отличный от него элемент. Максимальное значение L(i)i=1,...,N и соответствует решению задачи.

```
L[1]: = 1;
For  i:=2 to N do
     if A[i-1]: = A[i]  then
      L[i]:=L[i-1]+1
    else
      L[i]:=1;
IndL:=1;
For  i:=2 to N do
   if L[i]>L[IndL]  then
     IndL:=i;
```

ПРИМЕР.

В таблице с N строками и M столбцами, состоящей из 0 и 1, необходимо найти квадратный блок максимального размера, состоящий из одних единиц. Под блоком понимается множество элементов соседних (подряд идущих) строк и столбцов таблицы. Интересующая нас часть показана на рисунке.

```
1 1 1 1 1 1
0 1 1 1 0 1
1 1 1 1 1 1
1 1 0 1 1 1
1 0 1 1 0 1
```

Положение любого квадратного блока может быть определено его размером и положением одного из его углов.

Пусть T(i,j) есть функция, значение которой соответствует размеру максимального квадратного блока, состоящего из одних единиц, правый нижний угол которого расположен в позиции (i, j). Функция T(i, j) вычисляет элемент таблицы B[i, j]. Для примера на рис. значения T(i, j) будут иметь вид

```
i\j 1 2 3 4 5 6
1  1 1 1 1 1 1
2  0 1 2 2 0 1
3  1 1 2 3 1 1
4  1 2 0 1 2 2
5  1 0 1 1 0 1
```

Таким образом, наша задача свелась к вычислению максимального значения функции T при всевозможных значениях параметров i и j. Этой функции может быть поставлена в соответствие таблица размера N*M.

Определим сначала значения элементов таблицы B, расположенных в первой строке и в первом столбце. Получим: B(1, 1) = A[1, 1],

B(1, j) = A[1, j] при j>2,

B(i, 1) = A[i, 1] при i>2.

Эти соотношения следуют из того факта, что в этих случаях рассматриваемая область матрицы A содержит только один элемент матрицы.

При 2<i<N и 2<j<M для этой функции можно записать следующие рекуррентные соотношения: B[i, j] = 0, если A[i, j] = 0 и

B[i, j] = min{B[i - 1, j], B[i, j - 1], B[i - 1, j - 1]} + 1, если A[i, j] = 1

Первое соотношение показывает, что размер максимального единичного блока с правым нижним углом в позиции (i, j) равен нулю в случае A[i, j] = 0.

Убедимся в правильности второго соотношения. Действительно, величина B[i - 1, j] соответствует максимальному размеру единичного блока таблицы A с правым нижним углом в позиции (i - 1, j). Тогда размер единичного блока с правым нижним углом в позиции (i, j) не превышает величину B[i - 1, j] + 1, так как к блоку в позиции (i - 1, j) могла добавиться только одна строка. Величина B[i, j - 1] соответствует максимальному размеру единичного блока таблицы A с правым нижним углом в позиции (i, j - 1). Тогда размер единичного блока с правым нижним углом в позиции (i, j) не превышает величину B[i, j - 1] + 1, так как к блоку в позиции (i - 1, j) мог добавиться только один столбец. Величина B[i - 1, j - 1] соответствует максимальному размеру единичного блока таблицы A с правым нижним углом в позиции (i - 1, j - 1). Тогда размер единичного блока с правым нижним углом в позиции (i, j) не превышает величину B[i - 1, j - 1] + 1, так как к блоку в позиции (i - 1, j - 1) могли добавиться только одна строка и один столбец. Итак, размер единичного блока с правым нижним углом в позиции (i, j) равен min{B[i - 1, j], B[i, j - 1], B[i - 1, j - 1]} + 1.

```
B[1, 1]: = A[1, 1];
For  j:=2 to 6 do
 B[1, j]: = A[1, j];
for i:= 2 to 5 do
 B[i, 1]: = A[i, 1];
for i:=2 to  5 do
   for j:=2 to 6 do
      if A[i, j]: = 1 then
       begin
          B[i, j]: = min(B[i, j - 1], B[i - 1, j]);
           B[i, j]: = min(B[i, j], B[i - 1, j - 1]) + 1
```

```
  end
else
    B[i, j]: = 0;
```

ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОЙ РЕАЛИЗАЦИИ УЧАЩИМИСЯ

Задача. *Максимальная сумма*

В заданной числовой последовательности A[1..N] найти максимальную сумму подряд идущих элементов.

Входные данные:

Первая строка входного файла содержит число N (1<=N<=1000). Следующие строки содержат элементы последовательности, A[i] (-100<=A[i]<=100), разделенные пробелами и/или переводами строк.

Выходные данные:

Выходной файл должен содержать единственное число - максимальную возможную сумму.

Пример входного файла	Пример выходного файла
3 5 -3 6	8

Задача. *Минимальный штраф - 1.*

Задана матрица натуральных чисел A[1..N, 1..M], m<=n. За каждый проход через клетку (i, j) взимается штраф A[i, j]. Необходимо определить путь с минимальным суммарным штрафом, с которым можно пройти из клетки (1, 1) в клетку (n, m). При этом из текущей клетки можно переходить в любую из 3-х соседних клеток, стоящих в строке с номером, на 1 большим текущего номера строки.

Входные данные:

Первая строка входного файла содержит числа N и M (1<=N, M<=100). Следующие строки входного файла содержат N*M натуральных чисел A[i, j] (1<=A[i, j]<=100).

Выходные данные:

В первой строке выходного файла должен быть записан минимальный штраф. В каждой из следущих N строк должны быть записаны два по числа xi, yi -- i-ая клетка искомого пути.

Пример входного файла:	Пример выходного файла:
3 2 2 1 3 4 2 3	8 1 1 2 1 3 2

Задача. *Минимальный штраф – 2.*

Задана матрица натуральных чисел A[1..N, 1..M]. За каждый проход через клетку (i, j) взимается штраф A[i, j]. Необходимо определить путь с минимальным суммарным штрафом, с которым можно пройти из некоторой клетки первой строки в некоторую клетку N-ой строки. При этом из текущей клетки можно переходить в любую из 3-х соседних клеток, стоящих в строке с номером, на 1 большим текущего номера строки.

Входные данные:

Первая строка входного файла содержит числа N и M (1<=N, M<=100). Далее идет N*M натуральных чисел A[i, j] (1<=A[i, j]<=100)

Выходные данные:

Первая строка выходного файла должна содержать штраф. В каждой из следующих N строк должны быть записаны два числа xi, yi -- i-ая клетка искомого пути.

Пример входного файла	Пример выходного файла
3 2 2 1 3 4 2 3	6 1 2 2 1 3 1

6.5.3.6 ВОССТАНОВЛЕНИЕ РЕШЕНИЯ ПО МАТРИЦЕ

Пример.

Для заданной числовой последовательности A[1.. N] найти максимальную длину подпоследовательности, такой, что каждый последующий элемент подпоследовательности делится нацело на все предыдущие. Входными параметрами является число N и последовательность из N целых чисел, 1<N<100, |A[i]|<100, i=1,...,N. Кроме того, необходимо указать сами элементы последовательности.

Входные данные	Выходные данные
5 5 -3 6 0 -10	3 3 6 0

Обозначим через K(i) значение функции, которая равна длине максимальной подпоследовательности в числовой последовательности A, последним элементом которой равен элемент с индексом i.

Нам необходимо найти значение функции K(N). Таким образом, для решения задачи достаточно последовательно вычислить значения K(i) для всех значений i от 1 до N.

В силу условия задачи, для вычисления значения K(i+1) достаточно использовать предыдущие значения K(j), j=1,...i, причем только те из них, для которых выполняется условие A[i+1] mod A[j] = 0. Таким образом,

K(i+1)=max{K(j)}+1, j=1,...i, A[i+1]modA[j] = 0 или равно 1, если (i+1)-й элемент не делится ни на один предыдущий. Для приведенного примера функции К будет соответствовать массив К

i	1 2 3 4 5
A	5 -3 6 0 -10
K	1 1 2 3 2

Из матрицы К видно, что максимальная длина подпоследовательности равна 3, при этом последним элементом такой последовательности является 4-й элемент. Для определения номера предпоследнего (а он является последним элементом подпоследовательности, длина которой на единицу меньше максимальной длины), достаточно найти такой индекс j, для которого выполняются соотношения j=1,...i, A[i]modA[j] = 0 и K[i]-K[j] =1.

Зная предыдущий элемент (элемент с индексом 2), аналогичным образом можно найти элемент, стоящий перед ним. Процесс заканчивается тогда, когда будет найден элемент, который является начальным элементом подпоследовательности (для которого значение функции К равно 1).

Нетрудно заметить, что элементы подпоследовательности (структура решения) восстанавливается точно в обратном порядке, в котором происходило заполнение матрицы, соответствующей подзадачам решаемой задачи.

Для предыдущих задач были достаточно очевидны рекуррентные соотношения, причем параметрами функции были только количество исходных элементов. Однако очень часто на практике в качестве параметров выступают и другие значения, например общая сумма, максимальные значения элементов.

Пример.

Имеется 5 неделимых предметов. Для каждого предмета известна его масса (в кг.). Величины массы являются натуральными числами. Ваша цель состоит в том, чтобы определить, существует ли несколько предметов, суммарная масса предметов которого ровно 16 кг. Если такой набор существует, то требуется определить список предметов в наборе.

Пусть элемент M(i) таблицы M соответствует массе i-го предмета.

Через T обозначим функцию, значение которой равно 1, если такой набор имеется, и равно 0, если такого набора нет. Аргументами у этой функции будут количество предметов и требуемая суммарная масса набора.

Для нашей задачи T(5,16) определим подзадачи T(i,j), где i обозначает количество начальных предметов, из которых можно осуществлять выбор, а j определяет требуемую суммарную массу требуемого набора. Отметим, что определенный таким образом первый параметр i определяет как количество предметов для подзадачи, так и значения масс из таблицы M.

Определим сначала начальные значения функции T. При нулевых значениях одного из аргументов значение функции равны: T(0,j)=0 при j>1, {нельзя без предметов набрать массу j>0} T(i,0)=0 при i>=0. {всегда можно набрать нулевую массу }.

Определим возможные значения функции T(i,j) при ненулевых значениях аргументов.

Решение подзадачи, соответствующей функции T(i,j) может быть сведено к двум возможностям: следует брать предмет с номером i в набор или нет.

Если предмет не берется, то решение задачи с i предметами сводится к решению подзадачи с i-1 предметами, т.е. T(i,j)=T(i-1,j). Если предмет с номером i берется, то это уменьшает суммарную массу для i-1 первых предметов на величину M[i], т.е. T(i,j)=T(i-1,j-M[i]).

При этом необходимо учитывать, что вторая ситуация возможна только тогда, когда масса i-го предмета не больше значения j.

Теперь для получения решения нам необходимо выбрать лучшую из этих двух возможностей. Поэтому рекуррентное соотношение при i>1и j>1 имеет вид:

$$T(i,j)=T(i-1,j) \text{ при } j<M[i];$$

$$T(i,j)=\max(T(i-1,j),T(i-1,j-M[i])) \text{ при } j<M[i].$$

Пусть заданы следующие значения массы для 5 предметов:

M[1]=4;
M[2]=5;
M[3]=3;
M[4]=7;
M[5]=6.

Таблица значений функции T, которую мы также назовем T, выглядит следующим образом:

i\j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0
3	1	0	0	1	1	1	0	1	1	1	0	0	1	0	0	0	0
4	1	0	0	1	1	1	0	1	1	1	1	1	1	0	1	1	1
5	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Следовательно, T(5,16)=1, и набор существует. Для определения списка предметов в наборе будем поступать следующим образом. Рассмотрим элементы T(5,16) и T(4,16). Так как значения обоих этих элементов равны, то это значит, что можно набрать массу 16 кг. С помощью первых четырех предметов, т.е. предмет 5 в возможный набор можно не включать.

Теперь рассматриваем элементы T(4,16) и T(3,16). Их значения не равны, а это значит, что предмет 4 должен быть обязательно включен в возможный набор (без предмета 5). Поэтому предмет 4 включается в набор, а оставшаяся требуемая масса других элементов набора равна 16-M(4)=16-7=9. Рассматриваем элементы T(3,9) и T(2,9). Их значения равны, а значит, предмет 3 в набор не включаем.

Рассматриваем элементы T(2,9) и T(1,9). Их значения не равны, а это значит, что предмет 2 должен быть обязательно включен в возможный набор

(без предметов 3, 5). Поэтому предмет 2 включается в набор, а оставшаяся требуемая масса других элементов набора равна 9-M(2)=9-5=4.

Наконец, рассматриваем элементы T(1,4) и T(0,4). Их значения не равны, а это значит, что предмет 1 должен быть обязательно включен в возможный набор, а оставшаяся требуемая масса других элементов набора равна 0.

Таким образом, в искомый набор входят элементы 1, 2, 4.

Программная реализация:

```
T[0, 0] := 1;
for j := 1 to 16 do T[0, j] := 0;
for i := 1 to  5 do T[i, 0] := 0;
for i := 1 to 5 do begin
  for j := 1 to 16 do begin
    if j >= M[i] then begin
      T[i, j] = max(T[i - 1, j], T[i - 1, j - M[i]])
    end else begin
      T[i, j] = T[i - 1, j];
    end;
  end;
end;

sum := 16;
if T[5, 16] = 1 then begin
  for i := 5 downto 1 do begin
    if T[i, sum] = T[i - 1, sum] then begin
      writeln (i, "---No")
    end else begin
      writeln (i, "---Yes");
      sum := sum - M[i];
    end;
  end;
end else begin
  writeln("No solution");
end;
```

ПРИМЕР решения задачи методом ДП.

В таблице размером N*N, где N<13, клетки заполнены случайным образом цифрами от 0 до 9. Предложить Чебурашке алгоритм, позволяющий найти маршрут из клетки (1,1) в клетку (N,N) и удовлетворяющий следующим условиям:

1. любые две последовательные клетки в маршруте имеют общую сторону;
2. количество клеток маршрута минимально;
3. сумма цифр в клетках маршрута максимальна.

АЛГОРИТМ:

Будем искать наилучшие пути, идущие из клетки (1,1) во все остальные клетки таблицы, в частности и в клетку (N,N). Для этого в некоторой вспомогательной таблице B того же размера, что и A, будем отмечать суммы цифр оптимальных путей, ведущих в соответствующие клетки. Так в клетке(1,1) таблицы B окажется число A(1,1), потому что путь, ведущий в нее , состоит из одной клетки. Так же легко заполняются клетки (1,2) и (2,1) таблицы B. В них тоже ведут единственные пути. и суммы цифр вдоль них равны A(1,1)+A(1,2) и A(1,1)+A(2,1) соответственно.

Поясним сказанное на конкретном примере. Пусть таблица A размером 5*5 имеет вид, представленный на рис.2. Нумерация клеток идет от левого верхнего угла. Проследим за заполнением таблицы B. О клетках (1,2) и (2,1) уже говорилось. Чтобы заполнить клетку (2,2), заметим, что в нее можно попасть либо из клетки (1,2), либо из клетки (2,1). Следовательно, в клетку (2,2) таблицы B надо записать либо число B(1,2)+A(2,2), либо число B(2,1)+A(2,2), в зависимости от того, какое из них больше. Заполнение остальных клеток таблицы B аналогично.

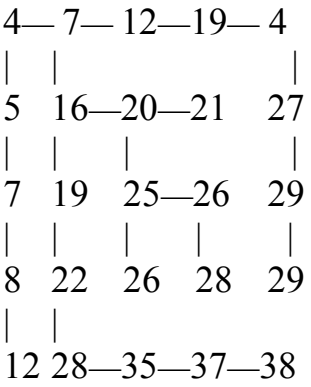
Если путь, ведущий в эту клетку один, то в клетку вписывается сумма цифр вдоль этого пути. Такими клетками являются клетки вида (1,J) и (J,1).

Если же в клетку (I,J) можно попасть из клеток для которых подсчитаны значения B, то необходимо выбрать M=max{B(I,J-1), B(I-1,J} и записать в клетку B(I,J) число M+A(I,J).

Клетку B(I,J) соединим чертой с той клеткой, где был достигнут максимум M. Для нахождения искомого маршрута достаточно пройти из клетки (N,N) в клетку (1,1) по черточкам.

4	3	5	7	5
1	9	4	1	3
2	3	5	1	2
1	3	1	2	0
4	6	7	2	1

таб. A



таб. B

Программная реализация:

```
var
 i,k,j,N:integer;
 a,b:array[1..13,1..13] of integer;
 c:array[1..25,1..2] of integer;

FUNCTION max(a1,a2:integer):integer;
```

```
{Функция определения max из двух чисел }
var
 d:integer;
begin
 if a1>a2
  then
   d:=a1
  else
   d:=a2;
 max:=d;
end;

BEGIN
 ReadLn(N);          {Размерность матрицы}
 for i:= 1 to N do
  for k:= 1 to N do
   ReadLn(a[i,k]);      {Ввод матрицы}

 b[1,1]:=a[1,1];           {Заполнение матрицы B}
 for i:=2 to N do
  begin
   b[1,i]:=a[1,i]+b[1,i-1];
   b[i,1]:=a[i,1]+b[i-1,1];
  end;
 for i:= 2 to N do
  for k:= 2 to N do
   b[i,k]:=a[i,k]+max(b[i-1,k],b[i,k-1]);

 i:=N;      {Чтение пути от конца к началу}
 k:=N;
 j:=2*N-1;
 c[1,1]:=1;
 c[1,2]:=1;
 while (i<>1) or (k<>1) do
  begin
   c[j,1]:=i;
   c[j,2]:=k;
   if i=1
    then Dec(k)
    else if k=1
         then Dec(i)
         else if max(b[i-1,k],b[i,k-1])=b[i,k-1]
              then
               Dec(k)
              else
```

```
                Dec(i);
          Dec(j);
         end;
        Write(c[1,1],',',c[1,2]);
        for i:=2 to 2*N-1 do
         Write('->',c[i,1],',',c[i,2]);
        WriteLn;
       END.
```

Результат работы:

1,1->1,2->2,2->3,2->4,2->5,2->5,3->5,4->5,5

ПРИМЕР решения задачи методом ДП:

ОПТИМАЛЬНАЯ РАССТАНОВКА СКОБОК.

В арифметическом выражении, операндами которого являются целые числа от 0 до 9, а операциями – бинарные операции «+» и «*», расставить скобки так, чтобы результат оказался максимальным.

ПРОГРАММНАЯ РЕАЛИЗАЦИЯ:

```
type pt=array[1..4] of byte;
var r_max,r_min,n,m,t,i,j,k:longint;     a:array[1..35,1..35] of longint;
op:array[1..35] of char;     b:array[1..35,1..35] of pt;
b_max,b_min:pt;ex:extended;
procedure init;{Процедура читает из файла числа}
var ch:char;    {Символьная переменная}
   flgmin:boolean;{Флаг отрицательного числа}
begin
 assign(input,'input.txt');{Связываем стандартный поток ввода с}
 reset(input);            {файлом "input.txt" и открываем его для чтения}
 t:=0;n:=0;               {Обнуляем переменные-счетчики}
 flgmin:=false;           {Флаг отриц. числа выставляем в false}
 while not eoln(input) do  {Читаем всю строку}
  begin
   read(ch);             {Читаем символ}
   case ch of            {Обрабатываем символ}
   '0'..'9':t:=t*10+(ord(ch)-48);{В t считываем текущее число}
   '*','+':begin              {Обнаружили арифметический знак}
        inc(n);               {Следовательно, увеличилось количество чисел}
        op[n]:=ch;            {Записываем в op операцию}
        if flgmin then        {Если число отрицательное то}
          a[n,n]:=-t          {Учитываем знак}
        else a[n,n]:=t;
        flgmin:=false;        {Флаг отриц. след. числа выставляем в false}
        t:=0;
       end;
   '-':flgmin:=true;          {Текущее число - отрицательное}
```

```
      end;
    end;
  inc(n);                        {Добавляем последнее число}
  if flgmin then t:=-t;
  a[n,n]:=t;
  close(input);                  {Закрываем файл}
end;
procedure run;                   {Основная подпрограмма}
procedure fillmax(a,b,c,d:longint);
begin b_max[1]:=a;b_max[2]:=b;b_max[3]:=c;b_max[4]:=d;end;
procedure fillmin(a,b,c,d:longint);
begin b_min[1]:=a;b_min[2]:=b;b_min[3]:=c;b_min[4]:=d;end;
begin
  for m:=1 to n-1 do             {Идем по диогоналям}
    for i:=1 to n-m do
      begin
        j:=i+m;{Считаем j координату}
        r_max:=-maxlongint;      {Максимальная сумма}
        r_min:=maxlongint;       {Минимальная сумма}
        for k:=i to j-1 do
          case op[k] of
          '+':begin              {Знак сложения}
                if r_max<a[i,k]+a[k+1,j] then {Сумма больше...}
                  begin fillmax(i,k,k+1,j);r_max:=a[i,k]+a[k+1,j];end;
                if r_min>a[k,i]+a[j,k+1] then  {Сумма меньше...}
                  begin fillmin(k,i,j,k+1);r_min:=a[k,i]+a[j,k+1];end;
              end;
          '*':begin              {Знак умножения}
                if r_max<a[i,k]*a[k+1,j] then
                  begin fillmax(i,k,k+1,j);r_max:=a[i,k]*a[k+1,j];end;
                if r_max<a[k,i]*a[j,k+1] then
                  begin fillmax(k,i,j,k+1);r_max:=a[k,i]*a[j,k+1];end;
                if r_max<a[i,k]*a[j,k+1] then
                  begin fillmax(i,k,j,k+1);r_max:=a[i,k]*a[j,k+1];end;
                if r_max<a[k,i]*a[k+1,j] then
                  begin fillmax(k,i,k+1,j);r_max:=a[k,i]*a[k+1,j];end;
                if r_min>a[i,k]*a[k+1,j] then
                  begin fillmin(i,k,k+1,j);r_min:=a[i,k]*a[k+1,j];end;
                if r_min>a[k,i]*a[j,k+1] then
                  begin fillmin(k,i,j,k+1);r_min:=a[k,i]*a[j,k+1];end;
                if r_min>a[i,k]*a[j,k+1] then
                  begin fillmin(i,k,j,k+1);r_min:=a[i,k]*a[j,k+1];end;
                if r_min>a[k,i]*a[k+1,j] then
                  begin fillmin(k,i,k+1,j);r_min:=a[k,i]*a[k+1,j];end;
              end;
```

```
        end;{Записываем лучшие результаты}
       a[i,j]:=r_max;a[j,i]:=r_min;b[i,j]:=b_max;b[j,i]:=b_min;
      end;
end;
procedure rec(x,y:byte);
begin        {Процедура по массиву b востанавливает положение скобок}
  if b[x,y,1]=b[x,y,2] then {Выводим 1-ое выражение}begin
     if (a[b[x,y,1],b[x,y,2]]<0) then write('[');
     write(a[b[x,y,1],b[x,y,2]]);   {Если выражение-число, то пишем число}
     if (a[b[x,y,1],b[x,y,2]]<0) then  write(']');end else
  begin write('(');rec(b[x,y,1],b[x,y,2]);write(')');end;
  if b[x,y,2]>b[x,y,1] then{Выводим знак операции}write(op[b[x,y,2]])
  else write(op[b[x,y,1]]);{Выводим 1-ое выражение}
  if b[x,y,3]=b[x,y,4] then begin
     if (a[b[x,y,3],b[x,y,4]]<0) then write('[');
     write(a[b[x,y,3],b[x,y,4]]);  {Если выражение-число, то пишем число}
     if (a[b[x,y,3],b[x,y,4]]<0) then write(']');end else
  begin write('(');rec(b[x,y,3],b[x,y,4]);write(')'); end;
end;
begin
  init;run;
  assign(output,'output.txt');
  rewrite(output);
  writeln(a[1,n]);{Выводим ответ}rec(1,n);{Выводим скобки}
  close(output);
end.
```

ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОЙ РЕАЛИЗАЦИИ УЧАЩИМИСЯ

1. ДВА РЮКЗАКА - 1

Дан массив чисел A[1..N], элементы которого являются натуральными числами. Требуется определить, можно ли эти числа разбить на два подмножества с одинаковой суммой элементов.

Входные данные:

В первой строке входного файла находится число N (1<=N<=100). Далее идет N натуральных чисел A[i] (1<=A[i]<=100).

Выходные данные:

Если искомое разбиение существует, в выходной файл необходимо вывести YES, иначе вывести NO.

Пример входного файла	Пример выходного файла
4 4 3 1 2	YES

2. *ДВА РЮКЗАКА – 2*.

Дан массив чисел A[1..N], элементы являются натуральными числами. Требуется разбить эти числа разбить на два подмножества, чтобы сумма элементов в подмножествах отличалась минимальным образом. Входными параметрами являются N, и N натуральных чисел. Ответом должны быть номера элементов первого множества.

Входные данные:

В первой строке входного файла находится число N (1<=N<=100). Далее идет N натуральных чисел A[i] (1<=A[i]<=100). Сумма всех A[i] не превосходит 1000.

Выходные данные:

В выходной файл необходимо вывести разницу p (p>=0) и затем вывести номера элементов из первого множества.

Пример входного файла	Пример выходного файла
4 4 3 1 2	0 1 3

6.6 ВОЛНОВЫЕ АЛГОРИТМЫ.

Данные алгоритмы достаточно полно разобраны в разделе «Графы» (раздел 7.4) и рассматривают способы поиска кратчайшего пути в графе. Тем менее эти алгоритмы относятся и к динамическому программированию. При реализации волновых алгоритмов часть таблицы, отведенная для запоминания решений подзадач, остается незаполненной, а часть действий алгоритма является лишней, что и отличает эти алгоритмы от рассмотренных выше. Но именно эта возможность использовать избыточную память и увеличивать количество производимых операций примерно в N раз делает программы, реализующие эти алгоритмы, простыми для понимания и реализации.

ЗАДАЧА О ЛАБИРИНТЕ.

Лабиринт задан массивом A размером N*N, в котором A[i,j]=1, если клетка «проходима», и A[i,j]=0, если “непроходима”. Путник изначально размещается в клетке [i0,j0]. Он может перемещаться в любую из соседних «проходимых» клеток, если у нее есть общая сторона с той, в которой он находится.

Определить, может ли путник выйти из лабиринта. Если да, то напечатать путь от выхода до начального положения путника. Выходом считается любая граничная точка массива A.

Алгоритм решения:

Запишем в клетку [i0,j0] число 2. Просмотрим все клетки лабиринта (или исключим из рассмотрения заведомо недостижимые за один ход). Если

у рассматриваемой «проходимой» клетки, помеченной единицей, есть сосед, помеченный двойкой (в общем случае числом К), то мы запишем в нее 3(в общем случае К+1). Т.о. на каждом шаге алгоритма числом К будут помечены все те клетки, до которых путник может добраться ровно за К-2 единичных перемещения (до которых за К-2 шага «докатилась волна»). Этот процесс закончится , когда очередное число будет вписано в граничную клетку либо, когда за весь просмотр массива ни одна из клеток не будет помечена числом К (выхода в этом случае нет). Печать решения: если последняя клетка помечена числом К, то предпоследняя – числом К-1, и т.д. Полученный путь по построению является кратчайшим. Дополнительные массивы можно не заводить, однако просмотр ряда клеток массива будет избыточным.

ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОЙ РЕАЛИЗАЦИИ УЧАЩИМИСЯ

Задача 1. *ПРО БОЛЬНОГО ВОРА.*

Вор, пробравшись в хранилище банка, обнаружил N СЛИТКОВ золота. Но стоимость каждого слитка была разной из-за проб золота в них (C[I]). Вес слитков тоже различался (M[I]). Поднять вор может только X кг (перед операцией его от волнения стукнул радикулит). Помогите больному вору забрать наиболее драгоценный груз.

Задача 2. *ЧИСЛА.*

Имеется N положительных чисел X1,X2...XN и число N. Выяснить можно ли получить N, складывая некоторые из чисел X1,X2...XN. Число действий должно быть порядка N^2.

Задача 3.

N различных станков один за другим объединены в конвейер. Имеется N рабочих. Задана матрица C[N,N], где C[i,j] производительность i-ого рабочего на j-ом станке. Определить

a) на каком станке должен работать каждый из рабочих, чтобы производительность была максимальной.
b) то же, но станки расположены параллельно и выполняют однородные операции.

Задача 4. *ТРЕУГОЛЬНИК.*

На рисунке изображен треугольник из чисел. Написать программу, которая вычисляет наибольшую сумму чисел, расположенных на пути, начинающемся в верхней точке треугольника и заканчивающемся на основании треугольника:

- каждый шаг на пути может осуществляться вниз по диагонали влево или вниз по диагонали вправо;
- число строк в треугольнике >1 и <100;
- треугольник составлен из целых чисел от 0 до 99.

```
                7
              8   3
            8   1   0
          2   7   4   4
        4   5   2   6   5
```

Задача 5. *АВТОЗАПРАВКА.*

Вдоль кольцевой дороги расположено m городов, в каждом из которых есть автозаправочная станция. Известна стоимость Z[i] заправки в городе с номером i и стоимость C[i] проезда по дороге, соединяющей i - й и (i+1)-й города, C[m] - стоимость проезда между первым и m-м городами. Для жителей каждого города определить город, в который им необходимо съездить, чтобы заправиться самым дешевым образом, и направление - «по часовой стрелке» или «против часовой стрелки», города пронумерованы по часовой стрелке.

АЛГОРИТМ: Введем два дополнительных массива

On, Ag: array[1..m] of record wh, qh:integer; end;.

On[i] означает, где следует заправляться (wh) и стоимость заправки (qh) жителям i-го города, если движение разрешено только по часовой стрелке. В этом случае жители города i имеют две альтернативы: либо заправляться у себя в городе, либо ехать по часовой стрелке. Во втором случае жителям города i надо заправляться там же, где и жителям города i+1, или в первом, если i=m. Итак, On[i]=min{Z[i],C[i]+On[i+1].qh}. Откуда известно значение On[i+1].qh? Необходимо найти город j с минимальной стоимостью заправки - On[j]:=(j,Z[j]). После этого можно последовательно вычислять значения On[j-1], On[j-2], ..., On[j+1]. **Аналогичные** действия необходимо выполнить при формировании массива Ag[i], после этого для жителей каждого города i следует выбрать лучший из On[i].qh и Ag[i].qh вариант заправки.

Задача 6. *РЕСТОРАН.*

В ресторане собираются N посетителей. Посетитель с номером i приходит во время Ti и имеет благосостояние Pi. Входная дверь ресторана имеет K состояний открытости. Состояние открытости двери может изменяться на одну единицу за одну единицу времени, т.е. она или открывается на единицу, или закрывается на единицу, или остается в текущем состоянии. В начальный момент времени дверь закрыта (состояние 0.) Посетитель с номером i входит в ресторан только в том случае, если дверь специально открыта для него, т.е когда состояние открытости двери совпадает с его степенью полноты Si, в противном случае посетитель обижается и уходит безвозвратно. Ресторан работает T часов. Цель швейцара - собрать посетителей с максимальным благосостоянием, правильно открывая и закрывая двери.

Входные данные:

В первой строке значения N, K, T, разделенные пробелами.

(1<=N<=100, 1<=K<=100, 0<=T<=30000). Во второй строке находятся времена прихода посетителей T1, T2...Tn, разделенные пробелами (0<=Ti<=T, для всех i=1,2...N). В третьей строке находятся величины благосостояния посетителей P1, P2...Pn, разделенные пробелами (0<=Pi<=300) , в четвертой строке находятся значения степени полноты посетителей Si, разделенные пробелами. Все исходные данные - целые числа. Выходные данные:

Число - максимальное суммарное благосостояние посетителей , либо 0 (если нельзя добиться прихода в ресторан ни одного посетителя.) Написать программу и тесты.

6.7 «ЖАДНЫЕ» АЛГОРИТМЫ

«Жадный» алгоритм - метод оптимизации, приспособленный к задачам, в которых процесс принятия решений может быть разбит на отдельные этапы (шаги).

Метод состоит в том, что оптимальное решение строится постепенно, шаг за шагом. На каждом шаге оптимизируется решение ТОЛЬКО ТЕКУЩЕГО ШАГА, не рассматривая оптимальность конечного результата, иначе говоря, последовательность локально оптимальных (жадных) выборов дает глобально оптимальное решение.

6.7.1 УСЛОВИЯ ПРИМЕНЕНИЯ «ЖАДНЫХ» АЛГОРИТМОВ

Если последовательность локально оптимальных (жадных) выборов (подзадач) дает глобально оптимальное решение (т.е на каждом шаге данный алгоритм берет «самый жирный кусок», а затем пытается сделать наилучший выбор среди оставшихся, каковы бы они не были).

ЗАДАЧИ

Задача 1. *ЗАДАЧА О ВОРЕ.*

Вор, пробравшись в хранилище банка, обнаружил N мешков золотого песка. Но стоимость мешков была разной из-за проб золота в них (C[I]). Вес мешков тоже различался (M[I]). Поднять вор может только X кг (перед операцией его от волнения стукнул радикулит). Помогите бедному вору выбрать наиболее драгоценный груз.

АЛГОРИТМ:

Необходимо набить рюкзак по MAX стоимости груза. Произведем сортировку стоимости содержимого мешков (C[I]/M[I]) по убыванию. Будем заполнять рюкзак вора, складывая мешки с золотом, начиная с самого дорогого мешка, пока есть место в рюкзаке. Если в конце заполнения мешок полностью не входит в рюкзак, золото следует отсыпать ровно столько, чтобы заполнить рюкзак до конца.

Задача 2. *ЗАДАЧА О ВЫБОРЕ ЗАЯВОК.*

Пусть даны N заявок на проведение занятий в одной и той же аудитории. Два разных занятия не могут перекрываться по времени. В каждой заявке указано начало и конец занятия (S[i] и F[i] для i- той заявки). Разные заявки могут пересекаться, и тогда можно удовлетворить только одну из них. Конец одной заявки может совпадать с началом другой. Выбрать такие заявки, чтобы их количество было MAX.

АЛГОРИТМ:

Упорядочим заявки в порядке возрастания времени окончания. На каждом шаге будем делать такой выбор заявки, чтобы оставшееся свободное время было максимальным.

Задача 3.*ЗАДАЧА О ВОСХОЖДЕНИИ.*

N альпинистов решили покорить вершину Эльбруса. Каждый I – тый альпинист может нести S[i] кг припасов и съедает за один день C[i] припасов. Какое максимальное число альпинистов дойдет до вершины и вернется обратно, если альпинисты могут делиться припасами на привале, а те, у которых не хватит припасов на восхождение и обратный путь, возвращаются назад. Скорость движения одинакова на любом отрезке пути, привал – один в день пути. Составить программу и тесты.

§7. ГРАФЫ

На алгоритмах теории графов основаны оптимальные решения многочисленных задач.

В данном разделе приведены лишь начальные алгоритмы: поиска в ширину, глубину и Дейкстры.

7.1 ОСНОВНЫЕ ПОНЯТИЯ И ОПРЕДЕЛЕНИЯ

Граф – математическая структура, применяемая в программировании при исследовании СВЯЗЕЙ МЕЖДУ ОБЪЕКТАМИ. Графы удобно рисовать, “изображать графически” – отсюда и их название.

Объект - это вершина. Ребра и дуги – связи между объектами.

ПРИМЕР задачи, решаемой с помощью графов:

На олимпиаду прибыло N человек. Некоторые из них знакомы между собой. Можно ли опосредованно перезнакомить их всех между собой?

(Незнакомые люди могут познакомиться только через общего знакомого).

ОПРЕДЕЛЕНИЕ:

Граф - абстрактное представление любой системы объектов, которая описана парой (V,E), где V – конечное множество т.н. узлов или вершин (объектов), а E – множество ребер (дуг), соединяющих все или некоторые из вершин. На рисунках вершины обозначаются номерами, а ребра и дуги - простыми или направленными линиями.

СЛОВАРЬ ТЕРМИНОВ

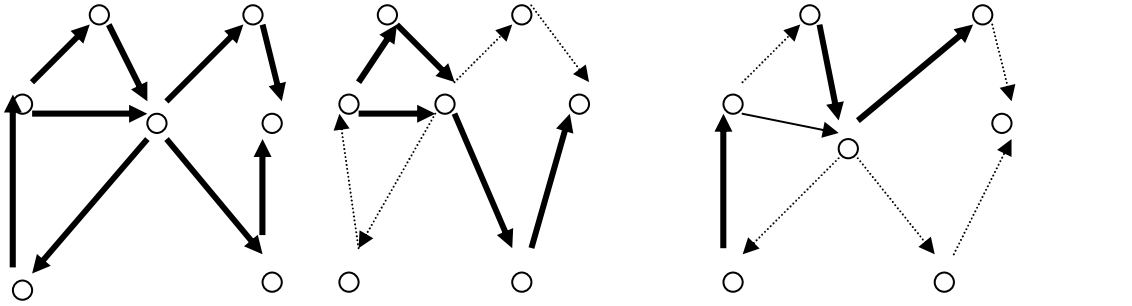
Каждая дуга соединяет две вершины, которые называются ее началом и концом. Дуга направлена от начала к концу. Если у дуги начало и конец совпадают, то она называется петлей.

Полный граф – граф без петель, в котором любые две различные вершины соединены ровно одной дугой (если любая пара U,W –ребро). В полном графе число ребер равно (N*N-N)/2, где N – число вершин.

Если выкинуть из графа несколько дуг, то его называют частичным графом.

Если выкинуть некоторые вершины и дуги, оставшиеся без начала и конца, то получим подграф (подмножество вершин графа и все соединяющие их ребра).

Если выкинем еще несколько дуг, то получим частичный подграф.



ГРАФ **ПОДГРАФ** **ЧАСТИЧНЫЙ ГРАФ**

Ребро (I,J) обозначает связь вершин I и J, которые называются смежными, а ребро – инцидентным им.

Дуга - ребро в ориентированном графе, обозначающее направление связи вершин.

Все дуги, входящие в вершину, образуют входящую звезду, выходящие – исходящую звезду, а все вместе - **звезду** вершины.

Степень вершины - это число инцидентных ей ребер (сумма количества дуг во входящей и выходящей звездах). Сумма степеней всех вершин в графе должна делиться на 2, иначе количество дуг получилось бы дробным.

Граф считается ориентированным, если задано множество упорядоченных ребер. В разреженном (неориентированном) графе число ребер намного меньше.

Маршрут между Z и W – это последовательность вершин (и ребер), соединяющих Z и W.

Цепь – маршрут без повторения ребер. Граф связен, если для любой пары вершин есть соединяющая цепь.

Компонента связности – максимальный связный подграф.

Цикл – замкнутый маршрут (конец маршрута совпадает с его началом).

Простой цикл – замкнутая цепь (конец цепи совпадает с началом).

Дерево - произвольный неориентированный граф без циклов.

Путь в ориентированном графе последовательность дуг, в которой конечная вершина всякой дуги, отличной от последней, является начальной вершиной следующей.

Простой путь - это путь, в котором каждая дуга используется не более одного раза.

Элементарный путь – это путь, в котором каждая вершина используется не более одного раза.

7.2 СПОСОБЫ ОПИСАНИЯ ГРАФОВ:

7.2.1 МАТРИЦА СМЕЖНОСТИ.

Данный способ удобен для небольших графов и графов с весами в небольших типах данных. Недостаток данного способа – требуется очень много памяти для хранения матрицы (например, матрица 10000*10000 типа longint занимает очень много памяти!).

Для создания матрицы смежности для графа с N вершинами затрачивается минимум N*N байт.

Матрица смежности - это двумерный массив A размерности N*N.

$$A[i,j] = \begin{cases} 1, \text{ если вершины i и j смежны} \\ 0, \text{ если вершины i и j не смежны} \end{cases}$$

Рассмотрим экономный способ отображения матрицы смежности. Представим каждую I-тую строку матрицы смежности множеством (Byte) номеров ее позиций, содержащих “1” (номера вершин, смежных с I-той вершиной).

```
const
  N=5;
type
  Mno=set of 1..N;
var
  J,K: byte;
 A:array[1..N] of Mno;      {Массив из N множеств}
begin
  for J:=1 to N do
      A[j]:=[];                  {Исходные пустые множества}
  WriteLn(‘Ввод для каждой J вершины номера K смежных с нею’);
  WriteLn( ‘вершин (только те, которые больше J), заканчивая’);
  WriteLn (‘перечень номеров вводом нуля и нажатием Enter’);
   for J:=1 to N-1 do
     begin
       Write(J,’-я вершина связна с ’);
        repeat
```

```
            Read (K);                          {K- номер вершины, смежной с J}
            if K<>0 then
                if K<J then
                    WriteLn ('Нужен  номер больший J')
                  else
                    A[J]:=A[J]+[K];  A[K]:=A[K]+[J]
        until K=0
      end;
   for K:=1 To N do
     begin
        WriteLn;
        {Контрольный вывод матрицы}
        for K:=1 to N do
        if K in A[J] then
           Write ('1 ')
        else
           Write ('0 ');
     end;
   ReadLn;
   ReadLn;
end.
```

7.2.2 ПЕРЕЧЕНЬ РЕБЕР

Для хранения перечня N ребер необходим массив R размерности N*2. Каждая строка массива описывает одно ребро.

R[i]=[start,finish]- начальная и конечная вершины.

7.2.3 СПИСОК СМЕЖНЫХ ВЕРШИН

Данный способ удобен для хранения графов с весами, в котором каждому ребру приписан некоторый вещественный «вес» (расстояние между вершинами, стоимость проезда и т.п.), т.е. задана весовая функция. В этом случае удобно хранить вес ребра (u,v) вместе с вершиной V в списке вершин, смежных с U.

Недостаток способа – для нахождения ребра из U в V нужно просматривать весь F(V).

Описание данных и процедура создания списковой структуры для представления графа:

```
Const max_graf=100;
Type   list=^elem;
       Elem=record
              Num : integer;
              Next : list;
```

```
                End;
Var
    Graf : array[1..max_graf] of elem;

Procedure CreateGraf(n:integer);
{n- число вершин графа}
Var       a : integer;
          Sosed,sosed1:list;
Begin
  For i:=1 to n do   {для каждой вершины}
   Begin
    New(sosed); {выделили память для нового элемента}
    Graf[i].next:=sosed; {ссылка на новый элемент}
    Writeln (‘для нового элемента ’,i,’введите номер очередного соседа или
0’);
    Repeat
      Readln(a);
      Sosed^.num:=a; {указатель на очередного сеседа}
      If a<>0 then
        Begin
          New(sosed1);
          Sosed^.next :=sosed1;
          Sosed:=sosed1;
        End;
     Until a=0 {больше соседей нет}
    End;
End;
```

Массив F[1..N] (N – число вершин) массив списков. Для каждой из N вершин список содержит смежные ей вершины в произвольном порядке (указатели «на»)

Для ориентированного и неориентированного графа количество требуемой памяти равно V+2*E (число вершин + число ребер).

7.3 ПОНЯТИЕ ДОСТИЖИМОСТИ

Если существует путь из вершины V в вершину U, то говорят, что U достижима из V.

Матрицу достижимости R определим следующим образом:

$$R[v,u] = \begin{cases} 1, \text{ если u достижима из v} \\ 0, \text{ в противном случае} \end{cases}$$

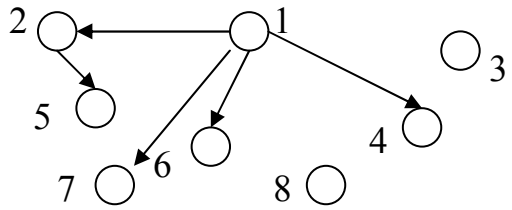
Множество R(v) – это множество таких вершин графа G, каждая из которых может быть достигнута из вершины V.

Пусть Г(v) – множество таких вершин графа G, которые достижимы из V с использованием путей длины 1.

Г2(v)- множество вершин, достижимых из V с использованием путей длины 2 и так далее. Тогда : R(v)={v} + Г(v) +Г2(v) +…. + ГN(v). Для графа на рисунке 2 :

R(1) = {1} + {2,5} + {6} + {4} + {7} = {1,2,4,5,6,7}

Рис. 2



Процедура Reach формирует матрицу достижимости:

```
procedure Reach;  {матрица R  - глобальная переменная}
var S,T : set of 1...N;
      i,j,l : integer;
begin
  Fillchar (R,Sizeof (R),O);
  For i:=1 to N do
    begin              {ищем вершины, достижимые  из вершины с номером i }
     T:=[i];
     Repeat
       S:=T;
      For l:=1 to N do
         If l in S then
            For J:=1 to N do
              If A[l,j] =1 Then T:=T+[j];
   Until S=t;   { если T не изменилось, то найдены все вершины графы, достижимые из вершины с номером I}
      For J:=1 To N do
        If J in T then R[i,j] :=1;
      end;
end;
```

7.4 ПОИСКИ КРАТЧАЙШИХ ПУТЕЙ В ГРАФЕ

К поиску кратчайших путей в графе относятся следующие задачи: найти наименьший по протяженности (стоимости, «пересадочности» и т.д.) путь в графе от вершины I к вершине J.

Рассмотрим некоторые оптимальные алгоритмы поиска.

7.4.1 ПОИСК В ГЛУБИНУ

Название основано на поиске «вглубь», пока это возможно (есть непройденные исходящие ребра), и возвращаться и искать другой путь, когда таких ребер нет.

Алгоритм метода:

Просмотр вершин графа начинается с некоторой фиксированной вершины V. Выбирается вершина U, смежная с V. Процесс повторяется с вершины U. Если на очередном шаге мы работаем с вершиной q и нет вершин, смежных с q и не просмотренных ранее (новых), то возвращаемся из вершины q к вершине, из которой мы попали в q. Если это вершина V, то просмотр закончен. Для фиксации признака, просмотрена вершина графа или нет, требуется структура данных типа:

Nnew : array[1..N] of Boolean

Пример:

Пусть граф описан матрицей смежности A. Просмотр начинается с первой вершины.

На рисунке 3А показан исходный граф, а на рисунке 3Б указана та очередность, в которой вершины графа просматривались в процессе поискав глубину:

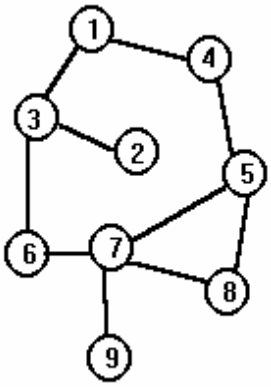


Рис. 3А

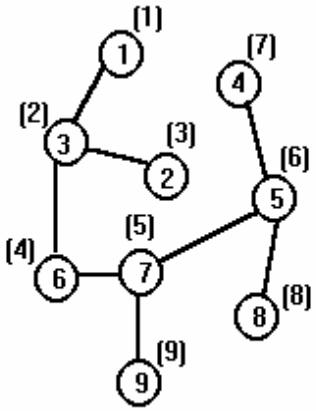


Рис. 3Б

Реализация рекурсивного алгоритма:

```
{Массивы Nnew  и A - глобальные}
procedure Pg (V: integer);        {поиск в глубину}
Var j : integer;
Begin
  Nnew [v]:=false;                {нет пути из вершины V}
  Write(V :3);
  For j:=1 To N Do        {для всех вершин от 1 до N – проверка наличия пути }
    If (A[v,j]<>0) And Nnew[j]  Then  Pg(j)      {идти от вершины, из которой
есть путь}
End;
```

Фрагмент вызывающего алгоритма:

```
Fillchar (Nnew, SizeOf (Nnew), True);
For i:=1 To N Do
  If Nnew[i] Then Pg(i);
```

Реализация данного алгоритма без рекурсий:

```
uses crt;
const max=10;
var is:array[1..max]of boolean;  {Была ли вершина}
   i,j,n:word;
   a:array[1..max,1..max]of byte; {Матрица смежности}
procedure pg(i:word);
var m:array[1..max]of word;j,k:word;  {Массив предыдущих}
begin
 m[i]:=0;
 j:=i;
 while j<>0 do  {Если не обошли всю компоненту…}
   begin
     is[j]:=false;  {Были}
     write(j,',');
     for k:=1 to n do
       if is[k] and (a[j,k]<>0) then break;  {Ищем соседа}
     if is[k] and (a[j,k]<>0) then  {Сосед есть…}
       begin
         m[k]:=j;  {Ставим ему предшественника}
         j:=k;     {Переходим в него}
       end
     else
       j:=m[j];    {Возвращаемся назад}
   end;
end;

begin
 fillchar(is,sizeof(is),true);
 clrscr;
 readln(n);
 for i:=1 to n do
   for j:=1 to n do
     begin
       write('a[',i,',',j,']=');
       readln(a[i,j]);
     end;
 readkey;
 clrscr;
 for i:=1 to n do
   if is[i] then
```

```
      begin
        pg(i);
        gotoxy(wherex-1,wherey);
        write(' ');
        writeln;
      end;
  readkey;
end.
```

7.4.2 ПОИСК В ШИРИНУ.

Название метода объясняется тем, что поиск ведется вширь – сначала просматриваются все соседние вершины, затем соседи соседей и т.д.

Алгоритм метода:

Пусть задан граф и зафиксирована начальная вершина. Алгоритм перечисляет все вершины, достижимые из начальной в порядке возрастания расстояния (числа ребер) от начальной вершины. В процессе поиска из графа выделяется часть, называемая «деревом поиска в ширину» с корнем в начальной вершине. Для каждой из них путь из корня в дереве поиска будет одним из кратчайших путей (из начальной вершины) в графе.

Алгоритм применим и к ориентированным, и к неориентированным графам.

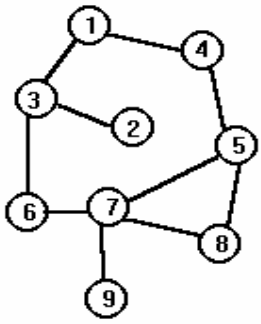


Рис. 4

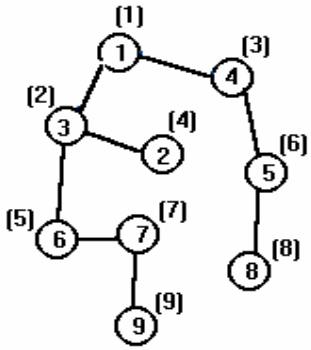


Рис. 4А

На рисунке 4А в скобках указана очередность их просмотра.

Процедура реализации метода:

```
Procedure PW (v: integer);
Var Og:array[1..N] of 0..N:   {очередь}
      Yk1, Yk2: integer; {указатели очереди, Yk1-запись, Yk2 – чтение}
       J : integer;
Begin
  Fillchar(Og, Sizeof(Og),O); Yk1:=0; Yk2:=0; {начальная инициализация}
  Inc(Yk1); Og[Yk1] := v; Nnew [v] :=false;
                                          {в очередь – вершину v }
   While Yk2<Yk1 Do                       { пока очередь не пуста}
     Begin
```

```
      Inc (Yk2); v:=Og[Yk2]; write(v:3);
                                          {берем элемент из очереди}
      For J:=1 To N Do   {просмотр всех вершин, связанных с v}
        If (A[v,j] <>0) And Nnew [j] Then
         Begin                          {если вершина ранее не просмотрена}
           Inc (Yk1);  Og[Yk1] :=J;  Nnew [ J] :=false;
                                          {заносим ее в очередь}
         End;
      End;
End;
```

Время работы алгоритма:

Время работы алгоритма равно квадрату числа вершин.

7.4.3 ВОЛНОВОЙ АЛГОРИТМ.

(кратчайшее расстояние от начальной вершины до остальных вершин.)

1 ВАРИАНТ

Задача *Лабиринт.*

Попав в лабиринт, состоящий из одинаковых квадратных комнат, каждая из которых имеет от 1 до 4 дверей в соседние комнаты, путник долго блуждал по нему, пока не нашёл клад. Во время поиска он составил описание своего маршрута, обозначая каждый переход из комнаты в комнату буквами: С(север), В(восток), Ю(юг), З(запад).

Опишите алгоритм, определяющий по заданной записи самый короткий путь назад.

Решение:

Подходящей структурой данных для представления информации о лабиринте является квадратная таблица L, каждая клетка которой соответствует комнате. Размеры таблицы можно установить заведомо большими лабиринта, для чего достаточно подсчитать количество букв в исходной записи переходов. Значение L(i,j) следует устанавливать так, чтобы по нему можно было определить возможные переходы в соседние комнаты, например, в виде четырёхразрядного двоичного числа, где первый разряд равен 1, если из комнаты (i,j) возможен переход на север, второй разряд соответствует переходу на юг, третий - на запад, а четвёртый - на восток. Вход в лабиринт находится в некоторой клетке (i0,j0).

Алгоритм проведём в два этапа.

На первом этапе занесём в нашу таблицу всю информацию о лабиринте, которая содержится в записи переходов путника. На втором этапе найдём кратчайший путь, ведущий из конечной клетки маршрута в начальную. Для этого применим так называемый волновой алгоритм, состоящий из прямого и обратного ходов. Во время прямого хода

просматриваются комнаты, достижимые от входа за 1 шаг, за 2 шага и т.д. Во время обратного хода строится кротчайший путь, ведущий назад.

Первый этап

Установим 0 во все значения L(i,j).

Установим текущее положение в лабиринте i:=i0, j:=j0.

НЦ пока есть символы в записи

читаем очередной символ S

ЕСЛИ S="С", то

устанавливаем 1 в первый разряд L(i,j),

устанавливаем 1 во второй разряд L(i+1, j) (считаем, что переход чередзверь возможен в обе стороны),

меняем текущее положение i:=i+1;

ЕСЛИ S="Ю", то

устанавливаем 1 во второй разряд L(i,j),

устанавливаем 1 в первый разряд L(i-1,j),

меняем текущее положение i:=i-1;

ЕСЛИ S="З", то

устанавливаем 1 в третий разряд L(i,j),

устанавливаем 1 во четвёртый разряд L(i,j+1),

меняем текущее положение j:=j+1;

ЕСЛИ S="В", то

устанавливаем 1 в четвёртый разряд L(i,j),

устанавливаем 1 во третий разряд L(i,j-1),

меняем текущее положение j:=j-1;

КЦ

Запоминаем ik:=i и jk:=j - координаты комнаты с кладом.

Второй этап

Прямой ход волнового алгоритма .

заведём числовую таблицу V того же размера, что и L. Значением каждого элемента V(i,j) таблицы будет минимально возможное количество шагов, которые ведут из комнаты (i0,j0) в комнату (i,j) в лабиринте. Сначала заполним таблицу V числами -1.

Установим номер очередного шага h:=0.

Присвоим элементу V(i0,j0) значение h.

НЦ ПОКА V(ik,jk)= -1

Просматриваем клетки таблицы (i,j), в которых V(i,j)=h. По информации, содержащейся в L, заносим число h+1 во все клетки таблицы V, соответствующие комнатам смежным комнате (i,j), которые ещё содержат -1 (т.е ещё не проходились). После просмотра увеличиваем h на 1.

КЦ

Обратный ход волнового алгоритма

Установим текущее положение i:=ik, j:=jk.

Установим h:=(i,j).

НЦ пока h<>0

Используя информацию таблицы L, находим комнату, смежную комнате (i,j), в которой записано число h-1, и устанавливаем текущее положение, соответствующее найденной комнате. В зависимости от того, в какую сторону эта комната находится от предыдущей, выводим один из символов : "С", "Ю", "З" или "В".

Уменьшаем h на 1.

КЦ

Программная реализация:

```
type
 ROOM=record
 N,S,W,E:integer
end;
var
 s:string;  i0,j0,ik,jk,i,j,index,h:integer;  l:array[1..50,1..50] of ROOM;
v:array[1..50,1..50] of integer;
begin
 ReadLn(s);
 i0:=25;  j0:=25;  i:=i0;  j:=j0;
 for index:=1 to ord(s[0]) do
  begin
   if s[index]='C' then
    begin
     l[i,j].N:=1;      l[i+1,j].S:=1;      i:=i+1;
    end;
   if s[index]='Ю' then
    begin
     l[i,j].S:=1;
     l[i-1,j].N:=1;
     i:=i-1;
    end;
   if s[index]='З' then
    begin
     l[i,j].W:=1;
     l[i,j-1].E:=1;
     j:=j-1;
    end;
   if s[index]='B' then
    begin
     l[i,j].E:=1;      l[i,j+1].W:=1;      j:=j+1;
    end;
  end;
 ik:=i;  jk:=j;
 for i:=1 to 50 do
  for j:=1 to 50 do
```

```
  v[i,j]:=-1;
 h:=0;
 v[i0,j0]:=h;
 while v[ik,jk]=-1 do
  begin
   for i:=1 to 50 do
    for j:=1 to 50 do
     if v[i,j]=h then
      begin
       if (l[i,j].N=1) and (v[i+1,j]=-1) then
        v[i+1,j]:=h+1;
       if (l[i,j].S=1) and (v[i-1,j]=-1) then
        v[i-1,j]:=h+1;
       if (l[i,j].W=1) and (v[i,j-1]=-1) then
        v[i,j-1]:=h+1;
       if (l[i,j].E=1) and (v[i,j+1]=-1) then
        v[i,j+1]:=h+1;
      end;
    h:=h+1;
  end;
 i:=ik;  j:=jk;   h:=v[i,j];
 while v[i,j]<>0 do
  begin
    if (i+1<=50) and (v[i+1,j]=h-1) and (l[i,j].N=1)
       then
         begin
           Write('C');
           i:=i+1;
         end
       else
          if (i-1>0) and (v[i-1,j]=h-1) and (l[i,j].S=1)
             then
                begin
                  Write('Ю');
                  i:=i-1;
                end              else              if (j-1>0) and(v[i,j-1]=h-1) and
(l[i,j].W=1)
                   then
      begin                        Write('З');                        j:=j-1;
                        end
                 else
                        if (j+1<=50) and(v[i,j+1]=h-1) and (l[i,j].E=1)
                          then
                             begin
                                Write('В');
```

```
                    j:=j+1;
                end;
        h:=h-1;
       end;
      Write(' ');
    end.
```

ВАРИАНТ 2

(с использованием динамической памяти):

```
    Procedure Wave (num_begin, num_end: integer);
        {номера вершин начала и конца пути}

    Var
        k     : integer;     {номер «фронта волны»}
        num   : integer;       {номер текущей вершины}
        flag  : boolean;     {признак необходимости строить     очередной
фронт}
    beg_qu, end_qu, elem_qu : list; {начало, конец элемента очереди}
    Procedure Add (num : integer; var end_qu: list );
    {добавление элемента к очереди}
    Var
        elem_qu: list;
    begin
        end_qu^.num:=num; {поместили элемент в конец очереди}
    new(elem_qu); {выделили память под следующий элемент}
    end_qu^.next:=elem_qu; {присоединили новый элемент}
    end_qu:elem_qu
    end;
    Procedure Extract (var num : integer; var begin_qu : list);
    {извлечь элемент из списка}
    var
    elem_qu: list;
    begin
    num:=beg_qu^.num; {считали первый элемент очереди}
    elem_qu:=beg_qu; {запомнили ссылку на первый элемент для
последующего уничтожения}
    beg_qu:=beg_qu^.next; {переносим указатель начала очереди на второй
элемент}
    Dispose (elem_qu); {освобождаем память, уничтожив первый элемент}
    end;
    begin
    new (elem_qu); {инициализация очереди и размещение в ней первого
элемента}
    beg_qu:=elem_qu; {очередь пока пуста}
```

```
end_qu:=elem_qu;
Graf [num begin].num:= 1; {начальный этап}
Add (num_begin, end_qu); {поместили начало пути в очередь}
Flag:=true; {нужно строить фронт}
While flag and (not goal) do {строим очередной фронт}
begin
Extract (num, beg_qu); {берём значение очередного элемента очереди}
k:= Graf [num].num; {число шагов до извлечённого элемента – 1}
{просмотреть соседей, поместить очередной фронт и добавить
помеченные элементы в очередь}
ref:=Graf[num].next;
repeat
a:=ref^.num;
if a<>0 then begin
{обработка очередного соседа
if Graf [a].num=0
{пометить вершину следующим фронтом}
then begin
Graf[a].num := k+1;
Add(a, end_qu)
End;
ref:=ref^.next
{переходим к следующему соседу}
end;
until a=0;
if Graf[num_end].num<>0
then goal:= true {дошли до цели}
else
if beg_qu = end_qu then flag false {очередь пуста}
end;
end.
```

ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОЙ РЕАЛИЗАЦИИ УЧАЩИМИСЯ

Задача 1. *ПУТЬ.*

Найти кратчайшее расстояние между двумя вершинами в графе. Найти все возможные пути между этими двумя вершинами в графе непересекающиеся по

a) ребрам
b) вершинам.

Задача 2.

Лабиринт задается матрицей смежности N*N, где C(i,j)=1, если узел i связан узлом j посредством дороги. Часть узлов назначается входами, часть

- выходами. Входы и выходы задаются последовательностями узлов X(1),..,X(p) и Y(1),..,Y(k) соответственно.

Найти максимальное число людей, которых можно провести от входов до выходов таким образом, чтобы:

a) их пути не пересекались по дорогам, но могли пересекаться по узлам;
b) их пути не пересекались по узлам;

Задача 3.

N шестеренок пронумерованы от 1 до N (N<= 10). Заданы M (0<= <=M<=45) соединений пар шестеренок в виде (i,j), 1<=i<j<=N (шестерня с номером i находится в зацеплении с шестерней j). Можно ли повернуть шестерню с номером 1?

Если да, то найти количество шестерен, пришедших в движение.

Если нет, то требуется убрать минимальное число шестерен так, чтобы в оставшейся системе при вращении шестерни 1 во вращение пришло бы максимальное число шестерен. Указать номера убранных шестерен (если такой набор не один, то любой из них) и количество шестерен, пришедших в движение.

Задача 4.

Имеется N прямоугольных конвертов и N прямоугольных открыток различных размеров. Можно ли разложить все открытки по конвертам, чтобы в каждом конверте было по одной открытке. Замечание. Открытки нельзя складывать, сгибать и т.п., но можно помещать в конверт под углом. Например, открытка с размерами сторон 5:1 помещается в конверты с размерами 5:1, 6:3, 4.3:4.3, но не входит в конверты с размерами 4:1, 10:0.5, 4.2:4.2.

Задача 5.

Составить программу для нахождения произвольного разбиения 20 студентов на 2 команды, численность которых отличается не более чем в 2 раза, если известно, что в любой команде должны быть студенты, обязательно знакомые друг с другом. Круг знакомств задается матрицей (20,20) с элементами

$$A(ij)=\begin{cases} 1, & \text{если i знаком с j} \\ 0, & \text{иначе} \end{cases}$$

Задача 6.

Имеется N человек и прямоугольная таблица A[1:N,1:N];элемент A[i,j] равен 1, если человек i знаком с человеком j, A[i,j]=A[j,i]. Можно ли разбить людей на 2 группы, чтобы в каждой группе были только незнакомые люди?

Задача 7.

На олимпиаду прибыло N человек. Некоторые из них знакомы между собой. Можно ли опосредованно перезнакомить их всех между собой?

(Незнакомые люди могут познакомиться только через общего знакомого).

Задача 8.

Пусть группа состоит из N человек. В ней каждый имеет (N/2) друзей и не больше K врагов. У одного из них есть книга, которую все хотели бы прочитать и потом обсудить с некоторыми из остальных.

Написать программу, которая:

- Находит способ передачи книги таким образом, чтобы она побывала у каждого в точности один раз, переходя только от друга к другу и наконец возвратилась к своему владельцу.
- Разбивает людей на S групп, где будет обсуждаться книга, таким образом, чтобы вместе с каждым человеком в ту же самую группу вошло не более P его врагов.

Примечание: предполагается, что S*P>=K.

Задача 9.

В заданном графе необходимо определить, существует ли цикл, проходящий по каждому ребру графа ровно один раз.

Задача 10.

Имеется N городов. Для каждой пары городов (I,J) можно построить дорогу, соединяющую эти два города и не заходящие в другие города. Стоимость такой дороги A(I,J). Вне городов дороги не пересекаются.

Написать алгоритм для нахождения самой дешевой системы дорог, позволяющей попасть из любого города в любой другой. Результаты задавать таблицей B[1:N,1:N], где B[I,J]=1 тогда и только тогда, когда дорогу, соединяющую города I и J, следует строить.

Задача 11.

В картинной галерее каждый сторож работает в течение некоторого непрерывного отрезка времени. Расписанием стражи называется множество пар [T1(i), T2(i)] - моментов начала и конца дежурства i-го сторожа из интервала [0,EndTime].

Для заданного расписания стражи требуется:

a) проверить, в любой ли момент в галерее находится не менее двух сторожей. Если условие (a) не выполнено, то:

b) перечислить все интервалы времени с недостаточной охраной (менее 2 сторожей).

c) добавить наименьшее число сторожей с заданной, одинаковой для всех длительностью дежурства, чтобы получить правильное расписание (т.е. удовлетворяющее условию (a)).

d) проверить, можно ли обойтись без добавления новых сторожей, если разрешается сдвигать времена дежурства каждого из сторожей с сохранением длительности дежурства.

e) если это возможно, то составить расписание с наименьшим числом сдвигов.

Задача 12.

Вводится N - количество домов и K - количество дорог. Дома пронумерованы от 1 до N. Каждая дорога определяется тройкой чисел - двумя номерами домов - концов дороги и длиной дороги. В каждом доме живет по одному человеку. Найти точку - место встречи всех людей, от которой суммарное расстояние до всех домов будет минимальным.

Если точка лежит на дороге, то указать номера домов – концов этой дороги и расстояние от первого из этих домов. Если точка совпадает с домом, то указать номер этого дома.

Примечание: длины дорог - положительные целые числа.

Задача 13.

N колец сцеплены между собой (задана матрица A(n*n), A(i,j)=1в случае, если кольца i и j сцеплены друг с другом и A(i,j)=0 иначе). Удалить минимальное количество колец так, чтобы получилась цепочка.

Задача 14.

Янка положил на стол N выпуклых K-гранников и N различных типов наклеек по K штук каждая. Ночью кто-то наклеил наклейки на грани, по одной на грань. Помогите Янке расставить многогранники так, чтобы наклейка каждого типа была видна ровно K-1 раз.

Задача 15.

Имеется N точек и M проводков. Проводком можно соединить некоторую пару различных точек, причем пара может быть соединена не более чем одним проводком. Все проводки должны быть использованы.

Пусть Di - количество проводков, которые будут соединены с точкой с номером i, i=1, ..., N.

Необходимо соединить N точек с помощью M проводков таким образом, чтобы сумма S=D1*D1 + D2*D2 + ... + Dn*Dn была максимальной.

Вывести величины Di в неубывающем порядке и. по требованию (priznak=1), список соединений.

ВВОД:
<Введите N:> N (N<=100)
<Введите M:> M (M<=1000)
<PRIZNAK=> PRIZNAK
ВЫВОД:
<Результирующая конфигурация:> Di в неубывающем порядке.
<Сумма S> S
<Список соединений>

<Точку 1 соединить с> список точек

.....

<Точку N соединить с> список точек

Задача 16.

Задано N городов с номерами от 1 до N и сеть из M дорог с односторонним движением между ними. Каждая дорога задается тройкой (i, j, k), где i - номер города, в котором дорога начинается, j - номер города, в котором дорога заканчивается, а k - ее длина (число k - натуральное). Дороги друг с другом могут пересекаться только в концевых городах.

Все пути между двумя указанными городами A и B можно упорядочить в список по неубыванию их длин (если есть несколько путей одинаковой длины, то выбираем один из них). Необходимо найти один из путей, который может быть вторым в списке.

Вывести его длину L и города, через которые он проходит.

ВВОД:

<Количество городов N:> N

<Количество дорог M:> M

<Начало, конец и длина дороги 1:> i1, j1, k1

.....

<Начало, конец и длина дороги M:> iM, jM, kM

<Города A и B, между которыми надо найти путь:> A, B

ВЫВОД:

<Пути нет>

или

<Путь проходит по городам> A, i1, i2, ..., B

<Длина пути> L

Задача 17. *Ларсона*

Пусть G - конечный неориентированный связный граф. Предположим, что он представляет собой систему тоннелей, в которых может прятаться беглец. Группа из S полицейских, двигаясь по туннелям, стремится схватить этого беглеца, который может двигаться с любой скоростью, стремясь избежать поимки. Требуется определить минимальное количество полицейских S, гарантирующих поимку беглеца.

ЛИТЕРАТУРА:

1. Шень А. Программирование: теоремы и задачи. М.:МЦНМО, 1995.
2. Окулов С.М., Пестов А.А., Пестов О.А. Информатика в задачах.: Киров: Вятский госпедуниверситет, 1998.
3. Андреева Е., Фалина И. Системы счисления и компьютерная арифметика.
   М.: Лаборатория базовых знаний, 2000.
4. Андреева Е. Принципы проверки учебных и олимпиадных задач по информатике. //Информатика №34, 2001.
5. Юркин А. Практикум по программированию. Барнаул: АГУ,1999.
6. Черкасова П. Компьютер и графы. Спб.: ЦПО "Информатизация образования", 2000.
7. Крючкова Е. Теория алгоритмов и формальных языков.: Барнаул: АлтГТУ, 2000.
8. Кормен Т., Лейзерсон .Ч, Ривест Р. Алгоритмы: построение и анализ. Москва : МЦНМО, 1999.
9. Столяр С. Алгоритмы. СПб.: ЦПО "Информатизация образования", 2000.