ОБРАБОТКА МАССИВОВ

ОБРАБОТКА ЧИСЛОВЫХ МАССИВОВ

Пример 1. Переворот одномерного целочисленного массива – перестановка его элементов в обратном порядке.

Выделим в отдельную функцию процедуру инвертирования:

```
void invert(int *a,int n)
{
 for(int j=0,tmp; j<n/2; j++)
 {
 tmp=a[j];
 a[j]=a[n-j-1];
 a[n-j-1]=tmp;
 }
}
```

ОБРАБОТКА ЧИСЛОВЫХ МАССИВОВ

Для проверки ее работоспособности можно воспользоваться следующей программой:

```
#include <stdio.h>
#include <conio.h>
#define N 20
 void invert(int *a,int n);
 void main()
 { int j,a[N];
 printf("Before reverse:\n");
  for(j=0; j<N; j++)
        { a[j]=j+1; printf("%3d",a[j]); }
       invert(a,N);
        printf("\nAfter reverse:\n");
  for(j=0; j<N; j++) printf("%3d",a[j]);
getch();
}
```

ОБРАБОТКА ЧИСЛОВЫХ МАССИВОВ

Пример 2. Перестановка головы и хвоста массива без использования промежуточного массива.

Алгоритм этой процедуры заключается в том, что надо последовательно выполнить 3 инвертирования – головы массива, хвоста массива и всего массива целиком. Для этой цели воспользуемся модификацией ранее написанной процедурой invert:

```
void invert1(int *a, int k, int n){
//k – индекс первого элемента инвертируемого фрагмента массива
 //n – количество инвертируемых элементов
 int j,tmp;
  for(j=k; j<k+n/2; j++)
  { tmp=a[j];
  a[j]=a[2*k+n-j-1];
  a[2*k+n-j-1]=tmp; }
}
```

ОБРАБОТКА ЧИСЛОВЫХ МАССИВОВ

Пример 2. Для проверки описанного выше алгоритма можно воспользоваться следующей программой:

```
#include <stdio.h>  #include <conio.h>  #define N 20  #define M 15
void invert1(int *a, int k, int n);
void main()
      { int j,a[N];
printf("Before reverse:\n");
for(j=0; j<N; j++)
       { a[j]=j+1; printf("%3d",a[j]); }
       invert1(a,0,M);
       printf("\nAfter reverse head:\n");
for(j=0; j<N; j++) printf("%3d",a[j]);
      invert1(a,M,N-M);
       printf("\nAfter reverse tail:\n");
 for(j=0; j<N; j++) printf("%3d",a[j]);
       invert1(a,0,N);
       printf("\nAfter reverse all:\n");
 for(j=0; j<N; j++)  printf("%3d",a[j]);
getch();
}
```

ОБРАБОТКА ЧИСЛОВЫХ МАССИВОВ

Пример 3. Количество "счастливых" билетов. Билеты для общественного транспорта хранятся в рулонах и идентифицируются номерами от 000000 до 999999. «Счастливым" считается билет, у которого сумма трех первых цифр совпадала с суммой трех последних цифр.

Самый простой способ заключается в организации 6 вложенных друг в друга циклов, где каждый счетчик перебирает цифры от 0 до 9, а во внутреннем цикле сравниваются суммы первых трех и последних трех счетчиков:

```
#include <iostream.h>
#include <conio.h>void main()
{ int n=0;
 for(int j1=0; j1<10; j1++)  for(int j2=0; j2<10; j2++)
 for(int j3=0; j3<10; j3++)  for(int j4=0; j4<10; j4++)
 for(int j5=0; j5<10; j5++)  for(int j6=0; j6<10; j6++)
 if(j1+j2+j3==j4+j5+j6) n++;
 cout << n;
 getch();
}
```

ОБРАБОТКА ЧИСЛОВЫХ МАССИВОВ

Работу этой программы можно ускорить почти в 1000 раз за счет использования массивов. Сумма трех цифр принадлежит диапазону [0,27]. Допустим, мы подсчитали, сколько раз встретилась каждая сумма – s0 (количество сочетаний, давших сумму 0), s1 (количество сочетаний, давших сумму 1), ... Тогда количество "счастливых" билетов будет равно (s0)

2+(s1)

2+ ....

Поэтому гораздо более быстрой программой будет следующая:

```
#include <iostream.h>   #include <conio.h>
void main()
{ int n=0,k,s[28];
 for(k=0; k<28; k++) s[k]=0;
 for(int j1=0; j1<10; j1++)
 for(int j2=0; j2<10; j2++)
 for(int j3=0; j3<10; j3++)            s[j1+j2+j3]++;
 for(k=0; k<28; k++)        n += s[k]*s[k];
 cout << n;
 getch();
}
```

ОБРАБОТКА ЧИСЛОВЫХ МАССИВОВ

Пример 4. Определение количества разных элементов в целочисленном массиве.

Первый вариант программы основан на предварительной сортировке исходного массива. После того как массив отсортирован, следует проанализировать соседние элементы и, как только встречается пара разных чисел, к счетчику надо добавлять 1.

```
#include <stdio.h>          #include <conio.h>
void sort(int *a,int n)
{ int tmp;
 for(int i=0;i<n-1;i++)
 for(int j=i+1;j<n; j++)
 if(a[j]<a[i])   {tmp=a[i]; a[i]=a[j]; a[j]=tmp; }
}
int difference(int *a,int n)
{ int i,m=1;
 sort(a,n); //сортировка исходного массива
 for(i=0; i<n-1; i++)
 if(a[i]!=a[i+1]) m++;
 return m;
}
```

ОБРАБОТКА ЧИСЛОВЫХ МАССИВОВ

```
void main()
{ int a0[5]={0,0,0,0,0};
 int a1[5]={1,1,1,1,1};
 int a2[5]={0,1,1,1,1};
 int a3[5]={0,0,1,1,2};
 int a4[5]={0,1,2,3,4};
 int a5[5]={1,2,3,4,5};
 printf("\na0:%d",difference(a0,5));
 printf("\na1:%d",difference(a1,5));
 printf("\na2:%d",difference(a2,5));
 printf("\na3:%d",difference(a3,5));
 printf("\na4:%d",difference(a4,5));
 printf("\na5:%d",difference(a5,5));
 getch();
}
```

```
//= Результат работы =
a0:1
a1:1
a2:2
a3:3a4:5
a5:5
```

ОБРАБОТКА ЧИСЛОВЫХ МАССИВОВ

Вариант 2. При первом просмотре определяем, содержится ли в исходном массиве хотя бы один нулевой элемент. Если содержится, то в переменную k0 заносим 1, в противном случае в k0 заносим 0. При втором проходе нулевые элементы исключаем из рассмотрения и сравниваем a[i] с a[j]. В случае равенства в элемент a[i] заносим 0. При третьем проходе подсчитываем ненулевые элементы и добавляем к сумме k0. Мы ограничимся только видоизмененной функцией difference:

```
int difference(int *a,int n)
{ int i,j,k0=0,m=0;
 for(i=0; i<n; i++)                  //первый проход
 if(a[i]==0) { k0=1; break; }        //поиск нулевого элемента
 for(i=0; i<n-1; i++)                //второй проход
 { if(a[i]==0) continue;             //обход нулей
 for(j=i+1; j<n; j++)                //поиск дубликатов
 if(a[i]==a[j])
 { a[i]=0; break; } }                //забой дубликата
  for(i=0;i<n;i++)                   //подсчет ненулевых элементов
 if(a[i]!=0) m++;
 return m+k0;
}
```

РАБОТА С ВЕКТОРАМИ

Пример 5. Вычисление *нормы вектора*.

```
#include <stdio.h>
#include <math.h>   //здесь находится прототип функции sqrt
#include <conio.h>
double norm(double *a, int n)
      { double s=0;
      for(int i=0; i<n; i++)  s += a[i]*a[i];
      return sqrt(s);}
void main()
{     double v1[5]={1.,2.,3.,4.,5.};
      printf("norm=%f",norm(v1,5));
      getch();}

//=== Результат работы ===norm=7.416198
```

Функцией norm можно воспользоваться и для того, чтобы вычислить "норму" любого фрагмента вектора – достаточно вместо первого аргумента задать адрес начальной компоненты (например, &v1[2]), а в качестве второго аргумента количество обрабатываемых компонент.

РАБОТА С ВЕКТОРАМИ

Пример 7. Нормирование вектора.

```
#include <iostream.h>
#include <math.h>   //здесь находится прототип функции sqrt
#include <conio.h>
void norm_vec(double *a, int n)
      { double s=0;
      for(int i=0; i<n; i++) s += a[i]*a[i];
      s= sqrt(s);
      for(int i=0; i<n; i++) a[i] /= s;
      }
void main()
      { double v1[5]={1.,2.,3.,4.,5.};
      norm_vec(v1,5);
      for(int i=0;i<5;i++)
      cout << v1[i]<< "  ";
      getch();
}
```

//=== Результат работы ===

0.13484 0.26968 0.40452 0.53936 0.6742

РАБОТА С ВЕКТОРАМИ

Пример 8. Вычисление скалярного произведения двух векторов.

```
#include <stdio.h>
#include <conio.h>
double scal_prod(double *a, double *b,int n)
{     double s=0;
      for(int i=0; i<n; i++) s += a[i]*b[i];
      return s;
}
void main()
      { double v1[5]={1.,2.,3.,4.,5.};      printf("scal_prod=%f",
      scal_prod(v1,v1,5));
      getch();}
//=== Результат работы ===scal_prod=55.000000
```

РАБОТА С ВЕКТОРАМИ

Пример 9. Сумма векторов.

```
#include <stdio.h>
#include <conio.h>
void sum_vec(double *a,double *b,double *c,int n)
      { for(int i=0; i<n; i++) c[i]=a[i]+b[i]; }
void main()
      { double v1[5]={1.,2.,3.,4.,5.};
        double v2[5]={1.,0.,1.,0.,1.};
      double v3[5];
      sum_vec(v1,v2,v3,5);
      for(int i=0;i<5;i++)
            printf("%3.0f",v3[i]);
      getch();
      }
//=== Результат работы ===  2  2  4  4  6
```

РАБОТА С МАТРИЦАМИ

Работу с двумерными массивами можно организовать двумя способами. Во-первых, операции над элементами двумерных массивов можно свести к операциям над одномерными массивами, используя приведенные индексы. Во-вторых, можно воспользоваться указателями на строки матрицы (как известно, имя массива одновременно является указателем на ее первую строку).

РАБОТА С МАТРИЦАМИ

Пример 10. Формирование единичной матрицы с приведенными индексами.

```
#include <stdio.h>
#include <math.h>
#include <conio.h>
void eye(int *a, int n)
{ int i,j;
  for(i=0; i<n; i++)
    for(j=0; j<n;j++)
      { if(i==j) a[i*n+j]=1;
        else a[i*n+j]=0;
      }
}
```

РАБОТА С МАТРИЦАМИ

```
void main()
{ int i,j,v[5][5];
  eye((int*)v,5);
  for(i=0;i<5;i++)
    { for(j=0;j<5;j++)
        printf("%3d",v[i][j]);
      printf("\n");
    }
  getch();
}
```

//=== Результат работы

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

РАБОТА С МАТРИЦАМИ

Пример 11. Сложение квадратных матриц.

Поскольку в операции сложения участвуют n^2 одноименных компонент матриц-слагаемых, то матрицы можно рассматривать как длинные вектора и производить сложение как с векторами:

```
#include <stdio.h>
#include <math.h>
#include <conio.h>
void add_mat(int *a,int *b,int *c,int n)
      { int i;
        for(i=0; i<n*n; i++)
            c[i]=a[i]+b[i];}
```

РАБОТА С МАТРИЦАМИ

```
void main()
{ int i,j,v3[3][3];
  int v1[3][3]={{1,2,3},{4,5,6},{7,8,9}};
  int v2[3][3]={{0,0,1},{0,0,2},{0,0,3}};
  add_mat((int*)v1,(int*)v2,(int*)v3,3);
  for(i=0;i<3;i++)
    { for(j=0;j<3;j++)
        printf("%3d",v3[i][j]);
      printf("\n");
    }
  getch();
}
```

//=== Результат работы ===

```
  1  2  4
  4  5  8
  7  8 12
```

РАБОТА С МАТРИЦАМИ

Пример 12. Умножение квадратных матриц с использованием приведенных индексов.

```
#include <stdio.h>
#include <conio.h>
void mult_mat(int *a,int *b,int *c,int n)
{ int i,j,k,s;
  for(i=0; i<n; i++)
    for(j=0; j<n;j++)
      { s=0;
        for(k=0;k<n;k++)
          s += a[i*n+k]*b[k*n+j]; //s=s+ai,k*bk,j
        c[i*n+j]=s;                    //ci,j=s
      }
}
```

РАБОТА С МАТРИЦАМИ

```
void main()
{ int i, j, v3[2][2];
  int v1[2][2]={{1,2}, {3,4}};
  int v2[2][2]={{5,6}, {7,8}};
  mult_mat((int*)v1, (int*)v2,(int*)v3,2);
  for(i=0;i<2;i++)
    { for(j=0;j<2;j++)
        printf("%4d",v3[i][j]);
      printf("\n");
    }
  getch();
}
```

//=== **Результат работы** ===

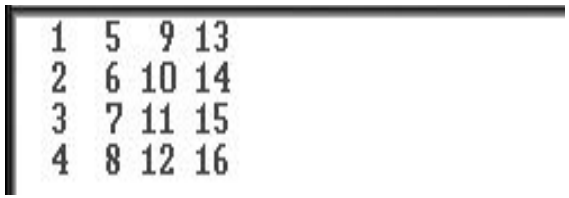
РАБОТА С МАТРИЦАМИ

Пример 13. Транспонирование матрицы с использованием массива указателей на строки.

```
#include <stdio.h>
#include <conio.h>
void transp(int *p[],int n)
{ int tmp,i,j;
  for(i=0; i<n-1; i++)
    for(j=i+1; j<n; j++)
      { tmp=p[i][j]; p[i][j]=p[j][i]; p[j][i]=tmp; }
}
void main()
{ int v[4][4]={{ 1, 2, 3, 4},
               { 5, 6, 7, 8},
               { 9,10,11,12},
               {13,14,15,16}};
```

РАБОТА С МАТРИЦАМИ

```
//массив указателей на строки
  int *p[4]={(int *)&v[0],(int *)&v[1],(int *)&v[2],(int *)&v[3]};
  transp(p,4);
  for(int i=0; i<4; i++)
    { for(int j=0;j<4;j++)
        printf("%3d",v[i][j]);
      printf("\n");
    }
  getch();
}
//=== Результат работы ===
```



РАБОТА С МАТРИЦАМИ

Массив указателей p на строки двумерного массива v может быть сформирован и другими способами:

```
int *p[4]={(int *)v,(int *)(v+1),(int *)(v+2),(int *)(v+3)};
int *p[4]={v[0],v[1],v[2],v[3]};
int *p[4]={*v,*(v+1),*(v+2),*(v+3)};
```

Очевидно, что указатель p[0] "смотрит" на элемент v[0][0]. Поэтому указатель p[0][1]=p[0]+1 "смотрит" на элемент v[0][1], указатель p[0][2] – на элемент v[0][2] и т.д. Можно было бы видоизменить заголовок функции transp следующим образом:

```
void transp(int **p,int n)
```

Все эти модификации ничего не меняют в алгоритмах работы программ.

ПОИСК

Задача поиска формулируется следующим образом: задан массив a, содержащий n однотипных элементов (чисел, строк, записей и т.п.). Нужно установить, содержится ли в этом массиве заданный объект q. При положительном ответе следует дополнительно сообщить порядковый номер (индекс j), найденного объекта (a[j]=q).

ПОСЛЕДОВАТЕЛЬНЫЙ ПОИСК

Классический алгоритм *последовательно поиска* в неупорядоченном массиве состоит из четырех следующих шагов:

шаг S1: Установить начальный индекс j=1 ;

шаг S2: Проверить условие q=a[j]. Если условие выполнено, то решение найдено и работа прекращается;

шаг S3: Увеличить индекс j на 1;

шаг S4: Проверить условие окончания цикла j<n+1. Если условие выполнено, повторяется шаг S2. В противном случае сообщить, что объект q в массиве a не содержится.

Трудоемкость классического последовательного поиска можно оценить только в среднем. В лучшем случае первое же сравнение может дать ответ (q=a[1]). В худшем случае придется перебрать все n элементов. В среднем на поиск будет затрачено n/2сравнений.

ПОСЛЕДОВАТЕЛЬНЫЙ ПОИСК

Функция ssearch, реализующая *последовательный поиск*, устроена довольно просто:

```
int ssearch(int q, int *a, int n)
{ register int j;
for(j=0; j<n;j++)
if(q==a[j]) return j;
return -1;
}
```

ДВОИЧНЫЙ ПОИСК

Двоичный поиск можно применить только в том случае, если исходный массив упорядочен, например, по возрастанию величин объектов. Тривиальные случаи типа q<a[1] или q>a[n] не рассматриваются, хотя ничего не стоит подключить к поиску и такие проверки. Идея двоичного поиска заключается в уменьшении вдвое зоны поиска на каждом шаге (отсюда и второе название метода – деление пополам). Сначала искомый объект q сравнивается со средним элементом массива. В зависимости от результата сравнения на следующий шаг остается первая или вторая половина массива. Оставшаяся половина вновь делится на 2, и так продолжается до тех пор, пока зона поиска не сузится до двух элементов. В этом случае либо объект q совпадает с одним из этих элементов, либо продолжает сохраняться строгое неравенство с обеими границами.

ДВОИЧНЫЙ ПОИСК

Функция bsearch, реализующая двоичный поиск, может быть организована следующим образом:

```
int bsearch(int q, int *a, int n)
{ register int left=0,right=n-1,mid;
if(q<a[0] || q>a[n-1]) return -1;
for(;left<=right;)
{ mid=(left+right)/2;
if(q<a[mid]) right=mid-1;
else if(q>a[mid]) left=mid+1;
else return mid;
}
return -1;
}
```

Максимальное количество шагов, которое требуется для двоичного поиска, оценивается ближайшим целым к $\log_2 n$. Для массива в 1000 элементов прямой поиск в среднем затрачивает 500 шагов, тогда как двоичный поиск ограничивается 10 шагами.

СОРТИРОВКА МАССИВОВ
СОРТИРОВКА МЕТОДОМ ПУЗЫРЬКА

Идея метода состоит в сравнении двух соседних элементов, в результате чего меньшее число (более легкий "пузырек") перемещается на одну позицию влево. Обычно просмотр организуют с конца, и после первого прохода самое маленькое число перемещается на первое место. Затем все повторяется от конца массива до второго элемента и т.д. Известен и другой вариант пузырьковой сортировки, в котором также сравнивают два соседних элемента, и если хотя бы одна из смежных пар была переставлена, то просмотр начинают с самого начала

СОРТИРОВКА МЕТОДОМ ПУЗЫРЬКА

Функция bubble, реализующая первый алгоритм пузырьковой сортировки приведена ниже:

```
void bubble(int *x, int n)
{ register int i,j;
  int tmp;
  for(i=1;i<n;i++)
    for(j=n-1;j>=i; j--)
      if(x[j-1]>x[j])
        { tmp=x[j-1]; x[j-1]=x[j]; x[j]=tmp; }
}
```

СОРТИРОВКА МЕТОДОМ ПУЗЫРЬКА

Более известный алгоритм пузырьковой сортировки реализован в функции bubble1. В ней использована флажковая переменнаяq, которая принимает ненулевое значение в случае перестановки какой-либо смежной пары:

```
void bubble1(int *x, int n)
{ register int i,j;
  int tmp,q;
m: q=0;
  for(i=1;i<n-1;i++)
    if(x[i]>x[i+1])
      { tmp=x[i]; x[i]=x[i+1]; x[i+1]=tmp; q=1;}
  if(q) goto m;
}
```

Пузырьковая сортировка эффективна, когда в исходных данных многие элементы уже упорядочены. Если исходный массив уже отсортирован, то работа функции ограничивается первым проходом. В худшем случае (массив упорядочен по убыванию) количество сравнений составляет n*(n-1)/2, а количество перестановок достигает 3*n*(n-1)/2. Среднее количество перестановок равно 3*n*(n-1)/4.

СОРТИРОВКА МЕТОДОМ ОТБОРА

Идея метода: находится элемент с наименьшим значением и меняется местами с первым элементом. Среди оставшихся элементов ищется наименьший, который меняется со вторым и т.д. Функция select, реализующая такую процедуру, приведена ниже:

```
void select(int *x, int n)
{ register int i,j,k;
  int q,tmp;
  for(i=0; i<n-1;i++)
    { q=0; k=i; tmp=x[i];
      for(j=i+1; j<n; j++)
        { if(x[j]<tmp)
            { k=j; tmp=x[j]; q=1; }
        }
      if(q) { x[k]=x[i]; x[i]=tmp; }
    }
}
```

СОРТИРОВКА МЕТОДОМ ОТБОРА

Оценка трудоемкости метода отбора:

- количество сравнений – n*(n-1)/2 ;
- количество перестановок:
  - в лучшем случае – 3*(n-1)
  - в худшем случае – n2/4+3*(n-1)
  - в среднем – n*(log n +0.577216)

СОРТИРОВКА МЕТОДОМ ВСТАВКИ

Идея метода: последовательное пополнение ранее упорядоченных элементов. На первом шаге сортируются два первых элемента. Затем на свое место среди них вставляется третий элемент. К трем упорядоченным добавляется четвертый, который занимает свое место в четверке и т.д. Примерно так игроки упорядочивают свои карты при сдаче их по одной. Функция insert, реализующая описанную процедуру:

```
void insert(int *x, int n)
{ register int i,j;
  int tmp;
  for(i=1;i<n;i++)
    { tmp=x[i];
      for(j=i-1;j>=0 && tmp<x[j]; j--)
        x[j+1]=x[j];
      x[j+1]=tmp;
    }
}
```

Трудоемкость метода: количество сравнений зависит от исходной упорядоченности массива. Если массив уже отсортирован, то все равно потребуется 2*(n-1) сравнение. Если массив упорядочен по убыванию, то число сравнений возрастает до n*(n+1)/2.

СОРТИРОВКА МЕТОДОМ ВСТАВКИ

Идея метода: последовательное пополнение ранее упорядоченных элементов. На первом шаге сортируются два первых элемента. Затем на свое место среди них вставляется третий элемент. К трем упорядоченным добавляется четвертый, который занимает свое место в четверке и т.д. Примерно так игроки упорядочивают свои карты при сдаче их по одной. Функция insert, реализующая описанную процедуру:

```
void insert(int *x, int n)
{ register int i,j;
  int tmp;
  for(i=1;i<n;i++)
    { tmp=x[i];
      for(j=i-1;j>=0 && tmp<x[j]; j--)
        x[j+1]=x[j];
      x[j+1]=tmp;
    }
}
```

Трудоемкость метода: количество сравнений зависит от исходной упорядоченности массива. Если массив уже отсортирован, то все равно потребуется 2*(n-1) сравнение. Если массив упорядочен по убыванию, то число сравнений возрастает до n*(n+1)/2.