```
int N;
struct DataStr * Data;
Data = (struct DataStr *)malloc(Nmax*sizeof(struct DataStr));
if(Data==NULL)exit(1);
N = readData(Data);
prnData(Data,N);
```

Рабочие функции:

```
int readData(struct DataStr *Data){
int n=0;
while(fscanf(stdin,"%s %s %d %d",
 Data[n].Fname,  Data[n].Lname),
 &(Data[n].id),  &(Data[n].dq)==4)n++;
return n;
}
void prnData(struct DataStr *Data, int N){
int n=0;
for(n=0;n<N;n++){
 fprintf(stdout,"%s %s %d %d\n",
 Data[n].Fname,  Data[n].Lname,
 Data[n].id,  Data[n].dq);
}
return;
}
```

1.185. С экрана последовательно вводятся строки, см. задачу 1.184, следующего формата

<Фамилия> <Имя> <id> <dq>

Требуется считать информацию, сохранив ее в памяти ЭВМ, и выдать на экран имена и <id> всех пользователей, имеющих значение <dq> не менее 1000, упорядочив их по алфавиту. Если встречаются пользователи с одинаковой фамилией, то их необходимо упорядочить по содержимому <id>.

Указание. Отсортировать введенный массив по совокупности полей <Фамилия>+<id>, реализовав соответствующую функцию сравнения, и распечатать пользователей с допустимым <dq>.

1.7 Разбор чисел и битовые операции

При решении задач данного раздела можно анализировать остатки от деления целых чисел, полученных с помощью оператора `%`, либо использовать битовые операции `<<, >> , ~, ^, |, &`, соответственно обеспечивающие сдвиг влево, вправо, отрицание (not), исключающее или (xor, сложение по модулю 2), побитовое или (or, логическое сложение), побитовое и (and, логическое умножение).

1.186. Получить массив целых чисел `int b[64]`, хранящий двоичное разложение неотрицательного целого числа $n = b[0] + b[1] \cdot 2 + \ldots + b[63] \cdot 2^{63}$, $n < 2^{64}$.
Решение.

```
int longint2binary(unsigned long int n, int *b)
{
   unsigned long int mask=1;
   int i;
     for(i=0;mask;i++){
        if(n&mask){
            b[i]=1;
        }
        else{
             b[i]=0;
        }
         mask=mask<<1;
      }
  return i;
}
```

В данном случае возвращаемое значение позволит проверить размер типа `long int`.

1.187. Получить массив целых чисел, соответствующий реальному битовому представлению заданного символа в памяти компьютера.
Решение.

```
void char_bits (char c, int *b)
{
   unsigned char mask=1<<7;
   int i;
     for(i=0;mask;i++){
        if(c&mask){
          bit[i]=1;
        }
        else{
          bit[i]=0;
        }
         mask>>=1;
      }
  return;
}
```

1.188. Получить массив целых чисел, соответствующий реальному битовому представлению целого числа типа `int` в памяти компьютера.
Решение. Так как для хранения переменных типа `int` используется несколько байт, а последовательность их размещения в памяти зависит от архитектуры ЭВМ, аппаратных настроек, выбранной сборки ОС и обычно бывает двух типов: от старшего к младшему (big-endian) или от младшего к старшему (little-endian), то недостаточно в решении задачи 1.201 формально заменить `char` на `int`.

```
void longint_bits(long int n, int *b)
{
   char *c;
   unsigned char mask=1;
   int Nb = sizeof(long int);
   int i;

   c=(char *)(&n);
   for(i=0;i<Nb;i++){
    char_bits (*c,b+i*8);
    c++;
   }
  return;
}
```

Отметим, что при работе с переменными, длина которых превышает машинное слово, возможен т.н. смешанный (mixed-endian) порядок байт.

1.189. В массиве целых чисел установите k-й по счету бит (с учетом сквозной нумерации слева на право) равным единице.

1.190. В массиве целых чисел установите k-й по счету бит (с учетом сквозной нумерации слева на право) равным нулю.

Указание. Использовать либо маску типа `mask=1<<i; mask=~mask` и операцию побитового умножения; либо, сначала проверить, что соответствующий бит равен 1, а далее использовать маску `mask=1<<i;` и операцию xor.

1.191. Для целого числа получить массив $b[\,]$ с цифрами его записи в k-ричной системе исчисления при $k \le 2^8$, $k \le 2^{64}$. Например, для числа 254 при $k = 16$ нужно получить $b[0] = 14$, $b[1] = 15$, $b[i] = 0$ при $i > 1$.

1.192. На основе задачи 1.191 реализовать функции сложения и вычитания "длинных" чисел, записанных в k-ричной системе исчисления.

1.193. С экрана вводятся две последовательности цифр, представляющие собой целую и дробную части некоторого положительного десятичного числа. Требуется найти его двоичное представление, сохранив результат в отдельных массивах (см. задачу 1.186).

1.194. Дано целое число. Получить целое число, записанное теми же цифрами в обратном порядке.

1.195. Вычислить представление числа m/n в виде десятичной дроби (т.е. вывести цифры, составляющие начало и период этой дроби).

Идеи реализации. Можно запасти массивы необходимых размеров и реализовать алгоритм деления "столбиком", запоминая получающиеся остатки. Период появится тогда, когда мы получим два одинаковых остатка. Можно также проанализировать делители чисел m и n, что даст возможность заранее вычислить длины начальной части и периода дроби. В этом случае массивы становятся не нужны, так как получаемые цифры частного можно сразу выводить на печать.

1.196. Требуется реализовать функцию возведения числа x в заданную целую степень N за не более чем $2\log_2 N$ умножений.

Идеи реализации. Найдем разложение $N = n_0 2^0 + n_1 2^1 + \ldots + n_k 2^k$. Тогда $x^N = x^{n_0}(x^2)^{n_1} \ldots (x^{2^k})^{n_k}$. Поэтому можно последовательно вычислить значения x^1, x^2, x^4, и т.д. и перемножить те из них, которые соответствуют единицам в двоичном представлении числа N.

1.197. Решите уравнение $b_1x+b_2y = 1$ в целых числах для взаимно простых b_1, b_2.

Указание. Известно, что данная задача, называемая диофантовым уравнением, имеет бесконечно много решений. Сначала найдем одно из них. Пусть $b_1 > b_2$. Тогда алгоритм Евклида порождает цепочку коэффициентов b_k, p_k и соответствующую систему уравнений (2):

$$(1)\ \begin{cases} b_1 = b_2p_2 + b_3, \\ \ldots\ldots\ldots \\ b_{k-1} = b_kp_k + b_{k+1}, \\ b_k = b_{k+1}p_{k+1} + b_{k+2}, \\ \ldots\ldots\ldots \\ b_{N-1} = b_Np_N + 1; \end{cases} \qquad (2)\ \begin{cases} b_1x_1 + b_2y_1 = 1, \\ \ldots\ldots\ldots \\ b_{k-1}x_{k-1} + b_ky_{k-1} = 1, \\ b_kx_k + b_{k+1}y_k = 1, \\ \ldots\ldots\ldots \\ b_{N-1}x_{N-1} + b_Ny_{N-1} = 1. \end{cases}$$

Сравнивая последние строки (1) и (2) находим значения $x_{N-1} = 1, y_{N-1} = -p_N$. Если же для k-го уравнения $b_kx_k+b_{k+1}y_k = 1$ найдено решение x_k, y_k, то из $(k-1)$-го соотношения $b_{k-1} = b_kp_k+b_{k+1}$ имеем $b_{k-1}y_k+b_k(x_k-p_ky_k) = 1$. Следовательно, $x_{k-1} = y_k$, $y_{k-1} = x_k - p_ky_k$. Так как при $k = N - 1$ решение $x_{N-1} = 1$, $y_{N-1} = -p_N$ получено, следовательно рекуррентная формула позволяет найти частное решение x_1, y_1 для исходной задачи $b_1x_1+b_2y_1 = 1$ и вычислить полное решение по формуле: $x = x_1+k\cdot b_2$, $y = y_1-k\cdot b_1$ при $k = 0, \pm1, \pm2, \ldots$.

Приведем одну из возможных реализаций рассмотренного алгоритма.

```
int DiophEqI(double * x, double *y, double * p, double *b){
 int n,N,k;
scanf("%d",b+1);
scanf("%d",b+2);
n=2;
while(1){
 p[n]=b[n-1]/b[n];
 b[n+1]=b[n-1]%b[n];
 if(b[n+1]==1)break;
 n++;
}
N=n;
for(k=2;k<=N;k++){
 printf("k=%d p=%d b=%d\n",k,p[k],b[k+1]);
}
x[N-1]=1; y[N-1]=-p[N];
for(k=N-1;k>1;k--){
 x[k-1]=y[k]; y[k-1]=x[k]-p[k]*y[k];
}
 printf("(%d) * (%d) + (%d) * (%d) = 1\n",b[1],x[1],b[2],y[1]);
```

```
return 1;
}
```

Указанный алгоритм можно переформулировать следующим образом. Запишем систему (3) тождеств для b_1, b_2. Затем, поделив b_1 на b_2 с остатком, найдем разложение $b_1 = b_2p_2 + b_3$ и вычтем из первого уравнения второе, умноженное на p_2. Получим систему (4):

$$(3)\ \begin{cases} 1 \cdot b_1 + 0 \cdot b_2 = b_1, \\ 0 \cdot b_1 + 1 \cdot b_2 = b_2; \end{cases} \qquad (4)\ \begin{cases} 1 \cdot b_1 - p_2 \cdot b_2 = b_3, \\ 0 \cdot b_1 + 1 \cdot b_2 = b_2. \end{cases}$$

Для системы уравнений (4) вычислим разложение $b_2 = b_3p_3 + b_4$ и вычтем из второго уравнения первое, умноженное на p_3. Такую процедуру будем повторять до тех пор, пока не получим равенство $x_1 \cdot b_1 + y_1 \cdot b_2 = b_{N+1} = 1$, т.е. найденные в результате x_1 и y_1 являются искомым частным решением.

Приведем одну из возможных реализаций метода для случая взаимно простых $b_{1,2}$ и $b_1 > b_2$.

```
int DiophEqII(void){
 int a[2][2];
 int b1,b2,b[2];
 int Imin,Imax;
 scanf("%d",&b1);
 scanf("%d",&b2);
  b[0]=b1;    b[1]=b2;
  a[0][0]=1;  a[0][1]=0;
  a[1][0]=0;  a[1][1]=1;
  Imax=0;  Imin=1;
while(1){
  if(b[Imin]==1) {x[0]=a[Imin][0]; x[1]=a[Imin][1];break;}
  p=b[Imax]/b[Imin];
  a[Imax][0] -= a[Imin][0]*p;
  a[Imax][1] -= a[Imin][1]*p;
  b[Imax] -= b[Imin]*p;
  Imax =(Imax+1)%2;
  Imin =(Imin+1)%2;
}

printf("(%d) * (%d) + (%d) * (%d) = 1\n",b1,x[0],b2,x[1]);
return 0;
}
```

1.198. Требуется вывести в файл все подмножества множества $\{0, \ldots, N-1\}$.

Идеи реализации. Если ограничиться значениями $N < 32$, то можно воспользоваться следующим приемом: для каждого целого числа $m < N$, набор единиц в двоичном представлении m соответствует некоторому подмножеству множества $\{0, \ldots, N-1\}$. Перебрав все числа $m = 0, \ldots, 2^N - 1$ (например, с помощью простейшего цикла), мы тем самым переберем все возможные подмножества.

1.199. Требуется вывести в файл все k-элементные подмножества множества $\{0, \ldots, N-1\}$.

Идеи реализации. Можно воспользоваться решением задачи 1.198 и отбирать только те значения m, которые содержат ровно k единиц в своем двоичном представлении. Другой способ см. 1.237.

1.200. Эффективно вычислите все простые числа, не превосходящие N.

Идеи реализации. Очевидный способ решения — последовательно проверять числа $1, 2, \ldots$ на простоту, см. задачу 1.27. Для больших N лучше воспользоваться алгоритмом типа "решето Эратосфена" — заводим битовый массив на N элементов и заполняем его 1, априори считая все числа простыми. Затем для $k = 2, 3, \ldots, k^2 \leq N$ проверяем значение k-ой ячейки, и если оно равно 1, то обнуляем все ячейки до N-ой с номерами, кратными k.

1.201. Эффективно вычислите все простые числа, лежащие между m и n.

Идеи реализации. Можно воспользоваться алгоритмом "двойное решето Эратосфена" — находим и сохраняем в отдельном массиве все простые числа, не превосходящие $\sqrt{n}$, а далее с их помощью вычеркиваем все "лишние" числа из требуемого промежутка. Отметим, что если в задаче 1.200 значение N велико, то последовательное применение данного подхода при правильно подобранных n и m позволяет ускорить работу алгортма за счет автоматического кэширования при работе с массивами.

1.202. Эффективно вычислить первые n простых чисел.

Идеи реализации. См. задачу 1.201.

1.203. Подсчитайте количество закодированных символов в текстовом файле, хранящемся в формате UTF-8.

Указание. Для записи кода одного символа в UTF-8 используется от одного до 4-х байт. Какое именно количество байт кодирует текущий символ определяется старшими битами. Если байт имеет вид (0*******), то он кодирует символ с кодом от 0 до 127 из первой части таблицы ASCII, иначе для текущего символа используется многобайтовое кодирование следующей структуры:

```
(0*******) --- при однобайтовом кодировании символа;
(110*****)(10******) --- если для кодирования символа выделено два
(1110****)(10******)(10******) --- три
(11110***)(10******)(10******)(10******) --- четыре байта.
```

При этом битовое представление кода символа из таблицы UNICODE последовательно вписывается справа налево на допустимые позиции, отмеченные символом *. Невостребованные левые биты заполняются нулями.

1.204. Напишите функцию конвертации текстового файла с кириллическим текстом, записанном в формате UTF-8, в текстовый файл с кодировкой koi8-r.

1.205. Для идентификации каждого сетевого интерфейса в пределах одного сегмента сети используется шестибайтовый MAC-адрес, обычно записываемый в виде шести пар шестнадцатиричных чисел. Первые n бит

MAC-адреса кодируют номер предприятия, а остальные — серийный номер устройства. Узнать MAC-адрес на Windows-ЭВМ можно командой *getmac*, набранной в командной строке. При этом на экран шестнадцатиричные пары будут выданы разделенными символом тире, например, A4-CE-D8-F4-AB-FB, (в Linux для этого используется символом двоеточие). Реализуйте функцию, получающую MAC-адрес в формате текстовой строки и значение n и возвращающую номер предприятия и серийный номер устройства в виде двух десятичных чисел.

1.206. Для идентификации компьютеров в разных подсетях используется четырехбайтовый IP-адрес, для удобства записываемый в виде четырех десятичных чисел разделенных точкой, например, 192.168.0.1. Таблицу соответствий между MAC-адресами и IP-адресами можно получить, набрав в командной строке команду *arp -a*. Первые n бит IP-адреса кодируют адрес сети, а остальные — адрес узла в сети. Реализуйте функцию, получающую IP-адрес в виде текстовой строки и величину n и и возвращающую два четырехбайтовых массива с адресом сети и адресом узла соответственно. Распечатайте найденные адрес сети и адрес узла в виде четырех десятичных чисел разделенных точкой. Модифицируйте функцию так, чтобы она допускала ввод IP-адреса и величины n форме одной строки, содержащей т.н. краткую слешнотацию: 192.168.161.1/24.

1.207. В терминологии сетей TCP/IP маской сети называют четырехбайтовый адрес, у которого первые n бит единицы, а остальные — нули. Побитно "перемножив" IP-адрес и маску — получим адрес сети, побитно "перемножив" IP-адрес и "инвертированную" маску — найдем адрес узла в сети.

Реализуйте функцию, получающую IP-адрес и маску сети в виде двух четырехбайтовых массивов, и возвращающую четырехбайтовые массивы с адресом сети и адресом узла. Отметим, что для адресации компьютеров в подсети не используются два адреса: адрес сети, т.е. адрес в котором все биты, отсекаемые маской, равны 0, и т.н. широковещательный адрес, в котором все биты, отсекаемые маской, равны 1. Включите в функцию проверку на корректность полученного адреса узла.

1.8 Задачи на обработку множества точек

В следующих задачах предполагается, что в файле записано несколько пар чисел, которые можно рассматривать как координаты множества точек на плоскости или как координаты множества концов отрезков на прямой. Требуется составить программы, которые читают исходные данные и отвечают на поставленные вопросы, или выдают значения параметров требуемых геометрических объектов. Следует стремиться к построению минимальных по сложности алгоритмов (сложность оценивается порядком числа операций в зависимости от числа точек или отрезков).

1.208. Множество точек определяет ломаную. Имеет ли она самопересечения?