1 Header Files

```
/*  bool.h */

#define TRUE    1
#define FALSE   0

typedef int bool;
```

```
/*  queue.h */

#define QUEUESIZE       1000

typedef struct {
  int q[QUEUESIZE+1];     /* body of queue */
  int first;         /* position of first element */
  int last;          /* position of last element */
  int count;         /* number of queue elements */
} queue;
```

```
/*  editdistance.h */

#define MAXLEN          101   /* longest string */

#define MATCH           0     /* symbol for match */
#define INSERT          1     /* symbol for insert */
#define DELETE          2     /* symbol for delete */

typedef struct {
  int cost;     /* cost of reaching this cell */
  int parent;   /* parent cell */
} cell;
```

```
/*  graph.h - Header file for graph data types */

#define MAXV  100   /* max number of vertices */
#define MAXDEGREE   50  /* max outdegree of vertex */

typedef struct {
  int v;      /* neighboring vertex */
  int weight;   /* edge weight */
} edge;

typedef struct {
  int edges[MAXV+1][MAXDEGREE];   /* adjacency info */
  int degree[MAXV+1];   /* outdegree of each vertex */
  int nvertices;        /* # vertices in the graph */
  int nedges;           /* # edges in the graph */
} graph;

typedef struct {
  edge edges[MAXV+1][MAXDEGREE];  /* adjacency info */
  int degree[MAXV+1];   /* outdegree of each vertex */
  int nvertices;        /* # vertices in the graph */
  int nedges;           /* # edges in the graph */
} wgraph;
```

```
/*  geometry.h - Header file for geometric data types */

#define PI  3.1415926
#define EPSILON 0.000001

typedef struct {
  double a;     /* x-coefficient */
  double b;     /* y-coefficient */
  double c;     /* constant term */
} line;

#define DIMENSION   2   /* dimension of points */
#define X     0   /* x-coordinate index */
#define Y     1   /* y-coordinate index */

typedef double point[DIMENSION];

#define MAXPOLY   200 /* max # points in a polygon */

typedef struct {
  int n;              /* # points in polygon */
  point p[MAXPOLY];   /* array of points in polygon */
} polygon;


typedef struct {
  point p1,p2;  /* endpoints of line segment */
} segment;

typedef point triangle[3];  /* triangle datatype */

typedef struct {
  int n;             /* # triangles in triangulation */
  int t[MAXPOLY][3];  /* nodes id's in triangulation */
} triangulation;

typedef struct {
  point c;      /* center of circle */
  double r;     /* radius of circle */
} circle;

/*  Comparison macros   */

#define max(A, B)     ((A) > (B) ? (A) : (B))
#define min(A, B)     ((A) < (B) ? (A) : (B))
```

2 Data Structures

```
/*  queue.c */

#include "queue.h"
#include "bool.h"


init_queue(queue *q)
{
  q->first = 0;
  q->last = QUEUESIZE-1;
  q->count = 0;
}

enqueue(queue *q, int x)
{
  if (q->count >= QUEUESIZE)
  printf("Warning: queue overflow enqueue x=%d\n",x);
  else {
    q->last = (q->last+1) % QUEUESIZE;
    q->q[ q->last ] = x;
    q->count = q->count + 1;
  }
}

int dequeue(queue *q)
{
  int x;

  if (q->count <= 0) printf("Warning: empty queue dequeue.\n");
  else {
    x = q->q[ q->first ];
    q->first = (q->first+1) % QUEUESIZE;
    q->count = q->count - 1;
  }

  return(x);
}

int empty(queue *q)
{
  if (q->count <= 0) return (TRUE);
  else return (FALSE);
}

print_queue(queue *q)
{
```

```
  int i,j;

  i=q->first;

   while (i != q->last) {
     printf("%c ",q->q[i]);
     i = (i+1) % QUEUESIZE;
   }
   printf("%2d ",q->q[i]);
   printf("\n");
}
```

```
#include <stl.h> /* C++ Standard Template Library */

/* Stack */
stack<int> S;
/* Methods: S.push(x), S.top(), S.pop(), S.empty */

/* Queue */
queue<char> Q;
/* Methods: Q.front(), Q.back(), Q.push(x), Q.pop,
   Q.empty() */

/* Dictionary */
/* Variety of containers such as hash_map */
/* Methods: H.erase(x), H.find(x), H.insert(x) */

/* Priority Queue */
priority_queue<int> Q;
/* Methods: Q.top(), Q.push(x), Q.pop(), Q.empty() */

/* Sets */
set<int, comparison> S;
/* Methods: S.member(x), set_union, set_intersection */
```

3 Strings

```
#include <ctype.h>  /* C character library */

int isalpha(int c);  /* true if upper or lower case */
int isupper(int c);  /* true if upper case */
int islower(int c);  /* true if lower case */
int isdigit(int c);  /* true if numerical digit */
int ispunct(int c);  /* true if punctuation symbol */
int isxdigit(int c); /* true if hexadecimal digit */
int isprint(int c);  /* true if printable character */

int toupper(int c);  /* convert to upper case */
int tolower(int c);  /* convert to lower case */
```

```
#include <string.h>  /* C++ string library */

char *strcat(char *dst, const char *src);  /* concat */
int strcmp(const char s1*, const char *s2); /* ==? */
char *strcpy(char *dst, const char *src);
size_t strlen(const char *s);  /* length of string */
char *strstr(const char *s1, const char *s2);
        /* search for s2 in s1 */
char *strtok(char *s1, const char *s2); /* it. search */

string::size()        /* string length */
string::empty()       /* empty? */
string::c_str()       /* return pointer to C-style s */
string::append(s)     /* append s to string */
string::erase(n,m)    /* delete m chars from n */
insert(n, const string &s);    /* ins s at n */
string::find(s)       /* search left */
string::rfind(s)      /* or right for given string */
string::first()       /* get first */
string::last()        /* or last character */
```

```
/*  editdistance.c - String comparison via dyn prog */

#include <string.h>
#include "editdistance.h"
#include "bool.h"

cell m[MAXLEN+1][MAXLEN+1]; /* dynamic prog table */

int string_compare(char *s, char *t)
{
  int i,j,k;    /* counters */
  int opt[3];   /* cost of the three options */

  for (i=0; i<MAXLEN; i++) {
  row_init(i);
  column_init(i);
  }

  for (i=1; i<strlen(s); i++)
  for (j=1; j<strlen(t); j++) {
    opt[MATCH] = m[i-1][j-1].cost + match(s[i],t[j]);
    opt[INSERT] = m[i][j-1].cost + indel(t[j]);
    opt[DELETE] = m[i-1][j].cost + indel(s[i]);

    m[i][j].cost = opt[MATCH];
    m[i][j].parent = MATCH;
    for (k=INSERT; k<=DELETE; k++)
    if (opt[k] < m[i][j].cost) {
      m[i][j].cost = opt[k];
      m[i][j].parent = k;
    }
  }

  goal_cell(s,t,&i,&j);
  return( m[i][j].cost );
}

reconstruct_path(char *s, char *t, int i, int j)
{
/*printf("trace (%d,%d)\n",i,j);*/

  if (m[i][j].parent == -1) return;

  if (m[i][j].parent == MATCH) {
  reconstruct_path(s,t,i-1,j-1);
  match_out(s, t, i, j);
  return;
  }
  if (m[i][j].parent == INSERT) {
    reconstruct_path(s,t,i,j-1);
  insert_out(t,j);
  return;
  }
  if (m[i][j].parent == DELETE) {
    reconstruct_path(s,t,i-1,j);
  delete_out(s,i);
  return;
  }
}

print_matrix(char *s, char *t, bool costQ)
{
  int i,j;    /* counters */
  int x,y;    /* string lengths */

  x = strlen(s);
  y = strlen(t);

  printf("   ");
  for (i=0; i<y; i++)
  printf("  %c",t[i]);
  printf("\n");


  for (i=0; i<x; i++) {
  printf("%c: ",s[i]);
  for (j=0; j<y; j++) {
    if (costQ == TRUE)
    printf(" %2d",m[i][j].cost);
    else
    printf(" %2d",m[i][j].parent);
```

```
  }
  printf("\n");
  }
}
```

```
/*  lcs.c - Longest common subsequence of two strings. */

#include <string.h>
#include "editdistance.h"
#include "bool.h"

extern cell m[MAXLEN+1][MAXLEN+1];   /* dyn prog table */


/*  For normal edit distance computation  */

goal_cell(char *s, char *t, int *i, int *j)
{
  *i = strlen(s) - 1;
  *j = strlen(t) - 1;
}

int match(char c, char d)
{
  if (c == d) return(0);
  else return(MAXLEN);
}

int indel(char c)
{
  return(1);
}

row_init(int i)   /* what is m[0][i]? */
{
  m[0][i].cost = i;
  if (i>0)
  m[0][i].parent =  INSERT;
  else
  m[0][i].parent = -1;
}

column_init(int i)  /* what is m[i][0]? */
{
  m[i][0].cost = i;
  if (i>0)
  m[i][0].parent = DELETE;
  else
  m[0][i].parent = -1;
}

/**********************************************/

match_out(char *s, char *t, int i, int j)
{
  if (s[i] == t[j]) printf("%c",s[i]);
}

insert_out(char *t, int j)
{
}

delete_out(char *s, int i)
{
}

/**********************************************/

main(){
  int i,j;
  int lcslen, complen;
  char s[MAXLEN],t[MAXLEN];     /* input strings */

  s[0] = t[0] = ' ';

  scanf("%s",&(s[1]));
  scanf("%s",&(t[1]));

  complen = string_compare(s,t);
  lcslen = (strlen(s) + strlen(t) - 2 - complen) / 2;

  printf("length of longest common subsequence = %d\n",
    lcslen);
  /*
    print_matrix(s,t,TRUE);
    printf("\n");
    print_matrix(s,t,FALSE);
  */
  goal_cell(s,t,&i,&j);
  /*
    printf("%d %d\n",i,j);
  */
  reconstruct_path(s,t,i,j);
  printf("\n");
}
```

4 Sorting

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int sort_function(const void *a, const void *b);

char list[5][4] = {"cat", "car", "cab", "cap", "can"};

int main(void)
{
  int x;

  qsort((void *)list, 5, sizeof(list[0]),
        sort_function);
  for (x=0; x<5; x++)
    printf("%s\n", list[x]);
    return 0;
}

int sort_function(const void *a, const void *b)
{
  return(strcmp((char *)a,(char *)b));
}
```

```
/*  sorting.c - Implementations of sorting algorithms */

#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define NELEM   100     /* size of test arrays */

/*  Swap the ith and jth elements of array s.   */

newswap(int s[], int i, int j)
{
    int tmp;    /* placeholder */

  tmp = s[i];
  s[i] = s[j];
  s[j] = tmp;
}

insertion_sort(int s[], int n)
{
  int i,j;    /* counters */

  for (i=1; i<n; i++) {
    j=i;
    while ((j>0) && (s[j] < s[j-1])) {
      swap(&s[j],&s[j-1]);
      j = j-1;
    }
  }
}

selection_sort(int s[], int n)
```

```
{
  int i,j;        /* counters */
  int min;    /* index of minimum */

    for (i=0; i<n; i++) {
        min=i;
    for (j=i+1; j<n; j++)
      if (s[j] < s[min]) min=j;
    swap(&s[i],&s[min]);
    }
}

/*  quicksort array s from the index l to index h.  */

quicksort(int s[], int l, int h)
{
  int p;      /* index of partition */

  if ((h-l)>0) {
    p = partition(s,l,h);
    quicksort(s,l,p-1);
    quicksort(s,p+1,h);
  }
}

int partition(int s[], int l, int h)
{
  int i;      /* counter */
  int p;      /* pivot element index */
  int firsthigh;    /* divider pos for pivot el */

  p = h;
  firsthigh = l;
  for (i=l; i<h; i++)
    if (s[i] < s[p]) {
      swap(&s[i],&s[firsthigh]);
      firsthigh ++;
    }
  swap(&s[p],&s[firsthigh]);
  return(firsthigh);
}



main()
{
  int s[NELEM+2];
  int n;
  int i,j;      /* counters */

  for (i=0; i<NELEM; i++) s[i] = i;
  random_permutation(s,NELEM);

  insertion_sort(s,NELEM);
  selection_sort(s,NELEM);
  quicksort(s,0,NELEM-1);

  for (i=0; i<NELEM; i++) printf("%d ",s[i]);
    printf("\n");
}
```

5 Arithmetic and Algebra

Rounding numbers:

$$round(X,k) = floor(10^k X + (1/2))/10^k$$

Solutions to $x^2 + p \cdot x + q$:

$$x_{1/2} = -\frac{p}{2} \pm \sqrt{\frac{p^2}{4} - q}$$

```
#include <math.h>  /* C math library */

double floor(double x);  /* chop off fractional part */
double ceil(double x);   /* raise to next int */
double fabs(double x);   /* absolute value of x */
double sqrt(double x);   /* square root */
double exp(double x);    /* compute e^x */
double log(double x);    /* compute ln */
double log10(double x);  /* compute base-10 log */
double pow(x,y);         /* compute x^y */
```

```
/*  bignum.c - Arithmetic for big numbers */

#include <stdio.h>

#define MAXDIGITS   100   /* maximum length bignum */

#define PLUS  1     /* positive sign bit */
#define MINUS     -1    /* negative sign bit */

typedef struct {
  char digits[MAXDIGITS];   /* represent the number */
  int signbit;    /* 1 if positive, -1 if negative */
  int lastdigit;    /* index of high-order digit */
} bignum;


print_bignum(bignum *n)
{
  int i;

  if (n->signbit == MINUS) printf("- ");
  for (i=n->lastdigit; i>=0; i--)
  printf("%c",'0'+ n->digits[i]);

  printf("\n");
}

int_to_bignum(int s, bignum *n)
{
  int i;      /* counter */
  int t;      /* int to work with */

  if (s >= 0) n->signbit = PLUS;
  else n->signbit = MINUS;

  for (i=0; i<MAXDIGITS; i++) n->digits[i] = (char) 0;

  n->lastdigit = -1;

  t = abs(s);

  while (t > 0) {
  n->lastdigit ++;
  n->digits[ n->lastdigit ] = (t % 10);
  t = t / 10;
  }

  if (s == 0) n->lastdigit = 0;
}

initialize_bignum(bignum *n)
{
  int_to_bignum(0,n);
}


int max(int a, int b)
{
  if (a > b) return(a); else return(b);
}

/*  c = a +-/* b;   */

add_bignum(bignum *a, bignum *b, bignum *c)
{
  int carry;    /* carry digit */
  int i;      /* counter */

  initialize_bignum(c);

  if (a->signbit == b->signbit)
    c->signbit = a->signbit;
  else {
```

```
  if (a->signbit == MINUS) {
    a->signbit = PLUS;
    subtract_bignum(b,a,c);
    a->signbit = MINUS;
  } else {
      b->signbit = PLUS;
      subtract_bignum(a,b,c);
      b->signbit = MINUS;
  }
  return;
  }

  c->lastdigit = max(a->lastdigit,b->lastdigit)+1;
  carry = 0;

  for (i=0; i<=(c->lastdigit); i++) {
  c->digits[i] = (char)
      (carry+a->digits[i]+b->digits[i]) % 10;
  carry = (carry + a->digits[i] + b->digits[i]) / 10;
  }

  zero_justify(c);
}


subtract_bignum(bignum *a, bignum *b, bignum *c)
{
  int borrow;   /* has anything been borrowed? */
  int v;      /* placeholder digit */
  int i;      /* counter */

  if ((a->signbit == MINUS) || (b->signbit == MINUS)) {
    b->signbit = -1 * b->signbit;
    add_bignum(a,b,c);
    b->signbit = -1 * b->signbit;
  return;
  }

  if (compare_bignum(a,b) == PLUS) {
  subtract_bignum(b,a,c);
  c->signbit = MINUS;
  return;
  }

  c->lastdigit = max(a->lastdigit,b->lastdigit);
  borrow = 0;

  for (i=0; i<=(c->lastdigit); i++) {
  v = (a->digits[i] - borrow - b->digits[i]);
  if (a->digits[i] > 0)
    borrow = 0;
  if (v < 0) {
    v = v + 10;
    borrow = 1;
  }

    c->digits[i] = (char) v % 10;
  }

  zero_justify(c);
}

compare_bignum(bignum *a, bignum *b)
{
  int i;      /* counter */

  if ((a->signbit == MINUS) && (b->signbit == PLUS))
      return(PLUS);
  if ((a->signbit == PLUS) && (b->signbit == MINUS))
      return(MINUS);
  if (b->lastdigit > a->lastdigit)
      return (PLUS * a->signbit);
  if (a->lastdigit > b->lastdigit)
      return (MINUS * a->signbit);
  for (i = a->lastdigit; i>=0; i--) {
  if (a->digits[i] > b->digits[i])
      return(MINUS * a->signbit);
  if (b->digits[i] > a->digits[i])
      return(PLUS * a->signbit);
  }
  return(0);
}

zero_justify(bignum *n)
{
  while ((n->lastdigit > 0) &&
         (n->digits[ n->lastdigit ] == 0))
  n->lastdigit --;

  if ((n->lastdigit == 0) && (n->digits[0] == 0))
  n->signbit = PLUS;  /* hack to avoid -0 */
}


digit_shift(bignum *n, int d)   /* multiply n by 10^d */
{
  int i;      /* counter */

  if ((n->lastdigit == 0) && (n->digits[0] == 0))
      return;
  for (i=n->lastdigit; i>=0; i--)
  n->digits[i+d] = n->digits[i];

  for (i=0; i<d; i++) n->digits[i] = 0;

  n->lastdigit = n->lastdigit + d;
}


multiply_bignum(bignum *a, bignum *b, bignum *c)
{
  bignum row;   /* represent shifted row */
  bignum tmp;   /* placeholder bignum */
  int i,j;    /* counters */

  initialize_bignum(c);

  row = *a;

  for (i=0; i<=b->lastdigit; i++) {
  for (j=1; j<=b->digits[i]; j++) {
    add_bignum(c,&row,&tmp);
    *c = tmp;
  }
  digit_shift(&row,1);
  }

  c->signbit = a->signbit * b->signbit;

  zero_justify(c);
}


divide_bignum(bignum *a, bignum *b, bignum *c)
{
  bignum row;       /* represent shifted row */
  bignum tmp;       /* placeholder bignum */
  int asign, bsign;     /* temporary signs */
  int i,j;      /* counters */

  initialize_bignum(c);

  c->signbit = a->signbit * b->signbit;

  asign = a->signbit;
  bsign = b->signbit;

  a->signbit = PLUS;
  b->signbit = PLUS;

  initialize_bignum(&row);
  initialize_bignum(&tmp);

  c->lastdigit = a->lastdigit;

  for (i=a->lastdigit; i>=0; i--) {
  digit_shift(&row,1);
  row.digits[0] = a->digits[i];
  c->digits[i] = 0;
  while (compare_bignum(&row,b) != PLUS) {
    c->digits[i] ++;
```

```
    subtract_bignum(&row,b,&tmp);
    row = tmp;
  }
  }

  zero_justify(c);

  a->signbit = asign;
  b->signbit = bsign;
}


main()
{
  int a,b;
  bignum n1,n2,n3,zero;

  while (scanf("%d %d\n",&a,&b) != EOF) {
  printf("a = %d  b = %d\n",a,b);
  int_to_bignum(a,&n1);
  int_to_bignum(b,&n2);

  add_bignum(&n1,&n2,&n3);
  printf("addition -- ");
  print_bignum(&n3);

  printf("compare_bignum a ? b = %d\n",
         compare_bignum(&n1, &n2));

  subtract_bignum(&n1,&n2,&n3);
  printf("subtraction -- ");
  print_bignum(&n3);

    multiply_bignum(&n1,&n2,&n3);
  printf("multiplication -- ");
    print_bignum(&n3);

  int_to_bignum(0,&zero);
  if (compare_bignum(&zero, &n2) == 0)
    printf("division -- NaN \n");
    else {
    divide_bignum(&n1,&n2,&n3);
    printf("division -- ");
      print_bignum(&n3);
  }
  printf("-------------------------\n");
  }
}
```

6 Combinatorics

Binomial coefficients:

$$\binom{n}{k} = \frac{n!}{(n-k)!k!} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Fibonacci numbers: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$. Closed form:

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

Catalan numbers: $C_0 = 1$ and then 2,5,14,42,132,429,1430,... Good for enumerating possibilities of things you can easily break up into two parts. E.g. how many ways are there to make a balanced set of parentheses? For 3 sets of parentheses, there are 5 ways: ((())), ()(()), (())(), (()()), and ()()(). The problem can be broken up by dividing the sequence by the first set of matched parentheses, i.e. the first left parenthesis and its partner. Clearly, there are some k sets of matched parentheses between these two, and some $n-k-1$ sets of parentheses after the right parenthesis.

$$C_n = \sum_{k=0}^{n-1} C_k C_{n-k-1} = \frac{1}{n+1}\binom{2n}{n}$$

Eulerian numbers: $\left\langle {n \atop k} \right\rangle$ counts the number of ways to form a permutation of n things with exactly k ascending sequences.

$$\left\langle {n \atop k} \right\rangle = k\left\langle {n-1 \atop k} \right\rangle + (n-k+1)\left\langle {n-1 \atop k-1} \right\rangle$$

Stirling numbers: $\left[{n \atop k} \right]$ counts the number of permutations on n elements with exactly k cycles.

$$\left[{n \atop k} \right] = \left[{n-1 \atop k-1} \right] + (n-1)\left[{n-1 \atop k} \right]$$

Set partitions: $\left\{ {n \atop k} \right\}$ counts the number of ways to partition n elemeents into exactly k sets.

$$\left\{ {n \atop k} \right\} = k\left\{ {n-1 \atop k} \right\} + \left\{ {n-1 \atop k-1} \right\}$$

A special case is $\left\{ {n \atop 2} \right\} = 2^{n-1} - 1$.

Integer partitions: An integer partition of n is an unordered set of positive integers which add up to n. Let $f(n,k)$ be the number of integer partitions of n with largest part of at most k. Then is holds that

$$f(n,k) = f(n-k,k) + f(n,k-1)$$

with the base cases $f(1,1) = 1$ and $f(n,k) = 0$ whenever $k > n$.

```
/*  binomial.c - Computation of binomial coefficient */

#define MAXN  100   /* largest n or m */

long binomial_coefficient(n,m)
int n,m;    /* computer n choose m */
{
  int i,j;  /* counters */
  long bc[MAXN][MAXN];  /* table of bin coeff values */

  for (i=0; i<=n; i++) bc[i][0] = 1;

  for (j=0; j<=n; j++) bc[j][j] = 1;


  for (i=1; i<=n; i++)
  for (j=1; j<i; j++)
    bc[i][j] = bc[i-1][j-1] + bc[i-1][j];

  return( bc[n][m] );
}

main()
{
  int a, b;
  long binomial_coefficient();

  while (1) {
  scanf("%d %d",&a,&b);
  printf("%d\n",binomial_coefficient(a,b));
  }
}
```

7 Number Theory

Greatest common divisor and least common multiple:

$$x \cdot y = lcm(x,y)\cot gcd(x,y)$$

Modular arithmetic:

$$\begin{aligned}(x+y) \bmod m &= ((x \bmod m) + (y \bmod m)) \bmod m \\ (x \cdot y) \bmod m &= ((x \bmod m) \cdot (y \bmod m)) \bmod m \\ x^y \bmod m &= (x \bmod m)^y \bmod m\end{aligned}$$

```
/*  Compute greatest common divisor */

#include<stdio.h>
#include<math.h>


long gcd1(long p, long q)
{
  if (q > p) return(gcd1(q,p));
  if (q == 0) return(p);
  printf(" gcd(%d,%d) &=& gcd(%d \\mod %d, %d) =
```

```
    gcd(%d,%d) \n",p,q,p,q,q,q,p%q);
  return( gcd1(q, p % q) );
}

/* Find gcd(p,q) and x,y s.t. p*x + q*y = gcd(p,q) */

long gcd(long p, long q, long *x, long *y)
{
  long x1,y1;   /* previous coefficients */
  long g;     /* value of gcd(p,q) */

  if (q > p) return(gcd(q,p,y,x));

  if (q == 0) {
  *x = 1;
  *y = 0;
  return(p);
  }

  g = gcd(q, p%q, &x1, &y1);

  *x = y1;
  *y = (x1 - floor(p/q)*y1);

  return(g);
}


main() {
    long p,q;
    long gcd(), gcd2();
    long x,y,g1,g2;

    while (scanf("%d %d",&p,&q)!=EOF) {

  printf("gcd of p=%d and q=%d = %d\n",
      p,q,g1=gcd1(p,q));
  printf(" %d*%d + %d*%d = %d\n",
      p,x,q,y,g2=gcd(p,q,&x,&y));
  if (g1 != g2) printf("ERROR: GCD\n");
  if ((p*x + q*y) != g1)
      printf("ERROR: DIOPHONINE SOLUTION WRONG!\n");
    }
}
```

```
/* primes.c - Compute prime factorization of int. */

#include<stdio.h>
#include<math.h>


prime_factorization(long x)
{
  long i;   /* counter */
  long c;   /* remaining product to factor */

  c = x;
  while ((c % 2) == 0) {
    printf("%ld\n",2);
    c = c / 2;
  }

  i = 3;
  while (i <= (sqrt(c)+1)) {
    if ((c % i) == 0) {
      printf("%ld\n",i);
      c = c / i;
    }
    else
      i = i + 2;
    }

  if (c > 1) printf("%ld\n",c);
}


main() {
  long p;

  while (scanf("%ld",&p)!=EOF) {
    printf("prime factorization of p=%ld \n",p);
      prime_factorization(p);

    }
}
```

8 Backtracking

```
/* backtrack.c */

#include "bool.h"

#define MAXCANDIDATES   100   /* max next extensions */
#define NMAX    100     /* maximum solution size */

typedef char* data; /* type to pass data to backtrack */

bool finished = FALSE;  /* found all solutions yet? */

backtrack(int a[], int k, data input)
{
  int c[MAXCANDIDATES]; /* candidates for next pos */
  int ncandidates;  /* next position candidate count */
  int i;  /* counter */

  if (is_a_solution(a,k,input))
  process_solution(a,k,input);
  else {
  k = k+1;
  construct_candidates(a,k,input,c,&ncandidates);
  for (i=0; i<ncandidates; i++) {
    a[k] = c[i];
    backtrack(a,k,input);
    if (finished) return;   /* terminate early */
  }
  }
}
```

```
/*  permutations.c - Construct all permutations. */

#include "bool.h"
#include "backtrack.h"

process_solution(int a[], int k)
{
  int i;      /* counter */

  for (i=1; i<=k; i++) printf(" %d",a[i]);
  printf("\n");
}

is_a_solution(int a[], int k, int n)
{
  return (k == n);
}

/* What are possible elements of next slot in perm? */

construct_candidates(int a[], int k, int n, int c[],
                     int *ncandidates)
{
  int i;      /* counter */
  bool in_perm[NMAX];   /* what is now in perm? */

  for (i=1; i<NMAX; i++) in_perm[i] = FALSE;
  for (i=0; i<k; i++) in_perm[ a[i] ] = TRUE;

  *ncandidates = 0;
  for (i=1; i<=n; i++)
  if (in_perm[i] == FALSE) {
    c[ *ncandidates] = i;
    *ncandidates = *ncandidates + 1;
  }
}
```

```
main()
{
  int a[NMAX];    /* solution vector */

  backtrack(a,0,3);
}
```

```
/*  subsets.c - Construct all subsets. */

#include "bool.h"
#include "backtrack.h"


process_solution(int a[], int k)
{
  int i;      /* counter */

  printf("{");
  for (i=1; i<=k; i++)
    if (a[i] == TRUE) printf(" %d",i);
  printf(" }\n");
}

is_a_solution(int a[], int k, int n)
{
  return (k == n);
}


/* What are possible elements of next slot in perm? */

construct_candidates(int a[], int k, int n, int c[],
                     int *ncandidates)
{
  c[0] = TRUE;
  c[1] = FALSE;
  *ncandidates = 2;
}


main()
{
  int a[NMAX];    /* solution vector */

  backtrack(a,0,3);
}
```

9 Graph Traversal

```
/*  bfs-dfs.c */

#include "bool.h"
#include "graph.h"
#include "queue.h"

bool processed[MAXV];  /* which vertices have been proc */
bool discovered[MAXV]; /* which vertices have been found */
int parent[MAXV];   /* discovery relation */

bool finished = FALSE; /* if true, cut off search */

initialize_search(graph *g)
{
  int i;        /* counter */

  for (i=1; i<=g->nvertices; i++) {
    processed[i] = discovered[i] = FALSE;
    parent[i] = -1;
  }
}

bfs(graph *g, int start)
{
  queue q;    /* queue of vertices to visit */
  int v;      /* current vertex */
  int i;      /* counter */

  init_queue(&q);
  enqueue(&q,start);
  discovered[start] = TRUE;

  while (empty(&q) == FALSE) {
  v = dequeue(&q);
  process_vertex(v);
  processed[v] = TRUE;
  for (i=0; i<g->degree[v]; i++)
    if (valid_edge(g->edges[v][i]) == TRUE) {
    if (discovered[g->edges[v][i]] == FALSE) {
    enqueue(&q,g->edges[v][i]);
    discovered[g->edges[v][i]] = TRUE;
    parent[g->edges[v][i]] = v;
    }
    if (processed[g->edges[v][i]] == FALSE)
    process_edge(v,g->edges[v][i]);
    }
  }
}


/*
bool valid_edge(edge e)
{
  if (e.residual > 0) return (TRUE);
  else return(FALSE);
}
*/


dfs(graph *g, int v)
{
  int i;      /* counter */
  int y;      /* successor vertex */

  if (finished) return;  /* allow for search term */

  discovered[v] = TRUE;
  process_vertex(v);

  for (i=0; i<g->degree[v]; i++) {
  y = g->edges[v][i];
  if (valid_edge(g->edges[v][i]) == TRUE) {
    if (discovered[y] == FALSE) {
    parent[y] = v;
    dfs(g,y);
    } else
    if (processed[y] == FALSE)
      process_edge(v,y);
  }
  if (finished) return;
  }

  processed[v] = TRUE;
}


find_path(int start, int end, int parents[])
{
if ((start == end) || (end == -1))
  printf("\n%d",start);
  else {
  find_path(start,parents[end],parents);
  printf(" %d",end);
  }
}
```

10 Graph Algorithms

```
/*  graph.c - A generic adjacency list-in-array
              graph data type. */

#include "bool.h"
#include "queue.h"
#include "graph.h"

initialize_graph(graph *g)
{
```

```
  int i;      /* counter */

  g -> nvertices = 0;
  g -> nedges = 0;
  for (i=1; i<=MAXV; i++) g->degree[i] = 0;
}

read_graph(graph *g, bool directed)
{
  int i;      /* counter */
  int m;      /* number of edges */
  int x, y;     /* vertices in edge (x,y) */

  initialize_graph(g);

  scanf("%d %d",&(g->nvertices),&m);
  for (i=1; i<=m; i++) {
    scanf("%d %d",&x,&y);
    insert_edge(g,x,y,directed);
  }
}

insert_edge(graph *g, int x, int y, bool directed)
{
  if (g->degree[x] > MAXDEGREE)
  printf("Warning: insertion(%d,%d) exceeds max
          degree\n",x,y);

  g->edges[x][g->degree[x]] = y;
  g->degree[x] ++;

  if (directed == FALSE)
    insert_edge(g,y,x,TRUE);
  else
    g->nedges ++;
}


delete_edge(graph *g, int x, int y, bool directed)
{
  int i;      /* counter */

  for (i=0; i<g->degree[x]; i++)
  if (g->edges[x][i] == y) {
    g->degree[x] --;
    g->edges[x][i] = g->edges[x][g->degree[x]];

    if (directed == FALSE)
      delete_edge(g,y,x,TRUE);

    return;
  }
  printf("Warning: deletion(%d,%d) not found in g.\n",
          x,y);
}

print_graph(graph *g)
{
  int i,j;    /* counters */

  for (i=1; i<=g->nvertices; i++) {
  printf("%d: ",i);
  for (j=0; j<g->degree[i]; j++)
    printf(" %d",g->edges[i][j]);
  printf("\n");
  }
}
```

```
/*  wgraph.c - A generic weighted graph data type */


#include "bool.h"
#include "wgraph.h"

initialize_graph(g)
graph *g;       /* graph to initialize */
{

  int i;      /* counter */

  g -> nvertices = 0;
  g -> nedges = 0;

  for (i=1; i<=MAXV; i++) g->degree[i] = 0;
}

read_graph(g,directed)
graph *g;           /* graph to initialize */
bool directed;      /* is this graph directed? */
{
  int i;      /* counter */
  int m;      /* number of edges */
  int x,y,w;    /* placeholder for edge and weight */

  initialize_graph(g);

  scanf("%d %d\n",&(g->nvertices),&m);

  for (i=1; i<=m; i++) {
  scanf("%d %d %d\n",&x,&y,&w);
  insert_edge(g,x,y,directed,w);
  }
}

insert_edge(g,x,y,directed,w)
graph *g;           /* graph to initialize */
int x, y;         /* placeholder for edge (x,y) */
bool directed;        /* is this edge directed? */
int w;      /* edge weight */
{
  if (g->degree[x] > MAXDEGREE)
  printf("Warning: insertion(%d,%d) exceeds degree
          bound\n",x,y);

  g->edges[x][g->degree[x]].v = y;
  g->edges[x][g->degree[x]].weight = w;
  /*g->edges[x][g->degree[x]].in = FALSE;*/
  g->degree[x] ++;

  if (directed == FALSE)
  insert_edge(g,y,x,TRUE,w);
  else
  g->nedges ++;
}


delete_edge(g,x,y,directed)
graph *g;           /* graph to initialize */
int x, y;           /* placeholder for edge (x,y) */
bool directed;        /* is this edge directed? */
{
  int i;      /* counter */

  for (i=0; i<g->degree[x]; i++)
  if (g->edges[x][i].v == y) {
    g->degree[x] --;
    g->edges[x][i] = g->edges[x][g->degree[x]];

    if (directed == FALSE)
    delete_edge(g,y,x,TRUE);

    return;
  }

  printf("Warning: deletion(%d,%d) not found in g.\n",
          x,y);
}

print_graph(g)
graph *g;           /* graph to print */
{
  int i,j;    /* counters */

  for (i=1; i<=g->nvertices; i++) {
  printf("%d: ",i);
  for (j=0; j<g->degree[i]; j++)
    printf(" %d",g->edges[i][j].v);
  printf("\n");
  }
}
```

```
bool processed[MAXV];  /* which vertices have been proc */ }
bool discovered[MAXV]; /* which vertices have been found */
int parent[MAXV];     /* discovery relation */


initialize_search(g)
graph *g;       /* graph to traverse */
{
  int i;        /* counter */

  for (i=1; i<=g->nvertices; i++) {
    processed[i] = FALSE;
    discovered[i] = FALSE;
    parent[i] = -1;
  }
}


dfs(g,v)
graph *g;   /* graph to traverse */
int v;      /* vertex to start searching from */
{
  int i;      /* counter */
  int y;      /* successor vertex */

  discovered[v] = TRUE;
  process_vertex(v);

  for (i=0; i<g->degree[v]; i++) {
  y = g->edges[v][i].v;
  if (discovered[y] == FALSE) {
    parent[y] = v;
    dfs(g,y);
  } else
    if (processed[y] == FALSE)
      process_edge(v,y);
  }

  processed[v] = TRUE;
}

process_vertex(v)
int v;  /* vertex to process */
{
  printf(" %d",v);
}

process_edge(x, y)
int x,y;        /* edge to process */
{
}


find_path(start,end,parents)
int start;      /* first vertex on path */
int end;        /* last vertex on path */
int parents[];  /* array of parent pointers */
{
  if ((start == end) || (end == -1))
    printf("\n%d",start);
  else {
    find_path(start,parents[end],parents);
  printf(" %d",end);
  }
}

connected_components(g)
graph *g;       /* graph to analyze */
{
  int c;      /* component number */
  int i;      /* counter */

  initialize_search(g);

  c = 0;
  for (i=1; i<=g->nvertices; i++)
  if (discovered[i] == FALSE) {
    c = c+1;
    printf("Component %d:",c);
    dfs(g,i);
    printf("\n");
  }
}
```

```
/*  connected.c */

#include "bool.h"
#include "graph.h"

extern bool processed[];  /* which vertices were proc */
extern bool discovered[]; /* which vertices were found */
extern int parent[];  /* discovery relation */


process_vertex(int v)
{
  printf(" %d",v);
}

process_edge(int x, int y)
{
}

bool valid_edge(int e)
{
  return (TRUE);
}


connected_components(graph *g)
{
  int c;      /* component number */
  int i;      /* counter */

  initialize_search(g);

  c = 0;
  for (i=1; i<=g->nvertices; i++)
  if (discovered[i] == FALSE) {
    c = c+1;
    printf("Component %d:",c);
    dfs(g,i);
    printf("\n");
  }
}

main()
{
  graph g;

  read_graph(&g,FALSE);
  print_graph(&g);
  connected_components(&g);
}
```

```
/*  topsort.c - Topologically sort a directed acyclic
                graph (DAG) */

#include "bool.h"
#include "graph.h"
#include "queue.h"


compute_indegrees(graph *g, int in[])
{
  int i,j;    /* counters */

  for (i=1; i<=g->nvertices; i++) in[i] = 0;

  for (i=1; i<=g->nvertices; i++)
  for (j=0; j<g->degree[i]; j++) in[ g->edges[i][j] ] ++;
}


topsort(graph *g, int sorted[])
{
```

```
  int indegree[MAXV];   /* indegree of each vertex */
  queue zeroin;     /* vertices of indegree 0 */
  int x, y;     /* current and next vertex */
  int i, j;     /* counters */

  compute_indegrees(g,indegree);
  init_queue(&zeroin);
  for (i=1; i<=g->nvertices; i++)
  if (indegree[i] == 0) enqueue(&zeroin,i);

  j=0;
  while (empty(&zeroin) == FALSE) {
  j = j+1;
  x = dequeue(&zeroin);
  sorted[j] = x;
  for (i=0; i<g->degree[x]; i++) {
    y = g->edges[x][i];
    indegree[y] --;
    if (indegree[y] == 0) enqueue(&zeroin,y);
  }
  }

  if (j != g->nvertices)
  printf("Not a DAG -- only %d vertices found\n",j);
}


main()
{
  graph g;
  int out[MAXV];
  int i;

  read_graph(&g,TRUE);
  print_graph(&g);

  topsort(&g,out);

  for (i=1; i<=g.nvertices; i++)
  printf(" %d",out[i]);
  printf("\n");

}
```

```
/*  dijkstra.c - Shortest path between two vertices. */

#include <stdlib.h>
#include "bool.h"
#include "wgraph.h"

#define MAXINT  100007

int parent[MAXV];       /* discovery relation */


dijkstra(graph *g, int start)
{
  int i,j;    /* counters */
  bool intree[MAXV];    /* is vertex in tree yet? */
  int distance[MAXV];   /* dist vertex is from start */
  int v;      /* current vertex to process */
  int w;      /* candidate next vertex */
  int weight;   /* edge weight */
  int dist;     /* best current distance from start */

  for (i=1; i<=g->nvertices; i++) {
  intree[i] = FALSE;
  distance[i] = MAXINT;
  parent[i] = -1;
  }

  distance[start] = 0;
  v = start;

  while (intree[v] == FALSE) {
  intree[v] = TRUE;
  for (i=0; i<g->degree[v]; i++) {
    w = g->edges[v][i].v;
    weight = g->edges[v][i].weight;
    if (distance[w] > (distance[v]+weight)) {
      distance[w] = distance[v]+weight;
      parent[w] = v;
    }
  }

  v = 1;
  dist = MAXINT;
  for (i=1; i<=g->nvertices; i++)
    if ((intree[i] == FALSE) && (dist > distance[i])) {
      dist = distance[i];
      v = i;
    }
  }
  /* for (i=1; i<=g->nvertices; i++)
       printf("%d %d\n",i,distance[i]); */
}

main()
{
  graph g;
  int i;

  read_graph(&g,FALSE);
  dijkstra(&g,1);

  for (i=1; i<=g.nvertices; i++)
    find_path(1,i,parent);
  printf("\n");

}
```

```
/*  prim.c - Compute minimum spanning tree. */

#include <stdlib.h>
#include "bool.h"
#include "wgraph.h"

#define MAXINT  100007

int parent[MAXV];       /* discovery relation */


prim(graph *g, int start)
{
  int i,j;    /* counters */
  bool intree[MAXV];   /* is vertex in tree yet? */
  int distance[MAXV];  /* dist vertex is from start */
  int v;      /* current vertex to process */
  int w;      /* candidate next vertex */
  int weight;   /* edge weight */
  int dist;     /* best current distance from start */

  for (i=1; i<=g->nvertices; i++) {
  intree[i] = FALSE;
  distance[i] = MAXINT;
  parent[i] = -1;
  }

  distance[start] = 0;
  v = start;

  while (intree[v] == FALSE) {
  intree[v] = TRUE;
  for (i=0; i<g->degree[v]; i++) {
    w = g->edges[v][i].v;
    weight = g->edges[v][i].weight;
    if ((distance[w] > weight) && (intree[w] == FALSE)) {
      distance[w] = weight;
      parent[w] = v;
    }
  }

  v = 1;
  dist = MAXINT;
  for (i=1; i<=g->nvertices; i++)
    if ((intree[i] == FALSE) && (dist > distance[i])) {
      dist = distance[i];
      v = i;
```

```
    }
  }
}

main()
{
  graph g;
  int i;

  read_graph(&g,FALSE);

  prim(&g,1);

printf("Out of Prim\n");

  for (i=1; i<=g.nvertices; i++) {
  /*printf(" %d parent=%d\n",i,parent[i]);*/
    find_path(1,i,parent);
  }
  printf("\n");

}
```

```
/*  floyd.c */

#include <stdlib.h>
#include "bool.h"
#include "wgraph.h"


typedef struct {
  int weight[MAXV+1][MAXV+1];   /* adjacency info */
  int nvertices;       /* # vertices in the graph */
} adjacency_matrix;


initialize_adjacency_matrix(adjacency_matrix *g)
{
  int i,j;    /* counters */

  g -> nvertices = 0;

  for (i=1; i<=MAXV; i++)
  for (j=1; j<=MAXV; j++)
    g->weight[i][j] = MAXINT;
}

read_adjacency_matrix(adjacency_matrix *g, bool directed)
{
  int i;      /* counter */
  int m;      /* number of edges */
  int x,y,w;    /* placeholder for edge and weight */

  initialize_adjacency_matrix(g);

  scanf("%d %d\n",&(g->nvertices),&m);

  for (i=1; i<=m; i++) {
  scanf("%d %d %d\n",&x,&y,&w);
  g->weight[x][y] = w;
  if (directed==FALSE) g->weight[y][x] = w;
  }
}


print_graph(adjacency_matrix *g)
{
  int i,j;    /* counters */

  for (i=1; i<=g->nvertices; i++) {
  printf("%d: ",i);
  for (j=1; j<=g->nvertices; j++)
    if (g->weight[i][j] < MAXINT)
    printf(" %d",j);
  printf("\n");
  }
}

print_adjacency_matrix(adjacency_matrix *g)
{
  int i,j;      /* counters */

  for (i=1; i<=g->nvertices; i++) {
    printf("%3d: ",i);
    for (j=1; j<=g->nvertices; j++)
        printf(" %3d",g->weight[i][j]);
    printf("\n");
  }
}


floyd(adjacency_matrix *g)
{
  int i,j;    /* dimension counters */
  int k;      /* intermediate vertex counter */
  int through_k;    /* distance through vertex k */

  for (k=1; k<=g->nvertices; k++)
  for (i=1; i<=g->nvertices; i++)
    for (j=1; j<=g->nvertices; j++) {
    through_k = g->weight[i][k]+g->weight[k][j];
    if (through_k < g->weight[i][j])
      g->weight[i][j] = through_k;
    }
}


main()
{
  adjacency_matrix g;

  read_adjacency_matrix(&g,FALSE);
  print_graph(&g);

  floyd(&g);

  print_adjacency_matrix(&g);
}
```

```
/*  netflow.c - Ford-Fulkerson algorithm. */

#include "bool.h"
#include "queue.h"
#include <stdio.h>
#include "geometry.h"
#define MAXV  100   /* maximum number of vertices */
#define MAXDEGREE   50    /* max outdeg of vertex */


typedef struct {
  int v;      /* neighboring vertex */
  int capacity;     /* capacity of edge */
  int flow;     /* flow through edge */
  int residual;     /* residual capacity of edge */
} edge;

typedef struct {
  edge edges[MAXV][MAXDEGREE];  /* adjacency info */
  int degree[MAXV];     /* outdegree of each vertex */
  int nvertices;     /* number of vertices in graph */
  int nedges;       /* number of edges in the graph */
} flow_graph;


main()
{
  flow_graph g;     /* graph to analyze */
  int source, sink;     /* source and sink vertices */
  int flow;     /* total flow */
  int i;      /* counter */

  scanf("%d %d",&source,&sink);
  read_flow_graph(&g,TRUE);

  netflow(&g,source,sink);

  print_flow_graph(&g);
```

```
  flow = 0;
  for (i=0; i<g.nvertices; i++)
  flow += g.edges[source][i].flow;

  printf("total flow = %d\n",flow);
}



initialize_graph(g)
flow_graph *g;      /* graph to initialize */
{
  int i;      /* counter */

  g -> nvertices = 0;
  g -> nedges = 0;

  for (i=0; i<MAXV; i++) g->degree[i] = 0;
}

read_flow_graph(g,directed)
flow_graph *g;      /* graph to initialize */
bool directed;      /* is this graph directed? */
{
  int i;      /* counter */
  int m;      /* number of edges */
  int x,y,w;    /* placeholder for edge and weight */

  initialize_graph(g);

  scanf("%d %d\n",&(g->nvertices),&m);

  for (i=1; i<=m; i++) {
  scanf("%d %d %d\n",&x,&y,&w);
  insert_flow_edge(g,x,y,directed,w);
  }
}

insert_flow_edge(flow_graph *g, int x, int y,
                 bool directed, int w)
{
  if (g->degree[x] > MAXDEGREE)
  printf("Warning: insertion(%d,%d) exceeds degree
          bound\n",x,y);

  g->edges[x][g->degree[x]].v = y;
  g->edges[x][g->degree[x]].capacity = w;
  g->edges[x][g->degree[x]].flow = 0;
  g->edges[x][g->degree[x]].residual = w;
  g->degree[x] ++;

  if (directed == FALSE)
    insert_flow_edge(g,y,x,TRUE,w);
  else
    g->nedges ++;
}


edge *find_edge(flow_graph *g, int x, int y)
{
  int i;      /* counter */

  for (i=0; i<g->degree[x]; i++)
  if (g->edges[x][i].v == y)
    return( &g->edges[x][i] );

  return(NULL);
}


add_residual_edges(flow_graph *g)
{
  int i,j;    /* counters */

  for (i=1; i<=g->nvertices; i++)
  for (j=0; j<g->degree[i]; j++)
    if (find_edge(g,g->edges[i][j].v,i) == NULL)
      insert_flow_edge(g,g->edges[i][j].v,i,TRUE,0);
}
```

```
print_flow_graph(flow_graph *g)
{
  int i,j;    /* counters */

  for (i=1; i<=g->nvertices; i++) {
  printf("%d: ",i);
  for (j=0; j<g->degree[i]; j++)
    printf(" %d(%d,%d,%d)",g->edges[i][j].v,
    g->edges[i][j].capacity,
    g->edges[i][j].flow,
    g->edges[i][j].residual);
  printf("\n");
  }
}

bool processed[MAXV];     /* which nodes were proc */
bool discovered[MAXV];    /* which nodes were found */
int parent[MAXV];     /* discovery relation */

bool finished = FALSE;  /* if true, cut off search */


initialize_search(g)
flow_graph *g;      /* graph to traverse */
{
  int i;        /* counter */

  for (i=1; i<=g->nvertices; i++) {
    processed[i] = FALSE;
    discovered[i] = FALSE;
    parent[i] = -1;
  }
}


bfs(flow_graph *g, int start)
{
  queue q;    /* queue of vertices to visit */
  int v;      /* current vertex */
  int i;      /* counter */

  init_queue(&q);
  enqueue(&q,start);
  discovered[start] = TRUE;

  while (empty(&q) == FALSE) {
  v = dequeue(&q);
  process_vertex(v);
  processed[v] = TRUE;
  for (i=0; i<g->degree[v]; i++)
    if (valid_edge(g->edges[v][i]) == TRUE) {
    if (discovered[g->edges[v][i].v] == FALSE) {
    enqueue(&q,g->edges[v][i].v);
    discovered[g->edges[v][i].v] = TRUE;
    parent[g->edges[v][i].v] = v;
    }
    if (processed[g->edges[v][i].v] == FALSE)
    process_edge(v,g->edges[v][i].v);
    }
  }
}


bool valid_edge(edge e)
{
  if (e.residual > 0) return (TRUE);
  else return(FALSE);
}

process_vertex(v)
int v;      /* vertex to process */
{
}

process_edge(x,y)
int x,y;        /* edge to process */
{
}
```

```
find_path(start,end,parents)
int start;      /* first vertex on path */
int end;    /* last vertex on path */
int parents[];      /* array of parent pointers */
{
  if ((start == end) || (end == -1))
    printf("\n%d",start);
  else {
    find_path(start,parents[end],parents);
    printf(" %d",end);
  }
}

int path_volume(flow_graph *g, int start, int end,
                int parents[])
{
  edge *e;    /* edge in question */
  edge *find_edge();

  if (parents[end] == -1) return(0);

  e = find_edge(g,parents[end],end);

  if (start == parents[end])
  return(e->residual);
  else
    return(min(path_volume(g,start,parents[end],parents),
                e->residual) );
}


augment_path(flow_graph *g, int start, int end,
             int parents[], int volume)
{
  edge *e;    /* edge in question */
  edge *find_edge();

  if (start == end) return;

  e = find_edge(g,parents[end],end);
  e->flow += volume;
  e->residual -= volume;

  e = find_edge(g,end,parents[end]);
  e->residual += volume;

  augment_path(g,start,parents[end],parents,volume);
}


netflow(flow_graph *g, int source, int sink)
{
  int volume;   /* weight of the augmenting path */

  add_residual_edges(g);

  initialize_search(g);
  bfs(g,source);

  volume = path_volume(g, source, sink, parent);

  while (volume > 0) {
  augment_path(g,source,sink,parent,volume);
  initialize_search(g);
  bfs(g,source);
  volume = path_volume(g, source, sink, parent);
  }
}
```

11 Dynamic Programming

```
/*  elevator.c -
    Elevator stop optimization via dyn programming.*/

/* the height of the building in floors */
#define NFLOORS     25
/* what is the capacity of the elevator? */
#define MAX_RIDERS  50
#define MAXINT  100007


/* what floor does everyone get off at? */
int stops[MAX_RIDERS];
int nriders;           /* number of riders */
int nstops;         /* number of allowable stops */

/* dynamic programming cost table */
int m[NFLOORS+1][MAX_RIDERS];
/* dynamic programming parent table */
int p[NFLOORS+1][MAX_RIDERS];


int min(int a, int b)
{
    if (a < b)  return(a); else return(b);
}

/*  m[i][j] denotes the cost of serving all the riders
  using j stops, the last of which is at floor i.
  Zero is the originating floor.
*/

int optimize_floors()
{
  int i,j,k;    /* counters */
  int cost;     /* costs placeholder */
  int laststop;     /* the elevator's last stop */

  for (i=0; i<=NFLOORS; i++) {
    m[i][0] = floors_walked(0,MAXINT);
    p[i][0] = -1;
  }

  for (j=1; j<=nstops; j++)
    for (i=0; i<=NFLOORS; i++) {
      m[i][j] = MAXINT;
      for (k=0; k<=i; k++) {
         cost = m[k][j-1] - floors_walked(k,MAXINT) +
        floors_walked(k,i) + floors_walked(i,MAXINT);
         if (cost < m[i][j]) {
          m[i][j] = cost;
        p[i][j] = k;
         }
      }
    }

  laststop = 0;
  for (i=1; i<=NFLOORS; i++)
    if (m[i][nstops] < m[laststop][nstops])
      laststop = i;

  return(laststop);
}


floors_walked(int previous, int current)
{
  int nsteps=0;     /* total distance traveled */
  int i;      /* counter */

  for (i=1; i<=nriders; i++)
    if ((stops[i] > previous) && (stops[i] <= current))
      nsteps += min(stops[i]-previous, current-stops[i]);

  return(nsteps);
}


reconstruct_path(int lastfloor, int stops_to_go)
{
  if (stops_to_go > 1)
    reconstruct_path( p[lastfloor][stops_to_go],
                      stops_to_go-1);

  printf("%d\n",lastfloor);
}

print_matrix(m)
int m[NFLOORS+1][MAX_RIDERS];
```

```
{
  int i,j;            /* counters */

    for (j=0; j<=nstops; j++) {
        for (i=0; i<=NFLOORS; i++)
            printf("%3d",m[i][j]);
        printf("\n");
    }
}


main()
{
  int i,j;      /* counters */
  int laststop;

  scanf("%d %d",&nriders,&nstops);


  for (i=1; i<=nriders; i++)
    scanf("%d",&(stops[i]));

    for (i=1; i<=nriders; i++)
        printf("%d\n",stops[i]);

  laststop = optimize_floors();

  print_matrix(&m);
  printf("\n");
  print_matrix(&p);

  printf("cost = %d\n",m[laststop][nstops]);

  reconstruct_path(laststop,nstops);

}
```

12 Grids

```
/*  order.c - Demonstrate traversal orders on a grid.*/

#include "geometry.h"

row_major(int n, int m)
{
    int i,j;  /* counters */

  for (i=1; i<=n; i++)
    for (j=1; j<=m; j++)
      process(i,j);
}

column_major(int n, int m)
{
  int i,j;  /* counters */

  for (j=1; j<=m; j++)
    for (i=1; i<=n; i++)
      process(i,j);
}


snake_order(int n, int m)
{
  int i,j;  /* counters */

  for (i=1; i<=n; i++)
    for (j=1; j<=m; j++)
      process(i, j + (m+1-2*j) * ((i+1) % 2));
}

diagonal_order(int n, int m)
{
  int d,j;  /* diagonal and point counters */
  int pcount; /* points on diagonal */
  int height; /* row of lowest point */

  for (d=1; d<=(m+n-1); d++) {
    height = 1 + max(0,d-m);
    pcount = min(d, (n-height+1));
    for (j=0; j<pcount; j++)
      process(min(m,d)-j, height+j);
  }
}

process(int i, int j)
{
  printf("(%d,%d)\n",i,j);
}

main()
{
  printf("row_major\n");
  row_major(5,5);

  printf("\ncolumn_major\n");
    column_major(3,3);

  printf("\nsnake_order\n");
    snake_order(5,5);

  printf("\ndiagonal_order\n");
    diagonal_order(3,4);

    printf("\ndiagonal_order\n");
    diagonal_order(4,3);
}
```

13 Geometry

```
/*  distance.c */

#define DIMENSION   3
#include <math.h>

typedef int point[DIMENSION];

main(){
  point a={6,2,3};
  point b={6,3,4};
  double distance();

  printf("distance = %f\n",distance(a,b));
}

double distance(point a, point b)
{
  int i;
  double d=0.0;

  for (i=0; i<DIMENSION; i++)
  d = d + (a[i]-b[i]) * (a[i]-b[i]);

  return( sqrt(d) );
}


/*  Basic geometric primitives and data types --
    Lines, Circles, Segments */

#include "bool.h"
#include "geometry.h"
#include <math.h>

points_to_line(point p1, point p2, line *l)
{
  if (p1[X] == p2[X]) {
  l->a = 1;
  l->b = 0;
  l->c = -p1[X];
  } else {
    l->b = 1;
  l->a = -(p1[Y]-p2[Y])/(p1[X]-p2[X]);
  l->c = -(l->a * p1[X]) - (l->b * p1[Y]);
  }
}

point_and_slope_to_line(point p, double m, line *l)
{
```

```
  l->a = -m;
  l->b = 1;
  l->c = -((l->a*p[X]) + (l->b*p[Y]));
}

bool parallelQ(line l1, line l2)
{
  return ( (fabs(l1.a-l2.a) <= EPSILON) &&
   (fabs(l1.b-l2.b) <= EPSILON) );
}

bool same_lineQ(line l1, line l2)
{
  return ( parallelQ(l1,l2) &&
          (fabs(l1.c-l2.c) <= EPSILON) );
}


intersection_point(line l1, line l2, point p)
{
  if (same_lineQ(l1,l2)) {
  printf("Warning: Identical lines, all points
          intersect.\n");
  p[X] = p[Y] = 0.0;
  return;
  }

  if (parallelQ(l1,l2) == TRUE) {
  printf("Error: Distinct parallel lines do not
          intersect.\n");
  return;
  }

  p[X] = (l2.b*l1.c - l1.b*l2.c) /
         (l2.a*l1.b - l1.a*l2.b);

  if (fabs(l1.b) > EPSILON)   /* test for vert line */
  p[Y] = - (l1.a * (p[X]) + l1.c) / l1.b;
  else
  p[Y] = - (l2.a * (p[X]) + l2.c) / l2.b;
}

closest_point(point p_in, line l, point p_c)
{
  line perp;    /* perpendicular to l through (x,y) */

  if (fabs(l.b) <= EPSILON) { /* vertical line */
  p_c[X] = -(l.c);
  p_c[Y] = p_in[Y];
  return;
  }

  if (fabs(l.a) <= EPSILON) { /* horizontal line */
  p_c[X] = p_in[X];
  p_c[Y] = -(l.c);
  return;
  }

  point_and_slope_to_line(p_in,1/l.a,&perp);
  /* non-degenerate line */
/*printf("perpendicular bisector "); print_line(perp);*/
  intersection_point(l,perp,p_c);
/*printf("closest point "); print_point(p_c);*/
}

double distance(point a, point b)
{
  int i;    /* counter */
  double d=0.0;     /* accumulated distance */

  for (i=0; i<DIMENSION; i++)
    d = d + (a[i]-b[i]) * (a[i]-b[i]);

  return( sqrt(d) );
}


copy_point(point a, point b)
{
  int i;    /* counter */
  for (i=0; i<DIMENSION; i++) b[i] = a[i];
}

swap_point(point a, point b)
{
  point c;  /* temporary point */

  copy_point(a,c);
  copy_point(b,a);
  copy_point(c,b);
}


points_to_segment(point a, point b, segment *s)
{
  copy_point(a,s->p1);
  copy_point(b,s->p2);
}

segment_to_points(segment s, point p1, point p2)
{
  copy_point(s.p1,p1);
  copy_point(s.p2,p2);
}


bool point_in_box(point p, point b1, point b2)
{
  return( (p[X] >= min(b1[X],b2[X])) &&
          (p[X] <= max(b1[X],b2[X])) &&
          (p[Y] >= min(b1[Y],b2[Y])) &&
          (p[Y] <= max(b1[Y],b2[Y])) );
}

bool segments_intersect(segment s1, segment s2)
{
  line l1,l2;   /* lines containing input segments */
  point p;  /* intersection point */

  points_to_line(s1.p1,s1.p2,&l1);
  points_to_line(s2.p1,s2.p2,&l2);

  if (same_lineQ(l1,l2))
  /* overlapping or disjoint segments */
    return( point_in_box(s1.p1,s2.p1,s2.p2) ||
      point_in_box(s1.p2,s2.p1,s2.p2) ||
      point_in_box(s2.p1,s1.p1,s1.p2) ||
      point_in_box(s2.p1,s1.p1,s1.p2) );

  if (parallelQ(l1,l2)) return(FALSE);

  intersection_point(l1,l2,p);
  return( point_in_box(p,s1.p1,s1.p2) &&
          point_in_box(p,s2.p1,s2.p2) );
}

double signed_triangle_area(point a, point b, point c)
{
  return( (a[X]*b[Y] - a[Y]*b[X] + a[Y]*c[X]
  - a[X]*c[Y] + b[X]*c[Y] - c[X]*b[Y]) / 2.0 );
}

double triangle_area(point a, point b, point c)
{
  return( fabs(signed_triangle_area(a,b,c)) );
}

bool ccw(point a, point b, point c)
{
  double signed_triangle_area();

  return (signed_triangle_area(a,b,c) > EPSILON);
}

bool cw(point a, point b, point c)
{
  double signed_triangle_area();

  return (signed_triangle_area(a,b,c) < EPSILON);
}
```

```
bool collinear(point a, point b, point c)
{
  double signed_triangle_area();

  return (fabs(signed_triangle_area(a,b,c)) <= EPSILON);
}

print_points(point p[], int n)
{
  int i;      /* counter */

  for (i=0; i<n; i++)
    printf("(%lf,%lf)\n",p[i][X],p[i][Y]);
}

print_polygon(polygon *p)
{
  int i;    /* counter */

  for (i=0; i<p->n; i++)
    printf("(%lf,%lf)\n",p->p[i][X],p->p[i][Y]);
}

print_point(point p)
{
  printf("%7.3lf %7.3lf\n",p[X],p[Y]);
}

print_line(line l)
{
  printf("(a=%7.3lf,b=%7.3lf,c=%7.3lf)\n",l.a,l.b,l.c);
}

print_segment(segment s)
{
  printf("segment: ");
  print_point(s.p1);
  print_point(s.p2);
}
```

14 Computational Geometry

```
/*  convex-hull.c */

#include "bool.h"
#include "geometry.h"
#include <math.h>

point first_point;    /* first hull point */

convex_hull(point in[], int n, polygon *hull)
{
  int i;    /* input counter */
  int top;  /* current hull size */
  bool smaller_angle();

  if (n <= 3) {     /* all points on hull! */
  for (i=0; i<n; i++)
      copy_point(in[i],hull->p[i]);
  hull->n = n;
  return;
  }

  sort_and_remove_duplicates(in,&n);
  copy_point(in[0],&first_point);

  qsort(&in[1], n-1, sizeof(point), smaller_angle);

  copy_point(first_point,hull->p[0]);
  copy_point(in[1],hull->p[1]);

  copy_point(first_point,in[n]);
  /* sentinel to avoid special case */
  top = 1;
  i = 2;

  while (i <= n) {
  if (!ccw(hull->p[top-1], hull->p[top], in[i]))
    top = top-1;  /* top not on hull */
  else {
    top = top+1;
      copy_point(in[i],hull->p[top]);
    i = i+1;
  }
  }

  hull->n = top;
}


sort_and_remove_duplicates(point in[], int *n)
{
  int i;       /* counter */
  int oldn;    /* number of points before deletion */
  int hole;    /* index marked for potential del */
  bool leftlower();

  qsort(in, *n, sizeof(point), leftlower);

  oldn = *n;
  hole = 1;
  for (i=1; i<(oldn-1); i++) {
  if ((in[hole-1][X] == in[i][X]) &&
      (in[hole-1][Y] == in[i][Y]))
      (*n)--;
    else {
      copy_point(in[i],in[hole]);
      hole = hole + 1;
    }
  }
  copy_point(in[oldn-1],in[hole]);
}


main(){
  point in[MAXPOLY];    /* input points */
  polygon hull;     /* convex hull */
  int n;      /* number of points */
  int i;      /* counter */

  scanf("%d",&n);
  for (i=0; i<n; i++)
  scanf("%lf %lf",&in[i][X],&in[i][Y]);

  convex_hull(in,n,&hull);

  print_polygon(&hull);
}


bool leftlower(point *p1, point *p2)
{
  if ((*p1)[X] < (*p2)[X]) return (-1);
  if ((*p1)[X] > (*p2)[X]) return (1);

  if ((*p1)[Y] < (*p2)[Y]) return (-1);
  if ((*p1)[Y] > (*p2)[Y]) return (1);

  return(0);
}

/*
bool leftlower(point *p1, point *p2)
{
  if (fabs((*p1)[X] - (*p2)[X]) > EPSILON) {
    if ((*p1)[X] < (*p2)[X]) return (-1);
    if ((*p1)[X] > (*p2)[X]) return (1);
  }

  if (fabs((*p1)[Y] - (*p2)[Y]) > EPSILON) {
    if ((*p1)[Y] < (*p2)[Y]) return (-1);
    if ((*p1)[Y] > (*p2)[Y]) return (1);
  }

  return(0);
}
*/

bool smaller_angle(point *p1, point *p2)
{
```

```
  if (collinear(first_point,*p1,*p2)) {
  if (distance(first_point,*p1) <=
      distance(first_point,*p2))
    return(-1);
  else
    return(1);
  }

  if (ccw(first_point,*p1,*p2))
    return(-1);
  else
    return(1);
}
```

```
/*  triangulate.c - Triangulate a polygon via
                    ear-clipping, compute area. */


#include "bool.h"
#include "geometry.h"
#include <math.h>


triangulate(polygon *p, triangulation *t)
{
  int l[MAXPOLY], r[MAXPOLY];
  /* left/right neighbor indices */
  int i;      /* counter */

  for (i=0; i<p->n; i++) {  /* initialization */
  l[i] = ((i-1) + p->n) % p->n;
  r[i] = ((i+1) + p->n) % p->n;
  }

  t->n = 0;
  i = p->n-1;
  while (t->n < (p->n-2)) {
  i = r[i];
  if (ear_Q(l[i],i,r[i],p)) {
    add_triangle(t,l[i],i,r[i],p);
    l[ r[i] ] = l[i];
    r[ l[i] ] = r[i];
  }
  }
}

add_triangle(triangulation *t, int i, int j, int k,
             polygon *p)
{
  int n;      /* number of triangles in t */

  n = t->n;

  t->t[n][0] = i;
  t->t[n][1] = j;
  t->t[n][2] = k;

  t->n = n + 1;
}

bool ear_Q(int i, int j, int k, polygon *p)
{
  triangle t;   /* coordinates for points i,j,k */
  int m;      /* counter */
  bool cw();

  copy_point(p->p[i],t[0]);
  copy_point(p->p[j],t[1]);
  copy_point(p->p[k],t[2]);

  if (cw(t[0],t[1],t[2])) return(FALSE);

  for (m=0; m<p->n; m++) {
  if ((m!=i) && (m!=j) && (m!=k))
    if (point_in_triangle(p->p[m],t)) return(FALSE);
  }

  return(TRUE);
}
```

```
bool point_in_triangle(point p, triangle t)
{
  int i;      /* counter */
  bool cw();

  for (i=0; i<3; i++)
  if (cw(t[i],t[(i+1)%3],p)) return(FALSE);

  return(TRUE);
}

double area_triangulation(polygon *p)
{
  triangulation t;    /* output triangulation */
  double total = 0.0;   /* total area so far */
  int i;        /* counter */
  double triangle_area();

  triangulate(p,&t);
  for (i=0; i<t.n; i++)
  total += triangle_area(p->p[t.t[i][0]],
    p->p[t.t[i][1]], p->p[t.t[i][2]]);

  return(total);
}

double area(polygon *p)
{
  double total = 0.0;   /* total area so far */
  int i, j;     /* counters */

  for (i=0; i<p->n; i++) {
    j = (i+1) % p->n;
    total += (p->p[i][X]*p->p[j][Y]) -
             (p->p[j][X]*p->p[i][Y]);
  }

  return(total / 2.0);
}



main(){
  polygon p;    /* input polygon */
  triangulation t;  /* output triangulation */
  int i;      /* counter */
  double area(), area_triangulate();

  scanf("%d",&p.n);
  for (i=0; i<p.n; i++)
    scanf("%lf %lf",&p.p[i][X],&p.p[i][Y]);

/*
  print_polygon(&p);
  triangulate(&p,&t);
  print_triangulation(&t);
*/

  printf(" area via triangulation = %f\n",
          area_triangulation(&p));
  printf(" area slick = %f\n", area(&p));

}

print_triangulation(triangulation *t)
{
  int i, j;     /* counters */

  for (i=0; i< t->n; i++) {
  for (j=0; j<3; j++)
      printf(" %d ",t->t[i][j]);
/*
  for (j=0; j<3; j++)
    printf(" (%5.3f,%5.3f)",t->t[i][j][X],t->t[i][j][Y]);
*/
  printf("\n");
  }
}
```

```
/*  plates.c

    Compute the number of circles in two different
    disk packings. Assuming we have an $w \times l$ box,
    how many unit disks can we pack in there assuming
    we have w disks on the bottom?
*/

#include        <stdio.h>
#include        <math.h>

/*  how many triangular-lattice layers of radius r balls
    fit in height h?
*/

int dense_layers(double w, double h, double r)
{
    double gap;     /* distance between layers */

  if ((2*r) > h) return(0);

  gap = 2.0 * r * (sqrt(3)/2.0);
  return( 1 + floor((h-2.0*r)/gap) );
}

int plates_per_row(int row, double w, double r)
{
  int plates_per_full_row; /* #plates in full/even row */

  plates_per_full_row = floor(w/(2*r));

  if ((row % 2) == 0) return(plates_per_full_row);

  if (((w/(2*r))-plates_per_full_row) >= 0.5)
    /* odd row full, too */
        return(plates_per_full_row);
    else
        return(plates_per_full_row - 1);
}

/*  How many radius r plates fit in a hexagonal-lattice
    packed w*h box?
*/

int dense_plates(double w, double l, double r)
{
  int layers;     /* number of layers of balls */

  layers = dense_layers(w,l,r);

  return (ceil(layers/2.0) * plates_per_row(0,w,r) +
    floor(layers/2.0) * plates_per_row(1,w,r) );
}

int grid_plates(double w, double h, double r)
{
    int layers;     /* number of layers of balls */

    layers = floor(h/(2*r));

    return (layers * plates_per_row(0,w,r));
}

/* Hexagonal coordinates start with the center of disk
   (0,0) at geometric point (0,0).  The hexagonal
   coordinate $(xh,yh)$ refers to the center of the disk
   on the horizontal row xh and positive-slope diagonal
   yh. The geometric coordinate of such a point is a
   function of the radius of the disk $r$.
*/

hex_to_geo(int xh, int yh, double r, double *xg,
           double *yg)
{
  *yg = (2.0 * r) * xh * (sqrt(3)/2.0);
  *xg = (2.0 * r) * xh * (1.0/2.0) + (2.0 * r) * yh;
}

geo_to_hex(double xg, double yg, double r, double *xh,
           double *yh)
{
  *xh = (2.0/sqrt(3)) * yg / (2.0 * r);
  *yh = (xg - (2.0 * r) * (*xh)*(1.0/2.0) )/(2.0 * r);
}

/* Under the hexagonal coordinate system, the set of
   hexagons defined by coordinates (hx,hy), where
   $0 <= hx <= xmax$ and $0 <= hx <= ymax$ forms a
   diamond-shaped patch, not a conventional axis-oriented
   rectangle.  To solve this problem, we define array
   coordinates so that (ax,ay) refers to the position in
   an axis-oriented rectangle with (0,0) as the lower
   righthand point in the matrix.
*/

array_to_hex(int xa, int ya, int *xh, int *yh)
{
  *xh = xa;
  *yh = ya - xa + ceil(xa/2.0);
}

hex_to_array(int xh, int yh, int *xa, int *ya)
{
  *xa = xh;
  *ya = yh + xh - ceil(xh/2.0);
}

int plates_on_top(int xh, int yh, double w, double l,
                  double r)
{
  int number_on_top = 0;    /* total plates on top */
  int layers;     /* number of rows in grid */
  int rowlength;      /* number of plates in row */
  int row;      /* counter */
  int xla,yla,xra,yra;    /* array coordinates */

    layers = dense_layers(w,l,r);

  for (row=xh+1; row<layers; row++) {
    rowlength = plates_per_row(row,w,r) - 1;

    hex_to_array(row,yh-row,&xla,&yla);
    if (yla < 0) yla = 0;       /* left boundary */

    hex_to_array(row,yh,&xra,&yra);
    if (yra > rowlength) yra = rowlength;
    /* right boundary */

    /*printf("row=%d yla=%d yra=%d\n",row,yla,yra);*/

    number_on_top += yra-yla+1;
  }

  return(number_on_top);
}

main()
{
  double w;       /* box width */
  double l;       /* box length */
  double r;       /* plate radius */

  int i,j;      /* counters */
  int xh,yh,xa,ya;
  double xhf,yhf,xg,yg;
  int xmax,ymax;

  printf("input box width, box length, and plate
          radius:\n");
  scanf("%lf %lf %lf",&w,&l,&r);
  printf("box width=%lf, box length=%lf, and plate
          radius=%lf:\n",w,l,r);

  printf("dense packing = %d\n", dense_plates(w,l,r));
  printf("grid packing = %d\n",  grid_plates(w,l,r));

  /* print all the possible hexes in the box */

  xmax = floor(w / (2*r));
```

```
  ymax = dense_layers(w,l,r);

/*
  for (i=0; i<=xmax; i++)
    for (j=0; j<=ymax; j++) {
      printf("array(%d,%d) ",i,j);
      array_to_hex(i,j,&xh,&yh);
      printf("to hex(%d,%d) ",xh,yh);
      hex_to_geo(xh,yh,r,&xg,&yg);
      printf("to geo(%4.2f,%4.2f) ",xg,yg);
      geo_to_hex(xg,yg,r,&xhf,&yhf);
      printf("to hex(%4.2f,%4.2f) ",xhf,yhf);
      hex_to_array(xh,yh,&xa,&ya);
      printf("to array(%d,%d)\n",xa,ya);
    }
*/

  for (i=0; i<xmax; i++)
    printf("(0,%d) has %d on top.\n",i,
           plates_on_top(0,i,w,l,r));
}
```

```
/*  superman.c - Compute Superman's flight path */

#define MAXN    100     /* maximum number of circles */

#include "bool.h"
#include "geometry.h"
#include <math.h>

point s;            /* Superman's initial position */
point t;            /* target position */
int ncircles;           /* number of circles */
circle c[MAXN];         /* circles data structure */

superman()
{
  line l;     /* line from start to target pos */
  point close;  /* closest point */
  double d;     /* distance from circle-center */
  /* length of intersection with circles */
  double xray = 0.0;
  /* length around circular arcs */
  double around = 0.0;
  double angle;     /* angle subtended by arc */
  double travel;    /* total travel distance */
  int i;      /* counter */
  double asin(), sqrt();
  double distance();

  points_to_line(s,t,&l);

  for (i=1; i<=ncircles; i++) {
    closest_point(c[i].c,l,close);
    d = distance(c[i].c,close);
    if ((d>=0) && (d < c[i].r) &&
      point_in_box(close,s,t)) {
      xray += 2*sqrt(c[i].r*c[i].r - d*d);
      angle = acos(d/c[i].r);
      around += ((2*angle)/(2*PI)) * (2*PI*c[i].r);
printf("circle %d (%7.3lf,%7.3lf,%7.3lf) is %7.3lf
  away from l, xray=%7.3lf around=%7.3lf angle=%7.3lf.\n",
  i,c[i].c[X],c[i].c[Y],c[i].r,d,xray,around,angle);
    }
  }

  travel = distance(s,t) - xray + around;
  printf("Superman sees through %7.3lf units, and
    flies %7.3lf units\n", xray, travel);
}


main(){
  int i;      /* counter */

  scanf("%lf %lf",&s[X],&s[Y]);
  scanf("%lf %lf",&t[X],&t[Y]);
  scanf("%d",&ncircles);
  for (i=1; i<=ncircles; i++)
    scanf("%lf %lf %lf",&c[i].c[X],&c[i].c[Y],&c[i].r);

    printf("%7.3lf %7.3lf\n",s[X],s[Y]);
    printf("%7.3lf %7.3lf\n",t[X],t[Y]);
    printf("%d\n",ncircles);
    for (i=1; i<=ncircles; i++)
        printf("%7.3lf %7.3lf %7.3lf\n",
                c[i].c[X],c[i].c[Y],c[i].r);

  superman();
}
```

15 Diverse Files

```
/*  war.c - Simulation of card game War */

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "bool.h"
#include "queue.h"

#define NGAMES  50
#define MAXSTEPS 100000

#define NCARDS  52  /* number of cards */
#define NSUITS  4   /* number of suits */


char values[] = "23456789TJQKA";
char suits[] = "cdhs";


/*  Rank the card with given value and suit.    */

int rank_card(char value, char suit)
{
    int i,j;  /* counters */

  for (i=0; i<(NCARDS/NSUITS); i++)
    if (values[i]==value)
      for (j=0; j<NSUITS; j++)
        if (suits[j]==suit)
          return( i*NSUITS + j );

  printf("Warning: bad input value=%d, suit=%d\n",
         value,suit);
}


/*  Return the suit and value of the given card. */

char suit(int card)
{
  return( suits[card % NSUITS] );
}

char value(int card)
{
  return( values[card/NSUITS] );
}


testcards(){
  int i;        /* counter */
  char suit(), value();     /* reconstructed card */

  for (i=0; i<NCARDS; i++)
    printf(" i=%d card[i]=%c%c rank=%d\n", i, value(i),
      suit(i), rank_card(value(i),suit(i)) );
}

/************************************************/

random_init_decks(a,b)
queue *a,*b;
{
  int i;        /* counter */
  int perm[NCARDS+1];
```

```
  for (i=0; i<NCARDS; i=i+1) {
        perm[i] = i;
    }

  random_permutation(perm,NCARDS);

  init_queue(a);
  init_queue(b);

  for (i=0; i<NCARDS/2; i=i+1) {
    enqueue(a,perm[2*i]);
    enqueue(b,perm[2*i+1]);
  }

  print_card_queue(a);
  print_card_queue(b);

}

war(queue *a, queue *b)
{
  int steps=0;      /* step counter */
  int x,y;      /* top cards */
  queue c;      /* cards involved in the war */
  bool inwar;     /* are we involved in a war? */

  inwar = FALSE;
  init_queue(&c);

/*printf("deck counts a=%d b=%d\n",a->count,b->count);*/


  while ((!empty(a)) &&
         (!empty(b) && (steps < MAXSTEPS))) {
/*
print_card_queue(a);
print_card_queue(b);
*/
    steps = steps + 1;
    x = dequeue(a);
    y = dequeue(b);
    enqueue(&c,x);
    enqueue(&c,y);
    if (inwar) {
      inwar = FALSE;
    } else {
      if (value(x) > value(y))
        clear_queue(&c,a);
      else if  (value(x) < value(y))
        clear_queue(&c,b);
      else if (value(y) == value(x))
        inwar = TRUE;
    }
    /*printf("x=%d y=%d |a|=%d |b|=%d \n",x,y,a->count,
            b->count);*/
  }

/*printf("deck counts a=%d b=%d\n",a->count,b->count);*/

  if (!empty(a) && empty(b))
    printf("a wins in %d steps \n",steps);
  else if (empty(a) && !empty(b))
    printf("b wins in %d steps \n",steps);
  else if (!empty(a) && !empty(b))
    printf("game tied after %d steps, |a|=%d |b|=%d \n",
      steps,a->count,b->count);
  else
    printf("a and b tie in %d steps \n",steps);
}


print_card_queue(q)
queue *q;
{
    int i,j;

    i=q->first;

    while (i != q->last) {
        /*printf("%2d ",q->q[i]);*/
        printf("%c%c ",value(q->q[i]),suit(q->q[i]));
        i = (i+1) % QUEUESIZE;
    }

    printf("%2d ",q->q[i]);
    printf("\n");
}


clear_queue(queue *a, queue *b)
{
  /*printf("war ends with %d cards \n",a->count);*/
  while (!empty(a))
    enqueue(b,dequeue(a));
}

old_main(){
  queue a,b;
  int i;

  /*testcards();*/

  for (i=1; i<=NGAMES; i++) {
    random_init_decks(&a,&b);
    war(&a,&b);
  }
}


main()
{
  queue decks[2];     /* player's decks */
  char value,suit,c;    /* input characters */
  int i;        /* deck counter */

  while (TRUE) {
    for (i=0; i<=1; i++) {
      init_queue(&decks[i]);

      while ((c = getchar()) != '\n') {
        if (c == EOF) return;
        if (c != ' ') {
          value = c;
          suit = getchar();
          enqueue(&decks[i],rank_card(value,suit));
        }
      }
    }

    war(&decks[0],&decks[1]);
  }
}
```

```
//*************************************************
// A solution to a previous prolem using floyd-warshall
//  path algorithm. (It works on sample input)

#include<iostream>
#include<iomanip>
#include<vector>
#include<math.h>
#include<queue>

const double NONE=1000000000;
//must be bigger than total weight of edges
const int MAX=100;

class graph{
    public:
    double w[MAX+1][MAX+1];
    int n;
    bool used[MAX+1];
    graph(){
      for (int i=0; i<MAX+1; i++){
        used[i]=false;
        for (int j=0; j<MAX+1; j++)
          w[i][j]=NONE;
      }
      n=0;
```

```
    }
    void addEdge(int a, int b, double w){
      addEdgeD(a,b,w);
      addEdgeD(b,a,w);
    }
    void addEdgeD(int a, int b, double weight){
      w[a][b]=weight;
      if (!used[a]) n++;
      used[a]=true;
      if (!used[b]) n++;
      used[b]=true;
    }

    void Floyd(){
      //replaces edges with lengths of shortest paths
      //to each edge
      for (int k=1; k<=MAX; k++)
      if (used[k]) for (int i=1; i<=MAX; i++)
      if (used[i]) for (int j=1; j<=MAX; j++)
      if (used[j])
        if (w[i][j] > w[i][k] + w[k][j])
          addEdgeD(i, j,w[i][k]+w[k][j]);
    }

    double operator()(int a, int b){
      return w[a][b];
    }
};


//************************************************
void output(double total, int cas){
  cout.setf(ios::showpoint);
    cout.setf(ios::fixed);
    cout.precision(3);
  cout<<"Case "<<cas<<": average length between pages = "
      <<total<<" clicks"<<endl;
}

int main(){
  int c=0;
  while(1){
    c++;
    int a,b;
    graph g;
    while(1){
      cin>>a>>b;
      if (a==0 && b==0) break;
      g.addEdgeD(a,b,1);
    }
    if (g.n==0) return 0;
    g.Floyd();
    double total=0;
    for (int i=1; i<=MAX; i++)
      for (int j=1; j<=MAX; j++){
        if (i==j) continue;
        if (!g.used[j] || !g.used[i]) continue;
      //  cout<<i<<' '<<j<<' '<<g(i,j)<<endl;
        total+=g(i,j);
      }
    total/=(g.n*(g.n-1));
    output(total,c);
  }
}
```

```
//solution to prob b "say cheese" from 2001 using
//dijstra''s algorithm/min spanning tree
//note: min spanning tree not tested!
//see the other graph class for the shorter
//floyd-warshall algorithm

//Also contains a short distance function

const double NONE=1000000000;
//must be more than total weight of all edges
const int MAXV=105;
const int MAXD=105;

class edge{
  public:
  int v;
  double w;
  edge(int vv=0, double ww=0){
    v=vv; w=ww;
  }
};

class graph{
  public:
    edge edges[MAXV+1][MAXD];
    int degree[MAXV+1];
    int n;
    int parent[MAXV];

    graph(int nn){
      for (int i=0; i<MAXV+1; i++){
        degree[i]=0;
      }
      n=nn;
    }

    void addEdge(int a, int b, double w){
      addEdgeD(a,b,w);
      addEdgeD(b,a,w);
    }

    void addEdgeD(int a, int b, double weight){

      edges[a][degree[a]]=edge(b,weight);
      degree[a]++;
    }



    double minSpanningTree(int source){
      //returns the total weight of MST, tree data
      // is in parent[]
      double cost=0;
      double dist[MAXV];
      bool done[MAXV];

      for (int i=0; i<n; i++){
        done[i]=false;
        dist[i]=NONE;
        parent[i]=-1;
      }
      dist[source]=0;
      int v=source;
      while(!done[v]){
        cost+=dist[v];
        done[v]=true;
        for (int i=0; i<degree[v]; i++){
          int w=edges[v][i].v;
          if ((dist[w]>edges[v][i].w) && (!done[w])){
            dist[w]=edges[v][i].w;
            parent[w]=v;
          }
        }
        v=0;
        for (int i=1; i<n; i++)
          if (!done[i] && dist[i]<dist[v]) v=i;
      }
      return cost;
    }

    vector<double> shortestPaths(int source){
      //returns the vector of shortest distances,
      //parent[v] is the vertex one visits before v
      //in shortest path
      //from source to v
      vector<double> dist(n);
      bool done[MAXV];

      for (int i=0; i<n; i++){
        done[i]=false;
        dist[i]=NONE;
        parent[i]=-1;
      }
      dist[source]=0;
      int v=source;
```

```
      while(!done[v]){
        done[v]=true;
        for (int i=0; i<degree[v]; i++){
          int w=edges[v][i].v;
          if ((dist[w]> (dist[v]+edges[v][i].w)) &&
              (!done[w])){
            dist[w]=dist[v]+edges[v][i].w;
            parent[w]=v;
          }
        }
        v=0;
        for (int i=1; i<n; i++)
          if (!done[i] && dist[i]<dist[v]) v=i;
      }
      return dist;
    }

    vector<int> inDegrees()
    //number of incoming edges
    {
      int i,j;
      vector<int> in(n,0);
      for (i=0; i<n; i++)
        for (j=0; j<degree[i]; j++) in[edges[i][j].v]++;
      return in;
    }


    vector<int> top_sort()
    {
      vector<int> sorted(n);

      queue<int> zero;    /* vertices of indegree 0 */
      int x, y;       /* current and next vertex */
      int i, j;

      vector<int> indegree=inDegrees();

      for (i=0; i<n; i++)
        if (indegree[i] == 0) zero.push(i);

      j=0;
      while (!zero.empty()) {
        j = j+1;
        x = zero.front();
        zero.pop();
        sorted[j] = x;
        for (i=0; i<degree[x]; i++) {
          y = edges[x][i].v;
          indegree[y] --;
          if (indegree[y] == 0) zero.push(y);
        }
      }

      if (j != n)
        cout<<"error";

      return sorted;
    }

};
//*************************************************

double distance(double x[], double y[], int n=3){
  double total=0;
  for (int i=0; i<n; i++){
    total+=(x[i]-y[i])*(x[i]-y[i]);
  }
  return sqrt(total);
}


//*************************************************
void output(double total, int cas){
    cout.setf(ios::fixed);
    cout.precision(0);
  total+=.4999;
  cout<<"Cheese "<<cas<<": Travel time  = "<<total
    <<" sec"<<endl;
}
```

```
double error=.0000000001;
int main(){
  int c=0;
  while(1){
    c++;
    double x[MAXV][3];
    double r[MAXV];
    int n;
    cin>>n;
    if (n==-1) return 0;
    graph g(n+2);

    for (int i=0; i<n+2; i++){
      for (int j=0; j<3; j++)
        cin>>x[i][j];
      if (i<n) cin>>r[i];
        else r[i]=0;
    }

    for (int i=0; i<n+2; i++)
    for (int j=0; j<n+2; j++){
      double d=distance(x[i], x[j])-r[i]-r[j];
      if (d<error) d=0;
      g.addEdge(i,j,d);
    }
    vector<double> d=g.shortestPaths(n);

    output(d[n+1]*10,c);
  }
}


//roman numerals
int toInt(string a){
  int start=0, total=0;
  while(start<a.length()){
    switch(a[start]){
      case 'D': total+=500;
          break;
      case 'M': total+=1000;
          break;
      case 'L': total+=50;
          break;
      case 'V': total+=5;
          break;
      case 'C': if (start+1==a.length())
            total+=100;
          else
            if (toInt(a.substr(start+1,1))>100)
              total-=100;
          else
            total+=100;
          break;
      case 'X': if (start+1==a.length())
          total+=10;
          else
          if (toInt(a.substr(start+1,1))>10)
            total-=10;
          else
            total+=10;
           break;
      case 'I': if (start+1==a.length())
          total+=1;
          else
          if (toInt(a.substr(start+1,1))>1)
            total-=1;
          else
            total+=1;
           break;
      }
      start++;
    }
  return total;
}


//An output function to length 20

void output(string res, int len=20){
  if (res.length()<len) cout<<res<<endl;
```

```
  else {
    cout<<res.substr(0,len)<<endl;
    output(res.substr(len,res.length()-len));
  }
}
```

Sphere distance:

$$d(f,s) = R{\cdot}acos((sin(l[f]){\cdot}sin(l[s]))+(cos(l[f])\cot cos(l[s]){\cdot}cos(dg)))$$

where R is the radius of the great circle, $dg = longitude(f) - longitude(s)$,
and $l[f] = lattitude(f)$.

Some useful Java commands:

- Converting to an arbitrary base:
  int my_int = Integer.parseInt(string_val,radix);
  String s = Integer.toString(my_int,radix);

- Formatting Doubles to Arbitrary Precision:
  NumberFormat formatter = new DecimalFormat("####.00");
  // 0's force a 0 to appear if no number is there
  // #'s allow no numnber to be shown
  Sring s = formatter.format(my_num);