# Índice

## 1.1. Heap

Funciones para transformar un array en una *heap*. La cantidad $N$ de datos en la *heap* se encuentra en el primer elemento del array, y los elementos en sí están en los índices $1, \ldots, N$. La función $SWAPH(a, b)$ intercambia dos elementos de la *heap* (posiblemente intercambiando también referencias a esos elementos en otras estructuras). La función *compare* compara dos elementos de la *heap*: si se usa $a < b$ se tiene una *min-heap*, y si se usa $a > b$ se tiene una *max-heap*. [REQUIERE: $SWAP(a, b)$]

```
#define SWAPH(a,b) ( SWAP(A[(a)],A[(b)]) )

bool compare(int a, int b) {return a < b;}

#define PARENT(n) ( (n)/2 )
#define LEFT(n) ( (n)<<1 )
#define RIGHT(n) ( ((n)<<1)+1 )

void heapify(int A[], int i, bool cmp(int, int)) {
    int best;
    int l = LEFT(i);
    int r = RIGHT(i);
    if (l <= A[0] && cmp(A[l], A[i]) == true) best = l;
    else best = i;
    if (r <= A[0] && cmp(A[r], A[best]) == true) best = r;
    if (best != i) {
        SWAPH(i, best);
        heapify(A, best, cmp);
    }
}

void build_heap(int A[], bool cmp(int, int)) {
    for (int i=A[0]/2; i>=1; i--) heapify(A, i, cmp);
}

void heapsort(int A[], bool cmp(int, int)) {
    build_heap(A, cmp);
    for (int i=A[0]; i>=2; i--) {
        SWAPH(1, i);
        A[0]--;
        heapify(A, 1, cmp);
    }
}

int top(int A[]) {return A[1];}

int pop(int A[], bool cmp(int, int)) {
    int max = A[1];
    SWAPH(1, A[0]);
    A[0]--;
    heapify(A, 1, cmp);
    return max;
}

void update(int A[], int i, bool cmp(int, int)) {
    while (i > 1 && cmp(A[PARENT(i)], A[i]) == false) {
        SWAPH(i, PARENT(i));
        i = PARENT(i);
    }
}

void insert(int A[], bool cmp(int, int)) {
    A[0]++;
    update(A, A[0], cmp);
}
```

## 1.2. Binary indexed tree

Funciones para trabajar con un array como un *binary indexed tree*. *get_cf* devuelve la suma de las frecuencias en el rango $[0, idx]$, *get_f* devuelve la frecuencia de $idx$, y *upd_f* incrementa la frecuencia de $idx$ en la cantidad $f$ (donde $f$ puede ser tanto positivo como negativo). Todas las funciones toman como argumento adicional el array en el que está el *bit*, y funcionan en tiempo $\mathcal{O}(logN)$. Para inicializar el *bit* hay que poner todos los elementos del array que se va a usar en 0, y definir la variable $MAXN$ como el tamaño máximo del array.

```
int get_cf(int idx, int *bit) {
    int cf = bit[0];
    while (idx > 0) {
        cf += bit[idx];
        idx &= idx-1;
    }
    return cf;
}

void upd_f(int idx, int f, int *bit) {
    if (idx == 0) bit[idx] += f;
    else while (idx < MAXN) {
        bit[idx] += f;
        idx += idx&(-idx);
    }
}

int get_f(int idx, int *bit) {
    int f = bit[idx];
    if (idx > 0) {
        int p = idx&(idx-1); idx--;
        while (p != idx) {
            f -= bit[idx];
            idx &= idx-1;
        }
    }
```

```
27     return f;
28 }
```

Para trabajar en dos dimensiones, hay que agregar las funciones $bi_get\_cf$, $bi_get\_f$ y $bi_upd\_f$, y definir $MAXM$ como el número máximo de columnas ($MAXN$ se transforma en el número máximo de filas).

```
1  int bi_get_cf(int idx, int idy, int **bit) {
2      int cf = get_cf(idy, bit[0]);
3      while (idx > 0) {
4          cf += get_cf(idy, bit[idx]);
5          idx &= idx-1;
6      }
7      return cf;
8  }
9
10 void bi_upd_f(int idx, int idy, int f, int **bit) {
11     if (idx == 0) upd_f(idy, f, bit[0]);
12     else while (idx < MAXM) {
13         upd_f(idy, f, bit[idx]);
14         idx += idx&(-idx);
15     }
16 }
17
18 int bi_get_f(int idx, int idy, int **bit) {
19     int f = get_f(idy, bit[idx]);
20     if (idx > 0) {
21         int p = idx&(idx-1); idx--;
22         while (p != idx) {
23             f -= get_f(idy, bit[idx]);
24             idx &= idx-1;
25         }
26     }
27     return f;
28 }
```

## 1.3. RMQ

### 1.3.1. Binary Search Tree

Funciones para armar el *binary search tree* de un array. *rmq_init* inicializa el árbol binario la primera vez que va a usarse (nótese que debe darse $s < e$); *rmq_query* devuelve el mínimo en el rango $[a, b)$; *rmq_update* actualiza el árbol binario luego de que se haya modificado la posición $p$ del array; $rmq\_find$ encuentra el primer elemento en el rango $[a, b)$ que es más chico que el valor $v$. La inicialización es en tiempo $\mathcal{O}(N)$, y todas las demás operaciones son en tiempo $\mathcal{O}(logN)$. Todas las funciones deben ser llamadas con los argumentos $n = 1$, $[s, e) = [0, N)$, donde $N$ es el tamaño del array, siendo además $m$ y *rmq* el array y su árbol binario de búsqueda respectivamente. Para encontrar el máximo en lugar del mínimo, hay que modificar la función de comparación *compare*. El array en el que se va a guardar el árbol binario debe tener tamaño $2 * MAXN$, donde $MAXN$ es el tamaño del array original.

```
1  #define LEFT(n) ( 2*(n) )
2  #define RIGHT(n) ( 2*(n)+1 )
3
4  bool compare(int a, int b) {
5      return a < b;
6  }
7
8  void rmq_init(int n, int s, int e, int *m, int *rmq) {
9      if (s+1 == e) rmq[n] = s;
10     else {
11         rmq_init(LEFT(n), s, (s+e)/2, m, rmq);
12         rmq_init(RIGHT(n), (s+e)/2, e, m, rmq);
13         if (compare(m[rmq[LEFT(n)]], m[rmq[RIGHT(n)]]) == true) rmq[n
               ] = rmq[LEFT(n)];
14         else rmq[n] = rmq[RIGHT(n)];
15     }
16 }
17
18 int rmq_query(int n, int s, int e, int *m, int *rmq, int a, int b)
       {
19     if (a >= e || b <= s) return -1;
20     else if (s >= a && e <= b) return rmq[n];
21     else {
22         int l = rmq_query(LEFT(n), s, (s+e)/2, m, rmq, a, b);
23         int r = rmq_query(RIGHT(n), (s+e)/2, e, m, rmq, a, b);
24         if (l == -1) return r;
25         else if (r == -1) return l;
26         else if (compare(m[l], m[r]) == true) return l;
27         else return r;
28     }
29 }
30
31 void rmq_update(int n, int s, int e, int *m, int *rmq, int p) {
32     if (s+1 == e) rmq[n] = s;
33     else {
34         if (p < (s+e)/2) rmq_update(LEFT(n), s, (s+e)/2, m, rmq, p);
35         else rmq_update(RIGHT(n), (s+e)/2, e, m, rmq, p);
36         if (compare(m[rmq[LEFT(n)]], m[rmq[RIGHT(n)]]) == true ) rmq[
               n] = rmq[LEFT(n)];
37         else rmq[n] = rmq[RIGHT(n)];
38     }
39 }
40
41 int rmq_find(int n, int s, int e, int *m, int *rmq, int a, int b,
       int v) {
42     if (a >= e || b <= s) return -1;
43     else if (s >= a && e <= b) {
44         if (compare(m[rmq[n]], v) == false) return -1;
45         else if (s+1 == e) return rmq[n];
```

```
46      }
47
48      int l = rmq_find(LEFT(n), s, (s+e)/2, m, rmq, a, b, v);
49      if (l == -1) return rmq_find(RIGHT(n), (s+e)/2, e, m, rmq, a, b,
            v);
50      else return l;
51  }
```

### 1.3.2. Sparse Table

Funciones para armar la *sparse table* de un array, que permite encontrar la posición del mínimo en un rango determinado en tiempo $\mathcal{O}(1)$. La función *st_init* inicializa la *sparse table*, mientras que *st_query* devuelve la posición del mínimo en el rango $[s, e)$. Las dos funciones toman como argumentos el array $m$, su tamaño $N$, y la *sparse table* $st[LOGMAXN][MAXN]$. Para obtener el máximo en lugar del mínimo hay que invertir las comparaciones de las líneas 7 y 15.

```
1  void st_init(int *m, int N, int **st) {
2      int i, j;
3
4      for (i=0; i<N; i++) st[0][i] = i;
5      for (j=1; (1<<j)<=N; j++) {
6          for (i=0; i+(1<<j)<=N; i++) {
7              if (m[st[j-1][i]] < m[st[j-1][i+(1<<(j-1))]]) st[j][i] =
                   st[j-1][i];
8              else st[j][i] = st[j-1][i+(1<<(j-1))];
9          }
10     }
11 }
12
13 int st_query(int *m, int N, int **st, int s, int e) {
14     int k = 31 - __builtin_clz(e-s);
15     if (m[st[k][s]] < m[st[k][e-(1<<k)]]) return st[k][s];
16     else return st[k][e-(1<<k)];
17 }
```

## 1.4. Sliding window minimum

$swm(int\, n[], int\, m[], int\, N, int\, K)$ genera la lista de mínimos en $n[0, \ldots, N-1]$ cuando se la recorre con una ventana de tamaño $K$. Los mínimos se guardan en $m[0, \ldots, N-1]$, donde $m[i]$ contiene el mínimo de la ventana que termina con el elemento $n[i]$ incluído, de modo que el mínimo de la primera ventana completa, $\{n[0], \ldots, n[K-1]\}$, está guadado en $m[K-1]$. El algoritmo requiere $\mathcal{O}(N)$ tiempo y $\mathcal{O}(K)$ memoria.

```
1 #define MAXN 1048576
2 #define MAXK 1024
3
4 int s[MAXK];
5
6 void swm(int n[], int m[], int N, int K) {
7     int i, S, E;
8
9     s[0] = 0; S = 0; E = 1; m[0] = n[0];
10    for (i=1; i<N; i++) {
11        if (s[S] == i-K) {
12            S++;
13            if (S == MAXK) S = 0;
14        }
15        while (S != E && n[s[E-1]] >= n[i]) {
16            E--;
17            if (E == -1) E = MAXK;
18        }
19        s[E++] = i;
20        if (E == MAXK) E = 0;
21        m[i] = n[s[S]];
22    }
23 }
```

## 1.5. Suffix array

Construcción en $\mathcal{O}\left(Nlog^2N\right)$ del suffix array y de los longest common prefixes de una cadena. El tamaño máximo de la cadena es $MAXN$, y la cantidad máxima de caracteres del alfabeto es $ALF$. Debe definirse la variable global $N$ como el tamaño de la cadena cuyo suffix array se desea construir, y debe transformarse la cadena original a un array de int's en el rango $[0, ALF)$. Los longest common prefixes calculados son entre cada elemento y el elemento anterior. [REQUIERE: *cstdlib*, *cstring*, ]

```
1 #define MAXN 131072
2 #define ALF 64
3
4 int DIST, prm[MAXN], N;
5
6 int compare(const void *a, const void *b) {
7     int da=*(int*)a + DIST, db=*(int*)b + DIST;
8     if (da<N && db<N) return prm[da]- prm[db];
9     else if (da<N) return 1;
10    else if (db<N) return -1;
11    else return db-da;
12 }
13
14 void build_sufarray(int a[], int m[]) {
15     int i, j, c[ALF], p[ALF], g[MAXN], gg[MAXN], s, e;
16
17     memset(c, 0, sizeof(c));
18     for (i=0; i<N; i++) c[a[i]]++;
```

```
19
20      p[0] = 0;
21      for (i=1; i<ALF; i++) p[i] = p[i-1]+c[i-1];
22
23      memcpy(c, p, sizeof(p));
24      for (i=0; i<N; i++) {
25         m[p[a[i]]] = i;
26         prm[i] = p[a[i]];
27         g[p[a[i]]] = c[a[i]];
28         p[a[i]]++;
29      }
30      g[N] = N;
31
32      for (i=1; i<N; i<<=1) {
33         s = e = 0;
34         while (e <= N) {
35            if (g[s] == g[e]) e++;
36            else {
37               DIST = i;
38               qsort(m+s, e-s, sizeof(int), compare);
39               gg[s] = s;
40               for (j=s+1; j<e; j++) {
41                  if (m[j]+i < N && m[j-1]+i < N && g[prm[m[j]+i]] ==
                        g[prm[m[j-1]+i]]) {
42                     gg[j] = gg[j-1];
43                  } else gg[j] = j;
44               }
45               s = e;
46            }
47         }
48         memcpy(g, gg, sizeof(gg));
49         for (j=0; j<N; j++) prm[m[j]] = j;
50      }
51  // for (j=0; j<N; j++) printf("%d\n", m[j]);
52  }
53
54  void build_lcp(int a[], int m[], int lcp[]) {
55      int i, j, h;
56
57      h = lcp[0] = 0;
58      for (i=0; i<N; i++) {
59         if (prm[i] > 0) {
60            j = m[prm[i]-1];
61            while (i+h < N && j+h < N && a[i+h] == a[j+h]) h++;
62            lcp[prm[i]] = h;
63            if (h > 0) h--;
64         }
65      }
66  // for (i=1; i<N; i++) printf("entre %d y %d hay %d\n", m[i-1], m[i
        ], lcp[i]);
67  }
```

## 2.1. GCD

```
int gcd(int a, int b) { return (b==0) ? a : gcd(b,a%b); }
```

## 2.2. Extended GCD e Inverso mod $m$

Dado un par $a, b$, calcula los coef $n, m$ de modo que $ma + nb = gcd(a, b)$; $inv(n, m)$ es el inverso de $n$ modulo $m$ ($(n, m) = 1$)

```
typedef pair<tint,tint> ptt;
ptt egcd(tint a, tint b) {
    if (b == 0) return make_pair(1, 0);
    else {
      ptt RES = egcd(b, a%b);
        return make_pair(RES.second, RES.first - RES.second*(a/b));
    }
}
tint inv(tint n, tint m) {
   ptt EGCD = egcd(n, m);
   return ( (EGCD.first %m) + m)%m;
}
```

## 2.3. ModExp

Cálculo de $a^b \pmod n$ en $\mathcal{O}(\log b)$.

```
int modexp(int a, int b, int n) {
   int r = 1;
    while (b != 0) {
       if (b&1 == 1) r = (r*a)%n;
      b >>= 1;
      a = (a*a)%n;
   }
   return r;
}
```

## 2.4. Chinese Reminder Theorem

Dados $n$ modulos coprimos 2 a 2, y correspondientes restos, da la unica solucion modulo MOD al sistema de congruencias $X \equiv r_i \pmod{modulo_i}$ ($MOD = \prod modulo_i$) Usa: tint, MAXN, forn, inv (con egcd)

```
#define MAXN 20
tint MOD, modulo[MAXN], resto[MAXN];
void init_MOD(int n) { MOD = 1; forn(i,n) MOD*= modulo[i];  }
tint crt(int n) {
   init_MOD(n);
   tint res = 0;
   forn(i,n) {
      tint coef = 1;
      forn(j,n) if (i!=j) coef*= modulo[j];
      coef*= inv(coef,modulo[i]);
      res+= (coef*resto[i]) %MOD;
   }
   return res %MOD;
}
```

## 2.5. Find root

Encuentra la raíz de la función $func$ en el intervalo $(s, e)$, haciendo búsqueda binaria. [REQUIERE: *EPS*]

```
double findroot(double func(double x),double s,double e){
   while (e-s > EPS) {
      double mid = (s+e)/2.0;
      if (func(mid)*func(s) > 0.0) s = mid;
      else e = mid;
   }
   return (s+e)/2.0;
}
```

## 2.6. Máscaras de bits

Para recorrer todos los subconjuntos del conjunto $s$ hacemos

```
for (int mask = s; mask != 0; mask = (mask-1) & s) {
   /* CODIGO */
}
```

Para recorrer todos los subconjuntos de *fixed* que no tienen ningún elemento de *pro* se hace

```
int perm = (((1<<N)-1) & ~pro);
int mask = fixed;
while( (mask | pro) != ((1<<N)-1)){
   /* CODIGO */
   mask = ((mask + pro + 1) & perm) | fixed;
}
```

O bien

```
int rest = (1<<N)-1)^pro^fixed;
for (int mask2 = rest; mask2 != 0; mask2 = (mask2-1)&rest) {
  int mask = mask2 | fixed;
  /* CODIGO */
}
```

Todas las funciones de esta sección toman como argumento una matriz en la forma de una lista de punteros a las filas de la matriz. Entonces, si tenemos una matriz **int** $A[N][N]$, para pasarla como argumento a una funcion debemos definir una **int** ***$AA[N]$ de modo tal que $AA[i] = A[i]$ para $i = 0, 1, \ldots, N-1$.

## 3.1. Operaciones elementales

### Suma de matrices

Calcula $C = A + B$ en $\mathcal{O}\left(N^2\right)$.

```
1  void sum(int **A, int **B, int **C, int N) {
2      for (int i=0; i<N; i++) {
3          for (int j=0; j<N; j++) {
4              C[i][j] = A[i][j] + B[i][j];
5          }
6      }
7  }
```

### Producto de matrices

Calcula $C = A \cdot B$ en $\mathcal{O}\left(N^3\right)$.

```
1  void dot(int **A, int **B, int **C, int N) {
2      for (int i=0; i<N; i++) {
3          for (int j=0; j<N; j++) {
4              C[i][j] = 0;
5              for (int k=0; k<N; k++) {
6                  C[i][j] += A[i][k]*B[k][j];
7              }
8          }
9      }
10 }
```

### Potencia de matrices

Calcula $B = A^k$ en $\mathcal{O}\left(N^3 log\, k\right)$. Nótese que se modifica la matriz original $A[N][N]$. [REQUIERE: dot(int **A, int **B, int **C, N)]

```
1  void pow(int **A, int **B, int k, int N) {
2      int i, j, C[N][N], *CC[N];
3
4      for (i=0; i<N; i++) {
5          CC[i] = C[i];
6          for (j=0; j<N; j++) B[i][j] = (i==j);
7      }
8
9      while (k > 0) {
10         if ((k&1) == 1) {
11             dot(A, B, CC, N);
12             for (i=0; i<N; i++) for (j=0; j<N; j++)
13                 B[i][j] = C[i][j];
14         }
15         dot(A, A, CC, N);
16         for (i=0; i<N; i++) for (j=0; j<N; j++)
17             A[i][j] = C[i][j];
18         k >>= 1;
19     }
20 }
```

## 3.2. Gauss-Jordan (con pivoteo)

Solución del sistema lineal $C \cdot \vec{x} = \vec{b}$. La matriz $A$ es de $M \times N$, y se define como $A = C|\vec{b}$. Si Se quiere resolver más de una ecuación con una única matriz $C$ y distintos términos independientes $\vec{b}_i$, se define $A = C|B$, donde $B$ es la matriz cuyas columnas son los vectores $\vec{b}_i$. Para encontrar $A^{-1}$, se utiliza $A = C|I_{M\times M}$. La solución del sistema al terminar el algoritmo queda donde antes estaban los términos independientes. El algoritmo es $\mathcal{O}\left(M^3\right)$, y hay que definir el tipo *tint* adecuadamente (generalmente con ***typedef double*** *tint*). [REQUIERE: ABS(n), SWAPD(a,b,buf)]

```
1  void gaussjordan(tint **A, int M, int N) {
2    int i, j, k, maxi; tint tmp;
3
4    for (i=0; i<M; i++) {
5      maxi = i;
6      for (k=i+1; k<M; k++) {
7        if (ABS(A[k][i]) > ABS(A[maxi][i])) maxi = k;
8      }
9
10     for (j=0; j<N; j++) SWAPD(A[i][j], A[maxi][j], tmp);
11
12     for (j=0; j<N; j++) {
13       if (j != i) A[i][j] /= A[i][i];
14     }
15     A[i][i] = 1;
16
17     for (k=0; k<M; k++) {
18       if (k != i) {
19         for (j=i+1; j<N; j++)
20           A[k][j] -= A[k][i]*A[i][j];
21         A[k][i] = 0;
22       }
23     }
24   }
25 }
26
27 ---------------------------
```

```
28
29 bool invert(double **A, double **B, int N) {
30    int i, j, k, jmax;
31    double tmp;
32
33    for (i=1; i<=N; i++) {
34       //Encontrar el maximo elemento de A en la columna i con fila
             >= i
35       jmax = i;
36       for (j=i+1; j<=N; j++) {
37          if (ABS(A[j][i]) > ABS(A[jmax][i])) jmax = j;
38       }
39
40       //Intercambiar las filas i y jmax
41       for (j=1; j<=N; j++) {
42          swap(A[i][j], A[jmax][j]);
43          swap(B[i][j], B[jmax][j]);
44       }
45
46       //Controlar que la matriz sea invertible
47       if (A[i][i] == 0.0) return false;
48
49       //Normalizar la fila i
50       tmp = A[i][i];
51       for (j=1; j<=N; j++) {
52          A[i][j] /= tmp;
53          B[i][j] /= tmp;
54       }
55
56       //Eliminar los valores no nulos de la columna i
57       for (j=1; j<=N; j++) {
58          if (i == j) continue;
59
60          tmp = A[j][i];
61          for (k=1; k<=N; k++) {
62             A[j][k] -= A[i][k]*tmp;
63             B[j][k] -= B[i][k]*tmp;
64          }
65       }
66    }
67    return true;
68 }
```

## 4.1. Prim

Cálculo del peso del *minimum spanning tree* de un grafo no dirigido en $\mathcal{O}(V^2 + E)$. Los $N$ nodos están en el array global $V[N]$, y tienen los atributos $v$ (que indica si el nodo ya está en el árbol), $c$ (el vector de nodos adyacentes) y $ind$ (la posición de la arista que debe ser considerada cuando vuelva a analizarse el nodo). El vector $c$ de cada nodo debe ser inicializado con los nodos adyacentes a él en orden creciente del valor de las aristas. El valor de la arista en sí se encuentra en la matriz $A[N][N]$. Si el grafo no es conexo, el algoritmo calcula sólo el peso del *minimum spanning tree* de la componente conexa del nodo $R$. [REQUIERE: *vector*]

```
int prim(int R, int N) {
    int i, j, BEST, MIN, MST;

    for (i=0; i<N; i++) {
        n[i].v = false;
        n[i].ind = 0;
    }
    n[R].v = true;

    MST = 0;
    for (j=0; j<N-1; j++) {
        BEST = -1;
        for (i=0; i<N; i++) {
            while (n[i].v==true && n[i].ind < n[i].c.size() && n[n[i].
                c[n[i].ind]].v==true) {
                n[i].ind++;
            }
            if (n[i].v==true && n[i].ind < n[i].c.size() && (BEST ==
                -1 || A[i][n[i].c[n[i].ind]] < MIN)) {
                BEST = n[i].c[n[i].ind];
                MIN = A[i][n[i].c[n[i].ind]];
            }
        }
        if (BEST != -1) {MST += MIN; n[BEST].v = true;}
        else {break;}
    }
    return MST;
}
```

## 4.2. Kruskal

Cálculo del peso del *minimum spanning tree* de una grafo no dirigido en $\mathcal{O}(E \log V)$. Los $N$ nodos están en el array global $n[N]$, y las $E$ aristas están ordenadas por pesos crecientes en el array global $e[E]$. Usar **void** *join_rank*(**int** $n$) para la máxima optimización, y **void** *join*(**int** $n$) si no es necesario. Los nodos tienen los atributos $f$ (el padre) y $r$ (el rango, necesario si se usa *join_rank*), mientras que las aristas tienen los atributos $w$ (el peso) y $a$ y $b$ (los extremos). Si el grafo no es conexo se obtiene el peso del *minimum spanning forest*, es decir del *minimum spanning tree* de cada componente conexa.

```
int getfather(int n) {
    if (V[n].f != n) V[n].f = getfather(V[n].f);
    return V[n].f;
}
void join(int a, int b) {V[getfather(a)].f = getfather(b);}
void join_rank(int a, int b) {
    int fa = getfather(a), fb = getfather(b);
    if (V[fa].r > V[fb].r) {V[fb].f = fa;}
    else if (V[fa].r < V[fb].r) {V[fa].f = fb;}
    else {V[fa].f = fb; V[fb].r++;}
}
int kruskal(int N, int E) {
     int i, MST;
    for (i=0; i<N; i++) { V[i].f = i, V[i].r = 0;}
    MST = 0;
    for (i=0; i<E; i++) if (getfather(e[i].a) != getfather(e[i].b))
        {
        join_rank(e[i].a, e[i].b);
       MST += e[i].w;
    }
    return MST;
}
```

## 4.3. Dijkstra

Cálculo de la mínima distancia desde $s$ hasta todos los demás nodos en un grafo con aristas de peso no negativo. El array de vectores $adj[i]$ contiene pares cuya primera componente es un nodo adyacente al nodo $i$, y cuya segunda componente es el peso de la arista que va de $i$ a ese nodo. Al terminar el algoritmo, $mindist[i]$ contiene la mínima distancia del nodo $s$ al nodo $i$. [REQUIERE: *queue*, *vector*, *forn(i,n)*, *decl(v,c)*, *forall(i,c)*, *mp*]

```
typedef pair<int,int> pii;
typedef priority_queue<pii> heap;

const int INF = 1<<29;
const int MAXN = 10000;

heap q;
bool vis[MAXN];
int mindist[MAXN];

void init() {
    while (!q.empty()) q.pop();
    forn(u,MAXN) {
        vis[u] = false;
```

```
15          mindist[u] = INF;
16      }
17  }
18
19  void addNode(int u, int d) {
20      if (mindist[u] <= d) return;
21      mindist[u] = d;
22      q.push(mp(-d,u));
23  }
24
25  void dijkstra(int s, vector<pii> adj[]) {
26      init();
27      addNode(s,0);
28      while (!q.empty()) {
29          pii t = q.top(); q.pop();
30          int u = t.second, d = -t.first;
31          if (vis[u]) continue;
32          vis[u] = true;
33          forall(it,adj[u]) {
34              int v = it->first, cost = it->second;
35              addNode(v,d + cost);
36          }
37      }
38  }
```

---

Igual al algoritmo de arriba, solo que usando *heap* para correr en $\mathcal{O}(V \log V + E)$. [REQUIERE: *heap*, $SWAP(a,b)$]

```
1  #define MAXN 100000
2  #define INF -1
3
4  struct edge {
5      int dest, w;
6  };
7
8  struct node {
9      int inh, d, prev;
10     vector<edge> c;
11 } n[MAXN];
12
13 #define SWAPH(a,b) ( SWAP(n[A[(a)]].inh,n[A[(b)]].inh), SWAP(A[(a)
       ],A[(b)]) )
14
15 bool compare(int a, int b) {return n[a].d < n[b].d;}
16
17 int AA[MAXN];
18
19 void dijkstra(int S, int D, int N) {
20     int i, cur, dest, it, *A;
21     A = AA;
22
23     for (i=0; i<N; i++) {
24         n[i].d = INF;
25         n[i].inh = -1;
26         n[i].prev = -1;
27     }
28     n[S].d = 0;
29
30     A[0] = 1; A[1] = S; n[S].inh = 1;
31     while (A[0] > 0) {
32         cur = pop(A, compare);
33         n[cur].inh = -1;
34         if (cur == D) break;
35         for (it=0; it<n[cur].c.size(); it++) {
36             dest = n[cur].c[it].dest;
37             if (n[dest].d==INF || n[dest].d>n[cur].d+n[cur].c[it].w){
38                 n[dest].d = n[cur].d + n[cur].c[it].w;
39                 n[dest].prev = cur;
40                 if (n[dest].inh == -1) {
41                     A[A[0]+1] = dest;
42                     n[dest].inh = A[0]+1;
43                     insert(A, compare);
44                 } else update(A, n[dest].inh, compare);
45             }
46         }
47     }
48 }
```

## 4.4. Bellman-Ford

```
1  void bellman_ford(int V, edge e[], int E) {
2    for (int i=0; i<V-1; i++) {
3      for (int j=0; j<E; j++) {
4        if (n[e[j].s].d != INF && n[e[j].e].d > n[e[j].s].d + e[j].w)
             {
5          n[e[j].e].d = n[e[j].s].d + e[j].w;
6        }
7      }
8    }
9  }
```

## 4.5. Floyd - Warshall

Camino minimo de todos a todos en $\mathcal{O}(n^3)$.

```
1  forn(k,n) forn(i,n) forn(j,n) A[i][j]<?=A[i][k]+A[k][j];
```

## 4.6. Edmonds-Karp: flujo máximo en $\mathcal{O}((N+E)F)$ ó $\mathcal{O}(NE^2)$

Calcula el flujo máximo entre el nodo *SOURCE* y el nodo *SINK* de un grafo no dirigido en el que la capacidad de la arista $(i, j)$ se encuentra en $A[i][j]$. Los nodos deben estar en el array $n[N]$, y deben tener los parámetros **int** *best* y **int** *prev* además de un vector de adyacencias *vector* < **int** > *c*. Hay que definir *INF* como un valor mayor o igual que el flujo máximo. [REQUIERE *vector*, MIN(a,b)]

```
int edmondskarp(int **A, int SOURCE, int SINK) {
    int s[MAXN], S, E, F[MAXN][MAXN], FLOW, tmp, it;
    bool v[MAXN];

    memset(F, 0, sizeof(F));
    FLOW = 0;
    while (1) {
        memset(v, false, sizeof(v));
        n[SOURCE].best = INF;
        n[SOURCE].prev = -1;
        s[0] = SOURCE; S = 0; E = 1; v[SOURCE] = true;
        while (S != E && s[S] != SINK) {
            for (it=0; it<n[s[S]].c.size(); it++) {
                tmp = n[s[S]].c[it];
                if (v[tmp]==false && A[s[S]][tmp]-F[s[S]][tmp]>0) {
                    v[tmp] = true;
                    n[tmp].prev = s[S];
                    n[tmp].best = MIN(n[s[S]].best, A[s[S]][tmp] - F[s[S
                        ]][tmp]);
                    s[E] = tmp;
                    E++;
                }
            }
            S++;
        }
        if (S==E) return FLOW;
        else {
            FLOW += n[SINK].best;
            tmp = SINK;
            while (n[tmp].prev != -1) {
                F[n[tmp].prev][tmp] += n[SINK].best;
                F[tmp][n[tmp].prev] = - F[n[tmp].prev][tmp];
                tmp = n[tmp].prev;
            }
        }
    }
}
```

## 4.7. Relabel to front: flujo máximo en $\mathcal{O}(N^3)$

```
usa: algorithm, list, forn
#define MAXN 200
typedef list<tint> lint;
typedef lint::iterator lintIt;
//usadas para el flujo
tint f[MAXN][MAXN]; //flujo
tint e[MAXN]; //exceso
tint h[MAXN]; //altura
lintIt cur[MAXN];
//esto representa el grafo que hay que armar
lint ady[MAXN]; //lista de adyacencias (para los dos lados)
tint c[MAXN][MAXN]; //capacidad (para los dos lados)
tint n; //cant de nodos
tint cf(tint i, tint j) { return c[i][j] - f[i][j]; }
void push(tint i, tint j) {
tint p = min(e[i], cf(i,j));
f[j][i] = -(f[i][j] += p);
e[i] -= p;
e[j] += p;
}
void lift(tint i) {
tint hMin = n*n;
for(lintIt it = ady[i].begin() ; it != ady[i].end() ; ++it) {
if (cf(i, *it) > 0) hMin = min(hMin, h[*it]);
}
h[i] = hMin + 1;
}
void iniF(tint desde)
{
forn(i,n) {
h[i] = e[i] = 0;
forn(j,n) f[i][j] = 0;
cur[i] = ady[i].begin();
}
h[desde] = n;
for(lintIt it = ady[desde].begin() ; it != ady[desde].end() ; ++it)
{
f[*it][desde] = -(f[desde][*it] = e[*it] = c[desde][*it]);
}
}
void disch(tint i) {
while(e[i] > 0) {
lintIt& it = cur[i];
if (it == ady[i].end()) {lift(i); it = ady[i].begin();}
else if (cf(i,*it) > 0 && h[i] == h[*it] + 1) push(i,*it);
else ++it;
}
}
tint calcF(tint desde, tint hasta) {
iniF(desde);
```

```
lint l;
forn(i,n) {if (i != desde && i != hasta) l.push_back(i);}
for(lintIt it = l.begin() ; it != l.end() ; ++it) {
tint antH = h[*it];
disch(*it);
if (h[*it] > antH) { //move to front
l.push_front(*it);
l.erase(it);
it = l.begin();
}
} return e[hasta];
}
void addEje(tint a, tint b, tint ca) {
//requiere reiniciar las capacidades
if (c[a][b] == 0) {//soporta muchos ejes mismo par de nodos
ady[a].push_back(b);
ady[b].push_back(a);
}
c[b][a] = c[a][b] += ca;
}
void iniGrafo(tint nn) { //requiere n ya leido
n=nn;
forn(i,n) {
forn(j,n) c[i][j] = 0;
//solo si se usa la version de addeje con soporte multieje
ady[i].clear();
}
}
```

## 4.8. Dinic: Flujo maximo en $\mathcal{O}(E.V^2)$

Generalmente es mas rapido que preflow y edmonds-karp; para redes con capacidades 1 corre en $\mathcal{O}(E\sqrt{(V)})$ Usa tint, MAXN, vi, forn, si, pb, queue, vector, algorithm

```
/* Dinic
Flujo maximo en O(E.V^2); generalmente es mas rapido que preflow y
    edmonds-karp;
para redes con capacidades 1 corre en O(E.sqrt(V))
Usa tint, MAXN, vi, forn, si, pb, queue, vector, algorithm
*/

typedef tint tipo;
const int INF = 1<<29;
struct edge {
   tipo c,f;
   tipo r() { return c-f; }
};
int N,S,T;
edge red[MAXN][MAXN];
vi adjG[MAXN];

void reset() {
   forn(u,N) forn(i,si(adjG[u])) {
      int v = adjG[u][i];
      red[u][v].f = 0;
   }
}
void initGraph(int n, int s, int t) {
   N = n; S = s; T = t;
   forn(u,N) {
      adjG[u].clear();
      forn(v,N) red[u][v] = (edge){0,0};
   }
}
void addEdge(int u, int v, int c) {
   if (!red[u][v].c && !red[v][u].c) { adjG[u].pb(v); adjG[v].pb(u)
       ; }
   red[u][v].c += c;
}

int dist[MAXN];
bool dinic_bfs() {
   forn(u,N) dist[u] = INF;
   queue<int> q; q.push(S); dist[S] = 0;
   while (!q.empty()) {
      int u = q.front(); q.pop();
      forn(i,si(adjG[u])) {
         int v = adjG[u][i];
         if (dist[v] < INF || red[u][v].r() == 0) continue;
         dist[v] = dist[u] + 1;
         q.push(v);
      }
   }
   return dist[T] < INF;
}
tipo dinic_dfs(int u, tipo cap) {
   if (u == T) return cap;
   tipo res = 0;
   forn(i,si(adjG[u])) {
      int v = adjG[u][i];
      if (red[u][v].r() && dist[v] == dist[u] + 1) {
         tipo send = dinic_dfs(v,min(cap,red[u][v].r()));
         red[u][v].f += send; red[v][u].f -= send;
         res += send; cap -= send;
         if (cap == 0) break;
      }
   }
   if (res == 0) dist[u] = INF;
   return res;
}
tipo dinic() {
   tipo res = 0; while (dinic_bfs()) res += dinic_dfs(S,INF);
       return res;
}
```

## 4.9. Karger: min-cut randomizado

Encuentra con alta probabilidad un min-cut del grafo con nodos $[0, \ldots, N)$ y aristas $e[0, \ldots, E)$. Debe definirse $MAX$ como el máximo número de iteraciones del algoritmo básico, y entonces el algoritmo es $\mathcal{O}((N+E) * MAX)$. [REQUIERE: *cstdlib*, *ctime*]

```
int f[MAXN];

struct edge {
    int a, b, w;
} e[MAXE];

int getf(int n) {
    if (f[n] != n) f[n] = getf(f[n]);
    return f[n];
}

int mincut(int N, int E) {
    int i, j, tmp, tmp1, tmp2, RES, EE, NN;
    RES = INF;
    for (i=0; i<MAX; i++) {
        for (j=0; j<N; j++) f[j] = j;

        EE = E; NN = N;
        while (EE > 0 && NN > 2) {
            tmp = rand()%EE;
            tmp1 = getf(e[tmp].a);
            tmp2 = getf(e[tmp].b);
            if (tmp1 != tmp2) {
                f[tmp1] = tmp2;
                NN--;
            }
            swap(e[tmp], e[EE-1]);
            EE--;
        }

        tmp = 0;
        for (j=0; j<EE; j++) if (getf(e[j].a) != getf(e[j].b)) tmp +=
             e[j].w;
        if (tmp < RES) RES = tmp;
    }
    return RES;
}
```

## 4.10. Flujo de costo mínimo en $\mathcal{O}\left(B\,N^3\right)$

Cálculo del flujo de costo mínimo en un grafo. No es seguro que ande si están presentes las aristas $(i,j)$ y $(j,i)$ con $i \neq j$. Corre en $\mathcal{O}\left(B\,N^3\right)$ donde $B$ es una cota para las capacidades. NO FUNCIONA si hay ciclos de costo negativo. [REQUIERE: *algorithm*, *forn*]

```
#define forn(i,n) for (int i=0; i<(n); ++i)
#define MAXN 120
const int INF = 1<<30;
struct Eje{
        int f,m,p,rp; //f flujo, m capacidad, p costo
        int d(){return m-f;}
};

Eje red[MAXN][MAXN];
int adyc[MAXN], ady[MAXN][MAXN];
int N,F,D;
void iniG(int n, int f, int d) {
     N=n; F=f; D=d;
      fill(red[0], red[N], (Eje){0,0,0,0});
      fill(adyc, adyc+N,0);
}
void aEje (int d, int h, int m, int p) {
      red[h][d].p = -(red[d][h].p = p);
      red[h][d].rp = -(red[d][h].rp = p); // reduced cost
      red[d][h].m = m;
      ady[d][adyc[d]++] = h; ady[h][adyc[h]++] = d;
}
int md[MAXN], vd[MAXN], used[MAXN];
void reduceCost() {
    forn(i,N) forn(j,N) if (red[i][j].d()>0) {
        red[i][j].rp += md[i]-md[j];
        red[j][i].rp = 0;
    }
}
int camAu(int &v) {
    fill (vd,vd+N,-1);
    fill (used,used+N,false);
    vd[F]=F; md[F]=0;
    int i = F, next = F, cant = 0;
    while (cant<N) {
       used[i]=true;
       forn(jj,adyc[i]) {
          int j= ady[i][jj], nd= md[i]+red[i][j].rp;
          if (red[i][j].d()>0) if (vd[j]==-1 || md[j]>nd)
             md[j]=nd,vd[j]=i;
       }
       int mn=INF;
       forn(k,N) if (!used[k] && vd[k]!= -1 && md[k] < mn)
          next=k,mn=md[k];
       cant++;
       if (next==i) break;
       i=next;
    }
    v=0;
    if (vd[D]==-1) return 0;
    reduceCost();
```

```
52      int f=red[vd[D]][D].d();
53      for (int n=D;n!=F;n=vd[n]) f <?= red[vd[n]][n].d();
54      for (int n=D;n!=F;n=vd[n]) {
55         red[n][vd[n]].f = -(red[vd[n]][n].f+=f);
56         v += red[vd[n]][n].p * f;
57      }
58      return f;
59  }
60
61  int flujo(int &r) {
62       fill (vd,vd+N,-1);
63      vd[F]=F; md[F]=0;
64       bool cambios = true;
65       for (int rep=0; rep<N && cambios; rep++) {
66           cambios = false;
67           forn(i,N) if (vd[i]!=-1) forn(jj,adyc[i]) {
68               int j = ady[i][jj], nd = md[i]+red[i][j].rp;
69               if (red[i][j].d()>0) if (vd[j]==-1||md[j]>nd)
70                  md[j]=nd,vd[j]=i,cambios=true;
71           }
72       }
73       reduceCost();
74      r=0; int v,f=0, c;      // r = minCost, f = maxFlow
75      while ((c=camAu(v))) r+=v, f+=c;
76      return f;
77  }
```

## 4.11. Maximum bipartite matching

[REQUIERE: *vector*, *forn(i,n)*, *decl(v,c)*, *forall(i,c)*, *pb*]

```
1  #define MAXN 102
2  typedef vector<int> vi;
3
4  vi next[MAXN];
5  int M, pre[MAXN];
6  bool vis[MAXN];
7
8  void initGraph(int M, int N) {
9      ::M = M;
10     forn(i,M) next[i].clear();
11     forn(i,N) pre[i] = -1;
12 }
13
14 void addEdge(int u, int v) {
15     next[u].pb(v);
16 }
17
18 bool augment(int u) {
19     if (u == -1) return true;
20     if (vis[u]) return false;
21     vis[u] = true;
22     forall(it,next[u]) {
23        int v = *it;
24        if (augment(pre[v])) {
25           pre[v] = u;
26           return true;
27        }
28     }
29     return false;
30 }
31
32 int matching() {
33     int res = 0;
34     forn(u,M) {
35        forn(v,M) vis[v] = false;
36        res += augment(u);
37     }
38     return res;
39 }
```

## 4.12. Puntos de articulación

```
1  const int INF = 1<<29;
2  const int MAXN = 100000 + 100;
3  const int MAXE = MAXN;
4
5  int m,n;
6  vector<pii> edges;
7  vi adj[MAXN];
8
9  void addE(int u, int v) {
10     int ne = si(edges);
11     edges.pb(mp(u,v));
12     adj[u].pb(ne); adj[v].pb(ne);
13 }
14
15 stack<int> estack; //stack de edges
16 int d[MAXN], b[MAXN], t;
17 int bc[MAXE], nbc;
18
19 struct param {
20     int le, u, v, i, ne;
21     pii e;
22 };
23
24 #define save(x) prm.top().x = x
25 #define load(x) x = prm.top().x
26
27 void NR_dfs(int le, int u) {
28     stack<param> prm;
29     param init; init.le = le; init.u = u;
30     prm.push(init);
```

```
31
32      int i,v,ne; pii e;
33      for (;;) {
34          restart:;
35          le = prm.top().le; u = prm.top().u;
36
37          b[u] = d[u]= t++;
38          for (i = 0; i < si(adj[u]); i++) {
39              ne = adj[u][i];
40              if (ne == le) continue;
41              e = edges[ne];
42              v = e.first ^ e.second ^ u;
43              if (d[v] == -1) {
44                  estack.push(ne);
45
46                  save(le); save(u); save(i); save(v); save(e); save(ne);
47                  init.le = ne; init.u = v; prm.push(init);
48                  goto restart;
49                  ret:;
50                  load(le); load(u); load(i); load(v); load(e); load(ne);
51
52                  //if (b[v] > d[u]) ne es bridge
53                  if (b[v] >= d[u]) { //u es punto de articulacion
54                      int last;
55                      do{ //saca edges de la componente biconexa
56                          last = estack.top();
57                          bc[last] = nbc;
58                          estack.pop();
59                      } while (last != ne);
60                      nbc++;
61                  }
62                  else b[u] = min(b[u],b[v]);
63              }
64              else if (d[v] < d[u]) {
65                  estack.push(ne);
66                  b[u] = min(b[u],d[v]);
67              }
68          }
69          prm.pop();
70          if (prm.empty()) return;
71          else goto ret;
72      }
73  }
```

## 4.13. Circuito euleriano

[REQUIERE: *algorithm*, *vector*, *list*, *string*, *forn*]

```
1  typedef string ejeVal;
2  #define MNT ""//MENORATODOS
3  typedef pair<ejeVal, tint> eje;
4  tint n; vector<eje> ady[MAXN]; tint g[MAXN];
5  //grafo (inG = in grado o grado si es no dir)
6  tint aux[MAXN];
7  tint pinta(tint f) {
8      if (aux[f]) return 0;
9      tint r = 1; aux[f] = 1;
10     forn(i,ady[f].size()) r+=pinta(ady[f][i].second);
11     return r;
12 }
13
14 tint compCon(){
15     fill(aux, aux+n, 0);
16     tint r=0;
17     forn (i,n){
18         if (!aux[i]){ r++; pinta(r); }
19         }
20     return r;
21     }
22
23 bool isEuler(bool path, bool dir) {
24     if (compCon() > 1) return false;
25     tint c = (path ? 2 : 0);
26     forn (i,n){
27         if (!dir ? ady[i].size() %2 : g[i] != 0) {
28             if (dir && abs(g[i]) > 1) return false;
29             c--;
30             if(c<0) return false;
31             }
32         }
33 return true;
34 }
35
36 bool findCycle(tint f, tint t, list<tint>& r) {
37     if (aux[f] >= ady[f].size()) return false;
38     tint va = ady[f][aux[f]++].second;
39     r.push_back(va);
40     return (va != t ? findCycle(va, t, r) : true);
41 }
42
43 list<tint> findEuler(bool path){//dir., sin valores rep.
44   if (!isEuler(path, true)) return list<tint>();
45   bool agrego = false;
46   if (path) {
47     tint i = max_element(g, g + n)-g;
48     tint j = min_element(g, g + n)-g;
49     if (g[i]!=0) {ady[i].push_back(eje(MNT,j));agrego=1;}
50   }
51   tint x = -1;
52   forn (i,n) {
53     sort(ady[i].begin(), ady[i].end());
54     if (x<0 || ady[i][0] < ady[x][0]) x=i;
55   }
56   fill(aux, aux+n, 0);
57   list<tint> r;
58   findCycle(x,x,r);
```

```
  if (!agrego) r.push_front(r.back());
  list<tint> aux; bool find=false;
  list<tint>::iterator it = r.end();
  do{
    if (!find) --it;
    for (find=findCycle(*it,*it,aux);!aux.empty();aux.pop_front()){
        it = r.insert(++it, aux.front());
    }
    } while (it != r.begin());
return r;
}
```

## 4.14. Componentes fuertemente conexas

Busca las CFC en un grafo dirigido. Usa: vector, cstring, forn, MAXN. N = cant de vertices, ady/adyt la lista de adyacencias/lista traspuesta. Devuelve la cant de CFC's, y cc[] termina con la CFC de cada uno.

```
int vis[MAXN], cc[MAXN], CC,N, o[MAXN];
vector<int> ady[MAXN],adyt[MAXN];
// N = numero de vertices, ady lista de adyacencias, adyt la
    traspuesta
void dfs1(int i) {
   vis[i] = 2;
   forn(j,si(ady[i])) if (vis[ady[i][j]] == 0)
      dfs1(ady[i][j]);
   o[CC++] = i;
}
void dfs2(int i) {
   vis[i] = 2;
   cc[i] = CC;
   forn(j,si(adyt[i])) if (vis[adyt[i][j]] == 0)
      dfs2(adyt[i][j]);
}
int cfc() {
   CC = 0; memset(vis,0,sizeof(vis));
   forn(i,N) if (vis[i] == 0) dfs1(i);
   CC = 0; memset(vis,0,sizeof(vis));
   for(int i = N-1; i>=0; i--) if (vis[o[i]] == 0) {dfs2(o[i]); CC
       ++;}
   return CC;
}
```

## 4.15. Lowest Common Ancestor

Funciones para trabajar con los *lowest common ancestors* de los pares de nodos de un árbol. El preprocesamiento es en $\mathcal{O}(N \log N)$, y $lca(a,b)$ opera en $\mathcal{O}(\log N)$. $dist(a,b)$ calcula la distancia entre los nodos $a$ y $b$, y también corre en $\mathcal{O}(\log N)$.

```
#define MAXN 16384
#define LOGMAXN 14

int N, p[MAXN][LOGMAXN];

struct node {
   int l;
   vector<int> c;
} n[MAXN];

void dfs(int cur) {
   for (int i=0; i<(int)n[cur].c.size(); i++) {
      if (n[cur].c[i] != p[cur][0]) {
         p[n[cur].c[i]][0] = cur;
         n[n[cur].c[i]].l = n[cur].l+1;
         dfs(n[cur].c[i]);
      }
   }
}

void init_lca() {
   n[0].l = 0; p[0][0] = 0;
   dfs(0);

   for (int i=1; i<LOGMAXN; i++) {
      for (int j=0; j<N; j++) {
         p[j][i] = p[p[j][i-1]][i-1];
      }
   }
}

int lca(int a, int b) {
   if (n[a].l < n[b].l) return lca(b, a);
   else {
      int i, dl = n[a].l-n[b].l;

      for (i=0; i<LOGMAXN; i++) {
         if (((dl>>i)&1) == 1) {
            a = p[a][i];
         }
      }

      if (a == b) return a;
      else {
         for (i=LOGMAXN-1; i>=0; i--) {
            if (p[a][i] != p[b][i]) {
               a = p[a][i];
               b = p[b][i];
            }
         }
         return p[a][0];
```

```
52          }
53      }
54  }
55
56  int dist(int a, int b) {
57      return n[a].l + n[b].l - 2*n[lca(a, b)].l;
58  }
```

## 4.16. Extras y adaptaciones

- Bellman Ford

  Para detectar ciclos negativos se usa

```
1  bool neg_cycle(int V, edge e, int E;) {
2      bellman_ford(V, e, E);
3      for (int i=0; i<E; i++) {
4          if (n[e[j].s].d != INF && n[e[j].e].d > n[e[j].s].d + e[j
               ].w) return 1;
5      }
6      return 0;
7  }
```

  Para encontrar la mínima distancia de todos los nodos de un grafo a un único destino, se usa el algoritmo de Bellman-Ford con las aristas invertidas.

- Grafos bipartitos

  * A graph is bipartite if and only if it does not contain an odd cycle. Therefore, a bipartite graph cannot contain a clique of size 3 or more.

  * A graph is bipartite if and only if it is 2-colorable.

  * The size of the minimum vertex cover is equal to the size of the maximum matching (König's theorem).

  * The size of the maximum independent set plus the size of the maximum matching is equal to the number of vertices.

  * For a connected bipartite graph the size of the minimum edge cover is equal to the size of the maximum independent set.

  * For a connected bipartite graph the size of the minimum edge cover plus the size of the minimum vertex cover is equal to the number of vertices.

  * The spectrum of a graph is symmetric if and only if it's a bipartite graph.

- Aplicaciones de flujos

  * Dado un $DAG$, se puede encontrar la mínima cantidad de caminos (disjuntos?) que cubran a todos los vértices (problemas de los taxis, cajas, mamushkas, CodeJam). Para ello, se construye el grafo bipartito $(A, B)$ con todas las aristas del grafo original, donde $A$ contiene a los nodos que tengan $out - deg > 0$, y $B$ contiene a los nodos que tengan $in - deg > 0$. La solución es $n - m$, donde $n$ es el número de nodos del grafo y $m$ es el maximum matching del grafo bipartito construido.

  * Given a directed graph $G = (V, E)$ and two vertices $s$ and $t$, we are to find the maximum number of independent paths from $s$ to $t$. Two paths are said to be independent if they do not have a vertex in common apart from $s$ and $t$. We can construct a network $N = (V, E)$ from $G$ with vertex capacities, where: $s$ and $t$ are the source and the sink of $N$ respectively; $c(v) = 1$ for each $v$ in $V$; $c(e) = 8$ for each $e$ in $E$ (vale ponerle capacidad 1 a la arisa, se usa a lo sumo una vez). Then the value of the maximum flow is equal to the maximum number of independent paths from $s$ to $t$.

  * Given a directed graph $G = (V, E)$ and two vertices $s$ and $t$, we are to find the maximum number of edge-disjoint paths from $s$ to $t$. This problem can be transformed to a maximum flow problem by constructing a network $N = (V, E)$ from $G$ with $s$ and $t$ being the source and the sink of $N$ respectively and assign each edge with unit capacity.

- Otros

  * A matrix $A$ is an adjacency matrix of a $DAG$ if and only if the eigenvalues of the $(0, 1)$ matrix $A + I$ are positive, where $I$ denotes the identity matrix.

  * Every tournament (grafo proveniente de asignarle direcciones a cada una de las aristas de un grafo no dirigido completo, o sea un grafo dirigido tal que para todo $u$ y $v$ entonces o bien $(u, v) \in E$ o bien $(v, u) \in E$) has an odd number of Hamiltonian paths (se demuestra facil por inducción).

  * Teorema de Euler: En todo dibujo de un grafo planar se satisface $v - e + f = 2$, donde $v$ es el número de vértices, $e$ es el número de aristas, y $f$ es el número de regiones en que se divide el plano.

  * En un grafo conexo y planar existe un nodo que tiene grado a lo sumo 5.

## 5.1. Knuth-Morris-Pratt

[REQUIERE: *vector*, *string*, *algorithm*, *sz(a)*, *pb*, *all(c)*, *tr(c,i)*, *present(c,x)*, *cpresent(c,x)*, *forn(i,n)*, *forsn(i,m,n)*]

```
#define MAXN 1000
typedef vector<int> vi;
typedef vector<vi> vvi;
typedef pair<int,int> ii;

int F[MAXN];

int preproc(string pattern){
    F[0] = F[1] = 0;
    forsn(i, 2, sz(pattern)+1){
        int j = F[i-1];
        while(pattern[j] != pattern[i-1] && j>0) j = F[j];
        F[i] = j + ((pattern[j] == pattern[i-1])?1:0) ;
    }
    return 0;
}

int KMP(string pat, string str){
    preproc(pat);
    int m = sz(pat);
    for(int i=0, j = 0; j<sz(str); j++){
        cout << j << ' ' << i ;
        if(str[j] == pat[i]){ if(++i == m) return j-m+1; }
        else{
            if(i>0) j--;
            i = F[i];
        }
    }
    return -1;
}

int KMP2(string pat, string str){
    preproc(pat);
    int m = sz(pat);
    for(int i=0, j = 0; j<sz(str); j++){
        while(i>0 && pat[i]!=str[j]) i = F[i];
        if(str[j] == pat[i]){ if(++i == m) return j-m+1; }
    }
    return -1;
}

int main(){
    string pattern, eaea;
    cout << "escriba patron : ";
    cin >> pattern;
    cout << "escriba cadena : ";
    cin >> eaea;
    cout << KMP2(pattern, eaea) << endl;;
    forn(i, sz(pattern)+1) cout << F[i] << ' '; cout << endl;

    return 0;
}
```

## 5.2. Aho-Corasik

Busca todas las apariciones de ciertos patrones como subcadenas de cierta otra cadena. Los patrones se guardan en *pat*, y *match* devuelve un *vvi* tal que el vector $i$-ésimo guarda una lista de los índices en donde el patrón $i$ aparece dentro de la cadena mayor. Las cadenas en *pat* no pueden repetirse.

```
struct node {
   int id;  char c; bool isfinal;
   node *parent, *pre, *pfinal;
   map<char,node*> child;

   node(node* parent = NULL, char c = ' ') {
      this->c = c; this->parent = parent;
      pre = pfinal = NULL;
      id = -1; isfinal = false;
   }
   void insert(const string& s, int id = -1) {
      int n = s.size();
      node* act = this;
      forn(i,n)
         if (act->child.count(s[i])) act = act->child[s[i]];
         else act = act->child[s[i]] = new node(act,s[i]);
      act->isfinal = true; act->id = id;
   }
   void clear() {
      forall(it,child) it->second->clear();
      child.clear(); c = ' '; parent = pre = pfinal = NULL;
   }
};

vs pat;
node *root;

void precompute() {
   root = new node();
   int np = si(pat);
   forn(i,np) root->insert(pat[i], i);

   queue<node*> q; forall(it,root->child) q.push(it->second);
   while (!q.empty()) {
      node *u = q.front(); q.pop();
      node *p = u->parent->pre; char c = u->c;
      while (p && !p->child.count(c)) p = p->pre;
      if (p == NULL) u->pre = root;
      else u->pre = p->child[c];
```

```
40
41          if (u->pre->isfinal) u->pfinal = u->pre;
42          else u->pfinal = u->pre->pfinal;
43          forall(it,u->child) q.push(it->second);
44      }
45  }
46
47  vvi match(const string& s) {
48      int n = si(s), np = si(pat);
49      vvi res(np);
50      node* act = root;
51      forn(i,n) {
52          while (act && !act->child.count(s[i])) act = act->pre;
53          if (act == NULL) act = root;
54          else {
55              act = act->child[s[i]];
56              node *fin = act;
57              while (fin && fin->isfinal) {
58                  int id = fin->id;
59                  res[id].pb(i-si(pat[id])+1);
60                  fin = fin->pfinal;
61              }
62          }
63      }
64      return res;
65  }
66
67  //Para limpiar el Trie
68  if(root != NULL)(*root).clear();
```

## 6.1. Área de un polígono

Cálculo del área del polígono p[1, ..., N-1] en $\mathcal{O}(N)$. Los segmentos del polígono tienen extremos p[i] y p[i+1] para i=0, ..., N-2, y el polígono se cierra con el segmento p[N-1] a p[0]. [REQUIERE: ABS(n)]

```
double polygon_area(point p[], int N) {
    double A = 0;
    for (int i=1; i<N-1; i++) {
        A += (p[i].x - p[0].x)*(p[i+1].y - p[0].y) - (p[i+1].x - p
            [0].x)*(p[i].y - p[0].y);
    }
    return ABS(A/2);
}
```

## 6.2. Número de enrollamiento

Cálculo del número de enrollamiento de un punto $P0$ respecto del polígono $p[0, \ldots, N]$, donde los segmentos $p[i]$ a $p[i+1]$ para $i = 0, \ldots, N-1$ forman el polígono, y $p[0] = p[N]$. Devuelve

```
int winding_number(point p[], int N, point P0) {
    int i, WN;

    for (i=0; i<N; i++) {
        if ( (p[i].x - P0.x)*(p[i+1].y - P0.y) - (p[i].y - P0.y)*(p[i
            +1].x - P0.x) == 0 ) {
            if (P0.x >= MIN(p[i].x,p[i+1].x) && P0.x <= MAX(p[i].x,p[i
                +1].x) && P0.y >= MIN(p[i].y,p[i+1].y) && P0.y <= MAX(
                p[i].y,p[i+1].y)) {
                return 1;//EN EL BORDE
            }
        }
    }

    WN = 0;
    for (i=0; i<N; i++) {
        if (p[i].y <= P0.y) {
            if (p[i+1].y > P0.y) {
                if ( (p[i].x-P0.x)*(p[i+1].y-P0.y) - (p[i].y-P0.y)*(p[i
                    +1].x-P0.x) > 0 ) WN++;
            }
        } else {
            if (p[i+1].y <= P0.y) {
                if ( (p[i].x-P0.x)*(p[i+1].y-P0.y) - (p[i].y-P0.y)*(p[i
                    +1].x-P0.x) < 0 ) WN--;
            }
        }
    }
    return WN;
}
```

## 6.3. Cápsula convexa

Calculo de la cápsula convexa de un conjunto de puntos. [REQUIERE: *vector*, *algorithm*]

```
struct pto { tint x,y;} r;
tint dist2(pto a, pto b)
   {return (a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y);}
tint crossprod(tint x1, tint y1, tint x2, tint y2)
   {return (x1*y2-x2*y1);}
tint area(pto a, pto b, pto c)//area con signo
   {return crossprod(b.x-a.x,b.y-a.y,c.x-a.x,c.y-a.y);}
bool comp(const pto &a, const pto &b)//lowerleft
   {return (a.y==b.y) ? (a.x<b.x) : (a.y<b.y);}
bool operator<(const pto &a, const pto &b) {
    tint tmp = area(r,a,b);
    if (tmp == 0) return (dist2(a,r) < dist2(b,r));
    else return (tmp > 0);
}//counterwise (para clockwise poner tmp < 0)
vector<pto> convex_hull(vector<pto> puntos) {
    if (puntos.size() < 3) return puntos;
    r =* (min_element(puntos.begin(),puntos.end(),comp));
    sort(puntos.begin(),puntos.end());//usa operator <
    int i = 0;
    vector<pto> ch;
    ch.push_back(puntos[i++]);
    ch.push_back(puntos[i++]);
    while (i < puntos.size()) {
        if (ch.size()>1 && area(ch[ch.size()-2],ch[ch.size()-1],
            puntos[i])<0) ch.pop_back();
        else ch.push_back(puntos[i++]);
    }
    return ch;
}//en 24 va <= si NO se quieren incluir puntos alineados
```

## 6.4. Par de puntos más cercanos

*run_closest_pair* $(P)$ encuentra el par de puntos más cercanos de $p[0, \ldots, P-1]$ en $\mathcal{O}(N \log N)$. Puede llegar a ser lento si hay muchos puntos superpuestos. [REQUIERE: *cmath*, *cstdlib*, *algorithm*]

```
#define SQ(x) ( (x)*(x) )
#define MAXP 131072
#define INF 1E9

typedef double tint;

struct pt {
    tint x, y;
} p[MAXP], s[MAXP];

```

```
int cmpx(const void *a, const void *b) {
    pt pa=*(pt*)a, pb=*(pt*)b;
    if (pa.x < pb.x) return -1;
    else if (pa.x > pb.x) return 1;
    else if (pa.y < pb.y) return -1;
    else if (pa.y > pb.y) return 1;
    else return 0;
}

double dist(const pt &a, const pt &b) {
    return sqrt(SQ(a.x-b.x) + SQ(a.y-b.y));
}

double closest_pair(int S, int E) {
    if (E-S == 1) return INF;
    else {
        int i, j, k, l, r, cur, mid = (S+E)/2;
        double tmp = min(closest_pair(S, mid), closest_pair(mid, E));

        i = S; j = mid; l = r = 0;
        while (i<mid || j<E) {
            if (i < mid && abs(p[i].x - p[mid].x) > tmp) i++;
            else if (j < E && abs(p[j].x - p[mid].x) > tmp) j++;
            else {
                if (i == mid || (j < E && p[i].y > p[j].y)) cur = j++;
                else cur = i++;

                while (l < r && p[cur].y - s[l].y > 2.0*tmp) l++;
                for (k=l; k<r; k++) tmp = min(tmp, dist(p[cur], s[k]));
                s[r++] = p[cur];
            }
        }

        i = k = S; j = mid;
        while (i<mid || j<E) {
            if (i == mid || (j < E && p[i].y > p[j].y)) s[k++] = p[j
                ++];
            else s[k++] = p[i++];
        }
        for (i=S; i<E; i++) p[i] = s[i];

        return tmp;
    }
}

double run_closest_pair(int P) {
    qsort(p, P, sizeof(pt), cmpx);
    return closest_pair(0, P);
}
```

## 6.5. Terna de puntos más cercanos

Código para encontrar la terna de puntos que minimicen el perímetro del triángulo formado por ellos. Se puede modificar para encontrar el par de puntos más cercanos.

```
#define MAXN 3002
const double INF = 1e20;

int n;
vector<pdd> p;

inline pdd inv(pdd& p) { return mp(p.second,p.first); }

int main() {
    while(cin >> n && n!=-1){
        forn(i,n) {
            double x,y; cin >> x >> y;
            p.pb(mp(x,y));
        }
        sort(all(p));

        double res = INF;
        set<pdd> sact;
        queue<pdd> qact;

        forn(i,n) {
            while (!qact.empty() && qact.front().first < p[i].first -
                res/2) {
                sact.erase(inv(qact.front()));
                qact.pop();
            }

            decl(low,sact.lower_bound(mp(p[i].second-res/2,-INF)));
            decl(hi,sact.upper_bound(mp(p[i].second+res/2,INF)));

            for (decl(it,low); it != hi; it++) {
                decl(it2,it); it2++;
                for (; it2 != hi; it2++) {
                    res <?= dist3(*it,*it2,inv(p[i]));
                }
            }
            sact.insert(inv(p[i]));
        }
        cout << res << endl;
    }
}
```

## 6.6. Librería de geometría

Ver aparte.

## 6.7. Teoremas y propiedades

### Teorema de Pick

Dado un polígono simple con vértices en los puntos de una grilla, el área $A$ del polígono se relaciona con el número $B$ de puntos en el borde y el número $I$ de puntos en el interior por medio de la expresión

$$A = I + B/2 - 1 \tag{6.1}$$

### Definiciones y propiedades de las elipses

Una elipse queda definida por la ecuación

$$AX^2 + BXY + CY^2 + DX + EY = 1 \tag{6.2}$$

donde $B^2 - 4AC > 0$. La expresión anterior puede llevarse a la forma

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1 \tag{6.3}$$

donde si $(x_0, y_0)$ es el centro de la elipse y $(x_a, y_a)$ es el versor del eje $x$, entonces $x = x_a\,(x - x_0) + y_a\,(y - y_0)$ y $y = -y_a\,(x - x_0) + x_a\,(y - y_0)$. El área de la elipse es entonces $\pi ab$, y se define su excentricidad $e$ como

$$e = \sqrt{\frac{a^2 - b^2}{a^2}} \tag{6.4}$$

La separación entre los focos es $ae = \sqrt{a^2 - b^2}$, y la distancia de un punto de la elipse a los focos es $L = 2a$. En forma paramétrica, la elipse toma la forma

$$\begin{aligned} x\,(t) &= x_0 + a\,cos\,(t)\,cos\,(\phi) - b\,sen\,(t)\,sen\,(\phi) \\ y\,(t) &= y_0 + a\,cos\,(t)\,sen\,(\phi) + b\,sen\,(t)\,cos\,(\phi) \end{aligned}$$

donde $t \in [0, 2\pi)$ y $\phi$ es el ángulo que forman el eje $x$ y el semieje mayor de la elipse.

## 7.1. Longest common subsequence

Cálculo de la *longest common subsequence* de dos cadenas.

```
#define MAXN 110
#define MAXM 110
int lcs[MAXN][MAXM];
int c[2][110];

int LCS(int N, int M){
    int i, j;

    for (i=1; i<N; i++) for (j=1; j<M; j++) lcs[i][j]=0;
    for (i=0; i<N; i++) lcs[i][0] = (c[0][i]==c[1][0]);
    for (i=0; i<M; i++) lcs[0][i] = (c[0][0]==c[1][i]);
    for (i = 1; i<N; i++){
        for (j = 1; j<M; j++){
            lcs[i][j] >?= lcs[i-1][j];
            lcs[i][j] >?= lcs[i][j-1];
            lcs[i][j] >?= lcs[i-1][j-1]+(c[0][i]==c[1][j]);
            }
        }
    return lcs[N-1][M-1];
}
```

## 7.2. Longest increasing subsequence

Encuentra e imprime la *longest increasing subsequence* en el array $X[MSX]$. $M[i]$ es la posición $k$ del menor valor $X[k]$ tal que $k < i$ y existe una *increasing subsequence* de longitud $i$ terminando en $X[k]$. $P[k]$ es la posición del predecesor de $X[k]$ en la *longest increasing subsequence* que termina en $X[k]$.

```
#define MAX 100010
tint X[MAX];
int P[MAX], M[MAX], ind1, ind2, i, j, k, L;

L = 0; M[0] = 0;
for (i=1; i<=N; i++) {
    ind1 = 0; ind2 = L;
    k = 0;
    while (ind1 < ind2) {//Notar que nunca mira M[ind1]
        j = (ind1+ind2)/2;
        if (j == ind1) {
            if (X[M[ind2]] < X[i]) k = ind2;
            else k = ind1;
            break;
        }
        if (X[M[j]] < X[i]) ind1 = j;
        else ind2 = j;
        k = ind1;
    }
    P[i] = M[k];
    if (k == L || X[i] < X[M[k+1]]) {
        M[k+1] = i;
        L += (k==L);
    }
}
cout << L << endl;//Longitud de la L. I. S.

cout << X[M[L]];//Imprime la L. I. S. al reves
i = P[M[L]];
while(i != 0){ cout << ' ' << X[i]; i = P[i]; }
cout << endl;
```

## 8.1. Algoritmo húngaro

Cálculo de la asignación máxima en un grafo bipartito de $n$ nodos en cada parte. Se maximiza el beneficio total considerando los beneficios individuales dados en la matriz $cost[n][n]$. Para minimizar el costo, se puede hacer $cost[i][j] \Rightarrow -cost[i][j]$. El algoritmo es $\mathcal{O}\left(n^3\right)$. [REQUIERE: cstring, MAX(a,b), MIN(a,b); INICIALIZAR: $n$, $cost[n][n]$]

El algoritmo de arriba, sólo que sirve para matrices no cuadradas. El costo se encuentra en la matriz $mat[m][n]$. [REQUIERE: *map*, *cstring*]

```
#define maxn 128
#define INF 10000000;
int mat[maxn][maxn];
int match1[maxn], match2[maxn];

int munkres(int m, int n) {
    int s[maxn],t[maxn],p,q,ret=0,i,j,k,tmp;
    int l1[maxn],l2[maxn];

    memset(match1, -1, sizeof(match1));
    memset(match2, -1, sizeof(match2));
    memset(l2, 0, sizeof(l2));
    for (i=0; i<m; i++)
        for (j=0; j<n ;j++) l1[i] = max(l1[i], mat[i][j]);

    for (i=0; i<m; i++) {
        memset(t, -1, sizeof(t));
        for (s[ p=q=0 ]=i; p<=q && match1[i]<0; p++) {
            k = s[p];
            for (j=0; j<n && match1[i]<0; j++) {
                if (mat[k][j] == l1[k]+l2[j] && t[j]<0) {
                    s[++q] = match2[j];
                    t[j] = k;
                    if (s[q] < 0) {
                        for (p=j; p>=0; j=p) {
                            match2[j] = k = t[j]; p = match1[k];
                            match1[k]=j;
                        }
                    }
                }
            }
        }
        if (match1[i] < 0) {
            i--; tmp = INF;
            for (k=0; k<=q; k++) {
                for (j=0; j<n; j++) {
                    if (t[j]<0 && (l1[s[k]] + l2[j] - mat[s[k]][j] < tmp
                        ) ) {
                        tmp = l1[s[k]] + l2[j] - mat[s[k]][j];
                    }
                }
            }
            for (j=0; j<n; j++) if (t[j] >= 0) l2[j] += tmp;
            for (k=0; k<=q; k++) l1[s[k]] -= tmp;
        }
    }
    for (i=0; i<m; i++) ret += mat[i][match1[i]];
    return ret;
}
```

## 8.2. Enteros largos

Libreria para trabajar con enteros largos. Se define $BASE = 10^{BEXP}$, y el número máximo de digitos que se puede representar es $BEXP * MAXN$. Hay que asegurarse de que $2 * BASE < 2^{32}$ y $MAXN * BASE^2 < 2^{64}$.

```
typedef unsigned int uint;
typedef unsigned long long ulong;

#define MAXN 16
#define BASE 100000000
#define BEXP 8

struct lint {
    uint d[MAXN];

    lint() {
        memset(d, 0, sizeof(d));
    }
    lint(int n) {
        for (int i=0; i<MAXN; i++, n/=BASE) {
            d[i] = n%BASE;
        }
    }
};

void print(lint n) {
    bool flag = false;
    for (int i=MAXN-1; i>=0; i--) {
        if (flag) printf("%0*d", BEXP, n.d[i]);
        else if (n.d[i] > 0 || i == 0) {printf("%d", n.d[i]); flag =
            true;}
    }
}

lint read(const char input[]) {
    lint RES;
    int i, j, k;

    for (i=strlen(input)-1, j=1, k=0; i>=0; i--, j*=10) {
        if (j == BASE) {j=1; k++;}
        RES.d[k] += j*(input[i]-'0');
    }
```

```
37      return RES;
38  }
39
40  lint sum(const lint &a, const lint &b) {
41      lint RES;
42      for (int i=0; i<MAXN; i++) RES.d[i] = a.d[i]+b.d[i];
43      for (int i=0; i<MAXN-1; i++) {RES.d[i+1] += RES.d[i]/BASE; RES.d
            [i] = RES.d[i]%BASE;}
44      return RES;
45  }
46  lint operator+(const lint &a, const lint &b) {return sum(a, b);}
47
48  lint sum(const lint &a, const uint &b) {
49      lint RES=a;
50      RES.d[0] += b;
51      for (int i=0; i<MAXN-1; i++) {RES.d[i+1] += RES.d[i]/BASE; RES.d
            [i] = RES.d[i]%BASE;}
52      return RES;
53  }
54  lint operator+(const lint &a, const uint &b) {return sum(a, b);}
55
56  lint sub(const lint &a, const lint &b) {//CUIDADO CON a < b!
57      lint RES;
58      int tmp[MAXN];
59
60      memset(tmp, 0, sizeof(tmp));
61      for (int i=0; i<MAXN; i++) {
62          tmp[i] += a.d[i]-b.d[i];
63          if (i < MAXN-1 && tmp[i] < 0) {
64              tmp[i+1]--;
65              tmp[i] += BASE;
66          }
67          RES.d[i] = tmp[i];
68      }
69      return RES;
70  }
71  lint operator-(const lint &a, const lint &b) {return sub(a, b);}
72
73  lint sub(const lint &a, const uint &b) {
74      return a-lint(b);
75  }
76  lint operator-(const lint &a, const uint &b) {return sub(a, b);}
77
78  lint mul(const lint &a, const lint &b) {
79      lint RES;
80      int i, j;
81      ulong tmp[MAXN];
82
83      memset(tmp, 0ULL, sizeof(tmp));
84      for (i=0; i<MAXN; i++) {
85          for (j=0; j<MAXN-i; j++) {
86              tmp[j+i] += ((ulong)a.d[i])*((ulong)b.d[j]);
87          }
88      }
89      for (i=0; i<MAXN-1; i++) {tmp[i+1] += (tmp[i]/BASE); RES.d[i] +=
            tmp[i]%BASE;}
90      return RES;
91  }
92  lint operator*(const lint &a, const lint &b) {return mul(a, b);}
93
94  lint div(const lint &a, const uint &b) {
95      lint RES;
96      ulong tmp=0ULL;
97
98      for (int i=MAXN-1; i>=0; i--) {
99          tmp = tmp*BASE + a.d[i];
100         RES.d[i] = tmp/b;
101         tmp %= b;
102     }
103     return RES;
104 }
105 lint operator/(const lint &a, const uint &b) {return div(a, b);}
106
107 uint mod(const lint &a, const uint &b) {
108     ulong tmp=0ULL;
109
110     for (int i=MAXN-1; i>=0; i--) tmp = (tmp*BASE + a.d[i])%b;
111     return tmp;
112 }
113 uint operator%(const lint &a, const uint &b) {return mod(a, b);}
114
115 bool operator<(const lint &a, const lint &b) {
116     for (int i=MAXN-1; i>0; i--) if (a.d[i] != b.d[i]) return a.d[i]
            < b.d[i];
117     return a.d[0] < b.d[0];
118 }
119
120 bool operator>(const lint &a, const lint &b) {
121     for (int i=MAXN-1; i>0; i--) if (a.d[i] != b.d[i]) return a.d[i]
            > b.d[i];
122     return a.d[0] > b.d[0];
123 }
124
125 bool operator<=(const lint &a, const lint &b) {return !(a>b);}
126 bool operator>=(const lint &a, const lint &b) {return !(a<b);}
127
128 bool operator==(const lint &a, const lint &b) {
129     for (int i=MAXN-1; i>0; i--) if (a.d[i] != b.d[i]) return false;
130     return a.d[0] == b.d[0];
131 }
132
133 bool operator!=(const lint &a, const lint &b) {return !(a==b);}
134
135 lint div(const lint &a, const lint &b) {
136     lint S(0), E=a+1, M;
137     while (E > S+1) {
```

```
138         M = (S+E)/2;
139         if (M*b > a) E = M;
140         else S = M;
141     }
142     return S;
143 }
144 lint operator/(const lint &a, const lint &b) {return div(a,b);}
145
146 lint mod(const lint &a, const lint &b) {
147     return a - (a/b)*b;
148 }
149 lint operator%(const lint &a, const lint &b) {
150     return mod(a, b);
151 }
```

## 8.3. Fracciones

```
1  int gcd(int a, int b) { return b != 0 ? gcd(b, a%b) : a; }
2  int lcm(int a, int b) { return a!=0 || b!=0 ? a / gcd(a,b) * b : 0;
       }
3
4  class fraction {
5      void norm() { int g = gcd(n,d); n/=g; d/=g; if( d<0 ) { n=-n; d
           =-d; } }
6      int n, d;
7
8  public:
9      fraction() : n(0), d(1) {}
10     fraction(int n_, int d_) : n(n_), d(d_) {
11         assert(d != 0);
12         norm();
13     }
14
15     fraction operator+(const fraction& f) const {
16         int m = lcm(d, f.d);
17         return fraction(m/d*n + m/f.d*f.n, m);
18     }
19     fraction operator-() const { return fraction(-n,d); }
20     fraction operator-(const fraction& f) const { return *this + (-f
           ); }
21     fraction operator*(const fraction& f) const { return fraction(n*
           f.n, d*f.d); }
22     fraction operator/(const fraction& f) const { return fraction(n*
           f.d, d*f.n); }
23
24     bool fraction::operator<(const fraction& f) const { return n*f.d
           < f.n*d; }
25
26     friend std::ostream& operator<<(std::ostream&, const fraction&);
27 };
28
```

```
29 ostream& operator<<(ostream& os, const fraction& f) {
30     os << f.n;
31     if(f.d != 1 && f.n != 0) os << '/' << f.d;
32     return os;
33 }
```

## 9.1. cmath

Las siguientes funciones pueden tomar como argumento **double**, **float** o **long double**, y devuelven un resultado del mismo tipo:

| | | |
|---|---|---|
| $sin\,(x)$ | $x \in \mathbb{R}$ | $sin\,(x) \in [-1, 1]$ |
| $cos\,(x)$ | $x \in \mathbb{R}$ | $cos\,(x) \in [-1, 1]$ |
| $tan\,(x)$ | $x \in \mathbb{R}$ | $tan\,(x) \in \mathbb{R}$ |
| $asin\,(x)$ | $x \in [-1, 1]$ | $asin\,(x) \in [-\pi/2, \pi/2]$ |
| $acos\,(x)$ | $x \in [-1, 1]$ | $acos\,(x) \in [0, \pi]$ |
| $atan\,(x)$ | $x \in \mathbb{R}$ | $atan\,(x) \in [-\pi/2, \pi/2]$ |
| $atan2\,(y, x)$ | $y, x \in \mathbb{R}$ | $atan2\,(y, x) \in [-\pi, \pi]$ |
| $cosh\,(x)$ | $x \in \mathbb{R}$ | $cosh\,(x) \in [1, +\infty)$ |
| $sinh\,(x)$ | $x \in \mathbb{R}$ | $sinh\,(x) \in \mathbb{R}$ |
| $tanh\,(x)$ | $x \in \mathbb{R}$ | $tanh\,(x) \in (-1, 1)$ |
| $exp\,(x)$ | $x \in \mathbb{R}$ | $exp\,(x) \in \mathbb{R}^+$ |
| $log\,(x)$ | $x \in \mathbb{R}^+$ | $log\,(x) \in \mathbb{R}$ |
| $log10\,(x)$ | $x \in \mathbb{R}^+$ | $log10\,(x) \in \mathbb{R}$ |
| $sqrt\,(x)$ | $x \in \mathbb{R}_0^+$ | $sqrt\,(x) \in \mathbb{R}_0^+$ |
| $ceil\,(x)$ | $x \in \mathbb{R}$ | $ceil\,(x) \in \mathbb{Z}$ |
| $floor\,(x)$ | $x \in \mathbb{R}$ | $floor\,(x) \in \mathbb{Z}$ |
| $fabs\,(x)$ | $x \in \mathbb{R}$ | $fabs\,(x) \in \mathbb{R}_0^+$ |

## 9.2. __builtin_

| | |
|---|---|
| int __builtin_ffs(unsigned int x) | Posición del primer 1 desde la derecha. |
| int __builtin_clz(unsigned int x) | Cantidad de ceros desde la izquierda. |
| int __builtin_ctz(unsigned int x) | Cantidad de ceros desde la derecha. |
| int __builtin_popcount(unsigned int x) | Cantidad de 1's en x. |
| int __builtin_parity(unsigned int x) | 1 si x es par, 0 si es impar. |
| double __builtin_powi(double x, int n) | $x^n$ sin garantías. |

## 9.3. iomanip

```
1| cout << fixed << setprecision(3);
```