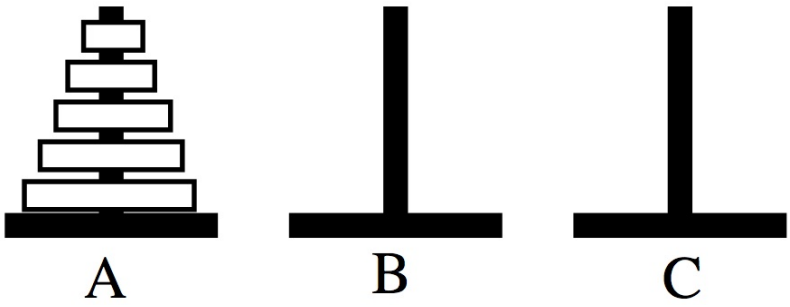

Figura 1: Representación esquemática de las Torres de Hanoi

1. Torres de Hanoi

En el conocidísimo rompecabezas de las torres de Hanoi el objetivo es trasladar los N discos inicialmente ensartados en el palo A hasta el palo C, paso a paso, y usando el palo B para las maniobras intermedias. En cada paso solo puede moverse un disco desde el palo en el que está a otro palo cualquiera, siempre y cuando el palo de destino no tenga discos, o el disco movido quede encima de un disco de mayor diámetro (ver diagrama).

En este ejemplo queremos desarrollar un procedimiento **iterativo** que, dado $N \geq 1$, imprima la secuencia de movimientos necesaria para resolver el rompecabezas. En concreto queremos diseñar un procedimiento

```
typedef char pole;
void hanoi(int N, pole org, pole aux, pole dst);
```

que imprime los movimientos necesarios para trasladar los N discos desde el palo `org` hasta el palo `dst`, usando el palo `aux` como palo auxiliar.

La solución recursiva resulta muy sencilla y constituye un ejemplo clásico de las ventajas del diseño recursivo. Si $N = 1$ basta mover el disco desde el palo `org` directamente al palo `dst`. Si $N > 1$ entonces moveremos, recursivamente, $N-1$ discos desde el palo `org` al palo `aux`, usando el palo `dst` como auxiliar. En el palo `org` sólo queda el disco de mayor diámetro y el palo `dst` está libre, con lo cual basta un movimiento para llevar a ese disco hasta `dst`. Tenemos entonces el disco de mayor diámetro en `dst`, $N-1$ discos en `aux` y el palo `org` libre. Como en `dst` está el disco de mayor diámetro, en la base del palo, podemos colocar encima de él cualquier otro disco. Para completar el rompecabezas bastará entonces trasladar, recursivamente, los $N-1$ discos desde el palo `aux` al palo `dst`, usando esta vez el palo `org` como palo auxiliar.

```
// Pre: N >= 1, (org,aux,dst) es una permutación de {'A', 'B', 'C'},
// cada uno de los N discos en el palo 'org' es de un diámetro inferior
// a cualquier disco que pueda haber en los palos 'aux' ó 'dst'

// Post: en el canal de salida 'cout' se imprimen todos los
// movimientos válidos necesarios para trasladar N discos situados en el
```

```
// palo 'org' al palo 'dst', empleando el palo 'aux' como intermediario
void hanoi(int N, pole org, pole aux, pole dst) {
    if (N==1)
      cout << "Mueve un disco de " << org
             << " a " << dst << endl;
    else {
      hanoi(N - 1, org, dst, aux);
      cout << "Mueve un disco de " << org
             << " a " << dst << endl;
      hanoi(N - 1, aux, org, dst);
    }
}
```

Escribir una implementación iterativa para el procedimiento `hanoi` no resulta tan sencillo. En lo que resta veremos una tecnica que (con algunas complicaciones adicionales) nos permitirá transformar de manera automática un algoritmo recursivo con recursividad múltiple (como en este caso) en un algoritmo iterativo equivalente, mediante el uso de una pila.

En el algoritmo iterativo una pila S contiene en todo momento las "tareas" pendientes de resolverse. Una tarea consiste en una tupla que representa la instancia a resolver; en nuestro caso el número de disco y las identidades de los palos de origen, auxiliar y de destino. En cada iteración el algoritmo desapila una tarea $curr = (n, org, aux, dst)$ de S. Si el numero de discos n es 1 la tarea puede ser resuelta de inmediato, moviendolo desde org a dst (y el palo auxiliar es irrelevante). Por otro lado si $n > 1$, tendremos tres tareas a resolver, por este orden: 1) $left = (n-1, org, dst, aux)$, 2) $middle = (1, org, \ldots, dst)$ y 3) $right = (n-1, aux, org, dst)$. Por esta razon se apilan en S, primero *right*, a continuación *middle* y por último *left*. Con ello completa una iteración. El bucle finaliza cuando no quedan tareas pendientes, esto es, cuando S queda vacía.

```
#include <iostream>
#include <stack>
#include <cassert>
using namespace std;

typedef char pole;

class HanoiTask {
private:
       int n_;
       pole org_, aux_, dst_;
public:
       HanoiTask(int n, pole org, pole aux, pole dst) {
           n_ = n;
           org_ = org; aux_ = aux; dst_ = dst;
       }
       // consultoras
       int ndiscos() const { return n_; }
       pole origen() const { return org_; }
```

```
        pole auxiliar() const { return aux_; }
        pole destino() const { return dst_; }

        // Pre: ndiscos() == 1
        // Post: escribe en el cout el movimiento necesario para
        //       realizar la tarea (mover un único disco)
        void escribe_movimiento() {
           assert(n_  == 1);
           cout << "Mueve disco de " << org_;
           cout << " a " << dst_ << endl;
        }
};

/* Pre: N >= 1, (org,aux,dst) es una permutación de {'A', 'B', 'C'},
   cada uno de los N discos en el palo 'org' es de un diámetro inferior
   a cualquier disco que pueda haber en los palos 'aux' ó 'dst' */

/* Post: en el canal de salida 'cout' se imprimen todos los
   movimientos válidos necesarios para trasladar N discos situados en el
   palo 'org' al palo 'dst', empleando el palo 'aux' como intermediario */
void hanoi(int N, pole org, pole aux, pole dst) {
  HanoiTask initial_task(N, org, aux, dst);
  stack<HanoiTask> S;
  S.push(initial_task);
  while (not S.empty()) {
    HanoiTask curr = S.top();
    S.pop();
    if (curr.ndiscos() == 1) {
      curr.escribe_movimiento();
    } else { // curr.ndiscos() > 1
      HanoiTask left_task(curr.ndiscos()-1, curr.origen(), curr.destino(),
                          curr.auxiliar());
      // el palo auxiliar es irrelevante en 'middle_task'
      HanoiTask middle_task(1, curr.origen(), curr.auxiliar(), curr.destino());

      HanoiTask right_task(curr.ndiscos()-1, curr.auxiliar(), curr.origen(),
                           curr.destino());
      S.push(right_task);
      S.push(middle_task);
      S.push(left_task);
    }
  }
}

int main() {
  int n;
  cout << "Num. discos: "; cin >> n;
  cout << endl;
  hanoi(n, 'A', 'B', 'C');
}
```

```
xxxxxxxxxx
x.x.x.x..x
x...x.x.xx
x.x.x...xx
x.x.x.x..x
xxx...x..x
x.x.xxx..x
x.x.x..xxx
x...x.xx.x
xx.xx.x..x
x..x..x..x
xx.x.xxxxx
xx...x..xx
x.......xx
xxxxxxxxxx
```

Figura 2: Matriz de caracteres representando un laberinto de $m = 13$ filas y $n = 8$ columnas ('X'=pared, '.'=posición libre)

2. Laberintos

En nuestro segundo ejemplo, mostraremos como utilizar una cola para encontrar un camino, si lo hay, entre dos posiciones dadas de un laberinto. De hecho, la utilización de la cola nos permitirá hallar un camino de longitud mínima, en el caso de que existan varios caminos alternativos entre las dos posiciones dadas.

Consideraremos laberintos rectangulares de $m \cdot n$ posiciones, organizadas en m filas y n columnas. Cada posición (i, j), $1 \leq i \leq m$, $1 \leq j \leq n$, puede estar o bien libre o bien ser una pared. Dado un laberinto L y dos posiciones libres, ini y fin, del laberinto, queremos una función que determine si existe al menos un camino entre ini y fin, y si así es, que nos devuelva uno cualquiera de longitud mínima. Un camino será una secuencia de posiciones válidas del laberinto, estando todas ellas libres y siendo cada par de posiciones del camino adyacente en el laberinto. Se dice que el camino es de longitud L si consta de $L+1$ posiciones, siendo ini la primera y fin la última. Por definición, para toda posición libre válida (i, j) existe un camino de longitud 0 entre la posición (i, j) y ella misma. También diremos que una posicion p está a distancia k de otra posición q si existen uno o más caminos de longitud k entre p y q, y no existe ningún camino de longitud inferior a k entre ambas posiciones.

Cada posición (i, j) tiene, salvo que sea adyacente al exterior del laberinto, cuatro posiciones adyacentes: $(i - 1, j)$ arriba, $(i, j + 1)$ a la derecha, $(i + 1, j)$ abajo y $(i, j - 1)$ a la izquierda. Por ejemplo, en el laberinto de la figura 2 existe un camino de longitud 11 entre las posiciones (2,2) y (6,1)—no existe ningún otro camino de menor longitud entre estas dos posiciones; entre las posiciones (2,2) y (8,7) no existe ningún camino.

Para encontrar un camino que nos lleva desde ini hasta fin, si lo hay, usaremos una cola Q de posiciones a visitar. El algoritmo recorrerá las posiciones libres del laberinto por orden creciente de distancia a ini. Es decir, en primer lugar se visitará ini, a continuación las posiciones libres válidas adyacentes a ini, las adyacentes libres a éstas (a distancia 2), y así sucesivamente. Puesto que el algoritmo no visita ninguna posición a distancia k de ini sin haber visitado previamente todas las posiciones a distancia inferior a k, garantizamos que el camino que encontremos es de longitud mínima (¡lo cual no significa que el algoritmo haga el mínimo número de iteraciones para encontrarlo!) Para evitar que la secuencia de posiciones visitadas entre "en bucle" revisitando una serie de posiciones una y otra vez (porque forman un ciclo) marcaremos las posiciones según las vayamos agregando a Q para ser visitadas. En cada iteración nuestro algoritmo extrae una posición p de la cola Q. Si $p = fin$ entonces hemos encontrado un camino entre ini y p y podemos finalizar nuestra búsqueda. En caso contrario, se encolan en Q todas las posiciones adyacentes a p válidas (que no quedan fuera del laberinto), libres y no marcadas (es decir, que ni han sido visitadas ya ni están ya en la cola Q). Las posiciones encoladas se marcan ya que han entrado en la cola Q para ser visitadas. El algoritmo de búsqueda terminará en cuanto visitemos fin o la cola Q quede vacia. En este último caso podemos concluir que no existe ningun camino entre ini y fin ya que el algoritmo visita todas las posiciones del laberinto accesibles desde ini.

En un primer nivel de refinamiento nuestro algoritmo tiene el siguiente aspecto:

```
// Pre: L.libre(ini), L.libre(fin),
//        todas las posiciones válidas de L sin marcar

... busca_camino(const Laberinto& L, pos ini, pos fin, ...) {
  queue<pos> Q;

  Q.push(ini); L.marcar(ini);
  bool camino_encontrado = false;
  while (not Q.empty() and not camino_encontrado) {
    pos p = Q.front(); Q.pop();
    L.marcar(p); // marcamos la posicion p
    if (igual(p, fin)) camino_encontrado = true;
    else {
      // p = (i,j)
      pos up(p.fila()-1, p.col());
      if (L.visitable(up)) {
          Q.push(up); L.marcar(up); ...
      }
      pos right(p.fila(), p.col()+1);
      if (L.visitable(right)) {
          Q.push(right); L.marcar(right); ...
      }
      pos down(p.fila()+1, p.col());
      if (L.visitable(down)) {
```

```
            Q.push(down); L.marcar(down); ...
        }
        pos left(p.fila(), p.col()-1);
        if (L.visitable(left)) {
            Q.push(left); L.marcar(left); ...
        }
      }
    }
    // 'camino_encontrado'==true ssi existe un camino
    // entre 'ini' y 'fin'
    ...
}
```

En esta solución preliminar damos por sentado que disponemos de las clases `pos` y `Laberinto`. La clase `pos` es muy simple y representa un par de enteros (i, j). La clase `Laberinto` nos permite representar laberintos, en concreto, saber para cada posición del laberinto si es una pared o está libre, y además de ofrecer operaciones para su construcción, destrucción, lectura, escritura, ... vemos que debería ofrecer al menos otros dos métodos adicionales: el método `marcar`, que permite marcar una posición válida de un laberinto, y el consultor `visitable` que devuelve cierto si y sólo si la posición p dada es visitable: es decir, si es válida (no exterior al laberinto), está libre (no es una pared) y no está marcada ("visitada o pendiente de visita").

Una vez definidas las clases `pos` y `Laberinto` tendríamos una solución completa para el problema de determinar la existencia de un camino entre las dos posiciones dadas, pero no sabemos de nigún camino, si lo hay, que las una. Para resolver esta parte del problema usaremos "miguitas de pan". En cada posición que visitemos dejaremos una indicación de cuál es la posición p desde donde llegamos. Esto se hace al encolar en Q la visita futura de la posición. Por ejemplo, si la posición *up* "encima" de p es válida y no está marcada se encolará Q; lo que haremos será indicar también que p es el *predecesor* de *up* en el camino que estamos explorando y que lleva desde *ini* hasta *up*. Serán necesarios dos métodos adicionales en la clase `Laberinto`: `asigna_pred`(q, p), que asigna la posición p como predecesora de q, y el consultor `predecesor`(p) que nos devuelve el predecesor de la posición p; si no se le ha asignado ninguno, el método devolverá la propia posición p. En particular, la posición *ini*, como es la de inicio, se tendrá a si misma como "predecesor".

Cuando el algoritmo encuentre un camino que lleva hasta fin podremos reconstruirlo de atrás hacia adelante: fin es la última posición del camino, su predecesor la penúltima posición, y así sucesivamente, hasta que llegamos a la posición *ini*. Como reconstruímos el camino en orden inverso resultará muy conveniente almacenarlo en una pila de posiciones. El usuario de la función recibe la pila y extrayendo iterativamente los contenidos de ésta, podrá, por ejemplo, imprimir el camino desde *ini* a fin; si el camino no existe la pila sera vacía. Nótese que un camino de longitud 0 sería una secuencia con una sola posición, $ini = fin$, y si lo representamos en una pila ésta no sería vacía.

```
#include "pos.hh"
```

```
#include "Laberinto.hh"
#include <iostream>
#include <stack>
#include <queue>

// Pre: L.libre(ini), L.libre(fin),
//      L.marcada(p)==false para toda posiciones válidas libre,
//      L.predecesor(p)==p para toda posición válida libre,
//      camino.empty()

// Post: devuelve, en la pila de posiciones 'camino', la secuencia
// de posiciones en un camino de longitud  mínima entre 'ini' y 'fin'
// en el laberinto L, si existe alguno; 'camino' es vacia si no existe
// ningún camino

void busca_camino(const Laberinto& L, pos ini, pos fin,
                  stack<pos>& camino) {
  queue<pos> Q;
  Q.push(ini);
  bool camino_encontrado = false;
  while (not Q.empty() and not camino_encontrado) {
    pos p = Q.front(); Q.pop();
    L.marcar(p); // marcamos la posicion p
    if (igual(p, fin)) camino_encontrado = true;
    else {
      // p = (i,j)
      pos up(p.fila()-1, p.col());
      if (L.visitable(up)) {
        Q.push(up); L.marcar(up);
        L.asigna_pred(up, p);
      }
      pos right(p.fila(), p.col()+1);
      if (L.visitable(right)) {
        Q.push(right); L.marcar(right);
        L.asigna_pred(right, p);
      }
      pos down(p.fila()+1, p.col());
      if (L.visitable(down)) {
        Q.push(down); L.marcar(down);
        L.asigna_pred(down, p);
      }
      pos left(p.fila(), p.col()-1);
      if (L.visitable(left)) {
        Q.push(left); L.marcar(left);
        L.asigna_pred(left, p);
      }
    }
  }

 // reconstruimos el camino en sentido inverso
```

```
  if (camino_encontrado) {
    pos p = fin;
    while (not igual(p,ini)) {
      camino.push(p);
      p = L.predecesor(p);
    }
    camino.push(p);
  }
}

int main() {
  Laberinto L;
  cin >> L;
  pos ini, fin;
  cin >> ini >> fin;
  stack<pos> camino;
  busca_camino(L, ini, fin, camino);
  if (camino.empty())
      cout << "No hay camino de " << ini << " a " << fin << endl;
  else {
    cout << "Camino entre " << ini << " y " << fin << ":";
    escribir_pila(camino)
  }
}
```

Dejamos como ejercicio para el lector la especificación detallada e implementación de las clases `pos` y `Laberinto`, así como de algunas funciones auxiliares, p.e., `escribir_pila`.