1 Agencia de viajes: enunciado

Una agencia de viajes mantiene una base de datos con exactamente N clientes y M destinos turísticos. En una situación real, estos valores fluctuarían, pero aquí los dejamos constantes, para simplificar. Cada cliente y cada destino viene dado por un identificador entre 1 y N o M según el caso. Además, para cada cliente de la agencia se almacenan sus destinos preferidos. Los destinos preferidos de un cliente pueden ser modificados en cualquier momento, pero no puede haber clientes con cero destinos preferidos. Notemos otra vez que en la realidad debería guardarse mucha más información de cada cliente y destino, pero en esta práctica no será necesario.

Ejemplo. El cliente 5 se puede dar de alta con destinos preferidos 1, 4 y 7. Al cabo de un tiempo, puede borrar los destinos 1 y 7 y añadir el 6, etc.

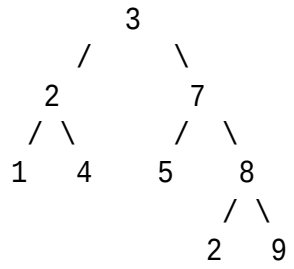
La intención de la agencia es optimizar sus recursos, por lo cual sólo organiza dos tipos de circuitos, que se describen más adelante. Estos circuitos se realizan para grupos del mayor tamaño posible, como parte de la estrategia de optimización.

Un circuito turístico está formado por uno o más destinos, sin repetir. Un destino satisface a un cliente si éste lo tiene entre sus preferencias. Un circuito satisface a un cliente si alguno de los destinos del circuito está entre las preferencias del cliente.

Circuitos tipo 1: La agencia dispone en todo momento de un conjunto de circuitos predefinidos. Cuando se deba organizar un circuito de tipo 1, se elige entre ellos el que satisfaga al mayor número de clientes. En un programa "realista" pediríamos devolver dicho circuito, pero para simplificar los cálculos, sólo hará falta informar de cuántos clientes satisface.

Los circuitos del conjunto compartirán destinos: todos comenzarán en el mismo destino, y en general compartirán "prefijos", es decir, dos circuitos dados tendrán varios de sus primeros destinos en común (correlativamente y empezando por el primero). Por esa razón, el conjunto de circuitos puede representarse mediante una estructura arborescente, y los circuitos serán las ramas de la misma. Para simplificar, supondremos que un destino sólo puede tener 0, 1 o 2 destinos a continuación.

Ejemplo: un conjunto de circuitos válido es 3,2,1, 3,2,4, 3,7,5, 3,7,8,2 y 3,7,8,9. Se puede representar mediante



Por último, el programa ha de permitir sustituir el conjunto de circuitos por otro nuevo cuantas veces se desee.

Circuitos tipo 2: Los circuitos de tipo 2 deben satisfacer a TODOS los clientes, pero intentando emplear el menor número posible de destinos. En esta modalidad no hay que basarse en el conjunto de los circuitos predefinidos del apartado anterior, sólo en las preferencias de los clientes.

Hay muchas estrategias posibles: algunas, muy costosas algorítmicamente, permiten obtener el circuito con el mínimo exacto de destinos necesarios; otras, mucho más eficientes, producen un circuito con una aproximación razonable a dicho mínimo. Nos inclinaremos por una de éstas últimas, que describimos brevemente a continuación.

Cuando haya que crear uno de estos circuitos, se elegirá como primer destino al que satisfaga a la mayor cantidad posible de clientes. De entre los destinos restantes, se selecciona aquel que satisfaga al mayor número de los clientes que faltan y se repite el proceso hasta que todos los clientes resulten satisfechos. En cada selección, si hay varios destinos empatados, el elegido es el de menor identificador. Nuevamente, para simplificar el programa, no hará falta mostrar el circuito construido sino sólo el número de destinos que contiene.

SE PIDE: Diseñar un programa modular que gestione la agencia. En primer lugar, debe inicializar las preferencias de todos los clientes y obtener un conjunto de circuitos inicial. Después tendrá que ir procesando las diversas operaciones que se le pidan, ya sean organizar circuitos de un tipo u otro, reemplazar el conjunto de circuitos por uno nuevo o modificar las preferencias de un cliente (añadiendo o quitando destinos).

La forma de comunicarse con el programa para que realice estas tareas será parecida a la de los ejercicios "Cubeta", "PseudoTetris", etc. Podéis diseñar un esquema provisional que ya refinaréis cuando conozcáis el juego de pruebas público.

La sintaxis de los datos y resultados, acompañada del juego de pruebas público, se conocerá dos semanas antes del día de la entrega del programa Java. Hasta entonces no podréis implementar de forma definitiva las operaciones de lectura y escritura necesarias para los tipos que utilicéis, aunque sí podréis especificarlas.

1.1 Esquema de programa principal

El programa principal leerá las preferencias de todos los clientes y obtendrá un conjunto de circuitos inicial. Después dará curso a las operaciones anteriormente mencionadas hasta que se le ordene parar. El esquema resultante, sin tener en cuenta la escritura de resultados, es

```
Esquema de programa principal:

  inicializar conjunto de clientes
  inicializar conjunto de circuitos
  leer opcion
  mientras no final hacer
    si opcion = 1 -> actualizar conjunto de circuitos
    [] opcion = 2 -> organizar circuito 1
    [] opcion = 3 -> organizar circuito 2
    [] opcion = 4 -> añadir destinos cliente
    [] opcion = 5 -> borrar destinos cliente
    fsi
    leer opcion
  fmientras
```

2 Especificación de los módulos

Atendiendo a los tipos de datos mencionados en el enunciado, necesitaremos un módulo para representar un conjunto de clientes, otro para las preferencias de un cliente y otro para el conjunto de circuitos predefinidos de la agencia. En una resolución normal, comenzaríamos por considerar las operaciones necesarias para el programa principal y las clasificaríamos en los diferentes módulos. Al pasar a su implementación, quizá descubriésemos que algún módulo necesita alguna operación adicional y la incorporaríamos en ese momento (sólo si es pública, es decir, si se usa en un módulo distinto al que pertenece). Sin embargo, en un documento de estas características, se presentan los módulos completamente acabados, sin necesidad de reflejar el proceso que ha dado lugar a su especificación final.

El conjunto de clientes da lugar al módulo `Cjt_Clientes`. Sus operaciones serán, en principio, la de lectura (es la única forma de crear un conjunto nuevo), añadir preferencias de un cliente del conjunto, borrar preferencias de un cliente del conjunto (estas operaciones obtienen sus datos de la secuencia de entrada) y organizar un circuito tipo 2, puesto que esta operación sólo depende de la información de los clientes del conjunto.

Sin embargo, por necesidades de la operación que organiza un circuito tipo 1 también hay que incluir una operación que calcule cuántos clientes del conjunto tienen a un cierto destino entre sus preferencias y, por otra parte, también necesitaremos una operación para marcar los clientes que tengan a un cierto destino entre sus preferencias, de modo que no se contabilice a un mismo cliente dos veces. Por supuesto, las operaciones de leer, añadir preferencias y quitar preferencias deben producir conjuntos de clientes no marcados.

```
Modulo Cjt_Clientes {datos};

Especificacion

Tipo Cjt_Clientes;

{ Descripción: contiene las preferencias de los N clientes, que son
destinos entre 1 y M}

Operaciones

funcion leer_cjt_clientes () retorna c:  Cjt_Clientes;
{Pre: - }
{Post: c es un conjunto de clientes de la secuencia de entrada}

funcion circuito_2 (c: Cjt_Clientes) dev n:nat;
{Pre: - }
{Post: n es el mínimo número de destinos que satisface a todos
los clientes de c, según la estrategia de la práctica}
```

```
accion altas_preferencias (e/s c: Cjt_Clientes);
{Pre: - }
{Post: se añaden preferencias a los clientes a partir de la secuencia
de entrada }

accion bajas_preferencias (e/s c: Cjt_Clientes);
{Pre: - }
{Post: se quitan preferencias a los clientes a partir de la secuencia
de entrada }

funcion num_cl_sat (c: Cjt_Clientes; i: nat) dev n:nat;
{Pre: 1<=i<=M }

{Post: n es el número de clientes no marcados de c satisfechos por el destino i}

accion marcar_cl_sat (e/s c: Cjt_Clientes; ent i: nat);
{Pre: 1<=i<=M }

{Post: se han marcado los clientes de c satisfechos por el destino i }
```

La información asociada a un cliente da lugar al módulo `Cliente`. Inicialmente, sus operaciones serán la de lectura, añadir una preferencia, borrar una preferencia y consultar una preferencia. Por necesidades de las operaciones de los otros módulos necesitaremos también una operacion que marque a un cliente (el circuito construido contiene un destino preferido por el cliente) y otra que consulte si un cliente está marcado.

```
Módulo Cliente {datos}

Especificación

Tipo Cliente;

{ Descripción: información de un cliente: preferencias, etc. }

Operaciones

funcion leer_cliente () retorna cl: Cliente;
{Pre: - }
{Post:  cl es un cliente de la secuencia de entrada}

accion añadir_preferencia (e/s cl: Cliente; ent i: nat);
{Pre: 1<=i<=M }
{Post:  cl pasa a tener el destino i como preferencia}
```

```
accion quitar_preferencia (e/s cl: Cliente; ent i: nat);
{Pre: 1<=i<=M }
{Post:  cl deja de tener el destino i como preferencia}

funcion consultar_preferencia (cl: Cliente; i: nat) dev b: bool;
{Pre: 1<=i<=M }
{Post:  b indica si el destino i es una de las preferencias de cl}

accion marcar (e/s cl: Cliente);
{Pre: -  }

{Post:  cl pasa a estar marcado}

funcion marcado (cl: Cliente) dev b: bool;
{Pre: - }

{Post:  b indica si cl está marcado}
```

Por último, el conjunto de circuitos predefinidos de la agencia da lugar al módulo `Cjt_Circuitos`. Sus únicas operaciones proceden directamente del enunciado y son la de lectura, que sirve tanto para el conjunto inicial como para las sucesivas actualizaciones, y la organización de un circuito tipo 1.

```
Modulo Cjt_Circuitos {datos}

Especificacion

Sobre Cjt_Clientes;

Tipo Cjt_Circuitos;

{ Descripción: representa los circuitos turísticos de una agencia }

Operaciones

funcion leer_cjt_circuitos () retorna c: Cjt_Circuitos;
{Pre: - }
{Post:  c es un conjunto de circuitos de la secuencia de entrada}

funcion circuito_1 (cci: Cjt_Circuitos; ccl: Cjt_Clientes) dev n:nat;
{Pre: - }
{Post: n es el máximo número de clientes de ccl que se pueden
satisfacer con un circuito de cci}
```

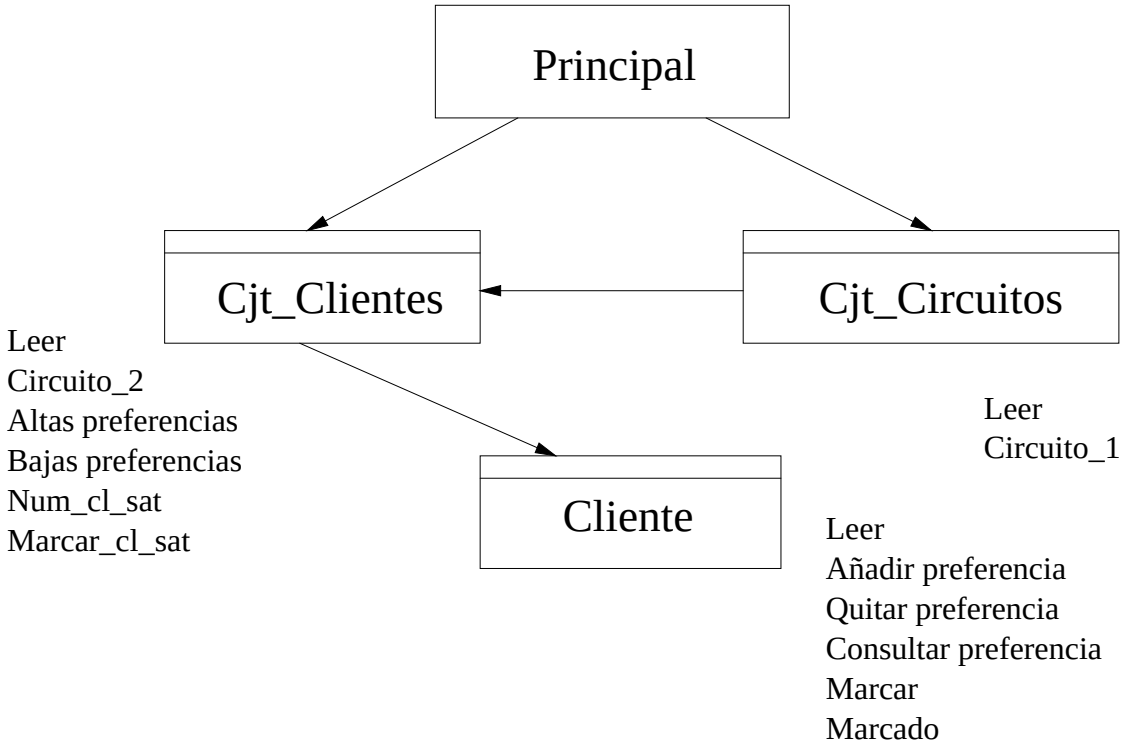
3 Programa principal detallado

Ya estamos en condiciones de escribir el programa principal definitivo. Las únicas operaciones que generan salida de resultados serán las organizaciones de los circuitos. Notad que estamos suponiendo que los datos leídos siempre son correctos, ya que no incluimos comprobaciones al respecto. Por último, puesto que los datos son naturales (clientes, preferencias, ...) usaremos números negativos para las opciones.

```
Programa principal

  var
    C_CLI: Cjt_Clientes
    C_CIRC: Cjt_Circuitos
    op: entero;     {Código de operación}
  fvar

  C_CLI:= leer_cjt_clientes ();
  C_CIRC:= leer_cjt_circuitos ();
  op:=leer();

  mientras op!=-6 hacer

    si op=-1 entonces C_CIRC:= leer_cjt_circuitos ();

    [] op=-2 entonces
                      var n: nat;
                      n:= circuito_1 (C_CIRC,C_CLI);
                      escribir(n);

    [] op=-3 entonces
                      var n: nat;
                      n:= circuito_2 (C_CLI);
                      escribir(n);

    [] op=-4 entonces altas_preferencias(C_CLI);

    [] op=-5 entonces bajas_preferencias(C_CLI);
    fsi;

    op:= leer()

  fmientras;
```

4 Diagrama modular

Con toda la información que hemos recogido hasta ahora, podemos expresar gráficamente las relaciones entre los módulos de nuestra solución mediante el correspondiente diagrama modular.



5 Implementación de los módulos

5.1 El módulo `Cjt_Clientes`

El tipo `Cjt_Clientes` se representa mediante un vector de `Cliente`, ya que su tamaño está acotado por un valor conocido. Su tamaño exacto es constante y no hace falta guardarlo. Las tres primeras operaciones consisten esencialmente en la lectura y almacenamiento de datos a partir de la secuencia de entrada y no las detallamos.

```
Modulo Cjt_Clientes;

Implementación

Sobre Cliente;

Tipo Cjt_Clientes = vector [1..N] de Cliente;

{ Descripción: contiene las preferencias de los N clientes, que son
destinos entre 1 y M}


Operaciones

funcion leer_cjt_clientes () retorna cjt:  Cjt_Clientes;
{Pre: - }
Itera leer_cliente
{Post: c es un conjunto de clientes de la secuencia de entrada}

accion altas_preferencias (e/s c: Cjt_Clientes);
{Pre: - }
Itera añadir_preferencia
{Post: se añaden preferencias a los clientes a partir de la secuencia
de entrada }

accion bajas_preferencias (e/s c: Cjt_Clientes);
{Pre: - }
Itera quitar_preferencia
{Post: se quitan preferencias a los clientes a partir de la secuencia
de entrada }
```

5.1.1 La operación `circuito_2`

Siguiendo la estrategia presentada en el enunciado, la función debe obtener el destino que más clientes satisfaga, luego el segundo (descontando los clientes satisfechos por el primero), y así hasta que todos los clientes resulten satisfechos.

```
funcion circuito_2 (c: Cjt_Clientes) dev n:nat;
{Pre: - }

  var nsat, max, sat: nat

  {Inv: nsat = número de clientes de c satisfechos por
   los n primeros destinos obtenidos por la estrategia;
   hemos marcado en c dichos clientes}
  {Cota: N-nsat}

  nsat:=0; n:=0;
  mientras nsat < N hacer
    <max,sat> := buscar_max_sat (c);
    marcar_cl_sat (c, max);
    nsat:=nsat+sat; n:=n+1;
  fmientras

{Post: n es el mínimo número de destinos que satisface a todos
los clientes de c, según la estrategia de la práctica}
```

Esta operación se ha obtenido mediante el siguiente razonamiento:

- *Inicializaciones.* Al principio no se ha obtenido aún ningun destino ($n = 0$), ni se ha marcado ningún cliente ($nsat = 0$).
- *Condición de salida.* Si ya están satisfechos todos los clientes ($nsat = N$), entonces n será el número de destinos buscado.
- *Cuerpo del bucle.* Para avanzar hacia la finalización debemos aumentar el número de clientes satisfechos (*nsat*). La estrategia obliga a hacerlo sumándole los clientes no marcados satisfechos por el destino que satisfaga a la máxima cantidad de clientes aún no marcados. Ambos valores, el destino (*max*) y su número de clientes satisfechos (*sat*) se obtienen llamando a la operación `buscar_max_sat`.

  Con toda esa información, para conservar el invariante debemos sumar dicha cantidad a *nsat*, incrementar en 1 el valor de n y marcar los nuevos clientes satisfechos.
- *Decrecimiento.* La diferencia entre N y *nsat* disminuye en cada vuelta, ya que al no haber clientes con cero preferencias, si quedan clientes por satisfacer ($nsat < N$) seguro que hay algún destino aún no probado que satisface a algun cliente nuevo, y por lo tanto *nsat* aumenta de valor.

5.1.2 La operación `buscar_max_sat`

Esta función obtiene el máximo de los clientes aún sin marcar satisfechos por los diversos destinos.

```
funcion buscar_max_sat (c: Cjt_Clientes) dev max, sat:nat;
{Pre: M>0 (hay al menos un destino) }

  var i,n: nat

  {Inv: max = destino en [1..i-1] que satisface a más clientes no marcados de c;
        sat = número de clientes no marcados de c satisfechos por max; 1<=i<=M+1}
  {Cota: M-i+1}

  i:=2; max:=1; sat:= num_cl_sat (c,1);
  mientras i<=M hacer
    n:=num_cl_sat (c,i);
    si n > sat entonces max:=i; sat:=n fsi;
    i:=i+1;
  fmientras

{Post:  max es el destino que satisface a más clientes no marcados de c;
sat es el número de clientes no marcados de c satisfechos por el destino max}
```

Notad que podemos ahorrar algún cálculo si en `circuito_2` mantenemos un vector local con los destinos ya usados, es decir, los que en alguna iteración hayan satisfecho a la mayor cantidad de clientes. Dicha información se deberá ir actualizando en `buscar_max_sat`. De este modo, una vez que un destino ya ha sido usado, no es necesario volver a calcular el número de clientes que satisface.

La derivación informal para obtener este código es la típica para cualquier problema de máximos:

- *Inicializaciones.* Si consideramos que el conjunto vacío no tiene definido un máximo, deberemos comenzar con `i=2`, de forma que el primer máximo válido es el primer destino, y el primer *sat* será el número de clientes por él satisfechos.

- *Condición de salida*. Deseamos tratar a todos los destinos, así que saldremos del bucle si `i=M+1`. Por tanto, seguiremos mientras `i <= M` (el invariante asegura que `i <= M+1`).

- *Cuerpo del bucle*. Como hay que avanzar hacia la condición de salida, lo mejor es incrementar la `i` de manera que se acerque a `M`. Pero antes hay que tratar el destino `i+1` para mantener el invariante, es decir, `n` ha de acabar siendo el destino en `[1..i]` que satisface a más clientes no marcados de `c` y `sat` ha de ser el número de clientes no marcados de `c` satisfechos por `max`. Previamente, `max` y `sat` cumplen dicha propiedad pero sólo considerando los destinos `[1..i-1]`. Por tanto, obtenemos en `n` el número de clientes de `c` no marcados satisfechos por el destino `i` y miramos si `n` es mayor que `sat`. En caso afirmativo, `i` es el nuevo máximo ya que si satisface a más clientes que el máximo anterior, también satisface a más clientes que todos los demás hasta el momento (por la propiedad transitiva). En caso contrario, `max` sigue siendo el destino que más clientes satisface hasta el momento y no tocamos nada. A continuación ya podemos incrementar la `i` para volver a alcanzar el invariante hasta `i-1`.

- *Decrecimiento.* A cada vuelta disminuye la distancia entre `M+1` y `i`.

5.1.3 La operación `num_cl_sat`

Esta función se encarga de contar los clientes aún sin marcar satisfechos por un destino.

```
funcion num_cl_sat (c: Cjt_Clientes; i: nat) dev n:nat;
{Pre: 1<=i<=M }

  var j: nat;

  {Inv: n = número de clientes de c en [1..j] no marcados
            satisfechos por el destino i; j<=N}
  {Cota: N-j}

  j:=0; n:=0;
  mientras j<N hacer
    si consultar_preferencia (c[j+1],i) /\ !marcado(c[j+1])
       entonces n:=n+1 fsi;
    j:=j+1;
  fmientras

{Post: n es el número de clientes no marcados de c satisfechos por el destino i}
```

La derivación informal para obtener este código es la típica para cualquier problema de conteo:

- *Inicializaciones.* Si empezamos con `j=0` aún no tenemos ningun cliente satisfecho, por tanto `n` también deberá ser `0`

- *Condición de salida.* Deseamos tratar a todos los destinos, así que saldremos del bucle si `j=N`. Por tanto, seguiremos mientras `j < N` (el invariante asegura que `j <= N`).

- *Cuerpo del bucle.* Como hay que avanzar hacia la condición de salida, lo mejor es incrementar en 1 la `j` de manera que se acerque a `N`. Pero antes hay que tratar el cliente `j+1` para mantener el invariante, es decir, `n` ha de acabar siendo el número de clientes no marcados de `c` en `[1..j+1]` que son satisfechos por el destino `i`. Previamente, `n` es el número de clientes no marcados de `c` en `[1..j]` que son satisfechos por el destino `i`. Por tanto, si el cliente `j+1` cumple al mismo tiempo que es satisfecho por el destino `i` y que no ha sido marcado todavía, `n` ha de aumentar en 1, y ha de conservar su valor en caso contrario. A continuación, ya podemos incrementar la `j` para volver a alcanzar el invariante hasta `j`.

- *Decrecimiento.* A cada vuelta disminuye la distancia entre `N` y `j`.

5.1.4 La operación `marcar_cl_sat`

Esta acción se encarga de marcar los clientes satisfechos por un destino.

```
accion marcar_cl_sat (e/s c: Cjt_Clientes; ent i: nat);
{Pre: 1<=i<=M }

  var j: nat;

  {Inv: se han marcado los clientes de c en [1..j-1] satisfechos
        por el destino i; 1<=j<=N+1}
  {Cota: N-j+1}

  j:=1;
  mientras j<=N hacer
    si consultar_preferencia (c[j],i) entonces marcar(c[j]) fsi;
    j:=j+1;
  fmientras

{Post: se han marcado los clientes de c satisfechos por el destino i }
```

Notad que se podrían saltar los clientes que ya estuviesen previamente marcados, pero el ahorro de no volver a marcar éstos se compensa con el de controlar quién lo está y quién no.

La derivación seguida es muy similar a la de las dos operaciones anteriores, con la única diferencia de que se trata de una acción.

- *Inicializaciones.* Si empezamos con `j=1` aún no hemos hemos de marcar ningun cliente, por tanto no hay que hacer nada más.
- *Condición de salida.* Deseamos tratar a todos los clientes, así que saldremos del bucle si `j=N+1`. Por tanto, seguiremos mientras `j <= N` (el invariante asegura que `j <= N+1`).
- *Cuerpo del bucle.* Como hay que avanzar hacia la condición de salida, lo mejor es incrementar la `j` de manera que se acerque a `N`. Pero antes hay que tratar el cliente `j` para mantener el invariante, es decir, que se hayan marcado los clientes de `c` en `[1..j]` satisfechos por el destino `i`.

  Previamente, se han marcado los clientes de `c` satisfechos por el destino `i` en `[1..j-1]`. Por tanto, si el cliente `j` es satisfecho por el destino `i` hay que marcarlo. En caso contrario, se queda como está. A continuación ya podemos incrementar la `j` para volver a alcanzar el invariante hasta `j-1`.
- *Decrecimiento.* A cada vuelta disminuye la distancia entre `N+1` y `j`.

5.2 El módulo `Cliente`

El tipo `Cliente` necesita almacenar sus preferencias y debe de permitir la posibilidad de ser marcado. Para la primera información empleamos un vector de booleanos y para la segunda un campo booleano. Las operaciones son muy sencillas y no necesitan más comentario.

```
Módulo Cliente

Implementación

Tipo Cliente = tupla
                 prefs : vector [1..M] de bool;
                 sat: bool;
               ftupla

funcion leer_cliente () retorna cl: Cliente;
{Pre: - }
Itera la lectura de las preferencias y poner sat a falso
{Post:  cl es un cliente de la secuencia de entrada}

accion añadir_preferencia (e/s cl: Cliente; ent i: nat);
{Pre: 1<=i<=M }
  cl.prefs[i]:=cierto
{Post:  cl pasa a tener el destino i como preferencia}

accion quitar_preferencia (e/s cl: Cliente; ent i: nat);
{Pre: 1<=i<=M }
  cl.prefs[i]:=falso
{Post:  cl deja de tener el destino i como preferencia}

funcion consultar_preferencia (cl: Cliente; i: nat) dev b: bool;
{Pre: 1<=i<=M }
  b:=cl.prefs[i]
{Post:  b indica si el destino i es una de las preferencias de cl}

accion marcar (e/s cl: Cliente);
{Pre: - }
  cl.sat:=cierto
{Post:  cl pasa a estar marcado}

funcion marcado (cl: Cliente) dev b: bool;
{Pre: - }
  b:=cl.sat
{Post:  b indica si cl está marcado}
```

5.3 El módulo `Cjt_Circuitos`

Como ya se dijo en el enunciado, la mejor representación para el conjunto de circuitos es un árbol binario de naturales. Por lo tanto la operación de lectura de un conjunto simplemente llamará a la de leer un árbol.

```
Modulo Cjt_Circuitos;

Implementación

Sobre Cjt_Clientes;

Tipo Cjt_Circuitos = arbol de nat

Operaciones

funcion leer_cjt_circuitos () retorna c: Cjt_Circuitos;
{Pre: - }
Leer arbol
{Post:  c es un conjunto de circuitos de la secuencia de entrada}
```

La operación `circuito_1` recorre todos los circuitos del conjunto (las ramas del árbol) y calcula el máximo de los clientes satisfechos por dichos circuitos.

```
funcion circuito_1 (cci: Cjt_Circuitos; ccl: Cjt_Clientes) dev n:nat;
{Pre: - }
  var n1,n2: nat;
  si es_nulo (cci) entonces n:=0
  [] !es_nulo (cci) entonces
        n:=num_cl_sat(ccl, raiz(cci));
        marcar_cl_sat(ccl,raiz(cci));
        n1:= circuito_1(hi(cci),ccl);
        n2:= circuito_1(hd(cci),ccl);
        {HI: n1 = máximo número de clientes de ccl que se pueden
             satisfacer con un circuito de hi(cci); n2 = máximo
             número de clientes de ccl que se pueden satisfacer
             con un circuito de hd(cci)}
        {Cota: tam(cci)}

        n:=n+max(n1,n2);
  fsi
{Post: n es el máximo número de clientes de ccl que se pueden
satisfacer con un circuito de cci}
```

La opción que hemos elegido es la de considerar primero los clientes satisfechos por el destino raíz, marcar estos y luego continuar el cálculo recursivamente. Se podría dejar el destino raíz

para el último, pero entonces el cálculo del máximo por la izquierda habría marcado a algunos clientes y el de la derecha a otros, por lo que sería un poco más complicado obtener el número de clientes no marcados que dicho destino satisface.

La derivación informal que hemos seguido es

- *Caso sencillo.* Si el árbol es nulo, es decir, no tiene destinos, no se satisface ningún cliente, por tanto el resultado es 0.
- *Caso recursivo.* Si el árbol es nulo, contamos los clientes satisfechos por el destino raíz (*n*) y los marcamos (esta info queda anotada en el conjunto de clientes). A partir de dichas marcas aplicamos las llamadas recursivas para saber cuántos clientes más satisfacen, respectivamente, el subconjunto de la izquierda y el de la derecha. Finalmente, basta con sumar *n* al máximo del número de clientes satisfechos por dichos subconjuntos.

  La llamada recursiva es correcta:

  1. Los parámetros son correctos. Como el árbol no es nulo, por la protección, sus subárboles son válidos.
  2. Los parámetros cumplen la precondición, ya que ésta es cierto.

- *Decrecimiento.* En cada llamada recursiva decrece el tamaño del árbol parámetro.

5.4 Comentarios finales

Una posible mejora de eficiencia consiste en tener calculado y guardado en todo momento el número de clientes no marcados satisfechos por cada destino. Eso convertiría a la operación `num_cl_sat` en trivial pero obligaría a modificar casi todas las demás operaciones del módulo `Cjt_Clientes` e, incluso, la operación `marcar_sat` empeoraría significativamente.

Por otra parte, notad que nuestro código para la operación `circuito_1` se basa fuertemente en el hecho de que dicha operación es una función, en particular, en que sus parámetros son de entrada. Así, por ejemplo, podemos suponer que después de la llamada recursiva con el hijo izquierdo, las marcas realizadas durante ese proceso no quedan registradas de cara a la llamada con el hijo derecho. Sin embargo, al pasar esta función a Java no podremos contar con esa propiedad y habrá que hacer una copia del conjunto de clientes antes de la llamadas para evitar este problema adicional. También habrá que añadir una operación para “limpiar” las marcas del conjunto de clientes antes de un cálculo de circuitos de cualquiera de los dos tipos, para no heredar las marcas de los cálculos realizados con anterioridad.