Pseudocódigo ayuda implementación algoritmo Ramificación y Poda

Consideraciones:

- En `main()` únicamente debéis definir la matriz **`B`** de beneficios y a continuación llamar a la función `AsignacionTrivial(B,s,&nNodos)` y `AsignacionPrecisa(B,s,&nNodos)`, donde `nNodos` debe representar el número de nodos explorados con valor inicial `0` y `s` el vector solución.
- Debéis usar el **TAD lista** dentro de cada una de estas funciones para representar la LNV. Podéis usar el utilizado para la práctica de Hash. En este caso, tal y como está en la diapositiva 80 de teoría, debéis definir la constante **`N`** (número de galeones y ciudades) y el **`NODO`** (o `TIPOELEMENTO`) de cada elemento de la lista **en el archivo `lista.h`**:

```
#define N 3
typedef struct {
    int n;                  //opcional, número de nodo
    int tupla[N];           //solución parcial del nodo
    int nivel;              //nivel del nodo
    int bact;               //beneficio actual
    int usadas[N];          //vector de ciudades asignadas para atacar (valores 0 o 1)
    float CI, BE, CS;       //cota inferior, beneficio estimado, cota superior
}TIPOELEMENTO;
typedef TIPOELEMENTO NODO;  //Para usar el nombre nodo en RyP
```

- Dentro de cada una de las funciones de asignación (`AsignacionTrivial()` y `AsignacionPrecisa()`) tendréis que realizar las inicializaciones necesarias (para el nodo raíz diapositiva 79, para el resto diapositivas 79-80):

```
////////////////////////////////////////////////////////////////////////////////
//INICIALIZACIONES
////////////////////////////////////////////////////////////////////////////////
TLISTA LNV;                  // Lista de nodos vivos
NODO raiz, x, y, s;          // Nodo raíz, nodo seleccionado (x) y nodo hijo de x (y)
float C=0;                   // Variable de poda

//Inicialización nodo raíz (diapositiva 79)
raiz.bact = 0;                  // No hay asignaciones. Beneficio acumulado es 0
raiz.nivel = -1;                // Primer nivel del árbol
raiz.tupla[N]={-1,-1,-1};       // tupla de solución sin ataques asignados
raiz.usadas[N]={0,0,0}          // todos los ataques están sin usar
CI(&raiz,B);                    // Cota inferior de raíz (trivial o precisa)
CS(&raiz,B);                    // Cota superior de raíz (trivial o precisa)
BE(&raiz);                      // Beneficio estimado de la raíz
raiz.n=1;                       // Nodo 1, si decidimos almacenar el número de nodo

//Inicialización bact nodo solución
s.bact=-1;
//Inicialización variable de poda (diapositiva 86)
C = raiz.CI;             // valor inicial variable de poda

//Inicialización lista de nodos vivos (diapositiva 86)
crearLista(&LNV);
// Guardo raíz como primer elemento de LNV
// Inserto siempre por el principio por estrategia LIFO: pila
// Una pila se puede representa mediante una lista en la que
// se inserta y suprime por el principio
insertarElementoLista(&LNV, primeroLista(LNV), raiz);
```

- Las funciones por desarrollar para utilizar dentro de `AsignacionTrivial()` y `AsignacionPrecisa()` son las siguientes (las cotas CI y CS para estimaciones triviales están definidos en la diapositiva 75 y las cotas para estimaciones precisas están definidas en las diapositivas 76 y 77):

```
// FUNCIONES PARA CALCULAR COTAS Y BENEFICIO
void BE(NODO *x);                          //beneficio estimado trivial-precisa
void CI_trivial(NODO *x);                  //cota inferior estimación trivial (D75)
void CS_trivial(NODO *x, int B[][N]);      //cota superior estimación trivial (D75)
void CI_precisa(NODO *x, int B[][N]);      //cota inferior estimación precisa (D76)
void CS_precisa(NODO *x, int B[][N]);      //cota superior estimación precisa (D77)

//FUNCIONES NECESARIAS PARA PROCEDIMIENTO DE RyP
int Solucion(NODO x);                     //determina si x es solución (D74)
NODO Seleccionar(TLISTA *LNV);            //Devuelve nodo según estrategia MB-LIFO y lo
                                          //elimina de la lista

//Funciones privadas necesarias para CI_precisa() y CS_precisa()
int _AsignacionVoraz(NODO x, int B[][N]); //Devuelve valor asignación voraz (D76)
int _MaximosTareas(NODO x, int B[][N]);   //Devuelve valor máximos ataques (D77)
//Función privada necesaria para calcular la solución voraz cuando CI==CS
NODO _SolAsignacionVoraz(NODO x, int B[][N]);
```

- La función `Seleccionar()` debe tener en cuenta que los nodos se insertan en la lista siempre por el principio, pues se sigue una estrategia LIFO pero, dentro de la lista, debéis seleccionar el primer nodo que encontréis con beneficio máximo, por tanto hay que recorrer la lista para buscarlo.
- Dentro del procedimiento descrito en la diapositiva 80 tenemos que generar los hijos de cada nodo `x` seleccionado. Esto está descrito en la diapositiva 73, en la que sustituimos la llamada a la función `Usada()` por el análisis del valor almacenado en el vector `x.usadas[]`:

```
AsignacionTrivial(int B[][N], NODO s)
   …
   //Inicialización
   …
   MIENTRAS LNV≠∅
      x = Seleccionar(LNV) //Selecciona x y lo elimina de LNV
      SI x.CS > C          //RAMIFICAMOS: GENERAMOS CADA HIJO
         PARA i = 0..N HACER             //PARA CADA HIJO y DE x
            y.nivel := x.nivel + 1
            y.tupla := x.tupla
            y.usadas := x.usadas
            SI NOT(x.usada[i]) ENTONCES   //NODO VÁLIDO
               y.tupla[y.nivel] := i      //Ciudad 'i' a galeón 'nivel'
               y.usadas[i] := 1
               y.bact := x.bact + B[y.nivel][i]
               //Calcular CI, CS y BE triviales para nodo y
               //Incrementar el número de nodos y almacenarlo en y.n
               …
               SI (Solución(y) AND y.bact>s.bact) ENTONCES //y es solución
                  s := y;
                  C := max(C,y.bact)
               SINO SI (NOT Solución(y) AND y.CS>C) ENTONCES
                  LNV := LNV + {y} //inserto por el principio (LIFO)
                  C := max(C, y.CI)
               FINSI
            FINSI
         FINPARA
      FINSI
   FINMIENTRAS
```

- En el caso de la estimación precisa de las cotas, hemos visto en el ejemplo de la diapositiva 95 que, cuando `CI==CS`, se aplica una solución voraz directa, por lo que hemos de contemplar ese caso en el algoritmo de la diapositiva del cuadro anterior añadiendo esto como primera condición (texto en azul) y cortando la exploración con un continue dado que desde ese punto ya se alcanza la solución voraz. También debemos considerar si ya la raíz cumple esto, dando lugar a una solución voraz desde el principio (segundo texto en azul, que podría evaluarse también antes del MIENTRAS, lo que evitaría meter la raíz en la LNV):

```
AsignacionPrecisa(int B[][N], NODO s)
   …
   //Inicialización
   …
   MIENTRAS LNV≠∅
      x = Seleccionar(LNV) //Selecciona x y lo elimina de LNV
      SI x.CS > C          //RAMIFICAMOS: GENERAMOS CADA HIJO
         PARA i = 0..N HACER             //PARA CADA HIJO y DE x
            y.nivel := x.nivel + 1
            y.tupla := x.tupla
            y.usadas := x.usadas
            SI NOT(x.usada[i]) ENTONCES   //NODO VÁLIDO
               y.tupla[y.nivel] := i      //Ciudad 'i' a galeón 'nivel'
               y.usadas[i] := 1
               y.bact := x.bact + B[y.nivel][i]
               //Incrementar el número de nodos y almacenarlo en y.n
               //Calcular CI, CS y BE precisas para nodo y
               …
               SI (NOT Solución(y) AND y.CS>=C AND y.CS==y.CI) ENTONCES
                  y=_SolAsignacionVoraz(y,B);
                  s:=y;
                  C:=max(C,y.bact);
                  continue;       //terminé la búsqueda por esta rama,
                                  //ya no analizo los demás hermanos
               SI (Solución(y) AND (y.bact>s.bact)) ENTONCES
                  s := y;
                  C := max(C,y.bact)
               SINO SI (NOT Solución(y) AND y.CS>C) ENTONCES
                  LNV := LNV + {y}   //inserto y en LNV
                  C := max(C, y.CI)
               FINSI
            FINSI
         FINPARA
      SINO SI (x.CS==C Y x.CS==x.CI) ENTONCES //nodo x seleccionado es solución voraz
         s=_SolAsignacionVoraz(x,B);
      FINSI
   FINMIENTRAS
```

- La función que calcula la solución voraz en el caso en que CI==CS es la siguiente:

```
NODO _SolAsignacionVoraz(NODO x, int B[][N])
    int Bmax // Para cada fila, beneficio máximo entre las ciudades no usadas
    int tmax // Para cada nivel, ciudad no usada de mayor beneficio

    PARA i = x.nivel+1..N-1 HACER              //PARA CADA nivel siguiente al actual
        //Busco en la fila i la ciudad con beneficio máximo no usada
        Bmax := -1
        PARA j=0..N-1 HACER                 //Pruebo ciudades j para galeón (nivel) i
            SI (NOT x.usadas[j] && B[i][j] > Bmax)
                Bmax := B[i][j]   // Se guarda el beneficio asociado a la asignación
                tmax := j         // Se guarda ciudad con beneficio máximo
            FINSI
        FINPARA
        // Actualizo x en nivel i con ciudad tmax con beneficio Bmax
        x.tupla[i] := tmax       // Guardo ciudad con Bmax para galeón i
        x.usadas[tmax] := 1      // La ciudad tmax fue usada
        x.bact := x.bact+Bmax    // Se actualiza el beneficio acumulado para el nodo
        //Incrementar el número de nodos y almacenarlo en x.n (si usamos este campo)
    FINPARA

    x.nivel := N - 1 // Necesario para que después pueda verificarse como solución
    DEVOLVER x       // Se devuelve el nodo hoja con la solución
```