TÉCNICAS AVANZADAS DE PROGRAMACIÓN

Contenido:

Técnicas Avanzadas de Programación

Backtracking, Programación Dinámica y Algoritmos Greedy

1. Introducción

Existen diferentes técnicas algorítmicas que pueden ser utilizadas para encontrar de manera rápida y eficiente soluciones algorítmicas a diferentes tipos de problemas, tomando en cuenta su estructura. Entre estas técnicas se tomarán en cuenta tres:

- Backtracking.
- Programación dinámica.
- Algoritmos greedy.

Durante el desarrollo de este tema se trabajará bajo el paradigma de programación imperativa.

2. Backtracking

El backtracking es un método sistemático para iterar entre todas las combinaciones posibles de un espacio de búsqueda (Skiena, 2003).

Backtracking o vuelta atrás es una técnica de programación que consigue la solución a un problema mediante la construcción de un grafo implícito de soluciones, de las cuales va descartando las que no son factibles para luego devolverse y continuar buscando hasta tener éxito.

2.1. Características

- La estructura implícita del backtracking se asemeja al recorrido en profundidad de un grafo dirigido.
- El grafo implícito es un árbol.
- Los nodos intermedios representan las soluciones parciales al problema.
- Es una técnica que hace uso de la recursividad.
- Es simple de implementar porque se basa en esquemas que están construidos, sin embargo, el desarrollo de las operaciones de los esquemas no es trivial.

2.2. Esquemas

La técnica del backtracking tiene tres posibles esquemas que se aplican según la naturaleza del problema:

a) Todas las soluciones.
b) Una solución.
c) La solución óptima.

Todas las Soluciones

```
proc buscar_todas_las_soluciones(T: paso)
inicio
    inicializar_alternativas()
    mientras ( existen_alternativas() ) hacer
         obtener_siguiente_alternativa()
         si ( es_alternativa_válida() ) entonces
              almacenar_paso()
              si ( es_solucion() ) entonces
                   procesar_solucion()
              sino
                   buscar_todas_las_soluciones(nuevo_paso)
              fsi
              borrar_paso()
         fsi
    fmientras
fproc
```

Una Solución

```
proc buscar_una_solucion(T: paso)
inicio
    inicializar_alternativas()
    mientras ( existen_alternativas() ∧ ¬sol_encontrada ) hacer
         obtener_siguiente_alternativa()
         si ( es_alternativa_válida() ) entonces
              almacenar_paso()
              si ( es_solución() ) entonces
                   sol_encontrada ← verdadero
                   procesar_solución()
              sino
                   buscar_una_solucion(nuevo_paso)
                   si ( ¬sol_encontrada ) entonces
                        borrar_paso()
                   fsi
              fsi
         fsi
    fmientras
fproc
```

La Solución Óptima

```
proc buscar_solucion_optima(T: paso)
inicio
    inicializar_alternativas()
    mientras ( existen_alternativas() ) hacer
         obtener_siguiente_alternativa()
         si ( es_alternativa_válida() ) entonces
              almacenar_paso()
              si ( es_solución() ) entonces
                   si ( mejor_que_actual() )
                        sustituir_solución()
                   fsi
              sino
                   buscar_solucion_optima(nuevo_paso)
              fsi
              borrar_paso()
         fsi
    fmientras
fproc
```

2.3. Problemas Clásicos

- Caminos mínimos en grafos (algoritmo de las ondas).
- El problema de las *n*-reinas.
- El problema del laberinto.
- El recorrido de un tablero de ajedrez con el caballo.

2.4. Ejemplo Completo

Problema: Búsqueda de caminos en un grafo.

Se tomará en cuenta el problema de búsqueda de caminos en un grafo, con diferentes enfoques, para ejemplificar en que caso se usa cada esquema de backtracking.

Búsqueda de todas las Soluciones

Planteamiento:

Dado un grafo dirigido, encontrar todos los caminos posibles entre dos vértices dados (vértice inicial y vértice final) de manera tal que los vértices contenidos en cada camino sean visitados una sola vez.

Solución:

a) Identificar los elementos principales en el problema.

Esquema a utilizar: Búsqueda de todas las soluciones (se pide encontrar todos los caminos posibles entre dos vértices dados del grafo).
Paso: Partiendo de un vértice cualquiera *v*, visitar un vértice adyacente *w*.
Solución: Cuando el vértice visitado (*w*) coincide con el vértice final (v_f).

b) Especificar las constantes, tipos y variables requeridas.

```
const
      N ← ? // N representa el número de vértices del grafo.
tipo
      vertice = entero
      { Se especifica este tipo suponiendo que los vértices del grafo
      pueden ser etiquetados con un número entero entre 1 y n }
      adyacencias = arreglo [1..n, 1..n] de real
      { Matriz de adyacencias que representa el grafo }
      visitado = arreglo [1..n] de lógico
      { Arreglo que indica si los vértices han sido visitados o no }
      camino = arreglo [1..n] de vertice
      { Arreglo en el que se almacena el camino que se construye }
var
      vertice: vi, vf   // Vértices inicial y final
```

```
        adyacencias: ady  // Matriz de adyacencias
        visitado: vis     // Vector de marca para los vértices visitados
        camino: c         // Vector que almacena el camino resultante
        entero: t         // Tamaño del camino en un momento determinado
```

c) Adaptar el esquema general a utilizar al problema particular.

```
// t = 0
proc buscar_todos_los_caminos(vertice: v)
var
      vertice: k
inicio
      k ← 0
      mientras (k ≠ n) hacer
            k ← k + 1
            si (visitable(v, k)) entonces
                  agregar_vertice(k)
                  si (k = vf) entonces
                        escribir_camino(c)
                  sino
                        buscar_todos_los_caminos(k)
                  fsi
                  eliminar_vertice(k)
            fsi
      fmientras
fproc
```

d) Definir las funciones y procedimientos necesarios.

```
func visitable(vertice: v, w) : logico
inicio
      retornar (ady[v, w] ≠ 0  ∧ ¬vis[w])
ffunc

proc agregar_vertice(vertice: v)
inicio
      c[t] ← v
      vis[v] ← verdadero
      t ← t + 1
fproc

proc eliminar_vertice(vertice : v)
inicio
      t ← t – 1
      c[t] ← 0
      vis[v] ← falso
fproc
```

Búsqueda de una Solución

Planteamiento:

Suponga que el problema es reformulado en los siguientes términos: Dado un grafo dirigido, encontrar algún camino entre dos vértices dados (vértice inicial y vértice final) de manera tal que los vértices contenidos en cada camino sean visitados una sola vez.

Solución:

a) Identificar los elementos principales en el problema.

Esquema a utilizar: Búsqueda de una solución (se pide encontrar un camino cualquiera entre dos vértices dados del grafo).
Paso: Partiendo de un vértice cualquiera *v*, visitar un vértice adyacente *w*.
Solución: Cuando el vértice visitado (*w*) coincide con el vértice final (v_f).

b) Especificar las constantes, tipos y variables requeridas.

Se utilizan las mismas constantes, tipos y variables que en el primer caso. Adicionalmente se requiere la siguiente variable:

```
var
      // ...
      logico: encontrado // Indica si ya se encontró un camino.
```

c) Adaptar el esquema general a utilizar al problema particular.

```
// t = 0
proc buscar_un_camino(vertice: v)
var
      vertice: k
inicio
      k ← 0
      mientras (k ≠ n ∧ ¬encontrado) hacer
            k ← k + 1
            si (visitable(v, k)) entonces
                  agregar_vertice(k)
                  si (k = vf) entonces
                        encontrado ← verdadero
                        escribir_camino(c)
                  sino
                        buscar_un_camino(k)
                        si(¬encontrado) entonces
                              eliminar_vertice(k)
                        fsi
                  fsi
            fsi
      fmientras
fproc
```

d) Definir las funciones y procedimientos necesarios.

Se mantienen las definiciones de las funciones y procedimientos utilizados en el caso anterior.

Búsqueda de la Solución Óptima

Planteamiento:

Suponga que el problema es reformulado en los siguientes términos: Dado un grafo dirigido y el costo de recorrer un arco, encontrar el camino de menor costo entre dos vértices dados (vértice inicial y vértice final) de manera tal que los vértices contenidos en cada camino sean visitados una sola vez. Nótese que si el costo de todos los arcos del grafo es 1 (grafo no pesado) el camino de costo mínimo resultante es el de longitud mínima.

Solución:

a) Identificar los elementos principales en el problema.

Esquema a utilizar: Búsqueda de la solución óptima (en este caso se tiene la función de costo asociada a un camino, la cual debe ser minimizada y el objetivo es encontrar el camino para el cual el valor de la función es el mínimo).
Paso: Partiendo de un vértice cualquiera *v*, visitar un vértice adyacente *w*.
Solución: Cuando el vértice visitado (*w*) coincide con el vértice final (v_f).

b) Especificar las constantes, tipos y variables requeridas.

Se utilizan las mismas constantes, tipos y variables que en el caso anterior. Adicionalmente se requieren los siguientes tipos y variables:

```
tipo
      // ...
      arreglo [1..N] de vértice: camino_optimo
      { Almacena el camino cuyo costo es costo_optimo }

var
      // ...
      real: costo_actual, costo_optimo
      { Variables que representan el costo de recorrer el camino
      actual y camino de menor costo conseguido hasta el momento,
      respectivamente }
```

c) Adaptar el esquema general a utilizar al problema particular.

```
// t = 0, costo_actual = 0 y costo_optimo = ∞
proc buscar_camino_optimo(vertice: v)
var
      vertice: k
inicio
      k ← 0
```

```
        mientras (k ≠ n) hacer
              k ← k + 1
              si (visitable(v, k)) entonces
                    agregar_vertice(v, k)
                    si (k = vf) entonces
                          si(costo_actual < costo_optimo) entonces
                                sustituir_camino()
                          fsi
                    sino
                          buscar_camino_optimo(k)
                    fsi
                    eliminar_vertice(v, k)
              fsi
        fmientras
  fproc
```

d) Definir las funciones y procedimientos necesarios.

Las definiciones de las funciones y procedimientos utilizados en el primer caso se modifican de la siguiente manera:

```
proc agregar_vertice(vertice: u, v)
inicio
      c[t] ← v
      costo_actual ← costo_actual + ady[u, v]
      vis[v] ← verdadero
      t ← t + 1
fproc

proc eliminar_vertice(vertice : u, v)
inicio
      t ← t - 1
      c[t] ← 0
      costo_actual ← costo_actual – ady[u, v]
      vis[v] ← falso
fproc

proc sustituir_camino()
var
      entero: i
inicio
      costo_optimo ← costo_actual
      para i ← 1 hasta t – 1 hacer
            camino_optimo[i] ← c[i]
      fpara
fproc
```

3. Programación Dinámica

Para hablar de programación dinámica, hay que hablar primero del principio de optimalidad de Bellman:

"*Dada una secuencia óptima de decisiones, toda subsecuencia de ella es, a su vez, óptima*"

La programación dinámica es una técnica basada en el principio de optimalidad de Bellman, la cual se aplica en problemas que pueden ser divididos en subproblemas, que son resueltos para luego combinar sus soluciones hasta obtener la solución del caso problema general, evitando calcular varias veces el mismo subproblema.

Se utiliza típicamente en problemas de optimización.

3.1. Elementos

Subestructuras óptimas: La solución óptima de un problema contiene las soluciones óptimas de sus subproblemas.

Subproblemas superpuestos: Se da cuando para resolver un problema algún subproblema debe ser recalculado.

Memorización: Se guardan los resultados parciales para acelerar los cálculos.

Reconstrucción de una solución óptima: Se guardan las decisiones hechas en cada subproblema. Es opcional.

3.2. Enfoques

Bottom-Up: Todos los subproblemas son resueltos de antemano para después decidir cuál subproblema se utilizará para obtener la solución del problema general.

Top-Down: Se aborda el problema general, el cual se divide en subproblemas que son resueltos almacenando las soluciones en caso de que sean necesarias nuevamente. Es una combinación de memorización y recursión.

Ejemplos:

```
// Función de Fibonacci desarrollada bajo el enfoque Bottom-Up
func fibonacci( entero: n; ref arreglo [0..n] de entero: memo ): entero
var
   entero: i
inicio
   si ( n ≤ 1 ) entonces
         retornar( n )
   sino
         memo[0] ← 0
         memo[1] ← 1
         para i ← 2 hasta n hacer
               memo[i] ← memo[i – 1] + memo[i – 2]
         fpara
         retornar( memo[n] )
   fsi
ffunc
```

```
// Función de Fibonacci desarrollada bajo el enfoque Top-Down
func fibonacci( entero: n; ref arreglo [0..n] de entero: memo ): entero
inicio
   si ( n ≤ 1 ) entonces
         retornar( n )
   sino
         si( memo[n – 1] = -1 ) entonces
               memo[n – 1] = fibonacci(n – 1, memo)
         fsi
         si( memo[n – 2] = -1 ) entonces
               memo[n – 2] = fibonacci(n – 2, memo)
         fsi
         memo[n] = memo[n – 1] + memo[n – 2]
         retornar( memo[n] )
   fsi
ffunc
```

3.3. Problemas Clásicos

- El problema de las monedas.
- El problema de la mochila.
- Búsqueda de caminos mínimos en un digrafo pesado (Algoritmo de Floyd-Warshall).

3.4. Ejemplo Completo

Problema: Se dispone de un conjunto finito M = $\{m_1, m_2, \ldots, m_n\}$ de tipos de monedas, donde cada m_i es un número natural que indica el valor de las monedas de tipo *i*, y se cumple $m_1 < m_2 < \ldots < m_n$. Suponiendo que la cantidad de monedas de cada tipo es ilimitada, se quiere pagar de forma exacta una cantidad $C > 0$ utilizando el menor número posible de monedas.

Normalmente para dar solución al problema se irían tomando monedas de mayor a menor denominación, tomando en cuenta las siguientes posibilidades:

1. Si el valor de la moneda m_i supera la cantidad *C* a pagar, se deben considerar para pagar *C* las monedas de menor denominación.
2. Si el valor de la moneda m_i es menor o igual que *C*, se tienen dos opciones:
   - 2.1. No tomar ninguna moneda de tipo *i* y considerar las monedas de menor denominación (igual al caso anterior).
   - 2.2. Tomar una moneda de tipo *i* y pagar el resto $C - m_i$ considerando nuevamente las monedas de los tipos 1 a *i*.

Estas posibilidades pueden ser resumidas en la siguiente función recursiva:

$$\text{monedas}(i, j) = \begin{cases} \text{monedas}(i-1, j) & \text{si } m_i > j \\ \min\{\text{monedas}(i-1, j), \text{monedas}(i, j-m_i) + 1\} & \text{si } m_i \leq j \end{cases}$$

Donde *i* es el tipo de moneda considerada y *j* es la cantidad restante a pagar. La solución vendría representada por monedas(*n*, *C*).

Los casos base de la recurrencia son los siguientes:

1. Se han considerado todos los tipos de moneda pero aún se tiene que pagar una cantidad mayor a 0.
2. No falta nada por pagar.

Ahora bien, se van calculando los valores de la recurrencia desde los casos base hasta encontrar la solución, guardando las soluciones intermedias en una tabla.

Ejemplo: Tomemos en cuenta que M = {1, 4, 6} y que se quieren devolver 8 unidades.

M\monedas	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
$m_0 = 0$	0	∞	∞	∞	∞	∞	∞	∞	∞
$m_1 = 1$	0								
$m_2 = 4$	0								
$m_3 = 6$	0								

Para cubrir el caso base 1, se coloca en la tabla una fila correspondiente a un tipo de moneda ficticio m_0, como en este caso no habría solución, se colocará como valor ∞ ya que se quiere minimizar el número de monedas.

Para cubrir el caso base 2, se coloca en la tabla una primera columna con valores 0, para representar que ya se ha pagado el monto completo.

Al calcular la primera fila se obtiene lo siguiente:

$monedas_{11} = \min\{\infty, 0 + 1\}$
$monedas_{12} = \min\{\infty, 1 + 1\}$
$monedas_{13} = \min\{\infty, 2 + 1\}$
$monedas_{14} = \min\{\infty, 3 + 1\}$
$monedas_{15} = \min\{\infty, 4 + 1\}$
$monedas_{16} = \min\{\infty, 5 + 1\}$
$monedas_{17} = \min\{\infty, 6 + 1\}$
$monedas_{18} = \min\{\infty, 7 + 1\}$

M\monedas	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
$m_0 = 0$	0	∞	∞	∞	∞	∞	∞	∞	∞
$m_1 = 1$	0	1	2	3	4	5	6	7	8
$m_2 = 4$	0								
$m_3 = 6$	0								

Al calcular la segunda fila se obtiene lo siguiente:

$monedas_{21} = 1$
$monedas_{22} = 2$
$monedas_{23} = 3$
$monedas_{24} = \min\{4, 0 + 1\}$
$monedas_{25} = \min\{5, 1 + 1\}$
$monedas_{26} = \min\{6, 2 + 1\}$
$monedas_{27} = \min\{7, 3 + 1\}$
$monedas_{28} = \min\{8, 1 + 1\}$

M\monedas	0	1	2	3	4	5	6	7	8
$m_0 = 0$	0	∞	∞	∞	∞	∞	∞	∞	∞
$m_1 = 1$	0	1	2	3	4	5	6	7	8
$m_2 = 4$	0	1	2	3	1	2	3	4	2
$m_3 = 6$	0								

Al calcular la tercera fila se obtiene lo siguiente:

$monedas_{31} = 1$
$monedas_{32} = 2$
$monedas_{33} = 3$
$monedas_{34} = 1$
$monedas_{35} = 2$
$monedas_{36} = \min\{3, 0 + 1\}$
$monedas_{37} = \min\{4, 3 + 1\}$
$monedas_{38} = \min\{2, 1 + 1\}$

M\monedas	0	1	2	3	4	5	6	7	8
$m_0 = 0$	0	∞	∞	∞	∞	∞	∞	∞	∞
$m_1 = 1$	0	1	2	3	4	5	6	7	8
$m_2 = 4$	0	1	2	3	1	2	3	4	2
$m_3 = 6$	0	1	2	3	1	2	1	2	2

Como se puede ver, la solución a este problema sería dar 2 monedas de 4.

Algoritmo:

```
const
   N ← ?       // Cantidad máxima de monedas
   C ← ?       // Cantidad máxima a regresar

tipo
   vector = arreglo [1..N] de entero
   tabla = arreglo [0..N, 0..C] de entero


proc inicializar(ref tabla: monedas)
var
   entero: i
inicio
   para i ← 0 hasta N hacer
         monedas[i, 0] ← 0
   fpara
   para i ← 0 hasta C hacer
         monedas[0, i] ← ∞
   fpara
fproc

func cantidad_monedas(vector: M; Entero: C): entero
var
   tabla: monedas
   entero: i
```

```
inicio
   inicializar(monedas)
   para i ← 1 hasta N hacer
         para j ← 1 hasta C hacer
               si(M[i] > j) entonces
                     monedas[i, j] ← monedas[i – 1, j]
               sino
                     monedas[i, j] ← min( monedas[i – 1, j],
                                          monedas[i, j – M[i]] + 1 )
               fsi
         fpara
   fpara
   retornar(monedas[N, C])
ffunc
```

Se puede ver que durante la construcción de la tabla, al ir construyendo la fila *i* sólo se necesita el valor de la fila anterior, *i* – 1, en la misma columna y un valor que se encuentra en la misma fila, por lo tanto se puede optimizar el uso de estructuras de datos si en vez de una tabla se usa un vector. Asimismo, el segundo ciclo se podría comenzar desde la columna correspondiente al valor m_i de la moneda que se esté procesando, dado que, como se puede ver en el ejemplo anterior, el número de monedas para armar cantidades inferiores a m_i se mantiene.

De acuerdo a estas optimizaciones, el algoritmo queda como sigue:

```
tipo
   vector = arreglo [1..N] de entero
   tabla = arreglo [0..C] de entero

proc inicializar(ref tabla: monedas)
var
   entero: i
inicio
   monedas[0] ← 0
   para i ← 1 hasta C hacer
         monedas[i] ← ∞
   fpara
fproc

func cantidad_monedas(vector: M; Entero: C): entero
var
   tabla: monedas
   entero: i
inicio
   inicializar(monedas)
   para i ← 1 hasta N hacer
         para j ← 1 hasta C hacer
               monedas[j] ← min( monedas[j], monedas[j – M[i]] + 1 )
         fpara
   fpara
   retornar(monedas[C])
ffunc
```

Ahora bien, hasta ahora solamente se tiene el número mínimo de monedas, más no la solución, es decir, cuantas monedas de cada tipo se necesitan para devolver la cantidad C. Esta información se puede calcular con los resultados dados en la tabla, verificando la siguiente ecuación:

$$\text{monedas}(j) = \text{monedas}(j - m_i) + 1$$

Si la ecuación es cierta, eso significa que se ha tomado una moneda de la denominación *i* para llegar a esa solución, por lo tanto se debe contar; sino, no se han tomado monedas de esa denominación y se deben considerar monedas de menor denominación.

```
{Pre: monedas[C] ≠ ∞}
proc calcular_monedas(vector: M; tabla: monedas; ref vector: cuantas)
var
   entero: i, j
inicio
   para i ← 1 hasta N hacer
         cuantas[i] ← 0
   fpara
   i ← N // Cantidad de monedas
   j ← C // Cantidad a pagar
   mientras(j > 0) hacer // No se ha pagado todo
         si(M[i] ≤ j ∧ monedas[j] = monedas[j – M[i]] + 1) entonces
               cuantas[i] = cuantas[i] + 1
               j = j – M[i]
         sino
               i ← i - 1
         fsi
   fmientras
fproc
```

4. Algoritmos Greedy

Los algoritmos greedy o algoritmos voraces son aquellos que toman las decisiones que parecen ser las mejores en un primer momento, basándose en la información disponible de modo inmediato, sin considerar los efectos de dichas decisiones en el futuro.

4.1. Características

- Selecciona el elemento más ventajoso en un momento.
- No reconsidera sus decisiones.
- Son fáciles de desarrollar.
- Son óptimos.
- Son típicamente utilizados en problemas de optimización.

4.2. Elementos

Conjunto de Candidatos: Son los elementos que considera el algoritmo para encontrar una solución.

Función Solución: Verifica si cierto conjunto de candidatos constituye una solución al problema.

Función de Factibilidad: Comprueba si cierto conjunto de candidatos es factible, esto es si al conjunto se le pueden agregar otros candidatos para obtener una solución.

Función de Selección: Indica el candidato más prometedor que no ha sido considerado.

Función Objetivo: Calcula el valor de la solución encontrada. No aparece de forma explícita en el algoritmo greedy.

4.3. Esquema

```
// Entrada: C, es el conjunto de candidatos.
// Salida: S, es el conjunto solución.

func Greedy(conjunto: C): conjunto
var
   conjunto: S
   elemento: x
inicio
      S ← ∅
      mientras ( C ≠ ∅ ∧ ¬es_solucion(S) ) hacer
         x ← seleccionar_candidato(C)
         C ← C - {x}
         si ( es_factible(S ∪ {x}) ) entonces
               S ← S ∪ {x}
         fsi
      fmientras
      si ( es_solucion(S) ) entonces
         retornar (S)
      sino
         retornar (∅)
      fsi
ffunc
```

4.4. Problemas Clásicos

- El problema de las monedas.
- El problema de la mochila.
- El problema del árbol cobertor de costo mínimo (Algoritmos de Kuskal y Prim).
- El problema de los caminos mínimos (en costo o longitud) de un digrafo pesado (Algoritmo de Dijkstra).

4.5. Ejemplo Completo

Problema: Se dispone de un conjunto finito M = $\{m_1, m_2, \ldots, m_n\}$ de tipos de monedas, donde cada m_i es un número natural que indica el valor de las monedas de tipo i, y se cumple $m_1 < m_2 < \ldots < m_n$. Suponiendo que la cantidad de monedas de cada tipo es ilimitada, se quiere pagar de forma exacta una cantidad $C > 0$ utilizando el menor número posible de monedas.

Algoritmo:

```
const
   N ← ?       // Cantidad máxima de monedas

tipo
   vector = arreglo [1..N] de entero

func cantidad_monedas(vector: M, entero: C; ref vector: cuantas): entero
var
   entero: resto, i, cantidad
inicio
   para i ← 0 hasta N hacer
         cuantas[i] ← 0
   fpara
   cantidad ← 0
   resto ← C
   i ← N
   mientras (resto ≠ 0 ∧ i > 0) hacer
         cuantas[i] ← resto div M[i]
         cantidad ← cantidad + cuantas[i]
         resto ← resto mod M[i]
         i ← i - 1
   fmientras
   si(resto ≠ 0) entonces
         cantidad ← -1    // Marca en caso de no obtener una solución
   fsi
   retornar(cantidad)
ffunc
```

Vale la pena destacar que con esta técnica, dependiendo del tipo de problema, podría encontrarse una solución no optima o podría no encontrarse solución alguna.

5. Referencias

- Brassard, G. y Bratley, P. (1997). Fundamentos de Algoritmia. Madrid, España: Prentice-Hall.
- Cormen, T., Leiserson, C., Rivest, R. y Stein Clifford. (2001). Introduction to Algorithms (Segunda Edición). Massachusetts, USA: McGraw-Hill.
- Martínez, A. A. y Rosquete, D. H. (2009). NASPI: Una Notación Algorítmica Estándar para la Programación Imperativa. Télématique. 8(3). 55 – 74.
- Ottogalli, K., Martínez, A. y León L. (2011). NASPOO: Una Notación Algorítmica Estándar para la Programación Orientada a Objetos. Télématique. 10(1). 81 – 102.
- Skiena, S. y Revilla, M. (2003). Programming Challenges: The Programming Contest Training Manual. New York, USA: Springer-Verlag.