1. (Algoritmos voraces) Un amigo te recomienda que entres en el mundo del trading de cripto que siempre vas a ganar, ya que tiene una bola de cristal que ve el futuro.

   Conocés el valor actual $v_1^0, \ldots, v_n^0$ de n criptomonedas. La bola de cristal indica el valor que criptomonedas durante los m días siguientes. Es decir, los valores $v_1^1, \ldots, v_1^m$ que tendrá la de 1 día, ..., dentro de m días respectivamente; los valores $v_2^1, \ldots, v_2^m$ que tendrá la cript día, ..., dentro de m días respectivamente, etcétera. En general, v_i^j es el valor que tendrá la de j días.

   Con esta preciada información podés diseñar un algoritmo que calcule el máximo dinero po de m días comprando y vendiendo criptomonedas, a partir de una suma inicial de dinero D

   Se asume que siempre habrá suficiente cantidad de cada criptomoneda para comprar y qu alguna por la compra y venta. También se asume que se pueden comprar fracciones de cr que no siempre las criptomonedas incrementan su valor.

   Se pide lo siguiente:

   (a) Indicar de manera simple y concreta, cuál es el criterio de selección voraz para constru

   (b) Indicar qué estructuras de datos utilizarás para resolver el problema.

   (c) Explicar en palabras cómo resolverá el problema el algoritmo.

   (d) Implementar el algoritmo en el lenguaje de la materia de manera precisa.

a) Se selecciona la criptomoneda que mayor ganancia porcentual tenga al siguiente día
b) **type** Crypto = **tuple**
   id: nat
   compra: float
   venta: float
   end tuple

   V: array[1..n,0..m] **of** float

c) Se selecciona la criptomoneda que mayor ganancia porcentual tenga al siguiente día, luego, dependiendo de la cantidad de dinero D, se compra tanto de esa cripto como se pueda para vender al siguiente día, lo mismo para los m-1 días.

```
fun cripto (V: array[1..n,0..m] of nat, D: float) ret S: float
      var cripto_aux: Crypto
      var ganancia: float
      var fracción: float

      S := D
      for day := 0 to m-1 do
            ganancia := 0.0
            fracción := 0.0
```

```
            cripto_aux = elegir_cripto(V,day)
            if( S ≥ cripto_aux.compra and S != 0) then
                  S := S - cripto_aux.compra
                  ganancia := ganancia + cripto_aux.venta
            else if (cripto_aux.compra > S and S != 0) then
                  fracción := S / cripto_aux.compra
                  ganancia := ganancia + fracción * cripto_aux.venta
            fi
            S := S + ganancia
      od
end fun

fun elegir_cripto(V: array[1..n,0..m] of nat, day: nat) ret result:
Crypto
      var cripto_aux: Crypto
      var max_aux: float
      var temp_max: float

      max_aux := -inf
      for i := 1 to n do
            if(V[i,day] > V[i,day+1]) then
                  temp_max := (V[i,day+1] - V[i,day] * 100) / V[i,day]
                  if (temp_max > max_aux) then
                        max_aux := temp_max
                        result.id := i
                        result.compra := V[i,day]
                        result.venta := V[i,day+1]
                  fi
            fi
      od
end fun
```

2. Finalmente tenés la posibilidad de irte N días (con sus respectivas noches) de vacacione armaste, cada día/noche i estarás en una ciudad C_i. Contás con M pesos en total de pres alojamiento y para cada ciudad conocés el costo k_i por noche del único hotel que tiene. Ca entre dormir en el hotel de la ciudad, lo que te costará k_i, o dormir en una carpa que lle Además, tenés una tabla que indica para cada ciudad i, la puntuación p_i del hotel.

   Se debe encontrar la máxima puntuación obtenible eligiendo en qué ciudades dormirás en h el presupuesto total gastado no supere el monto M. Notar que si decidís dormir en carp puntuación correspondiente para la misma será 0.

   (a) (Backtracking) Resolvé el problema utilizando la técnica de backtracking dando una ello:

   - Especificá precisamente qué calcula la función recursiva que resolverá el problem mentos toma y la utilidad de cada uno.
   - Da la llamada o la expresión principal que resuelve el problema.
   - Definí la función en notación matemática.

   (b) (Programación dinámica) Implementá un algoritmo que utilice Programación Dinámic lema.

   - ¿Qué dimensiones tiene la tabla que el algoritmo debe llenar?
   - ¿En qué orden se llena la misma?
   - ¿Se podría llenar de otra forma? En caso afirmativo indique cuál.

a)

Función recursiva:

dormir(i, m) “Máximo puntaje obtenible decidiendo si dormir o no en la ciudad i sin superar el monto m ”

Llamada principal:

dormir(n,M)

Función Matemática:

$$
\text{dormir}(i,m) \begin{cases} 0 & \text{, si } i = 0 \vee m = 0 \\ \text{dormir}(i-1,m) & \text{, si } i > 0 \,\&\, m > 0 \,\&\, k_i > m \\ \max(\text{dormir}(i-1,m),\ p_i + \text{dormir}(i-1,\ m-k_i)) & \text{, si } i > 0 \,\&\, m > 0 \,\&\, m > k_i \end{cases}
$$

b)

Dimensiones:

La tabla será de 2 dimensiones

Orden:

Se llena de arriba hacia abajo y de izquierda a derecha. Se puede llenar de arriba hacia abajo y de derecha a izquierda.

```
fun dormir(p: array[1..n] of nat, k: array[1..n] of float, M:
float) ret puntaje: nat
      {- variables -}
      var dp: array[0..n,0..M] of nat

      {- casos base -}
      for i := 0 to n do
            dp[i,0] := 0
      od
      for j := 0 to M do
            dp[0,j] := 0
      od
      {- caso recursivo -}
      for i := 1 to n do
            for j := 1 to M do
                  if(k[i] > j) then
                        dp[i,j] := dp[i-1,j]
                  else
                        dp[i,j] := max(dp[i-1,j], p[i] + dp[i-1,
j - k[i]])
                  fi
            od
      od
      puntaje := dp[n,M]
end fun
```

3. Para cada uno de los siguientes algoritmos determinar **por separado** cada uno de los sigu

   (a) ¿Qué hace? ¿Cuáles son las precondiciones necesarias para haga eso?

   (b) ¿Cómo lo hace?

   (c) El orden del algoritmo, analizando los distintos casos posibles.

   (d) Proponer nombres más adecuados para los identificadores (de variables, funciones y p

```
fun s(p: array[1..n] of nat, v,w: nat) ret y: nat
    y:= v
    for i := v+1 to w do
        if p[i] < p[y] then y:= i fi
    od
end fun

fun t(p: array[1..n] of nat, v,w: nat) ret y: nat
    y:= v
    for i := v+1 to w do
        if p[y] < p[i] then y:= i fi
    od
end fun

proc r(p: array[1..n] of nat)
     for i := 1 to n div 2 do
         swap(p, i, s(p, i, n-i+1));
         swap(p, n-i+1, t(p, i+1, n-i+1));
     od
end fun
```

Algoritmo 1:

```
fun s(p: array[1..n] of nat, v,w: nat) ret y: nat
     y:= v
     for i := v+1 to w do
          if p[i] < p[y] then
               y:= i
          fi
     od
end fun
```

¿Qué hace?

Toma un arreglo y dos índices que indican un segmento, devuelve la posición del mínimo elemento dentro de ese segmento.

¿Cómo lo hace? ¿Precondiciones?

Recorre el arreglo de izquierda a derecha y en cada paso compara los elementos a partir del indicado por el primer índice, siempre que encuentre un elemento menor, se actualiza el resultado y seguimos con el siguiente hasta llegar al último índice indicado.

{- Pre: v,w ≤ n -}

Orden del algoritmo

O(w-v)

Nombre:

indice_del_minimo

Algoritmo 2:

```
fun t(p: array[1..n] of nat, v,w: nat) ret y: nat
      y:= v
      for i := v+1 to w do
            if p[y] < p[i] then
                  y:= i
            fi
      od
end fun
```

¿Qué hace?

Toma un arreglo y dos índices que indican un segmento, devuelve la posición del máximo elemento dentro de ese segmento.

¿Cómo lo hace? ¿Precondiciones?

Recorre el arreglo de izquierda a derecha y en cada paso compara los elementos a partir del indicado por el primer índice, siempre que encuentre un elemento mayor, se actualiza el resultado y seguimos con el siguiente hasta llegar al último índice indicado.

{- Pre: v,w ≤ n -}

Orden del algoritmo

O(w-v)

Nombre:

indice_del_mayor

Algoritmo 3:

```
proc r(p: array[1..n] of nat)
      for i := 1 to n div 2 do
            swap(p, i, s(p, i, n-i+1));
            swap(p, n-i+1, t(p, i+1, n-i+1));
      od
end fun
```

¿Qué hace?

Ordena un arreglo.

¿Cómo lo hace? ¿Precondiciones?

Recorre la mitad del arreglo de izquierda a derecha, para cada posición busca el mínimo en adelante con la función s, una vez encontrado los intercambia, pasa a la siguiente posición y busca el máximo con la función t y lo manda al final del arreglo.

Luego lo mismo para todo el arreglo menos los elementos ya ubicados al principio y al final del mismo.
{PRE: n = 2k}

Orden del algoritmo

El bucle de la función r se ejecuta n div 2 veces y por cada iteración tenemos una complejidad de O(n - i + 1 - i) = O(n - 2i + 1) y O(n - i + 1 - i - 1) = O(n - 2i) pero nos quedamos con O(n - 2i + 1) por tener una complejidad mayor, por lo que la complejidad total es de:
O(n/2) * O(n - 2i + 1) ≡ O(n) * O(n) = O(n^2)

Nombre:

ordenar_arreglo

4. Considere la siguiente especificación del tipo Listas de algún tipo T.

```
spec List of T where

constructors
    fun empty() ret l : List of T
    {- crea una lista vacía. -}

    proc addl (in e : T, in/out l : List of T)
    {- agrega el elemento e al comienzo de la lista l. -}

destroy
    proc destroy (in/out l : List of T)
    {- Libera memoria en caso que sea necesario. -}

operations
    fun is_empty(l : List of T) ret b : bool
    {- Devuelve True si l es vacía. -}

    fun head(l : List of T) ret e : T
    {- Devuelve el primer elemento de la lista l -}
    {- PRE: not is_empty(l) -}

    proc tail(in/out l : List of T)
    {- Elimina el primer elemento de la lista l -}
    {- PRE: not is_empty(l) -}

    proc addr (in/out l : List of T,in e : T)
    {- agrega el elemento e al final de la lista l. -}

    fun length(l : List of T) ret n : nat
    {- Devuelve la cantidad de elementos de la lista l -}

    proc concat(in/out l : List of T
    {- Agrega al final de l todos los e
       en el mismo orden.-}

    fun index(l : List of T,n : nat) r
    {- Devuelve el n-ésimo elemento
    {- PRE: length(l) > n -}

    proc take(in/out l : List of T,i
    {- Deja en l sólo los primeros n
       elementos, eliminando el resto

    proc drop(in/out l : List of T,i
    {- Elimina los primeros n eleme

    fun copy_list(l1 : List of T) ret
    {- Copia todos los elementos de
```

(a) Utilizando como representación un arreglo de N elementos de tipo T y un natural, defi
representa la lista, implementá los constructores y las operaciones tail, concat, length y d

(b) ¿La representación elegida para implementar el tad tiene alguna limitación? En caso afi
es. ¿Alguna operación debe tener una precondición extra?

(c) ¿Qué orden tiene la operación tail implementada en el inciso anterior? ¿Se podría modifi
datos utilizada para que tail sea constante? Explicá cómo.

(d) Implementá una función que reciba dos listas de enteros y devuelva otra lista que conten
pares todos los elementos de la primera lista (en el mismo orden), y en las posicione
elementos de la segunda lista (en el mismo orden).
Para ello utilizá el tipo **abstracto**, sin acceder a su representación interna.

a)

```
Implement List of T where

      type List of T = tuple
                           elems: array[1..N] of T
                           size: nat
                       end tuple

      Constructors

            fun empty() ret l : List of T
                  l.size := 0
            end fun

            proc addl(in e : nat, in/out l : List of T)
                  for i := n down to 1 do
                        l.elem[i+1] := l.elem[i]
                  od
                  l.elems[1] := e
                  l.size := l.size + 1
            end proc


end proc

      Operations

            {- PRE: nos is_empty(l) -}
            proc tail(in/out l : List of T)
                  for i := 1 to l.size - 1 do
                        l.elems[i] := l.elems[i+1]
                  od
                  l.size := l.size - 1
```

```
end proc

proc concat(in/out l : List of T, in l0 : List of T)
      if not is_empty(l0) then
            if is_empty(l) then
                  l.size := l0.size
                  for i := 1 to l0.size
                        l.elems[i] := l0.elems[i]
                  do
            else
                  var j,k: nat
                  j := l.size + 1
                  k := 1
                  l.size := l.size + l0.size
                  for i := j to l.size
                        l.elems[i] := l0.elems[k]
                        k := k + 1
                  do
            fi

      fi
end proc

fun length(l : List of T) ret n : nat
      n := l.size
end fun

proc drop(in/out l : List of T, in n: nat)
      if (not is_empty(l)) then
            var j: nat
            j := 1
            for i := n + 1 to l.size do
                  l.elems[j] := l.elems[i]
                  j := j + 1
            od
            l.size := l.size - n
      fi
end proc
```

b)

Limitación:

l.size ≤ N en todo momento

Precondición necesaria:

```
proc drop(in/out l : List of T, in n: nat)
{- PRE: n ≤ length(l) -}
```

c)

Es de orden O(n) con n = length(l) - 1 - i. Se puede modificar la estructura de datos tq:

```
type Node of T = tuple
                     elem: T
                     next: pointer to (Node of T)
                 end tuple

type List of T = tuple
                     first: pointer to (Node of T)
                     size: nat
                 end tuple
```

Teniendo la referencia al primer elemento de la lista enlazada, solo basta con eliminar ese nodo y asignar el puntero first al siguiente elemento.

d)

```
{PRE: lenght(l0) == length(l)}
{- Los índices van de 1..N, no de 0..N-1 -}
fun (l: List of T, l0: List of T) ret S: List of T
      S := empty_list()
      while(is not_empty(l)) do
            addr(S,head(l0))
            tail(l0)
            addr(S,head(l))
            tail(l)
      od
end fun
```

5. (Para alumnos libres) Cuando se utiliza backtracking, se recorre un grafo implícito que e todas las posibles soluciones, para elegir la más apropiada.

   A partir del algoritmo que utiliza backtracking para resolver el problema de la moneda, di de búsqueda para monedas de denominaciones 3, 5 y 7 y monto a pagar 19.

Te la debo