1. (Algoritmos voraces) Es principio de mes y tenés que ayudar a tu abuelo a pag
es medio desconfiado y solo paga él mismo por ventanilla en efectivo, nada de t

Para cada factura i sabés qué día d_i va a llegar al domicilio y el día de venc
ir a pagar si no te ha llegado aún la factura al domicilio, y tenés que pagarla
puede pagar también el mismo día que vence). Como sos un excelente estudia
un algoritmo que obtenga qué facturas se pagarán cada día, de manera tal que
de veces posible a la ventanilla de pago.

Se pide lo siguiente:

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

Enunciado:

- n facturas
- para cada factura i se conoce d_i dia que llega a domicilio y v_i día de vencimiento
- No podes pagar si todavía no llegó
- Tenes tiempo hasta el día del vencimiento (inclusive)
- Se pide que facturas se pagarán cada día, de manera tq el abuelo vaya la menor cantidad de veces a la ventanilla

a) Se elige pagar, en el día de vencimiento, aquella factura que vence antes que cualquier otra y además pagar las que llegaron antes de este día y aún no vencen

b)
```
type Factura = tuple
                    id: nat
                    llegada: nat
                    vencimiento: nat
               end tuple

type Pagos = tuple
                day: nat
                facturas: Set of Factura
             end tuple
```

c) El algoritmo toma un conjunto de Factura como parámetro y devuelve una lista de Pagos, la cual indica en que día se realizan los pagos de ciertas facturas. Se copia el conjunto de entrada y se
seleccionan la factura que vence primero más las que llegaron antes de este dia y aun no han vencido, se arma un conjunto de esto para agregar a la solución junto con el dia que se realizará el pago luego se eliminan y se repite hasta que el conjunto de entrada esté vacío

d)

```
fun pagar(in F: Set of Factura) ret S: List of Pagos
     var f_aux: Set of Factura
     var a_pagar: Pagos

     S := empty_list()
     f_aux := set_copy(F)
     while(not is_empty_set(f_aux)) do
          seleccionar_facturas(f_aux, a_pagar)
          addr(S,a_pagar)
          diferencia(f_aux, a_pagar.facturas)
          destruir_set(a_pagar.facturas)
          a_pagar.facturas := empty_set()
     od
     destroy_set(f_aux)
end fun

proc seleccionar_facturas(in F: Set of Factura, in/out p:
Pagos)
     var f_aux, f_aux2: Set of Factura
     var min_aux: nat
     var factura, factura_final: Factura

     {- buscamos la que vence primero -}
     min_aux := inf
     f_aux2 := set_copy(F)
     f_aux := set_copy(F)
     while(not is_empty_set(f_aux)) do
          factura := get(f_aux)
          if(factura.vencimiento < min_aux) then
               min_aux := factura.vencimiento
               factura_final := factura
          fi
          elim(f_aux, factura)
     od

     {- seleccionamos las que llegaron antes del primer
vencimiento -}
     p.day := factura final.vencimiento
     add_Set(p.facturas, factura_final)
     elim(f_aux2, factura_final)
     while(not is_empty_set(f_aux2)) do
          factura := get(f_aux2)
          if(factura.llegada ≤ p.day) then
               add_Set(p.facturas, factura)
          fi
          elim(f_aux2, factura)
     od
```

```
end proc
```

2. Sos la única programadora de una flamante empresa que provee desarrollo en dist
posibles a los cuales ofrecer servicio y la posibilidad de trabajar H horas co
$i \in \{1..n\}$ ya calculaste la cantidad de horas h_i que requiere de trabajo, y la
Tenés la posibilidad de pedirle a un amigo que te ayude con algunos proyectos, e
de las horas (división entera) realizarlo, pero vas a cobrar la mitad del dinero (
tu amigo). Tu tarea es calcular la máxima ganancia que podés obtener eligien
recurrir a la ayuda de tu amigo.

   (a) (Backtracking) Resolvé el problema utilizando la técnica de backtracking
   ello:
      - Especificá precisamente qué calcula la función recursiva que resolverá
      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 Programaci
   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.

Enunciado:

- n proyectos
- H horas como máximo de trabajo
- para cada proyecto i ∈ {1..n} tenemos h_i horas requeridas para el trabajo y p_i la paga
- Un amigo nos puede ayudar a realizar un proyecto, nos tomaría la mitad del tiempo y ganamos la mitad de la paga
- Se pide calcular la máxima ganancia obtenible

a)

i) Función recursiva

p(i, k) “máxima paga obtenible realizando, o no, i proyectos sin superar k horas de trabajo”

ii) Llamada principal:

p(n,H)

iii) Función matemática:

casos:

no tengo más proyectos i = 0

tengo proyectos pero me quedé sin tiempo k = 0 & i > 0
tengo proyectos y me alcanza el tiempo
tengo proyectos y no me alcanza el tiempo (que me ayude mi amigo)

$p(i,k)$ | 0 , si i = 0 v k = 0
(Sin tiempo o sin proyectos)
| $\max(p(i - 1,k), p_i + p(i - 1, k - h_i))$, si i > 0 & k > 0 & $k \geq h_i$
(Nos alcanza el tiempo)
| $\max(p(i - 1,k), p_i \text{ div } 2 + p(i - 1, k - h_i \text{ div } 2))$, si i > 0 & k > 0 & $h_i > k$ & $k \geq h_i \text{ div } 2$
(Nos alcanza el tiempo con ayuda)

| p(i-1, k) , si i > 0 & k > 0 & $h_i > k$ & $h_i \text{ div } 2 > k$
(No nos alcanza el tiempo con ayuda)

No se bien si debe haber un caso imposible en donde deba devolver -inf, puesto a que estamos calculando la máxima suma posible y en algún caso, quedarse sin tiempo no equivale a ganar menos plata. Mientras tanto, nuestro amigo nos ayuda solo cuando el tiempo no nos alcanza. Si contamos con la ayuda de nuestro amigo y de igual manera no nos alcanza, veremos si podemos hacer el siguiente proyecto.

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

    {- casos base -}
    for i := 0 to n do dp[i,0] := 0 od
    for k := 0 to H do dp[0,k] := 0 od

    {- casos recursivos -}
    for i := 1 to n do
        for k := 1 to H do
            if ( k ≥ h[i] ) then
                dp[i,k] := max(dp[i-1,k], p[i] + dp[i-1, k-h[i]]
            else if h[i] > k and k ≥ h[i] div 2
                dp[i,k] := max(dp[i-1,k], p[i] div 2 + dp[i-1, k - h[i] div 2])
            else
                dp[i,k] := dp[i-1,k]
            fi
        od
```

```
        od
        puntaje := dp[n,H]
    end fun
```

- ¿Qué dimensiones tiene la tabla que el algoritmo debe llenar?
  2 dimensiones.
- ¿En qué orden se llena la misma?
  De arriba hacia abajo, de izquierda a derecha.
- ¿Se podría llenar de otra forma? En caso afirmativo indique cuál.
  De arriba hacia abajo, de derecha a izquierda.

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

   (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 procedimientos).

```
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(i, n-i+1));
         swap(p, n-i+1, t(i+1, n-i+1));
     od
end fun
```

Algoritmo 1:

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

Nos da la posición del elemento menor dentro de un segmento

{PRE: $v \leq w \leq n$ }

(b) ¿Cómo lo hace?

Toma una secuencia y dos índices que indican el segmento, desde el índice derecho hasta el último índice, se compara el siguiente con el anterior, y si es menor se actualiza la variable a retornar. En cada paso se comparan los elementos con el menor encontrado guardando la posición.

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

O(w-v)

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

indice_del_menor

Algoritmo 2:

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

Nos da la posición del elemento mayor dentro de un segmento

{PRE: $v \leq w \leq n$ }

(b) ¿Cómo lo hace?

Lo mismo que el anterior.

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

O(w-v)

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

indice_del_mayor

Algoritmo 3:

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

Ordena una secuencia de números de forma creciente

{PRE: n = 2k}

(b) ¿Cómo lo hace?

De izquierda a derecha, se itera una cantidad de veces indicada por la mitad de elementos. En cada paso se busca el mínimo con la función s para intercambiarlo al primer lugar, y el máximo con la función t para intercambiarlo al último lugar, luego el mínimo para intercambiarlo al segundo lugar y el máximo para el anteúltimo lugar. En cada paso el segmento en donde se busca es menor ya que no buscamos en las posiciones ya intercambiadas.

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

El bucle se ejecuta n div 2 veces, por cada iteración tenemos dos complejidades, una de o(n - 2i + 1) y otra de o(n - 2i) como la más compleja es o(n - 2i + 1) la complejidad total es de:

O(n/2) * O(n - 2i + 1) = O(n^2)

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

ordenar_arreglo

4. Considere la siguiente especificación del tipo Pila de elementos de tipo T:

spec Stack **of** T **where**

constructors

fun empty_stack() **ret** s : Stack **of** T
{- *crea una pila vacía.* -}

proc push (**in** e : T, **in**/**out** s : Stack **of** T)
{- *agrega el elemento* e *al tope de la pila* s. -}

destructors

proc destroy(**in**/**out** s : Stack **of** T)

copy

fun copy(s : Stack **of** T) **ret** s : Stack **of** T

operations

fun is_empty_stack(s : Stack **of** T) **ret** b : Bool
{- *Devuelve* True *si la pila es vacía* -}

fun top(s : Stack **of** T) **ret** e : T
{- *Devuelve el elemento que se encuentra en el tope de* s. -}
{- **PRE:** not is_empty_stack(s) -}

proc pop (**in**/**out** s : Stack **of** T)
{- *Elimina el elemento que se encuentra en el tope de* s. -}
{- **PRE:** not is_empty_stack(s) -}

(a) Utilizando como representación un arreglo de N elementos y un número natural, implementá los constructores y todas las operaciones indicando el orden de complejidad de cada una.

(b) ¿La representación elegida tiene alguna limitación? Si es así, ¿cuál? Justificá la respuesta.

(c) ¿Podría implementarse el tipo Pila utilizando como representación interna un conjunto de elementos? Justificá la respuesta.

(d) Utilizando el tipo **abstracto**, implementá un procedimiento *invert* que invierta los elementos de una pila de elementos de tipo T. ¿Qué orden de complejidad tiene la implementación?

a)

```
Implement Stack of T where

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

```
constructors

    {- O(1) -}
    fun empty_stack() ret s : Stack of T
        s.size := 0
    end fun

    {- O(1) -}
    proc push (in e : T, in/out s : Stack of T)
        s.size := s.size + 1
        s.elems[s.size] := e
    end proc

destructors

    {- O(1) -}
    proc destroy(in/out s : Stack of T)
        skip
    end proc

copy

    {- O(s.size) -}
    fun copy(s : Stack of T) ret c : Stack of T
        c.size := s.size
        for i := 1 to s.size do
            c.elems[i] := s.elems[i]
        od
    end fun

operations

    {- O(1) -}
    fun is_empty_stack(s : Stack of T) ret b : Bool
        b := 0 == s.size
    end fun

    {- O(1) -}
    fun top(s : Stack of T) ret e : T
        e := s.elems[s.size]
    end fun

    {- O(1) -}
    proc pop (in/out s : Stack of T)
        s.size := s.size - 1
    end fun
```

```
end implement
```

b) La limitación es que size ≤ N en todo momento.

c) No, un conjunto no tiene orden.

d)

```
proc invert(in/out s: Stack of T)
      var s_aux: Stack of T
            if(not is_empty_stack(s)) then
                  s_aux := copy(s)
                  while(not is_empty_stack(s)) do
                        pop(s)
                  od
                  while(not is_empty_stack(s_aux) do
                        push(top(s_aux),s)
                        pop(s_aux)
                  od
            fi
end proc
```

5. (Para alumnos libres) Dar la forma general de los algoritmos divide y vencerás, identificar sus características, explicarlas y mencionar ejemplos de uso conocido de esa técnica.