1. (Algoritmos voraces) Es el siglo XX y no existe Netflix ni ningún servicio de TV por demanda. En esta época la gente contrata servicio de TV por cable donde te envían un librito con la programación de cada canal día por día.

Como el fin de semana estará lloviendo, planeás encerrarte a ver películas. Del librito de programación seleccionaste n películas que te interesan (que se transmiten en distintos canales) y para cada película i, con $1 <= i <= n$, tenés el horario de comienzo c_i y de final f_i. Por supuesto no podés ver dos películas a la vez.

Debés encontrar cuáles de las n películas vas a ver, de manera que la cantidad sea máxima. Se pide lo siguiente:

(a) Indicar de manera simple y concreta, cuál es el criterio de selección voraz para construir la solución?
(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 películas.
- cada película i con 1≤ i ≤ n tenemos c_i horario de comienzo y f_i horario de fin.
- No se pueden ver dos películas a la vez:
- Cuales de las n películas vas a ver, tq la cantidad sea máxima.

Criterio de Selección:

- Se selecciona aquella película que termine antes entre todas las disponibles luego de una hora h.

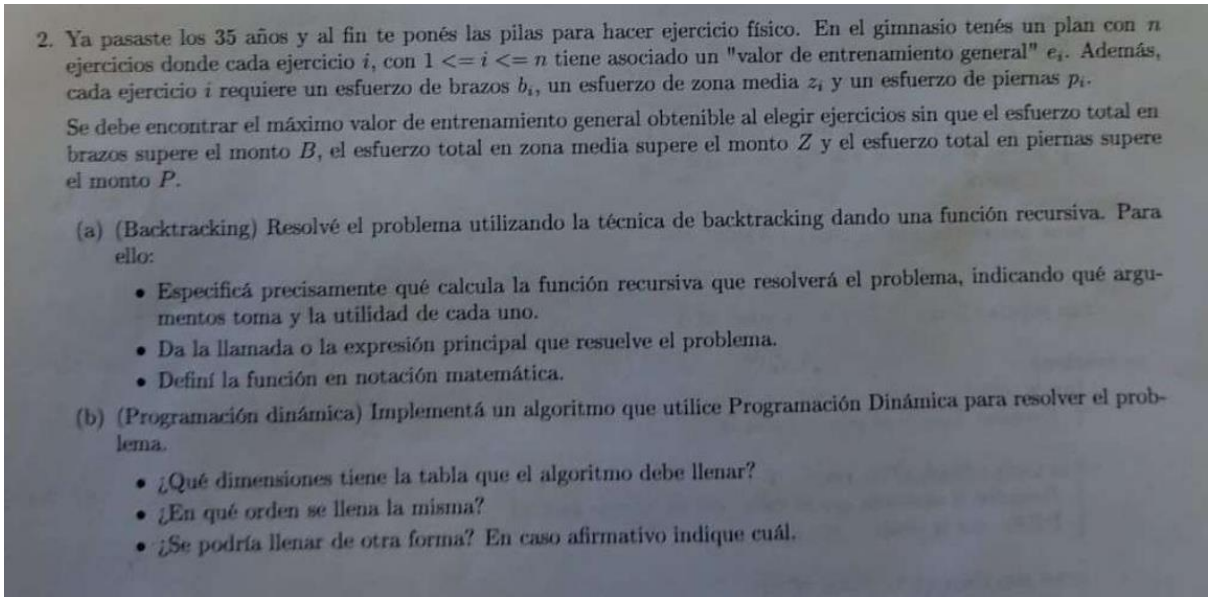
Estructura de datos:

-
```
type Película = tuple
                    id: nat
                    comienzo: float
                    final: float
                end tuple
```

Explicación:

- El algoritmo toma un conjunto de películas y selecciona aquella película que termine antes de todas las disponibles, la elimina del conjunto, la agrega a la solución, actualiza la hora en donde estamos parados con la hora final de la película, elimina todas aquellas películas que comenzaron en el horaria de la elegida y continúa hasta que el conjunto de películas es vacío.

```
fun pelis( P: Set of Película ) ret S: List of Película
      var p_aux: Set of Pelicula
      var h: float
      var p: Película
      h := 0.0
      S := empty_list()
      p_aux := copy_set(P)
      while( not is_empty_set(p_aux)) do
            p := elegir(p_aux)    {- Seleccionar la que termine primero -}
            elim(p_aux, p)
            addr(S, p)
            h := h + p.final
            elim_pelis(p_aux, h) {- eliminar las películas perdidas -}
      od
      destroy_set(p_aux)
end fun

fun elegir(P: Set of Película) ret res: Película
      var p_aux: Set of Pelicula
      var min_aux: float
      min_aux := inf
      p_aux := copy_set(P)
      while( not is_empty_set(p_aux)) do
            p := get(p_aux)
            if p.comienzo < min_aux then
                  min_aux := p.final
                  res := p
            fi
            elim(p_aux, p)
      od
      destroy_set(p_aux)

end fun

proc elim_pelis(in/out P: Set of Película, in H: float)
      var p_aux: Set of Pelicula
      p_aux := copy_set(P)
      while( not is_empty_set(p_aux)) do
            p := get(p_aux)
            if p.final ≤ H then
                  elim(P,p)
            fi
            elim(p_aux, p)
      od
      destroy_set(p_aux)
end proc
```



2. Ya pasaste los 35 años y al fin te ponés las pilas para hacer ejercicio físico. En el gimnasio tenés un plan con n ejercicios donde cada ejercicio i, con $1 <= i <= n$ tiene asociado un "valor de entrenamiento general" e_i. Además, cada ejercicio i requiere un esfuerzo de brazos b_i, un esfuerzo de zona media z_i y un esfuerzo de piernas p_i.

   Se debe encontrar el máximo valor de entrenamiento general obtenible al elegir ejercicios sin que el esfuerzo total en brazos supere el monto B, el esfuerzo total en zona media supere el monto Z y el esfuerzo total en piernas supere el monto P.

   (a) (Backtracking) Resolvé el problema utilizando la técnica de backtracking dando una función recursiva. Para ello:
   - Especificá precisamente qué calcula la función recursiva que resolverá el problema, indicando qué argumentos 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 Dinámica para resolver el problema.
   - ¿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:

- Plan con n ejercicios donde cada ejercicio tiene un valor e_i asociado que indica el valor de entrenamiento general.
- Además cada ejercicio requiere
  - b_i esfuerzo de brazos
  - z_i esfuerzo de zona media
  - p_i esfuerzo de piernas
- Se debe encontrar el máximo valor de entrenamiento general, sin que el esfuerzo en brazos supere B, el esfuerzo en zona media supere Z y el esfuerzo en piernas supere a P

Backtracking:

Función recursiva:

gym(i , br, zm, pn) “máximo valor obtenible de entrenamiento general realizando i ejercicios sin superar esfuerzo en brazo br, en zona media zm y piernas pn ”

Llamada principal:

gym(n, B, Z, P)

Función matemática:

gym(i,br,zm,pn) =

0	, si i = 0
-inf	, si i > 0 ^ (br = 0 v zm = 0 v pn = 0)
gym(i-1, br, zm, pn)	, si i > 0 ^ (b_i > br v z_i > zm v p_i > pn)
max(gym(i-1, br, zm, pn), v_i + gym(i-1, br - b_i, zm - z_i, pn - p_i))	, si i > 0 ^ (b_i ≤ br ^ z_i ≤ zm ^ p_i ≤ pn)

Se asume que b_i z_i y p_i siempre es mayor estricto que 0

Programación dinámica:

```
fun gym(n: nat, B: nat, P: nat, Z: nat, v: array[1..n] of nat, b: array[1..n] of
nat, z: array[1..n] of nat, p: array[1..n] of nat) ret solución: nat

{- variables -}
var dp: array[0..n, 0..B, 0..Z, 0..P] of nat

{- caso base -}
{- No hay ejercicios -}
for i := 0 to B do
  for j := 0 to Z do
    for k := 0 to P do
      dp[0,i,j,k] := 0
    od
  od
od
{- me quedan ejercicios pero no tengo energía en brazos -}
for i := 1 to n do
  for j := 0 to Z do
    for k := 0 to P do
      dp[i,0,j,k] := -inf
    od
  od
od
{- me quedan ejercicios pero no tengo energía en zona media -}
for i := 1 to n do
  for j := 0 to B do
    for k := 0 to P do
      dp[i,j,0,k] := -inf
    od
  od
od
{- me quedan ejercicios pero no tengo energía en piernas -}
for i := 1 to n do
  for j := 0 to B do
    for k := 0 to Z do
      dp[i,j,k,0] := -inf
    od
  od
od
{- caso recursivo -}
for i := 1 to n do
  for j := 1 to B do
    for k := 1 to Z do
      for l := 1 to P do
        if ( b[i] > j or z[i] > k or p[i] > l ) then
          dp[i,j,k,l] := dp[i-1,j,k,l]
        else if ( b[i] ≤ j and z[i] ≤ k and p[i] ≤ l ) then
           dp[i,j,k,l] :=
             max(dp[i-1, j, k, l], v[i] + dp[i-1, j - b[i], k - z[i], l - p[i]))
        fi
      od
    od
  od
od
solucion := dp[n,B,Z,P]
end fun
```

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).

```
proc q(in/out a : array[1..N] of int, in x : nat)
    for j:= 1 to x do
        m:= j
        for k:= j+1 to x do
            if a[k] < a[m] then m:= k fi
        od
        swap(a,j,m)
    od
end proc
proc p(in/out a : array[1..N] of int, in i : nat)
    q(a, i-1)
    r(a, i+1)
end proc
```

```
proc r(in/out a : array[1..N] of int, in y : nat)
    for j:= y to n do
        m:= j
        while m > y ∧ a[m] < a[m-1] do
            swap(a,m,m-1)
            m:= m-1
        od
    od
end proc
```

Algoritmo 1:

¿Qué hace? ordena x posiciones de un arreglo

Precondiciones: {- PRE: 1 ≤ x ≤ n -}

¿Cómo lo hace? Busca el mínimo para el primer índice, comparando todos los elementos consiguientes que se encuentran hasta el índice x inclusive, una vez encontrado los intercambia, luego busca el mínimo para el segundo índice de la misma forma, hasta tener x elementos ordenadas dentro del arreglo.

¿Orden? x^2

¿Nombre? ordenar_x_elementos

$$\sum_{j=1}^{x} \sum_{k=j+1}^{x} 1$$

$$\sum_{j=1}^{x} (x - j)$$

$$\sum_{j=1}^{x} (x - j) = (x - 1) + (x - 2) + (x - 3) + \cdots + 1 + 0$$

$$\sum_{j=1}^{x-1} j = \frac{(x - 1)x}{2}$$

$$O\left(\frac{(x-1)x}{2}\right) = O(x^2)$$

Algoritmo 2:

¿Qué hace? ordena un arreglo desde la posición y+1 inclusive en adelante
Precondiciones: {- PRE: 1 ≤ y ≤ n -}
¿Cómo lo hace? Busca el mínimo para todas las posiciones del arreglo desde y en adelante, en cada paso se busca el mínimo comparando con elementos anteriores, si este es encontrado, se intercambia con la posición actual y luego se comprueba lo mismo para la posición anterior tq sea mayor a y.
¿Orden? (n - y)^2
¿Nombre? ordenar_y_a_n_elementos

1. Para $j = y$, el bucle `while` no se ejecuta.
2. Para $j = y + 1$, el bucle `while` puede ejecutarse 1 vez.
3. Para $j = y + 2$, el bucle `while` puede ejecutarse 2 veces.
4. ...
5. Para $j = n$, el bucle `while` puede ejecutarse $n - y$ veces.

Sumando estas ejecuciones en el peor caso:

$$\sum_{j=y}^{n}(j-y)$$

Desglosamos la sumatoria:

$$\sum_{j=y}^{n}(j-y) = \sum_{k=0}^{n-y}k = 0+1+2+\cdots+(n-y)$$

Esta es una serie aritmética cuya suma puede calcularse con la fórmula:

$$\sum_{k=0}^{n-y} k = \frac{(n-y)(n-y+1)}{2}$$

Entonces, la complejidad total del algoritmo en el peor caso es proporcional a:

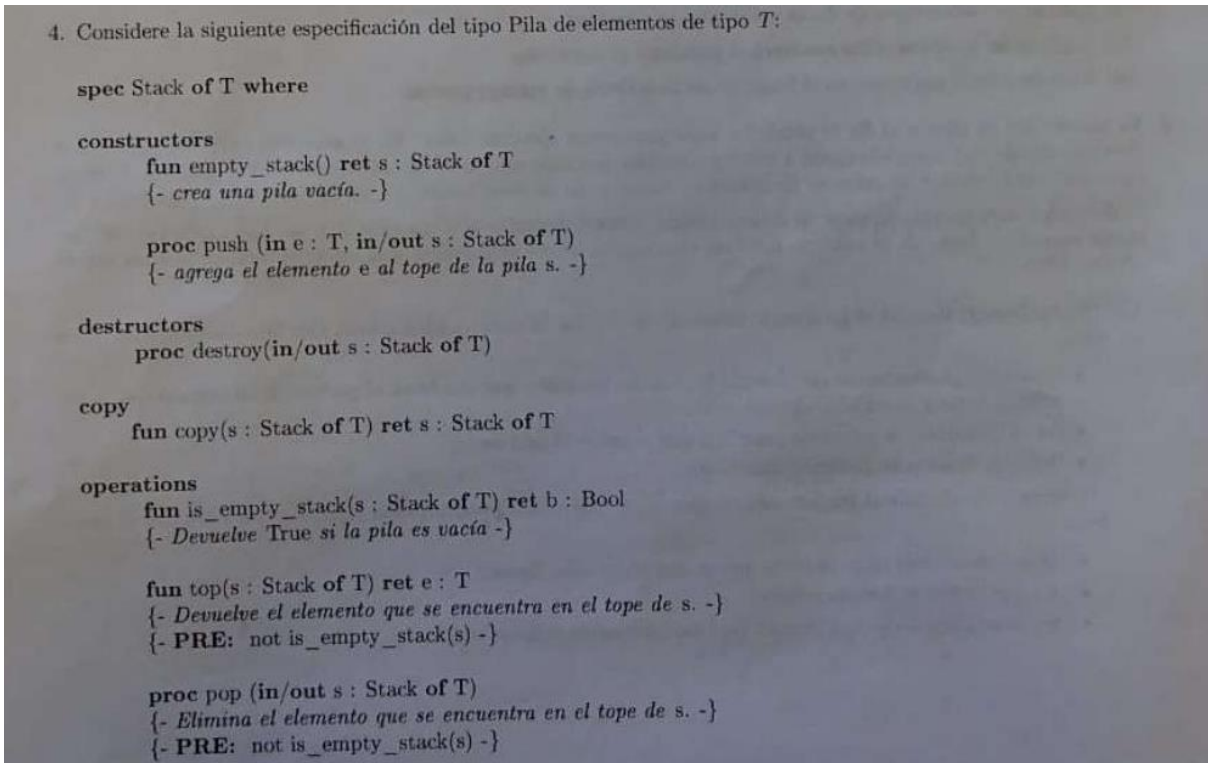
$$O\left(\frac{(n-y)(n-y+1)}{2}\right) = O((n-y)^2)$$

Conclusión

El orden de complejidad del algoritmo es $O((n-y)^2)$. Este resultado es consistente con el análisis de algoritmos de inserción, los cuales típicamente tienen una complejidad cuadrática en el peor caso debido a los desplazamientos necesarios para insertar cada elemento en su posición correcta.

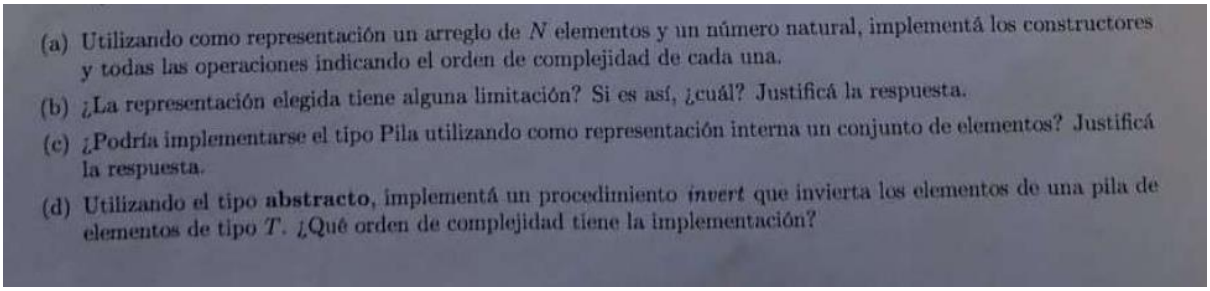
Algoritmo 3:

¿Qué hace? Ordena una secuencia de números
Precondiciones: {- PRE: 1 ≤ i ≤ n -}
¿Cómo lo hace? Llama al proc q indicando que se desea ordenar el arreglo hasta el índice i-1, luego llama al proc r que ordena el arreglo desde el índice i+1 en adelante. q busca el mínimo en todo el segmento y lo ubica en la posición actual avanzando de derecha a izquierda, mientras que r intercambia elementos si los anteriores al que se está mirando son mayores hasta ordenar todos los elementos desde el índice dado hasta la posición actual, luego lo mismo para la siguiente posición.
¿Orden? (n - y)^2 + n^2
¿Nombre? ordenar_elementos



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?

```
Implement Stack of T where

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

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

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

{- O(1) -}
proc destroy(in/out s: Stack of T)
      skip {- no se debe liberar memoria -}
end proc

{- O(n) en donde n es la cantidad de elementos en la pila -}
fun copy (s: Stack of T) ret q: Stack of T
      q := empty_stack()
      q->size := s->size;
      if(not is_empty_stack(s)) then
            for i := 1 to s->size do
                  q->elems[i] := s->elems[i]
            od
      fi
end fun

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




{- O(1) -}
```

```
{-Pre: not is_empty_stack(s)-}
fun top(s: Stack of T) ret e: T
      e := s->elems[s->size]
end fun

{- O(1) -}
{-Pre: not is_empty_stack(s)-}
proc pop (in/out s: Stack of T)
      s->size--
end proc
```

b) Esta representación tiene una limitación y es que el arreglo es de N elementos, entonces s->size debe de ser menor o igual a N en todo momento

c) Una pila no puede ser representada por un conjunto de elementos porque en el TAD conjuntos no existe el orden, entonces no tendríamos noción del "tope" de la pila

d)

```
{- O(n) en donde n es la cantidad de elementos en la pila -}
proc invert(in/out s: Stack of T)
      var q: Stack of T
      var j: nat
      if(not is_empty_stack(s)) then
            q := copy(s)
            while( not is_empty_stack(s)) do
                  pop(s)
            od

            while( not is_empty_stack(q)) do
                  push(top(q), s)
                  pop(q)
            od
      destroy(q)
      fi
end proc
```