1. (Backtracking) No es posible correr una carrera de 800 vueltas sin reemplazar cada tanto las cubiertas (las ruedas) del auto. Como los mecánicos trabajan en equipo, cuando se cambian las cubiertas se reemplazan simultáneamente las cuatro. Reemplazar el set de cuatro cubiertas insume un tiempo T fijo, totalmente independiente de cuál sea la calidad de las cubiertas involucradas. Hay diferentes sets de cubiertas: algunas permiten mayor velocidad que otras, y algunas tienen mayor vida útil que otras, es decir, permiten realizar un mayor número de vueltas. Sabiendo que se cuenta con n sets de cubiertas, que $t_1, t_2, \ldots, t_n$ son los **tiempos por vuelta** que pueden obtenerse con cada uno de ellos, y que $v_1, v_2, \ldots, v_n$ es la vida útil medida en **cantidad de vueltas** de cada uno de ellos, se pide encontrar el tiempo de carrera mínimo cuando la misma consta de m vueltas.

   (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.

a)

Enunciado:

- Cambiar las 4 cubiertas simultáneamente implica un tiempo T fijo
- Hay diferentes tipos de cubierta:
  - Algunas dan más velocidad que otras
  - Algunas tienen mayor vida útil que otras (mayor número de vueltas)
- Se cuenta con n sets de cubiertas
- $t_1, t_2, \ldots, t_n$ tiempos por vueltas que se obtienen con cada set
- $v_1, v_2, \ldots, v_n$ es la vida útil del set medida en cantidad de vueltas
- Encontrar el tiempo de carrera mínimo cuando la misma es de m vueltas

Observaciones:

- Cada vez que se cambia a otro set, se consume un tiempo T y además tenemos el tiempo t_i que debemos de multiplicarlo por la cantidad de vueltas realizadas, es decir $t_i * v_i + T$ es el tiempo total que pasó desde que se cambió el set hasta que se gastó.
- Cada vez que cambiamos un set tenemos aseguradas m - v_i vueltas, puesto que se recorre hasta que el set se desgaste.
- En la función debo de tener:
  - Conteo de las vueltas realizadas {1,2,...,m}
  - Conteo de los set que utilizo {1,2,...,n}

- Entonces debemos ver si hacemos v_i vueltas con el set i de manera tq la próxima llamada será con m - v_i vueltas, o usamos el siguiente set el cual nos dará un mejor o peor rendimiento.

Función recursiva:

cars(i,j) “mínimo tiempo de carrera teniendo en cuenta los i sets de ruedas para j vueltas restantes”

Llamada principal:

cars(n,m)

Función matemática:

Casos:

- Ya no tengo más sets de ruedas y no quedan más vueltas restantes (se asume que tenemos suficientes sets para terminar la carrera)
- Ya no tengo más sets de ruedas pero quedan vueltas por recorrer (caso imposible)
- Tengo sets de ruedas y tengo que realizar vueltas (caso recursivo)
  - Cambiar el set i, restar las vueltas realizadas
  - No usar el set i, y probar con el set i-1
  - OBS: Puede suceder que la vida útil del set i sea mayor o igual a las vueltas restantes, en este caso, se prueba usar el set y se pone en 0 las vueltas restantes.

cars(i,j)

0	, si j = 0
inf	, si i = 0 & j > 0
min(T + t_i * v_i + cars(i-1,j - v_i), cars(i-1,j))	, si i > 0 & j > 0 & v_i ≤ j
min(T + j * t_i , cars(i-1,j))	, si i > 0 & j > 0 & j < v_i

b)

Programación dinámica:

```
fun cars(v: array[1..n] of nat, t: array[1..n] of float, T: nat, M: nat) ret min_tiempo: float
{- variables -}
var dp: array[0..n, 0..M] of float

{- caso base -}
for i := 0 to n do dp[i,0] := 0 od
for j := 1 to M do dp[0,j] := inf od

{- caso recursivo -}
```

```
for i := 1 to n do
      for j := 1 to M do
            if v[i] ≤ j then
                  dp[i,j] := min(T + t[i] * v[i] + dp[i-1,j-v[i]], dp[i-
1,j])
            else
                  dp[i,j] := min(T + t[i] * j, dp[i-1,j])
            fi
      od
od
      min_tiempo := dp[n,M]
end fun
```

Dimensiones: 2
Orden: Se llena de arriba hacia abajo (fila 1,2,..,n) y en cada fila la columna se llena de izquierda a derecha. Se puede llenar de arriba a abajo y de derecha a izquierda.

2. (Algoritmos voraces) Llega el final del cuatrimestre y tenés la posibilidad de rendir algunas de las materias $1, ..., n$. Para cada materia i conocés la fecha de examen f_i, y la cantidad de días inmediatamente previos d_i que necesitás estudiarla de manera exclusiva (o sea, no podés estudiar dos materias al mismo tiempo). Dar un algoritmo voraz que obtenga la mayor cantidad de materias que podés rendir.

   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.

a) Se selecciona aquel final que se rinda antes que cualquier otro

b)
```
type Materia = tuple
                  id: nat
                  dia_cursada: nat
                  dias_previos: nat
               end tuple
```

c) El algoritmo toma un conjunto de Materias y selecciona aquella que se rinda primero, luego se verifica que sea factible rendirla, es decir, si tenemos dias a favor para estudiar agregamos a la solución y si no nos alcanzan los días no lo agregamos y vemos con el siguiente, cada vez que se pueda realizar un examen actualizamos el dia. La solución será una lista de Materias que indican que materias se pudo rendir.

d)

```
fun exámenes(E: Set of Materia) ret M: List of Materia
      var e_aux: Set of Materia
```

```
      var day: nat
      var materia: Materia

      M := empty_list()
      e_aux := copy_set(E)
      day := 1
      while(not is_empty_set(e_aux)) do
            materia := seleccionar_materia(e_aux)
            if(materia.dia_cursada − day ≥ materia.dias_previos) then
                  addr(M, materia)
                  day := materia.dia_cursada + 1
            fi
            elim(e_aux, materia)
      od
      destroy_set(e_aux)
end fun

fun seleccionar_materia(E: Set of Materia) ret res: Materia
      var e_aux: Set of Materia
      var materia: Materia
      var min_aux: int

      min_aux := inf
      e_aux := copy_set(E)
      while(not is_empty_set(e_aux)) do
            materia := get(e_aux)
            if(materia.dia_cursada < min_aux) then
                  min_aux := materia.dia_cursada
                  res := materia
            fi
            elim(e_aux, materia)
      od
      destroy_set(e_aux)
end fun
```

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

   (a) ¿Qué hace?

   (b) ¿Cómo lo hace?

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

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

```
proc p(a: array[1..n] of nat)
    var d: nat
    for i:= 1 to n do
        d:= i
        for j:= i+1 to n do
            if a[j] < a[d] then d:= j fi
        od
        swap(a, i, d)
    od
end proc
```

```
fun f(a: array[1..n] of nat) ret b : array[1..n] of nat
    var d: nat
    for i:= 1 to n do b[i] := i od
    for i:= 1 to n do
        d:= i
        for j:= i+1 to n do
            if a[b[j]] < a[b[d]] then d:= j fi
        od
        swap(b, i, d)
    od
end fun
```

Algoritmo 1:

```
proc p(a: array[1..n] of nat)
      var d: nat
      for i:= 1 to n do
            d:= i
            for j:= i+1 to n do
                  if a[j] < a[d] then d:= j  fi
            od
            swap(a, i, d)
      od
end proc
```

(a) ¿Qué hace?

Ordena un arreglo

(b) ¿Cómo lo hace?

Selecciona el menor de todos y lo intercambia con el elemento que se encuentra y la primera posición, selecciona el menor de todos los restantes y lo intercambia con el elemento que se encuentra en la segunda posición, en cada paso ordena un elemento hasta terminar

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

Independientemente de que esté ordenado o no, siempre se ejecutan las mismas comparaciones.

Tenemos un bucle que se ejecuta n veces, y por cada iteración tenemos otro bucle el cual si

ejecuta comparaciones que por cada i el bucle se ejecuta (n - i) veces. Es decir sumatoria desde i = 1 a n de (n - i), es decir, n-1 + n-2 + … + 0 es la sumatoria de n, lo cual lo podemos ver como sumatoria desde i = 1 hasta n de i o equivalentemente, n(n+1)/2.

En conclusión el algoritmo es de orden O(n^2)

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

selection_sort

Algoritmo 2:

```
fun f(a: array[1..n] of nat) ret b : array[1..n] of nat
      var d: nat
      for i:= 1 to n do
            b[i] := i
      od
      for i:= 1 to n do
            d:= i
            for j:= i+1 to n do
                  if a[b[j]] < a[b[d]] then
                  d:= j
            od
            swap(b, i, d)
      od
end fun
```

(a) ¿Qué hace?

Nos da un arreglo que nos indica el orden en el cual deben estar los índices de otro arreglo para que este último esté ordenado.

(b) ¿Cómo lo hace?

Inicializa un arreglo con las posiciones del otro arreglo, luego busca el menor para todos los elementos e intercambia la posición de este con la primer posición del primer arreglo, busca el mínimo para el resto del arreglo e intercambia la posición con la segunda posición del primer arreglo, de esta forma en cada paso el primer arreglo indica en qué posición debe de estar el indice del otro arreglo para estar ordenado

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

O(N^2)

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

ordenar_indices

4. (TADs) Considerá la siguiente especificación del tipo Conjunto de elementos de algún tipo T.

spec Set **of** T **where**

constructors

fun empty_set() **ret** s : Set **of** T
{- *crea un conjunto vacío.* -}

proc add (**in** e : T, **in**/**out** s : Set **of** T)
{- *agrega el elemento* e *al conjunto* s. -}

destroy

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

operations

fun is_empty_set(s : Set **of** T) **ret** b : bool
{- *Devuelve* True *si* s *es vacío.* -}

fun cardinal(s : Set **of** T) **ret** n : nat
{- *Devuelve la cantidad de elementos que tiene* s -}

fun member(e : T, s : Set **of** T) **ret** b : bool
{- *Devuelve True si el elemento* e *pertenece al conjunto* s -}

proc inters (**in**/**out** s : Set **of** T,**in** s0 : Set **of** T)
{- *Elimina de* s *todos los elementos que NO pertenecen a* s0 -}

{- **PRE:** not is_empty_set(s) -}
fun get(s : Set **of** T) **ret** e : T
{- *Devuelve algún elemento (cualquiera) de* s -}

proc elim (**in**/**out** s : Set **of** T,**in** e : T)
{- *Elimina el elemento* e *del conjunto* s *en caso que esté.* -}

proc union (**in**/**out** s : Set **of** T,**in** s0 : Set **of** T)
{- *Agrega a* s *todos los elementos de* s0 -}

proc diff (**in**/**out** s : Set **of** T,**in** s0 : Set **of** T)
{- *Elimina de* s *todos los elementos de* s0 -}

fun copy_set(s1 : Set **of** T) **ret** s2 : Set **of** T
{- *Crea un nuevo conjunto* s2 *con todos los elementos de* s1 -}

(a) Implementá los constructores del TAD Conjunto de elementos de tipo T, y las operaciones member, elim e inters, utilizando la siguiente representación:

implement Set **of** T **where**

type Set **of** T = **tuple**
elems : array[0..N-1] **of** T
size : nat
end tuple

¿Existe alguna limitación con esta representación de conjuntos? En caso afirmativo indicá si algunas de las operaciones o constructores tendrán alguna precondición adicional.
NOTA: Si necesitás alguna operación extra para implementar lo que se pide, debes implementarla también.

(b) Utilizando el tipo **abstracto** Conjunto de elementos de tipo T, implementá una función que reciba un conjunto de enteros s, un número entero i, y obtenga el entero perteneciente a s que está *más cerca* de i, es decir, un $j \in s$ tal que para todo $k \in s$, $|j-i| <= |k-i|$. Por ejemplo si el conjunto es $1, 5, 9$, y el entero 7, el resultado puede ser 5 o 9.

a)

```
fun empty_set() ret s: Set of T
      s.size := 0
end fun

proc add (in e : T, in/out s : Set of T)
      if not member(e,s) then
            s.size := s.size + 1
            s.elems[s.size - 1] := e
      fi
end proc

proc destroy (in/out s : Set of T)
      skip
ens proc

fun member(e : T, s : Set of T) ret b : bool
      var i: nat
      i := 0
      b := false
      if not is_empty_set(s) then
            while i < s.size and not b do
                  if(s.elems[i] == e) then
```

```
                    b := true
                fi
                i++
            od
        fi
end fun

proc elim (in/out s : Set of T,in e : T)
        if(member(e,s)) then
            var s0: Set of T
            s0 := empty_set()
            add(e,s0)
            diff(s,s0) {- deja a size en su valor correcto -}
        fi
end proc

proc inters (in/out s : Set of T,in s0 : Set of T)
        if(not is_empty_set(s0)) then
            for i := 0 to s.size-1 do
                if(not member(s.elems[i]), s0) then
                    elim(s,s.elems[i]))
                fi
            od
        fi
end proc
```

b)

(b) Utilizando el tipo **abstracto** Conjunto de elementos de tipo T, implementá una función que reciba un conjunto de enteros s, un número entero i, y obtenga el entero perteneciente a s que está *más cerca* de i, es decir, un $j \in s$ tal que para todo $k \in s$, $|j-i| <= |k-i|$. Por ejemplo si el conjunto es $1, 5, 9$, y el entero 7, el resultado puede ser 5 o 9.

```
fun cerca(s: Set of int, i: int) ret res: int
        var s_aux, s_aux2: Set of int
        var elem: int
        var mas_cercano: int
        var b: bool

        mas_cercano := inf
        s_aux := copy_set(s)
        while(not is_empty_set(s_aux)) do
            s_aux2 := copy_set(s)
            b := true
```

```
            elem := get(s_aux)
            while(not is_empty_set(s_aux2)) do
                  elem2 := get(s_aux2)
                  if | elem - i | > | elem2 - i | then
                        b := false
                  fi
                  elim(s_aux2, elem2)
            od
            destroy_set(s_aux2)
            s_aux2 := empty_set()
            if b then
                  if elem - i < mas_cercano then
                        mas_cercano := elem
                  fi
            fi
            elim(elem, s_aux)
      od
      res := mas_cercano
end fun
```

Agarramos un elemento cualquiera de s, si este cumple la condición para todos los elementos del conjunto, entonces en un posible resultado, para asegurarnos debemos ver si es menor o no a otros cercanos ya calculados

s = (1,5,9) i = 7

j = 1 → | 1 - 7 | ≤ | 1 - 7 | se cumple
j = 1 → | 1 - 7 | ≤ | 5 - 7 | no se cumple
j = 1 → | 1 - 7 | ≤ | 9 - 7 | no se cumple
esta 6 unidades cerca

j = 5 → | 5 - 7 | ≤ | 1 - 7 | si se cumple
j = 5 → | 5 - 7 | ≤ | 5 - 7 | si se cumple
j = 5 → | 5 - 7 | ≤ | 9 - 7 | si se cumple
esta dos unidades cerca

j = 9 → | 9 - 7 | ≤ | 1 - 7 | si se cumple
j = 9 → | 9 - 7 | ≤ | 5 - 7 | si se cumple
j = 9 → | 9 - 7 | ≤ | 9 - 7 | si se cumple
esta 2 unidades cerca

si el elemento j del set cumple con la condición, bien, ahora veamos si es el que más cerca está