Ej1)

(Algoritmos voraces) Te vas **n días** de vacaciones al medio de la montaña, lejos de toda civilización. Llevás con vos lo imprescindible: una carpa, ropa, una linterna, un buen libro y comida preparada para **m raciones diarias**, con **m > n**. Cada ración i tiene una fecha de vencimiento **vi** , contada en días desde el momento en que llegás a la montaña. Por ejemplo, una vianda con fecha de vencimiento 4, significa que se puede comer hasta el día número 4 de vacaciones inclusive. Luego ya está fuera de estado y no puede comerse. Tenés que encontrar la mejor manera de organizar las viandas diarias, de manera que la cantidad que se vencen sin ser comidas sea mínima.
Deberás indicar para cada día j, $1 \leq j \leq n$, qué vianda es la que comerás, asegurando que nunca comas algo vencido. 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:

- me voy n dias
- llevo m raciones con m > n
- cada ración y tiene una fecha de vencimiento v_i contada en días desde que llegaste
- Se debe organizar las viandas tq la cantidad que se vencen sin ser comidas sea mínima
- Se debe decir que comerás tal dia asegurando que no esté vencido

Criterio de selección:

- Se selecciona siempre la más próxima a vencer desde el día j en adelante

Estructuras de datos:

-
```
type Comida = tuple
                  id: nat
                  vencimiento: nat
                end tuple
```

- Solución:
  - `Viandas: array[1..n] of Comida` (El elemento indica que se consume y la posición indica el dia)

Explicación:

El algoritmo toma un conjunto de Comida, selecciona al candidato más óptimo para agregarlo a la solución si este es factible. Luego no solo elimina al candidato del conjunto si no también a aquellos candidatos que ya no son factibles, así hasta que el conjunto sea vacío o la solución esté completa. La solución será un arreglo en donde cada posición indica el día y el elemento que se debe consumir ese día. Se asume que se llevan suficientes alimentos con vencimiento suficiente para poder comer todos los días algo no vencido.

Algoritmo:

```
{- PRE: ∀ c ∈ C, c.vencimiento ≥ 1 -}
fun food(C: Set of Comida) ret S: array[1..n] of Comida
    var c_aux: Set of Comida
    var comida: Comida
    var i: nat

    i := 1
    c_aux := copy_set()
    while ( not is_empty_set(c_aux) and i ≤ n) do
        comida := elegir_comida(c_aux)    {- alimento con mínimo vencimiento -}
        S[i] := comida
        elim(c_aux, comida)               {- eliminarlo de los candidatos -}
        elim_comida_vencida(c_aux, i)     {- eliminó aquellos vencimientos ≤  i -}
        i++
    od
    destroy_set(c_aux)
end fun


fun elegir_comida (C: Set of Comida) ret vianda: Comida
    var c_aux: Set of Comida
    var comida: Comida
    var min_aux: nat

    min_aux := inf
    c_aux := copy_set()
    while (not is_empty_set(c_aux)) do
        comida := get(c_aux)
        if(comida.vencimiento < min_aux) then
            min_aux := comida.vencimiento
            vianda := comida
        fi
        elim(c_aux, comida)
    od
    destroy_set(c_aux)
end fun

proc elim_comida_vencida(in/out C: Set of Comida, in I: nat)
    var c_aux: Set of Comida
    var comida: Comida

    c_aux := copy_set()
    while (not is_empty_set(c_aux)) do
        comida := get(c_aux)
```

```
        if(comida.vencimiendo ≤ I) then
            elim(C, comida)
        fi
        elim(c_aux, comida)
    od
    destroy_set(c_aux)
end proc
```

Ej2)

Se tiene un tablero de 9x9 con números enteros en las casillas. Un jugador se coloca en una casilla a elección de la primera fila y se mueve avanzando en las filas y moviéndose a una columna adyacente o quedándose en la misma columna. En cada movimiento, el jugador suma los puntos correspondientes al número de la casilla, pero nunca puede pisar una casilla de manera tal que el puntaje acumulado, contando esa casilla, dé un valor negativo. El juego termina cuando se llega a la novena fila, y el puntaje total es la suma de los valores de cada casilla por la que el jugador pasó.

Se debe determinar el máximo puntaje obtenible, si es que es posible llegar a la última fila.

Se pide lo siguiente:

(a) Especificá precisamente qué calcula la función recursiva que resolverá el problema, indicando qué argumentos toma y la utilidad de cada uno.

(b) Da la llamada o la expresión principal que resuelve el problema.

(c) Definí la función en notación matemática.

Enunciado:

- Tablero 9x9 con números enteros en las casillas
- Se comienza desde la primer fila, cualquier columna
- En cada movimiento se suman los puntos al número correspondiente de la casilla
- No puede pisar una casilla si la suma de lo acumulado con esa casilla da negativo
- Se termina el juego cuando se llega a la novena fila
- El puntaje es la suma total de los valores de cada casilla por la que el jugador pasó
- Si es posible llegar a la última fila se debe obtener el máximo puntaje obtenible
- Cada casilla tiene asociado un número entero cij (i, j = 1, . . . , n) que indica el puntaje a asignar cuando la ficha esté en la casilla.

- El jugador puede moverse a:
    - La casilla que está inmediatamente abajo
    - La casilla que está abajo y a la izquierda (si la ficha no está en la columna extrema izquierda)
    - La casilla que está abajo y a la derecha (si la ficha no está en la columna extrema derecha)
    - Siempre y cuando la sumatoria actual, con la próxima casilla sea ≥ 0 el jugador podrá avanzar

Llamada recursiva:

F(i,j) “máximo puntaje obtenible partiendo desde la fila 1 hasta la fila i con j ∈ {1,2,...,9}”

Llamada principal:

$\max_{j \in \{1,2,..,9\}} F(9,j)$

Función matemática:

Casos:

- caso base cuando llegamos a la primer fila i = 1 y $t_{1j} \geq 0$
- caso base cuando llegamos a la primer fila i = 1 y $t_{1j} < 0$ (imposible)
- caso cuando me paso de los límites laterales j = 0 v j = 9 (imposible)
- caso cuando estoy en el medio $1 \leq i \leq 9$ & $1 < j < 9$ (recursivo)

F(i,j)

| t_{1j} , si i = 1 & $t_{1j} \geq 0$ (base)

| -inf , si i = 1 & $t_{1j} < 0$ (base)

| -inf , si i ≥ 1 & j = 0 v j = 10 (base)

| -inf , si i > 1 & 9 > j > 1 & x < 0

| x , si i > 1 & 9 > j > 1 & x ≥ 0

en donde $x = t_{ij} + \max(F(i-1, j-1), F(i-1, j), F(i-1, j+1))$

3. (Programación Dinámica) Implementá un algoritmo que utilice Programación Dinámica para resolver el problema del inciso anterior.

   - ¿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.

```
fun juego(t: array[1..9, 1..9] of nat) ret puntaje: int
      {- variables -}
      var dp: array[1..9, 0..10] of nat
      var x: int
      var max_res: int
      {- casos base -}
      for j := 1 to 9 do
            if( t[1,j] ≥ 0) then
                  dp[1,j] := t[1,j]
            else
                  dp[1,j] := -inf
      od
      for i := 1 to 9 do
            dp[i,0] := -inf
            dp[i,10] := -inf
      od
      {- casos recursivos -}
      for i := 2 to 9 do
```

```
            for j := 1 to 9 do
                x := t[i,j] + max(dp[i-1, j-1], dp[i-1, j], dp[i-1,
j+1])
                if x ≥ 0 then
                    dp[i,j] := x
                else
                    dp[i,j] := -inf
                fi
            od
      od
      max_res := -inf
      for j := 1 to 9 do
            if( dp[9,j] > max_res) then
                max_res := dp[9,j]
            od
      od
      puntaje := max_res
end fun
```

- La tabla tiene 2 dimensiones
- Se completa desde arriba hacia abajo y de izquierda a derecha
- Se puede llenar desde la izquierda o desde la derecha

4. 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 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
```

¿Qué hace? p ordena un arreglo de naturales de forma creciente, no hay precondiciones para el algoritmo.
¿Cómo lo hace? p toma una secuencia de números y lo recorre de izquierda a derecha, guarda la posición del elemento actual en otra variable y la toma como “el mínimo encontrado” y si en posiciones por delante hay algún elemento menor a este, se actualiza la posición del mínimo encontrado para finalmente hacer un intercambio entre la posición del mínimo encontrado y la posición actual.

¿Qué orden es?

- Caso complejo =

```
for i:= 1 to n do (n veces)
        d := i ( n veces )
        for j:= i+1 to n do ( n - 1 ) ( n - 2) … ( n - (n-1) ) veces
                if a[j] < a[d] then d:= j  ( comparación )
        od
        swap ( n veces )
od
```

- Conteo de comparaciones =

(n - 1) + (n - 2) + … + 1 = sumatoria desde j hasta n - 1 de 1
= n(n-1)/2 = n^2

¿Nombre? ordenar_array

```
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
```

¿Qué hace? f nos devuelve un arreglo en donde sus elementos indican que índices de otro arreglo deben ir en esos lugares para que el otro arreglo este ordenado.
¿Cómo lo hace? f toma un arreglo de n elementos y lo recorre de izquierda a derecha, guarda en la solución todas las posiciones del arreglo en orden. Luego guarda en una variable la posición del elemento actual y lo interpreta como “el mínimo” y si en las posiciones del arreglo que me indican los elementos de la solución, encuentra otro menor, actualiza la variable. Finalmente intercambia los elementos de la solución entre la posición actual y la variable del mínimo encontrado.
¿Qué orden es?

```
fun f(a: array[1..n] of nat) ret b : array[1..n] of nat
        var d: nat
        for i:= 1 to n do ( n veces, no es comparación)
                b[i] := i
        od
        for i:= 1 to n do ( n veces )
                d:= i ( n veces, no importa )
                for j:= i+1 to n do (por cada i se ejecuta (n-1) (n-2) … (n - (n-1) veces))
                        if a[b[j]] < a[b[d]] then d:= j  (SI IMPORTA)
```

```
            od
            swap(b, i, d) ( n veces, no importa )
        od
end fun
```

Contando las comparaciones realizadas:
(n-1) (n-2) … (n - (n-1))) = (n-1)n/2

¿Nombre? ordenar_arreglo_sin_ordenarlo

5. Dada la especificación del tad Cola:

spec Queue **of** T **where**

constructors
fun empty_queue() **ret** q : Queue **of** T
{- *crea una cola vacía.* -}

proc enqueue (**in**/**out** q : Queue **of** T, **in** e : T)
{- *agrega el elemento* e *al final de la cola* q. -}

operations
fun is_empty_queue(q : Queue **of** T) **ret** b : Bool
{- *Devuelve* True *si la cola es vacía* -}

fun first(q : Queue **of** T) **ret** e : T
{- *Devuelve el elemento que se encuentra al comienzo de* q. -}
{- **PRE:** not is_empty_queue(q) -}

proc dequeue (**in**/**out** q : Queue **of** T)
{- *Elimina el elemento que se encuentra al comienzo de* q. -}
{- **PRE:** not is_empty_queue(q) -}

Implementá el TAD utilizando punteros (siguiendo la idea de nodos enlazados), de manera que todas las operaciones (y los constructores) sean de orden **constante**.

```
implement Queue of T where

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

type Queue of T = tuple
                    front: pointer to (Node of T)
                    last: pointer to (Node of T)
                  end tuple

fun empty_queue() ret q : Queue of T
      alloc(q)
      q->front := null
      q->last := null
end fun
```

```
proc enqueue (in/out q: Queue of T, in e: T)
      var new_node: pointer to (Node of T)
      alloc(new_node)
      new_node->elem := e
      new_node->next := null
      if not is_empty_queue(q) then
            q->last->next := new_node
      else
            q->first := new_node;
            q->last := new_node;
      fi
end proc

fun is_empty_queue(q: Queue of T) ret b: Bool
      b := q->first == null and q->last == null
end fun

fun first(q: Queue of T) ret e: T
      e := q->first
end fun

proc dequeue (in/out q: Queue of T)
      var killme: pointer to (Node of T)
      killme := q->first;
      q->first := q->first->next
      free(killme)
end proc
```