# Práctico 3

## 1.3)

3. Se desea realizar un viaje en un automóvil con autonomía $A$ (en kilómetros), desde la localidad $l_0$ hasta la localidad $l_n$ pasando por las localidades $l_1, \ldots, l_{n-1}$ en ese orden. Se conoce cada distancia $d_i \leq A$ entre la localidad $l_{i-1}$ y la localidad $l_i$ (para $1 \leq i \leq n$), y se sabe que existe una estación de combustible en cada una de las localidades.

   Escribir un algoritmo que compute el menor número de veces que es necesario cargar combustible para realizar el viaje, y las localidades donde se realizaría la carga.

   Suponer que inicialmente el tanque de combustible se encuentra vacío y que todas las estaciones de servicio cuentan con suficiente combustible.

Demorar siempre lo máximo posible la carga

Voy a representar las localidades en un de `strings`, y las distancias en un arreglo de `nats`, el resultado va a ser una tupla, de una lista de un `nat`, con la cantidad de localidades, y una lista de `strings` con las localidades

```
type cargas = tuple
                  cantidad : nat
                  localidades : list of string
              end tuple

{- PRE: ∀j ∈ [1..n] : d[j] ≤ A -}
fun cargasDeCombustible(A : real,
                        l : array[0..n] of string,
                        d : array[1..n] of real)
                       ret cargas : cargas
     var autonomía : real {- Va a representar cuánto combustible hay en el tanque -}
     autonomía ≔ 0
     cargas.cantidades ≔ 0
     cargas.localidades ≔ empty_list()
     for j ≔ 0 to n - 1 do
          if autonomía < d[j + 1] →
               autonomía ≔ A
               addr(cargas.localidades, l[j])
               cargas.cantidades ≔ cargas.cantidades + 1
          fi
          autonomía ≔ autonomía - d[j + 1]
     od
end fun
```

1.4)

4. En numerosas oportunidades se ha observado que cientos de ballenas nadan juntas hacia la costa y quedan varadas en la playa sin poder moverse. Algunos sostienen que se debe a una pérdida de orientación posiblemente causada por la contaminación sonora de los océanos que interferiría con su capacidad de inter-comunicación. En estos casos los equipos de rescate realizan enormes esfuerzos para regresarlas al interior del mar y salvar sus vidas.

   Se encuentran $n$ ballenas varadas en una playa y se conocen los tiempos $s_1, s_2, \ldots, s_n$ que cada ballena es capaz de sobrevivir hasta que la asista un equipo de rescate. Dar un algoritmo voraz que determine el orden en que deben ser rescatadas para salvar el mayor número posible de ellas, asumiendo que llevar una ballena mar adentro toma tiempo constante $t$, que hay un único equipo de rescate y que una ballena no muere mientras está siendo regresada mar adentro.

Vos a representar cada ballena con una tupla, de un identificador como `string`, y el tiempo de supervivencia como `nat`, el imput de la función va a ser un arreglo de esas tuplas.
La idea es rescatar siempre a la que más pronto se moriría (pero que todavía está viva)

```
type ballena = tuple
                   id : string
                   sup : nat
               end tuple

fun orden_ballenas(b1, b2 : ballena) ret res : bool
     res ≔ b1.sup ≤ b2.sup
end fun

fun rescateDeBallenas(tiempos : array[1..n] of ballena,
                      t : nat)
                      ret ordenRescate : list of ballena
     var array_sorted : array[1..n] of ballena
         tt : nat {- Variable para llevar la cuenta del tiempo transcurrido -}
     array_sorted ≔ sort_by(orden_ballenas, array)
     tt ≔ 0
     ordenRescate ≔ empty_list()
     for j ≔ 1 to n do
          if tt ≤ (array_sorted[j]).sup →
               addr(ordenRescate, array_sorted[j])
               tt ≔ tt + t
          fi
     od
end fun
```

Con conjuntos:

```
fun rescateDeBallenas(tiempos : Set of ballena,
                      t : nat)
```

```
                    ret ordenRescate : list of ballena
    var tiempos2 : Set of ballena
        tt : nat {- Variable para llevar la cuenta del tiempo transcurrido -}
        min_ballena : ballena
    tiempos2 := copy_set(tiempos)
    tt := 0
    ordenRescate := empty_list()
    while ¬is_empty_set(tiempos2) do
        min_ballena := min_from_set_by(orden_ballenas, tiempos2)
        remove_set(tiempos2, min_ballena)
        if (tt ≤ ballena→sup) →
            addr(ordenRescate, min_ballena)
            tt := tt + t
        fi
    od
    set_destroy(tiempos2)
end fun
```

## 1.5)

5. Sos el flamante dueño de un teléfono satelital, y se lo ofrecés a tus $n$ amigos para que lo lleven con ellos cuando salgan de vacaciones el próximo verano. Lamentablemente cada uno de ellos irá a un lugar diferente y en algunos casos, los períodos de viaje se superponen. Por lo tanto es imposible prestarle el teléfono a todos, pero quisieras prestárselo al mayor número de amigos posible.

   Suponiendo que conoces los días de partida y regreso ($p_i$ y $r_i$ respectivamente) de cada uno de tus amigos, ¿cuál es el criterio para determinar, en un momento dado, a quien conviene prestarle el equipo?

   Tener en cuenta que cuando alguien lo devuelve, recién a partir del día siguiente puede usarlo otro. Escribir un algoritmo voraz que solucione el problema.

La idea es que se lo presto al que primero lo va a devolver

Voy a representar a la gente con una tupla con su nombre, su fecha de partida, y su fecha de retorno, y el resultado como una lista de personas

```
type persona =
    tuple
        nombre : string
        partida : nat
        retorno : nat
    end tuple


fun préstamos(ps : array[1..n] of persona) ret res : list of persona
    var ps_ordenado : array[1..n] of persona
        da : nat {- Variable para llevar la cuenta del día actual -}
    ps_ordenado := sort_by_retorno(ps) {- sort_by_retorno devuelve una versión ordenada según fecha de retorno de ps -}
```

```
      da := 0
      res := empty_list()
      for j := 1 to n do
            if da < (ps_ordenado[j]).partida →
                  addr(res, ps_ordenado[j])
            fi
      od
end fun
```

## 1.6)

6. Para obtener las mejores facturas y medialunas, es fundamental abrir el horno el menor número de veces posible. Por supuesto que no siempre es fácil ya que no hay que sacar nada del horno demasiado temprano, porque queda cruda la masa, ni demasiado tarde, porque se quema.

   En el horno se encuentran $n$ piezas de panadería (facturas, medialunas, etc). Cada pieza $i$ que se encuentra en el horno tiene un tiempo mínimo necesario de cocción $t_i$ y un tiempo máximo admisible de cocción $T_i$. Si se la extrae del horno antes de $t_i$ quedará cruda y si se la extrae después de $T_i$ se quemará.

   Asumiendo que abrir el horno y extraer piezas de él no insume tiempo, y que $t_i \leq T_i$ para todo $i \in \{1, \ldots, n\}$, ¿qué criterio utilizaría un algoritmo voraz para extraer todas las piezas del horno en perfecto estado (ni crudas ni quemadas), abriendo el horno el menor número de veces posible? Implementarlo.

La idea es cuando llega un tiempo $T_i$ abrir el horno, y sacar todas las piezas cuyo $t_i$ sea menor al actual (osea, las que ya están listas)

El resultado de la función es una lista con los tiempos en los que se tiene que abrir el horno (cada vez que se abre el horno se debe sacar todo lo que ya está listo)

```
{- PRE: ⟨∀i ∈ {1..n} : t[i] ≤ T[i]⟩ -}
fun horno(t, T : array[1..n] of Num) ret res : List of Num
      var ord_T : array[1..n] of Nat
          j : Nat
          e : Num
      ord_T := indices_ordenado(T)
      {- indices_ordenado es una función que devuelve un arreglo de índices de T, que tiene las posiciones en las que iría cada elemento si se ordenase (de menor a mayor) T, osea, mapear ord_T con (λi → T[i]) daría una versión ordenada de T -}
      j := 1
      res := empty_list()
      while j ≤ n do
            {- Saco la pieza actual, y todas las que siguen con T mayor que puedo sacar hasta que hay una que todavía no puedo sacar (t mayor) -}
            e := T[ord_T[j]]
            addl_list(res, e)
            while t[ord_T[j]] ≤ e do
                  j := j + 1
            od
      od
end fun
```

1.7)

7. Un submarino averiado descansa en el fondo del océano con $n$ sobrevivientes en su interior. Se conocen las cantidades $c_1, \ldots, c_n$ de oxígeno que cada uno de ellos consume por minuto. El rescate de sobrevivientes se puede realizar de a uno por vez, y cada operación de rescate lleva $t$ minutos.

   (a) Escribir un algoritmo que determine el orden en que deben rescatarse los sobrevivientes para salvar al mayor número posible de ellos antes de que se agote el total $C$ de oxígeno.

   (b) Modificar la solución anterior suponiendo que por cada operación de rescate se puede llevar a la superficie a $m$ sobrevivientes (con $m \leq n$).

Primero sacar a los que consumen más oxígeno

Devuelvo una lista con los índices en el arreglo `c` de las personas que hay que ir sacando

```
fun rescate(c : array[1..n] of Num, t : Num) ret res : List of Nat
      var is_c : array[1..n] of Nat
          h : Num
      is_c ≔ index_sort(c)
      {- index_sort es una función que devuelve un arreglo de índices de c, que tiene las posiciones en
las que iría cada elemento si se ordenase (de menor a mayor) c, osea, mapear is_c con (λi → c[i])
daría una versión ordenada de c -}
      h ≔ 0
      res ≔ empty_list()
      for j ≔ 1 to n do
            if h ≤ c[is_c[j]] →
                  addl_list(res, is_c[j])
                  h ≔ h + t
            fi
      od
end fun

fun rescate(c : array[1..n] of Num, t : Num, m : nat) ret res : List of Nat
      var is_c : array[1..n] of Nat
          h : Num
          i, j : Nat
      is_c ≔ index_sort(c)
      {- index_sort es una función que devuelve un arreglo de índices de c, que tiene las posiciones en
las que iría cada elemento si se ordenase (de menor a mayor) c, osea, mapear is_c con (λi → c[i])
daría una versión ordenada de c -}
      h ≔ 0
      j ≔ 1
      res ≔ empty_list()
      while j ≤ n do
            i ≔ 1
            while i ≤ m ∧ j ≤ n do
                  if h ≤ c[is_c[j]] →
```

```
                    addl_list(res, is_c[j])
                    i ≔ i + 1
               fi
               j ≔ j + 1
          od
          h ≔ h + t
     od
end fun
```

## 1.8)

8. Usted vive en la montaña, es invierno, y hace mucho frío. Son las 10 de la noche. Tiene una voraz estufa a leña y $n$ troncos de distintas clases de madera. Todos los troncos son del mismo tamaño y en la estufa entra solo uno por vez. Cada tronco $i$ es capaz de irradiar una temperatura $k_i$ mientras se quema, y dura una cantidad $t_i$ de minutos encendido dentro de la estufa. Se requiere encontrar el orden en que se utilizarán la menor cantidad posible de troncos a quemar entre las 22 y las 12 hs del día siguiente, asegurando que entre las 22 y las 6 la estufa irradie constantemente una temperatura no menor a $K1$; y entre las 6 y las 12 am, una temperatura no menor a $K2$.

```
type tronco =
     tuple
          k : real
          t : real
     end tuple

fun uso_de_troncos(leña : set of tronco) ret quema : list of troncos
     var copia_leña : set of tronco

     copia_leña ≔ copy_set(copia_leña)

end fun


fun orden_de_temperatura(t1, t2 : tronco) ret res : bool
     res ≔ t1.k ≤ t2.k
end fun
```

## 1.9)

9. (*sobredosis de limonada*) Es viernes a las 18 y usted tiene ganas de tomar limonada con sus amigos. Hay $n$ bares cerca, donde cada bar $i$ tiene un precio $P_i$ de la pinta de limonada y un horario de happy hour $H_i$, medido en horas a partir de las 18 (por ejemplo, si el happy hour del bar $i$ es hasta las 19, entonces $H_i = 1$), en el cual la pinta costará un 50% menos. Usted toma una cantidad fija de 2 pintas por hora y no se considera el tiempo de moverse de un bar a otro. Se desea obtener el menor dinero posible que usted puede gastar para tomar limonada desde las 18 hasta las 02 am (es decir que usted tomará 16 pintas) eligiendo en cada hora el bar que más le convenga.

El criterio es tomar en la que sea más barata en el momento

```
fun sobredosisDeLimonada(H : array[1..n] of Nat, P : array[1..n] of Num) ret r : Num
      var m : Num
      r ≔ 0
      for h ≔ 0 to 7 do
            m ≔ ∞
            for i ≔ 1 to n do
                  m ≔ m `min` ifThenElse(H_i < h, P_i, P_i/2)
            od
            r ≔ r + m
      od
end fun
```

Otra forma:

```
fun sobredosisDeLimonada(H : array[1..n] of Nat, P : array[1..n] of Num) ret r : Num
      var m : Num
      r ≔ 0
      for h ≔ 0 to 7 do
            r ≔ r + ⟨Min i ∈ {1..n} : ifThenElse(H _i < h, P_i, P_i/2)⟩
      od
end fun
```

## 2.3)

3. Usted quiere irse de vacaciones y debe elegir una ciudad entre K posibles que le interesan. Como no dispone de mucho dinero, desea que el viaje de ida hacia la ciudad pueda realizarse con a lo sumo L litros de nafta.

   (a) Dé un algoritmo que, dado un grafo representado por una matriz E : **array**[1..n,1..n] **of** Nat, donde el elemento E[i,j] indica el costo en litros de nafta necesario para ir desde la ciudad i hasta la ciudad j; un conjunto C de vértices entre 1 y n, representando las ciudades que quieren visitarse; un vértice v, representando la ciudad de origen del viaje; y un natural L, indicando la cantidad de litros de nafta total que puede gastar; devuelva un conjunto D de aquellos vértices de C que puede visitar con los L litros.

   (b) Ejecute el algoritmo implementado en el inciso anterior para el grafo descripto en el siguiente gráfico, con vértices 1,2,...11, tomando C = {11,5,10,7,8} como las ciudades de interés, disponiendo de L = 40 litros de nafta. ¿Cuáles son los posibles destinos de acuerdo a su presupuesto?

   Ayuda: Utilice el algoritmo de Dijkstra.

```
{- PRE: ∀c ∈ C : 1 ≤ c ≤ n -}
```

```
fun ciudadesVisitables(E : array[1..n,1..n] of num, C : Set of nat, v : nat, L : num)
ret D : Set of nat
      var res_Dijkstra : array[1..n] of num
          C_copy : Set of nat
          temp : nat
      C_copy ≔ copy_set(C)
      D ≔ empty_set()
      res_Dijkstra ≔ Dijkstra(E, v)
      while ¬is_empty_set(C_copy) do
            temp ≔ estract_from_set(C_copy) // Extrae y elimina
            if res_Dijkstra[temp] ≤ L →
                  set_add(D, temp)
            fi
      od
      destroy_set(C_copy)
end fun
```

## 3.1)

1. Modifique el código del algoritmo que resuelve el problema de la moneda utilizando backtracking, de manera que devuelva qué monedas se utilizan, en vez de sólo la cantidad.

```
fun cambio(j : Nat, C : Set of Nat) ret S : Set of Nat
      var c : Nat
          C_aux, cambio_C_aux, cambio_C_mas_c : Set of Nat
      if j = 0 →
            S ≔ empy_set()
      elif is_empty(C) →
            abort
      else
            C_aux ≔ set_copy(C)
            c ≔ get(C)
            elim(C_aux, c)
            cambio_C_aux ≔ cambio(j, C_aux)
            cambio_C_mas_c ≔ set_add(c, cambio(j-c, C))
            S ≔ (□ j < c ∨ set_length(cambio_C_aux) < set_length(cambio_C_mas_c)
                        → cambio_C_aux
                  □ si no
                        → cambio_C_mas_c
                  )
            set_destroy(C_aux)
      fi
end fun
```

## 3.3)

3. Una panadería recibe $n$ pedidos por importes $m_1, \ldots, m_n$, pero sólo queda en depósito una cantidad $H$ de harina en buen estado. Sabiendo que los pedidos requieren una cantidad $h_1, \ldots, h_n$ de harina (respectivamente), determinar el máximo importe que es posible obtener con la harina disponible.

`mejoresPedidos(i, h)` ≐ "El máximo importe con `h` harina teniendo en cuenta los primeros `i` elementos"

```
mejoresPedidos : (Nat, Num) → Num
mejoresPedidos(i, h) ≐
      □ i = 0 ∨ h ≤ 0 → 0
      □ h_i > h → mejoresPedidos(i - 1, h)
      □ si no → max
                (m_i + mejoresPedidos(i - 1, h - h_i))
                (mejoresPedidos(i - 1, h))
```

La llamada principal sería:
`mejoresPedidos(n, H)`

## 3.4)

4. Usted se encuentra en un globo aerostático sobrevolando el océano cuando descubre que empieza a perder altura porque la lona está levemente dañada. Tiene consigo $n$ objetos cuyos pesos $p_1, \ldots, p_n$ y valores $v_1, \ldots, v_n$ conoce. Si se desprende de al menos $P$ kilogramos logrará recuperar altura y llegar a tierra firme, y afortunadamente la suma de los pesos de los objetos supera holgadamente $P$. ¿Cuál es el menor valor total de los objetos que necesita arrojar para llegar sano y salvo a la costa?

La llamada principal del problema es `tiradaObjetos(n, P)`

`tiradaObjetos(i, h)` ≐ "El menor valor que se puede arrogar para perder `h` kilogramas, usando los primeros `i` objetos"

```
tiradaObjetos : (nat, num) → num
tiradaObjetos(i, h) ≐
      □ h ≤ 0 → 0
      □ i = 0 → ∞
      □ si no → min
                (v_i + tiradaObjetos(i-1, h - p_i))
                (tiradaObjetos(i-1, h))
```

## 3.5)

5. Sus amigos quedaron encantados con el teléfono satelital, para las próximas vacaciones ofrecen pagarle un alquiler por él. Además del día de partida y de regreso ($p_i$ y $r_i$) cada amigo ofrece un monto $m_i$ por día. Determinar el máximo valor alcanzable alquilando el teléfono.

```
alquileres : (Nat, {Nat}) → Num
```

`alquileres(i, d)` ≐ "El valor máximo alcanzable alquilando a primeros `i` amigos, sin usar los días de `d`"

```
alquileres(i, d) ≐
      □ i = 0 → 0
      □ {pᵢ,...,rᵢ} ∩ d = {} → max
                                (mᵢ * (rᵢ - pᵢ) + alquileres(i - 1, d ∪ {pᵢ,...,rᵢ})
                                alquileres(i - 1, d)
      □ si no → alquileres(i - 1, d)
```

La llamada que obtiene el resultado sería:
`alquileres(n, {})`

```
alquileres : nat → num
```

`alquileres(d)` ≐ "El máximo valor alcanzable a partir del dia `d` (inclusive)"

```
alquileres(d) ≐
      □ d > max(p) → 0
      □ si no → max alquileres(d + 1) ⟨max i : pᵢ = d : mᵢ + alquileres(rᵢ + 1)⟩
```

La llamada principal que resuelve el problema es:

```
alquileres(min(p))
```

## 3.6)

6. Un artesano utiliza materia prima de dos tipos: $A$ y $B$. Dispone de una cantidad $MA$ y $MB$ de cada una de ellas. Tiene a su vez pedidos de fabricar $n$ productos $p_1, \ldots, p_n$ (uno de cada uno). Cada uno de ellos tiene un valor de venta $v_1, \ldots, v_n$ y requiere para su elaboración cantidades $a_1, \ldots, a_n$ de materia prima de tipo $A$ y $b_1, \ldots, b_n$ de materia prima de tipo $B$. ¿Cuál es el mayor valor alcanzable con las cantidades de materia prima disponible?

```
pedidos : (nat, num, num) → num
```

`pedidos(i, ma, mb)` ≐ “El mayor valor alcanzable con los primeros `i` pedidos, si se tienen `ma` y `mb` madera de tipo `A` y `B` respectivamente”

```
pedidos(i, ma, mb) ≐
      □ i = 0 → 0
      □ a_i < ma ∨ b_i < mb → pedidos(i - 1, ma, mb)
      □ si no → max
                  (v_i + pedidos(i - 1, ma - a_i, mb - b_i))
                  pedidos(i - 1, ma, mb)
```

La llamada que obtiene el resultado sería:
`pedidos(n, MA, MB)`

## 3.7)

7. En el problema de la mochila se buscaba el máximo valor alcanzable al seleccionar entre $n$ objetos de valores $v_1, \ldots, v_n$ y pesos $w_1, \ldots, w_n$, respectivamente, una combinación de ellos que quepa en una mochila de capacidad W. Si se tienen dos mochilas con capacidades $W_1$ y $W_2$, ¿cuál es el valor máximo alcanzable al seleccionar objetos para cargar en ambas mochilas?

```
dosMochilas : (nat, num, num) → num
```

`dosMochilas(i, c1, c2)` ≐ “El máximo valor alcanzable con los primeros `i` elementos, poniendo hasta `c1` y `c2` peso en las mochilas $W_1$ y $W_2$ respectivamente”

```
dosMochilas(i, c1, c2) ≐
      □ i = 0 → 0
      □ w_i < c1 ∧ w_i < c2 → max
                              (v_i + dosMochilas(i - 1, c1 - w_i, c2))
                              (v_i + dosMochilas(i - 1, c1, c2 - w_i))
                              dosMochilas(i - 1, c1, c2)
      □ w_i < c1 → max
                    (v_i + dosMochilas(i - 1, c1 - w_i, c2))
                    dosMochilas(i - 1, c1, c2)
      □ w_i < c2 → max
                    (v_i + dosMochilas(i - 1, c1, c2 - w_i))
                    dosMochilas(i - 1, c1, c2)
      □ si no → dosMochilas(i - 1, c1, c2)

dosMochilas(i, c1, c2) ≐
      □ i = 0 ∨ (c1 = 0 ∧ c2 = 0) → 0
      □ c1 < 0 ∨ c2 < 0 → -∞
      □ si no → max
                  (v_i + dosMochilas(i - 1, c1 - w_i, c2))
                  (v_i + dosMochilas(i - 1, c1, c2 - w_i))
                  dosMochilas(i - 1, c1, c2)
```

La llamada que obtiene el resultado sería:
`dosMochilas(n, W1, W2)`

## 3.8)

8. Una fábrica de automóviles tiene dos líneas de ensamblaje y cada línea tiene $n$ estaciones de trabajo, $S_{1,1}, \ldots, S_{1,n}$ para la primera y $S_{2,1}, \ldots, S_{2,n}$ para la segunda. Dos estaciones $S_{1,i}$ y $S_{2,i}$ (para $i = 1, \ldots, n$), hacen el mismo trabajo, pero lo hacen con costos $a_{1,i}$ y $a_{2,i}$ respectivamente, que pueden ser diferentes. Para fabricar un auto debemos pasar por $n$ estaciones de trabajo $S_{i_1,1}, S_{i_2,2} \ldots, S_{i_n,n}$ no necesariamente todas de la misma línea de montaje ($i_k = 1, 2$). Si el automóvil está en la estación $S_{i,j}$, transferirlo a la otra línea de montaje (es decir continuar en $S_{i',j+1}$ con $i' \neq i$) cuesta $t_{i,j}$. Encontrar el costo mínimo de fabricar un automóvil usando ambas líneas.

```
ensamblaje : (nat, (1|2)) → num
```

`ensamblaje(i, e)` ≐ "El costo mínimo de recorrer el ensamblaje desde $S_{e,i}$ hasta el final"

```
ensamblaje(i, e) ≐
      □ i > n → 0
      □ si no → min
                  (a_{e,i} + ensamblaje(i + 1, e))
                  (t_{e,i} + a_{e',i} + ensamblaje(i + 1, e')
```

La llamada que obtiene el resultado es:
`min ensamblaje(1, 1) ensamblaje(1, 2)`

## 3.9)

9. El juego ↖U↑P↗ consiste en mover una ficha en un tablero de $n$ filas por $n$ columnas desde la fila inferior a la superior. La ficha se ubica al azar en una de las casillas de la fila inferior y en cada movimiento se desplaza a casillas adyacentes que estén en la fila superior a la actual, es decir, la ficha puede moverse a:

   - la casilla que está inmediatamente arriba,
   - la casilla que está arriba y a la izquierda (si la ficha no está en la columna extrema izquierda),
   - la casilla que está arriba y a la derecha (si la ficha no está en la columna extrema derecha).

   Cada casilla tiene asociado un número entero $c_{ij}$ $(i, j = 1, \ldots, n)$ que indica el puntaje a asignar cuando la ficha esté en la casilla. El puntaje final se obtiene sumando el puntaje de todas las casillas recorridas por la ficha, incluyendo las de las filas superior e inferior.

   Determinar el máximo y el mínimo puntaje que se puede obtener en el juego.

```
up_max : (nat, nat) → num
```

`up_max(i, j)` ≐ “El máximo puntaje obtenible saliendo desde el casillero `(i, j)` (contando $c_{i,j}$)”
`i` es la fila, y `j` la columna

```
up_max(i, j) ≐
      □ i > n → 0
      □ j < 1 ∨ j > n → -∞
      □ si no → c_{i,j} + max
                        up_max(i + 1, j - 1)
                        up_max(i + 1, j)
                        up_max(i + 1, j + 1)
```

La llamada principal que resuelve el problema es:

```
maximum [up_max(1, j) | j ← [1..n]]
⟨max j ∈ [1..n] : up_max(1, j)⟩
```

```
up_min : (nat, nat) → num
```

`up_min(i, j)` ≐ “El mínimo puntaje obtenible saliendo desde el casillero `(i, j)` (contando $c_{i,j}$)”
`i` es la fila, y `j` la columna

```
up_min(i, j) ≐
      □ i = n → c_{n,j}
      □ j < 1 ∨ j > n → ∞ {- Si me salgo para un costado, devuelvo el neutro de min -}
      □ si no → c_{i,j} + min
                        up_min(i + 1, j - 1)
                        up_min(i + 1, j)
                        up_min(i + 1, j + 1)
```

La llamada principal que resuelve el problema es:

```
minimum [up_min(1, j) | j ← [1..n]]
⟨Min j ∈ [1..n] : up_min(1, j)⟩
```

Los dos últimos ejercicios, también pueden resolverse planteando un grafo dirigido y recurriendo al algoritmo de Dijkstra. ¿De qué manera? ¿Serán soluciones más eficientes?

Si, y sería mucho más eficiente

## 4.1)

1. Dar una definición de la función cambio utilizando la técnica de programación dinámica a partir de la siguiente definición recursiva (backtracking):

$$cambio(i,j) = \begin{cases} 0 & j = 0 \\ \infty & j > 0 \wedge i = 0 \\ \min_{q \in \{0,1,\ldots,j \div d_i\}}(q + cambio(i-1, j - q * d_i)) & j > 0 \wedge i > 0 \end{cases}$$

`d : array[1..n] of Nat`
`d` tiene las denominaciones
`M : Nat`
`M` tiene el monto a pagar

```
fun cambio(d : array of Nat) out res : Nat
      var A : array[0..n,0..M]
      for i ≔ 0 to n do
            A_{i,0} ≔ 0 -- Caso base cuando no hay nada para pagar
      od
      for j ≔ 1 to M do
            A_{0,j} ≔ ∞ -- Caso base cuando no hay denominaciones, y si quedan cosas para pagar
      od
      for i ≔ 1 to n do
            for j ≔ 1 to M do
                  A_{i,j} ≔ A_{i-1,j} -- Caso no usar la denominación
                  if d_i ≠ 0 then
                        for q ≔ 1 to j `div` d_i do
                              A_{i,j} ≔ A_{i,j} `min` (q + A_{i-1,j-q*d_i})
                        od
                  fi
            od
      od
end fun
```

## 4.4)

4. Para cada una de las soluciones que propuso a los ejercicios del 3 al 9 del práctico de backtracking, dar una definición alternativa que utilice la técnica de programación dinámica. En los casos de los ejercicios 3, 5 y 7 modificar luego el algoritmo para que no sólo calcule el valor óptimo sino que devuelva la solución que tiene dicho valor (por ejemplo, en el caso del ejercicio 3, cuáles serían los pedidos que debería atenderse para alcanzar el máximo valor).

4.4 - 3)

3. Una panadería recibe $n$ pedidos por importes $m_1, \ldots, m_n$, pero sólo queda en depósito una cantidad $H$ de harina en buen estado. Sabiendo que los pedidos requieren una cantidad $h_1, \ldots, h_n$ de harina (respectivamente), determinar el máximo importe que es posible obtener con la harina disponible.

```
mejoresPedidos : (Nat, Num) → Num
mejoresPedidos(i, h) ≐
      □ i = 0 ∨ h ≤ 0 → 0
      □ hᵢ > h → mejoresPedidos(i - 1, h)
      □ si no → max
                (mᵢ + mejoresPedidos(i - 1, h - hᵢ))
                (mejoresPedidos(i - 1, h))
```

La llamada principal sería:
`mejoresPedidos(n, H)`

```
fun mejoresPedidos(m : array[1..n] of num, {- mᵢ es el importe del pedido i -}
                   h : array[1..n] of nat, {- hᵢ es la cantidad de harina que requiere el pedido i -}
                   H : num {- H es la cantidad de harina que hay disponible -}
                  ) out res : nat
      var p : array[0..n, 0..H] of nat {- pᵢ,ⱼ es el máximo importe con j harina usando solo los primeros i pedidos -}
          ha : nat
      ha ≔ K
      for i ≔ 1 to n do {- Inicializo la columna 0, en la que no hay harina -}
            pᵢ,₀ ≔ 0
      od
      for j ≔ 1 to H do {- Inicializo la fila 0, en la cual no hay pedidos -}
            p₀,ⱼ ≔ 0
      od
      for i ≔ 1 to n do
            for j ≔ 1 to H do
                  pᵢ,ⱼ ≔ (□ hᵢ > j → pᵢ₋₁,ⱼ
                          □ si no  → max (mᵢ + pᵢ₋₁,ⱼ₋hᵢ) pᵢ₋₁,ⱼ
                         )
            od
      od
      res ≔ pₙ,H
end fun
```

```
fun mejoresPedidos(
            m : array[1..n] of num, {- mᵢ es el importe del pedido i -}
            h : array[1..n] of nat, {- hᵢ es la cantidad de harina que requiere el pedido i -}
            H : num {- H es la cantidad de harina que hay disponible -}
            ) out res : set of nat {- Devuelve el conjunto de los índices de los pedidos que forman la solución óptima  -}
```

```
      var p : array[0..n, 0..H] of (set of nat)
      {- p_{i,j} es el conjunto de pedidos que maximizan el importe con j harina usando solo los primeros i pedidos -}
      for i ≔ 1 to n do {- Inicializo la columna 0, en la que no hay harina -}
            p_{i,0} ≔ {} {- Conjunto vacío -}
      od
      for j ≔ 1 to H do {- Inicializo la fila 0, en la cual no hay pedidos -}
            p_{0,j} ≔ {}
      od
      for i ≔ 1 to n do {- Itero sobre los pedidos -}
            for j ≔ 1 to H do {- Itero sobre la cantidad de harina-}
                  if h_i > j  ∨  m_i + sum(p_{i-1,j-h_i}) < sum(p_{i-1,j}) →
                        p_{i,j} ≔ p_{i-1,j}
                  else
                        p_{i,j} ≔ p_{i-1,j-h_i} ∪ {i}  {- ∪ es la unión entre conjuntos -}
                  fi
            od
      od
      res ≔ p_{n,H}
end fun

fun mejoresPedidos(
            m : array[1..n] of num, {- m[i] es el importe del pedido i -}
            h : array[1..n] of nat, {- h[i] es la cantidad de harina que requiere el pedido i -}
            H : num {- H es la cantidad de harina que hay disponible -}
            ) out res : set of nat {- Devuelve el conjunto de los índices de los pedidos que forman la solución óptima  -}

      var p : array[0..n, 0..H] of (set of nat)
      {- p[i,j] es el conjunto de pedidos que maximizan el importe con j harina usando solo los primeros i pedidos -}
      for i ≔ 1 to n do {- Inicializo la columna 0, en la que no hay harina -}
            p[i,0] ≔ {} {- Conjunto vacío -}
      od
      for j ≔ 1 to H do {- Inicializo la fila 0, en la cual no hay pedidos -}
            p[0,j] ≔ {}
      od
      for i ≔ 1 to n do {- Itero sobre los pedidos -}
            for j ≔ 1 to H do {- Itero sobre la cantidad de harina -}
                  if h[i] > j  ∨  m[i] + sum(p[i-1,j-h[i]]) < sum(p[i-1,j]) →
                        p[i,j] ≔ p[i-1,j] {- Si la harina j no alcanza para el pedido i, o si alcanza, pero el importe de no usar el pedido i es mayor al de usarlo -}
                  else
                        p[i,j] ≔ p[i-1,j-h[i]] ∪ {i}  {- ∪ es la unión entre conjuntos -}
                  fi
            od
      od
      res ≔ p[n,H]
end fun
```

4.4 - 4)

4. Usted se encuentra en un globo aerostático sobrevolando el océano cuando descubre que empieza a perder altura porque la lona está levemente dañada. Tiene consigo $n$ objetos cuyos pesos $p_1, \ldots, p_n$ y valores $v_1, \ldots, v_n$ conoce. Si se desprende de al menos $P$ kilogramos logrará recuperar altura y llegar a tierra firme, y afortunadamente la suma de los pesos de los objetos supera holgadamente $P$. ¿Cuál es el menor valor total de los objetos que necesita arrojar para llegar sano y salvo a la costa?

La llamada principal del problema es `tiradaObjetos(n, P)`

`tiradaObjetos(i, h)` ≐ "El menor valor que se puede arrogar para perder `h` kilogramas, usando los primeros `i` objetos"

```
tiradaObjetos : (nat, num) → num
tiradaObjetos(i, h) ≐
      □ h ≤ 0 → 0
      □ i = 0 → ∞
      □ si no → min
                (v_i + tiradaObjetos(i-1, h - p_i))
                (tiradaObjetos(i-1, h))
```

```
fun tiradaObjetos(p : array[1..n] of nat, v : array[1..n] of num, P : nat) out res :
num
      var A : array[0..n, 0..P]
      for i ≔ 0 to n do
            A_{i,0} ≔ 0
      od
      for j ≔ 1 to P do
            A_{0,j} ≔ ∞
      od
      for i ≔ 1 to n do
            for j ≔ 1 to P do
                  A_{i,j} ≔ min (v_i + A_{i-1,h-p_i}) A_{i-1,h}
            od
      od
      res ≔ A_{n,P}
end fun
```

4.4 - 5)

`alquileres : nat → num`

`alquileres(d)` ≐ "El máximo valor alcanzable a partir del dia `d` (inclusive)"

```
alquileres(d) ≐
      □ d > max(p) → 0
      □ si no → max alquileres(d + 1) ⟨max i : p_i = d : m_i + alquileres(r_i + 1)⟩
```

La llamada principal que resuelve el problema es:

```
alquileres(min(p))
```

```
fun alquileres(p, r : array[1..n] of nat, m : array[1..n] of num) out res : num
      var A : array[min(p)..max(r)] of num
          min_d, max_d : nat
      min_d ≔ min(p)
      max_d ≔ max(r)
      A[max_d] ≔ 0  {- Caso base -}
      for d ≔ max_d - 1 downto min_d do
            A[d] ≔ A[d + 1]
            for j ≔ 1 to n do
                  if p[j] = d →
                        A[d] ≔ max A[d] A[r[j] + 1]
                  fi
            od
      od
      res ≔ A[min_d]
end fun

fun alquileres(p, r : array[1..n] of nat,
               m : array[1..n] of num
              ) out res : set of nat {- Conjunto de los índices de los alquileres que forman la solución -}
      var A : array[min(p)..max(r)] of set of nat
          min_d, max_d : nat
      min_d ≔ min(p)
      max_d ≔ max(r)
      A[max_d] ≔ {}  {- Caso base: conjunto vacío -}
      for d ≔ max_d - 1 downto min_d do
            A[d] ≔ A[d + 1]
            for j ≔ 1 to n do
                  if p[j] = d ∧ sum A[d] < sum A[r[j] + 1]→
                        A[d] ≔ {d} ∪ A[r[j] + 1]
                  fi
            od
      od
      res ≔ A[min_d]
end fun
```

4.4 - 6)

6. Un artesano utiliza materia prima de dos tipos: $A$ y $B$. Dispone de una cantidad $MA$ y $MB$ de cada una de ellas. Tiene a su vez pedidos de fabricar $n$ productos $p_1, \ldots, p_n$ (uno de cada uno). Cada uno de ellos tiene un valor de venta $v_1, \ldots, v_n$ y requiere para su elaboración cantidades $a_1, \ldots, a_n$ de materia prima de tipo $A$ y $b_1, \ldots, b_n$ de materia prima de tipo $B$. ¿Cuál es el mayor valor alcanzable con las cantidades de materia prima disponible?

`pedidos : (nat, num, num) → num`

`pedidos(i, ma, mb)` ≐ "El mayor valor alcanzable con los primeros `i` pedidos, si se tienen `ma` y `mb` madera de tipo `A` y `B` respectivamente"

```
pedidos(i, ma, mb) ≐
      □ i = 0 → 0
      □ a_i < ma ∨ b_i < mb → pedidos(i - 1, ma, mb)
      □ si no → max
                  (v_i + pedidos(i - 1, ma - a_i, mb - b_i))
                  pedidos(i - 1, ma, mb)
```

La llamada que obtiene el resultado sería:
`pedidos(n, MA, MB)`

```
fun pedidos(v : array[1..n] of num, a, b : array[1..n] of nat, A, B : nat) out res : num
      var tabla : array[0..n,0..A,0..B] of nat
      {- tabla[i,ma,mb] es el mayor valor alcanzable con los primeros i pedidos, si se tienen ma y mb madera de tipo A y B respectivamente -}
      for ma ≔ 0 to A do
            for mb ≔ 0 to B do
                  tabla[0,ma,mb] ≔ 0 {- Inicializo el caso sin pedidos -}
            od
      od
      for i ≔ 1 to n do {- Inicializo el caso de que no hay una de las maderas  -}
            tabla[i,0,0] ≔ 0
            for ma ≔ 1 to A do {- Inicializo el caso de que no hay madera tipo B -}
                  if b[i] = 0 ∧ a[i] ≤ ma →
                        tabla[i,ma,0] ≔ max tabla[i-1,ma,0] tabla[i-1,ma-a[i],0]
                  else
                        tabla[i,ma,0] ≔ tabla[i-1,ma,0]
                  fi
            od
            for mb ≔ 0 to B do {- Inicializo el caso de que no hay madera tipo A -}
                  if a[i] = 0 ∧ b[i] ≤ mb →
                        tabla[i,0,mb] ≔ max tabla[i-1,mb,0] tabla[i-1,mb-b[i],0]
                  else
                        tabla[i,mb,0] ≔ tabla[i-1,mb,0]
                  fi
            od
```

```
      od
      for i ≔ 1 to n do
            for ma ≔ 1 to A do
                  for mb ≔ 1 to B do
                        if a[i] < ma ∨ b[i] < mb →
                              tabla[i,ma,mb] ≔ tabla[i-1,ma,mb]
                        else
                              tabla[i,ma,mb] ≔ max
                                                (v[i] + tabla[i-1,ma-a[i],mb-b[i]])
                                                tabla[i-1,ma,mb]
                        fi
                  od
            od
      od
      res ≔ tabla[n,A,B]
end fun
```

4.4 - 7)

7. En el problema de la mochila se buscaba el máximo valor alcanzable al seleccionar entre $n$ objetos de valores $v_1, \ldots, v_n$ y pesos $w_1, \ldots, w_n$, respectivamente, una combinación de ellos que quepa en una mochila de capacidad W. Si se tienen dos mochilas con capacidades $W_1$ y $W_2$, ¿cuál es el valor máximo alcanzable al seleccionar objetos para cargar en ambas mochilas?

```
dosMochilas : (nat, num, num) → num
```

`dosMochilas(i, c1, c2)` ≐ "El máximo valor alcanzable con los primeros `i` elementos, poniendo hasta `c1` y `c2` peso en las mochilas $W_1$ y $W_2$ respectivamente"

```
dosMochilas(i, c1, c2) ≐
      □ i = 0 ∨ (c1 = 0 ∧ c2 = 0) → 0
      □ c1 < 0 ∨ c2 < 0 → -∞
      □ si no → max
                  (v_i + dosMochilas(i - 1, c1 - w_i, c2))
                  (v_i + dosMochilas(i - 1, c1, c2 - w_i))
                  dosMochilas(i - 1, c1, c2)
```

La llamada que obtiene el resultado sería:
`dosMochilas(n, W₁, W₂)`

```
fun dosMochilas(v : array[1..n] of num,
                w : array[1..n] of nat,
                W₁, W₂ : nat
               ) out res : num
      var A : array[0..n,0..W₁,0..W₂] of num
      for c1 ≔ 0 to W₁ do
            for c2 ≔ 0 to W₂ do
```

```
                A_{0,c1,c2} ≔ 0
            od
        od
        for i ≔ 0 to n do
            A_{i,0,0} ≔ 0
        od
        for i ≔ 0 to n do
            for c1 ≔ 0 to W_1 do
                for c2 ≔ 0 to W_2 do
                    A_{i,c1,c2} ≔ max
                                    A_{i-1,c1,c2}
                                    ifThenElse(c1 - w_i ≥ 0, v_i + A_{i-1,c1-w_i,c2}, -∞)
                                    ifThenElse(c2 - w_i ≥ 0, v_i + A_{i-1,c1,c2-w_i}, -∞)
                od
            od
        od
        res ≔ A_{n,W_1,W_2}
end fun


fun ifThenElse(a : Bool, b, c : T) out res : T
        if a →
            res ≔ b
        else
            res ≔ c
        fi
end fun
```

## 4.4 - 8)

8. Una fábrica de automóviles tiene dos líneas de ensamblaje y cada línea tiene $n$ estaciones de trabajo, $S_{1,1}, \ldots, S_{1,n}$ para la primera y $S_{2,1}, \ldots, S_{2,n}$ para la segunda. Dos estaciones $S_{1,i}$ y $S_{2,i}$ (para $i = 1, \ldots, n$), hacen el mismo trabajo, pero lo hacen con costos $a_{1,i}$ y $a_{2,i}$ respectivamente, que pueden ser diferentes. Para fabricar un auto debemos pasar por $n$ estaciones de trabajo $S_{i_1,1}, S_{i_2,2} \ldots, S_{i_n,n}$ no necesariamente todas de la misma línea de montaje ($i_k = 1, 2$). Si el automóvil está en la estación $S_{i,j}$, transferirlo a la otra línea de montaje (es decir continuar en $S_{i',j+1}$ con $i' \neq i$) cuesta $t_{i,j}$. Encontrar el costo mínimo de fabricar un automóvil usando ambas líneas.

```
ensamblaje : (nat, (1|2)) → num
```

ensamblaje(i, e) ≐ “El costo mínimo de recorrer el ensamblaje desde $S_{e,i}$ hasta el final”

```
ensamblaje(i, e) ≐
    □ i > n → 0
    □ si no → min
                (a_{e,i} + ensamblaje(i + 1, e))
```

$(t_{e,i} + a_{e',i} + \text{ensamblaje}(i + 1, e')$

La llamada que obtiene el resultado es:
`min ensamblaje(1, 1) ensamblaje(1, 2)`

```
fun ensamblaje(a, t : array[1..2,1..n] of num) ret res : num
      var A : array[1..2,1..n+1] of num
      {- A_{e,i} representa el costo mínimo de recorrer el ensamblaje desde S_e hasta el final -}
      for e ≔ 0 to 1 do
            A_{e,n+1} ≔ 0
      od
      for i ≔ n downto 1 do
            for e ≔ 1 to 2 do
                  A_{e,i} ≔ (a_{e,i} + A_{e,i+1}) `min` (t_{e,i} + a_{e',i} + A_{i+1,e'})
            od
      od
      res ≔ A_{1,1} `min` A_{2,1}
end fun
```

## 4.4 - 9)

9. El juego ↖U↑P↗ consiste en mover una ficha en un tablero de $n$ filas por $n$ columnas desde la fila inferior a la superior. La ficha se ubica al azar en una de las casillas de la fila inferior y en cada movimiento se desplaza a casillas adyacentes que estén en la fila superior a la actual, es decir, la ficha puede moverse a:

   - la casilla que está inmediatamente arriba,
   - la casilla que está arriba y a la izquierda (si la ficha no está en la columna extrema izquierda),
   - la casilla que está arriba y a la derecha (si la ficha no está en la columna extrema derecha).

   Cada casilla tiene asociado un número entero $c_{ij}$ $(i, j = 1, \ldots, n)$ que indica el puntaje a asignar cuando la ficha esté en la casilla. El puntaje final se obtiene sumando el puntaje de todas las casillas recorridas por la ficha, incluyendo las de las filas superior e inferior.

   Determinar el máximo y el mínimo puntaje que se puede obtener en el juego.

```
fun up_max(c : array[1..n,1..n] of num) ret res : num
      var A : array[1..n+1,0..n+1] of num
      {- A_{i,j} representa el máximo puntaje obtenible saliendo desde el casillero (i, j) (contando c_{i,j}) -}
      for j ≔ 1 to n do
            A_{n+1,j} ≔ 0 {- Caso base cuando se llega al final del tablero (neutro +) -}
      od
      for i ≔ 1 to n+1 do {- Caso base cuando se sale por un costado del tablero (neutro max) -}
            A_{i,0} ≔ -∞
            A_{i,m+1} ≔ -∞
      od
```

```
      for i ≔ n downto 1 do
            for j ≔ 1 to n do
                  A_{i,j} ≔ c_{i,j} + (A_{i+1,j-1} `max` A_{i+1,j} `max` A_{i+1,j+1})
            od
      od
      res ≔ -∞
      for i ≔ 1 to n do
            res ≔ res `max` A_{1,i}
      od
end fun


fun up_min(c : array[1..n,1..n] of num) ret res : num
      var A : array[1..n+1,0..n+1] of num
      {- A_{i,j} representa el mínimo puntaje obtenible saliendo desde el casillero (i, j) (contando c_{i,j}) -}
      for j ≔ 1 to n do
            A_{n+1,j} ≔ 0 {- Caso base cuando se llega al final del tablero (neutro +) -}
      od
      for i ≔ 1 to n+1 do {- Caso base cuando se sale por un costado del tablero (neutro min) -}
            A_{i,0} ≔ ∞
            A_{i,m+1} ≔ ∞
      od
      for i ≔ n downto 1 do
            for j ≔ 1 to n do
                  A_{i,j} ≔ c_{i,j} + (A_{i+1,j-1} `min` A_{i+1,j} `min` A_{i+1,j+1})
            od
      od
      res ≔ ∞
      for i ≔ 1 to n do
            res ≔ res `min` A_{1,i}
      od
end fun
```