1. (Algoritmos voraces)

   Dado un grafo dirigido G con costos no negativos en sus aristas, representado
   vértice v del mismo, el algoritmo de Dijkstra calcula, para cada vértice w del gr
   mínimo de v a w.

   (a) De qué manera podés modificar el algoritmo de Dijkstra (llamémosle algori
   de calcular costos de caminos desde v, calcule costos de caminos hacia v.
   se dijo, y dado un vértice v del mismo, calcule, para cada vértice w del gr
   mínimo de w a v. Escribí el algoritmo.

   (b) ¿Cómo podrías utilizar los algoritmos de Dijkstra y de Artskjid para cal
   costo del camino de costo mínimo de ida y vuelta de v a w. Incluso si no
   intentar resolver éste utilizando ambos algoritmos.

a)

```
fun Artskjid (L: array[1..n,1..n] of Nat, v: Nat) ret D:
array[1..n] of Nat
      var c: Nat
      var C: Set of Nat
      for i := 1 to n do
            add(C,i)
      od
      elim(C,v)
      for i:= 1 to n do
            D[i]:= L[i,v] {- costo inicial de ir de cada vértice w a v -
      }
      od
      do (not is_empty_set(C))→
            c:= "elijo elemento c de C tal que D[c] sea mínimo"
            elim(C,c)
            for j in C do
                  D[j]:= min(D[j],D[c]+L[j,c])
                  {- mínimo entre ir de j a v o ir de j a c y luego a v -}
            od
      od
end fun
```

b)

```
fun ida_vuelta (L: array[1..n,1..n] of Nat, v: Nat) ret D:
array[1..n] of Nat
      var I: array[1..n] of nat
      var v: array[1..n] of nat
      I := Dijsktra(L,v)
      V := Artskjid(L,v)
```

```
        for i := 1 to n do
            D[i] := I[i] + V[i]
        od
    end fun
```

2. (Backtracking) Como sabés que esta medianoche aumentarán todos los precios, parte posible del dinero D de que disponés. Hay n objetos para comprar, c Suponemos que $D < \sum_{i=1}^{n} v_i$, por lo que deberás elegir cuáles objetos compra debe determinar el máximo monto que podés gastar sin exceder el dinero D di

   Se pide lo siguiente:

   (a) Especificá precisamente qué calcula la función recursiva que resolverá el p 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:

- Se quiere gastar la máxima cantidad de dinero D posible
- Hay n objetos para comprar
- Sus precios son $v_i,...,v_n$
- Se debe determinar el máximo monto que podés gastar sin exceder el dinero D disponible

a) Función recursiva:

`compras(i, d)` “máximo monto que se puede gastar comprando i objetos sin exceder el dinero d disponible ”

b) Llamada principal:

`compras(n,D)`

c) Función matemática:

compras(i,d)

| 0	, si i = 0 v d = 0
| compras(i-1,d)	, si i > 0 & d > 0 & v_i > d
| max(compras(i-1,d), v_i + compras(i-1,d - v_i))	, si i > 0 & d > 0 & d ≥ v_i

3. (Programación Dinámica) Dados $c_1, c_2, \ldots, c_n$ y la siguiente definición recursiva escribí un programa que utilice la técnica de programación dinámica para calcu

$$m(i,j) = \begin{cases} c_i & \text{si } i = \\ m(i-1,j) + m(i,j+1) & \text{si } i > \end{cases}$$

Ayuda: Antes de comenzar, hacé un ejemplo para entender la definición rec $4, c_1 = 3, c_2 = 1, c_3 = 2, c_4 = 5$, y calcular $m(4,1)$.

n = 4
c1 = 3
c2 = 1
c3 = 2
c4 = 5

calcular m(4,1)

m(4,1) = m(3,1) + m(4,2)
= m(2,1) + m(3,2) + m(4,2)
= m(1,1) + m(2,2) + m(3,2) + m(4,2)
= c1 + c2 + m(3,2) + m(4,2)
= c1 + c2 + m(2,2) + m(3,3) + m(4,2)
= c1 + c2 + c2 + c3 + m(4,2)
= c1 + c2 + c2 + c3 + m(3,2) + m(4,3)
= c1 + c2 + c2 + c3 + m(3,3) + m(2,2) + m(4,3)
= c1 + c2 + c2 + c3 + c3 + c2 + m(4,3)
= c1 + c2 + c2 + c3 + c3 + c2 + m(3,3) + m(4,4)
= c1 + c2 + c2 + c3 + c3 + c2 + c3 + c4
= 3 + 1 + 1 + 2 + 2 + 1 + 2 + 5 = 17

	1	2	3	4
1	3	j > i	j > i	j > i
2	4	1	j > i	j > i
3	7	3	2	j > i
4	17	10	7	5

La tabla se puede llenar desde arriba hacia abajo, de izquierda a derecha puesto que necesita de la fila anterior y la columna siguiente para calcular el valor de una casilla. i = 2,3,..,n y j = i, ..., 1

```
fun suma( c: array[1..n] of nat ) ret res: nat
```

```
      {- variables -}
      dp: array[1..n,1..n] of nat

      {- casos base -}
      for i := 1 to n do
            dp[i,i] := c[i]
      od

      {- caso recursivo -}
      for i := 2 to n do
            for j := i-1 downto 1 do
                  dp[i,j] := dp[i-1,j] + dp[i,j+1]
            od
      od
      res := dp[4,1]
end fun
```

4. Para cada uno de los siguientes algoritmos determinar **por separado** ca

   (a) ¿Qué hace? ¿Cuáles son las precondiciones necesarias para haga es

   (b) ¿Cómo lo hace?

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

   (d) Proponer nombres más adecuados para los identificadores (de varia

```
fun f(a: array[1..n] of nat, i,j,x:nat) ret b: bool
    var d,e,f : nat
    d,f,b := i,j,false
    while (d ≤ f) do
          e:= d+f div 2
          if x < a[e] → f:= e-1
          [] x = a[e] → b:= true
          [] x > a[e] → d:= e+1
          fi
    od
end fun
```

```
fun g(a,b: ar
    var i : na
    i := 1
    while (i
          c[i]
          i:=
    od
end fun
```

Algoritmo 1:

```
fun f(a: array[1..n] of nat, i,j,x:nat) ret b: bool
```

```
      var d,e,f : nat
      d,f,b := i,j,false
      while (d ≤ f) do
            e:= d+f div 2 {- calcula el índice medio entre j e i -}
            if x < a[e] → f:= e-1
            [] x = a[e] → b:= true
            [] x > a[e] → d:= e+1
      od
end fun
```

¿Qué hace? ¿Cuáles son las precondiciones necesarias para que haga eso?

Nos dice si un natural se encuentra dentro de un segmento

{PRE: a[i] ≤ a[i+1] ∀ i ∈ {1,...,n-1}} dado que usa busqueda binaria

¿Cómo lo hace?

El algoritmo toma dos índices, un natural y la secuencia. Recorre el segmento indicado por los índices, de manera tq se calcula la posición media entre ambos, y si su elemento es igual al natural se devuelve true. Si es menor, el índice derecho decrementa en 1, si es mayor el índice izquierdo aumenta en 1, esto se ejecuta siempre y cuando el índice izquierdo sea menor o igual al derecho

¿Qué orden tiene?

En el peor de los casos, x no se encuentra en el segmento y en cada paso no solo hacemos una comparación más si no que se divide d+f por 2, por lo que debemos ver inicialmente cuántas veces podemos efectuar esta división más las comparaciones del condicional de dentro:

$\log_2(f - d + 1) \equiv \log_2(j - i + 1)$

Proponer un nombre:

esta_x

Algoritmo 2:

```
fun g(a,b: array[1..n] of nat) ret c: array[1..n] of bool
      var i : nat
      i := 1
      while (i ≤ n) do
            c[i]:= f(a, 1, n, b[i])
            i:= i+1
      od
end fun
```

¿Qué hace? ¿Cuáles son las precondiciones necesarias para que haga eso?

Nos dice que elementos de la secuencia b están en a.

¿Cómo lo hace?

Recorre una secuencia de izquierda a derecha y en cada posición guardamos el resultado de llamar a f y buscar en toda la secuencia de a, el elemento de b indicado por la posición actual.

¿Qué orden tiene?

El bucle se ejecuta n veces, dado que f tiene una complejidad de log_2(j - i + 1) en total el orden es de: n * log_2(n - 1 + 1) = o(n * log(n))

Proponer un nombre:

elementos_iguales

5. El TAD VipQueue es una variante del tipo abstracto Queue que cuenta co el cual modifica la VipQueue agregándole un elemento de modo "prefere operación adicional **hayVip** que indica si en la VipQueue hay algún elem

   El resto de las operaciones tienen el mismo tipo que en la versión Qu modificado:

   La operación **first** devuelve el primer elemento que haya ingresado co ingresado de modo normal, en caso que no haya ningún vip. La operac resulta de eliminar el elemento que la operación **first** devolvería.

   Se pide:

   (a) Escribí la especificación completa del tipo VipQueue.

   (b) Implementá el tipo VipQueue utilizando una estructura que contenga naturales.

a)

```
Spec VipQueue of T where

    Constructors

        fun empty_queue() ret q: VipQueue of T
        {- devuelve una cola vacía. -}

        proc enqueue(in/out q: VipQueue of T, in e: T)
        {- Agrega un elemento normal a la cola. -}

        proc enqueueVip(in/out q: VipQueue of T, in e: T)
        {- Agrega un elemento vip a la cola. -}

    Copy

        fun copy_queue( q: VipQueue of T) ret c: VipQueue of T

        {- Copia la cola q en c. -}

    Destroy
```

```
        proc copy_queue(in/out q: VipQueue of T) ret c: VipQueue
of T            {- Libera memoria en caso de ser necesario. -}

    Operations

        fun hay_Vip(q: VipQueue of T) ret b: bool
        {- Nos dice si en q hay un elemento vip -}

        fun is_empty_queue(q: VipQueue of T) ret b: bool
        {- Nos dice si una cola es vacía. -}

        fun first(q: VipQueue of T) ret e : T
        {- Devuelve el primer elemento que haya ingresado como
        vip, o el primer elemento normal. -}
        {- PRE: not is_empty_queue(q) -}

        proc dequeue(in/out q: VipQueue of T)
        {- Elimina el elemento que first devuelva -}
        {- PRE: not is_empty_queue(q) -}

end spec
```

b)

```
Implement VipQueue of T where

    type VipQueue = tuple
                        normal: array[1..N] of T
                        vip: array[1..N] of T
                        normal_size: nat
                        vip_size: nat
                    end tuple


    Constructors

        fun empty_queue() ret q: VipQueue of T
            q.normal_size := 0
            q.vip_size := 0
        end fun

        proc enqueue(in/out q: VipQueue of T, in e: T)
            q.normal_size := q.normal_size + 1
            q.normal[q.normal_size] := e
        end proc

        proc enqueueVip(in/out q: VipQueue of T, in e: T)
            q.vip_size := q.vip_size + 1
```

```
            q.vip[q.vip_size] := e
        end proc

    Copy

        fun copy_queue(q: VipQueue of T) ret c: VipQueue of T

            c := empty_queue()
            if(not is_empty_queue(q)) then
                c.vip_size := q.vip_size
                c.normal_size := q.normal_size

                for i := 1 to c.normal_size do
                    c.normal[i] := q.normal[i]
                od
                for j := 1 to c.vip_size do
                    c.vip[j] := q.vip[i]
                od
            fi
        end fun

    Destroy

        proc copy_queue(in/out q: VipQueue of T) ret c: VipQueue
of T            skip
        end proc

    Operations

        fun is_empty_queue(q: VipQueue of T)
            b := q.normal_size == 0 and q.vip_size == 0
        end fun

        fun hay_Vip(q: VipQueue of T) ret b: bool
            b := q.vip_size > 0
        end fun

        fun first(q: VipQueue of T) ret e : T
            if hay_Vip(q) then
                e := q.vip[1]
            else
                e := q.normal[1]
            fi
        end fun

        proc dequeue(in/out q: VipQueue of T)
            if hay_Vip(q) then
                for i := 1 to q.vip_size - 1 do
```

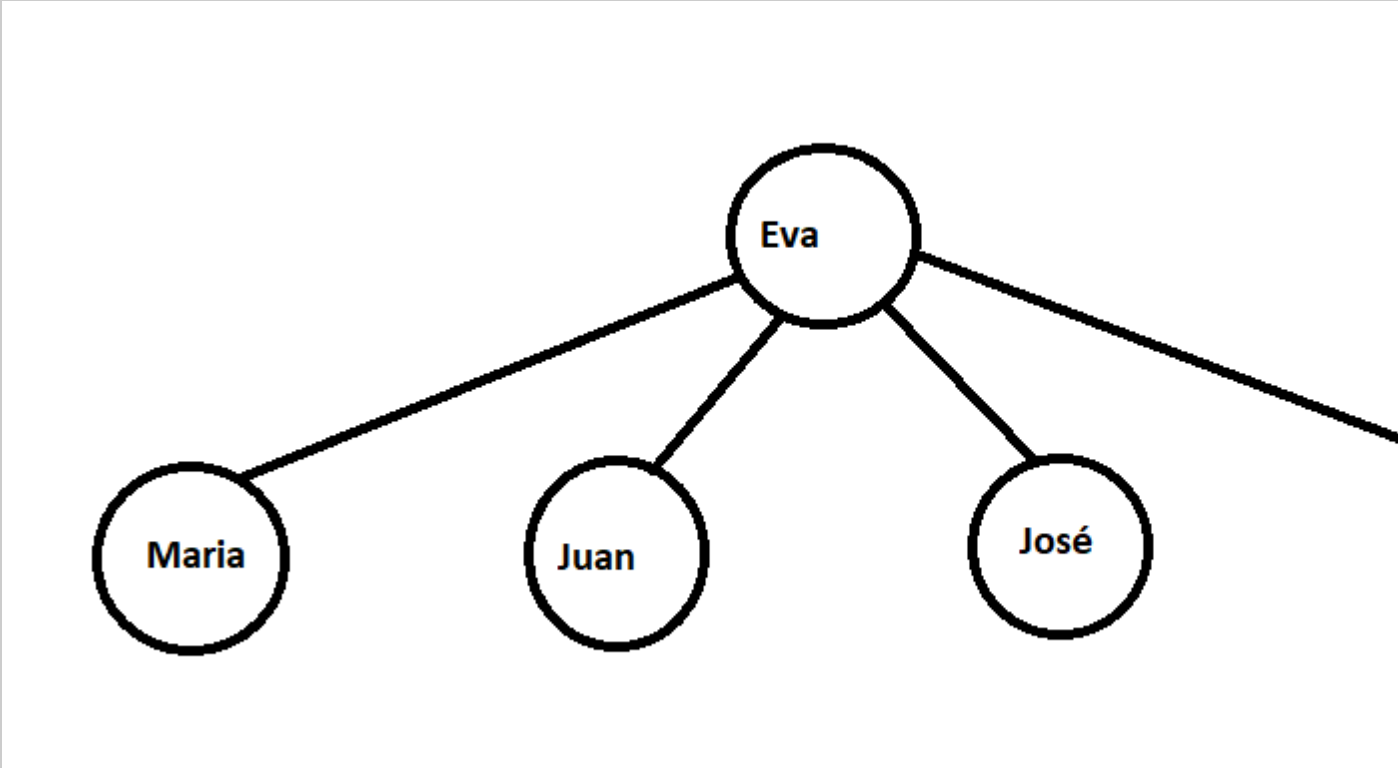
```
                    q.vip[i] := q.vip[i+1]
                od
                q.vip_size := q.vip_size - 1
            else
                for j := 1 to q.normal_size - 1 do
                    q.normal[j] := q.normal[j+1]
                od
                q.normal_size := q.normal_size - 1
            fi
        end proc

end implement
```

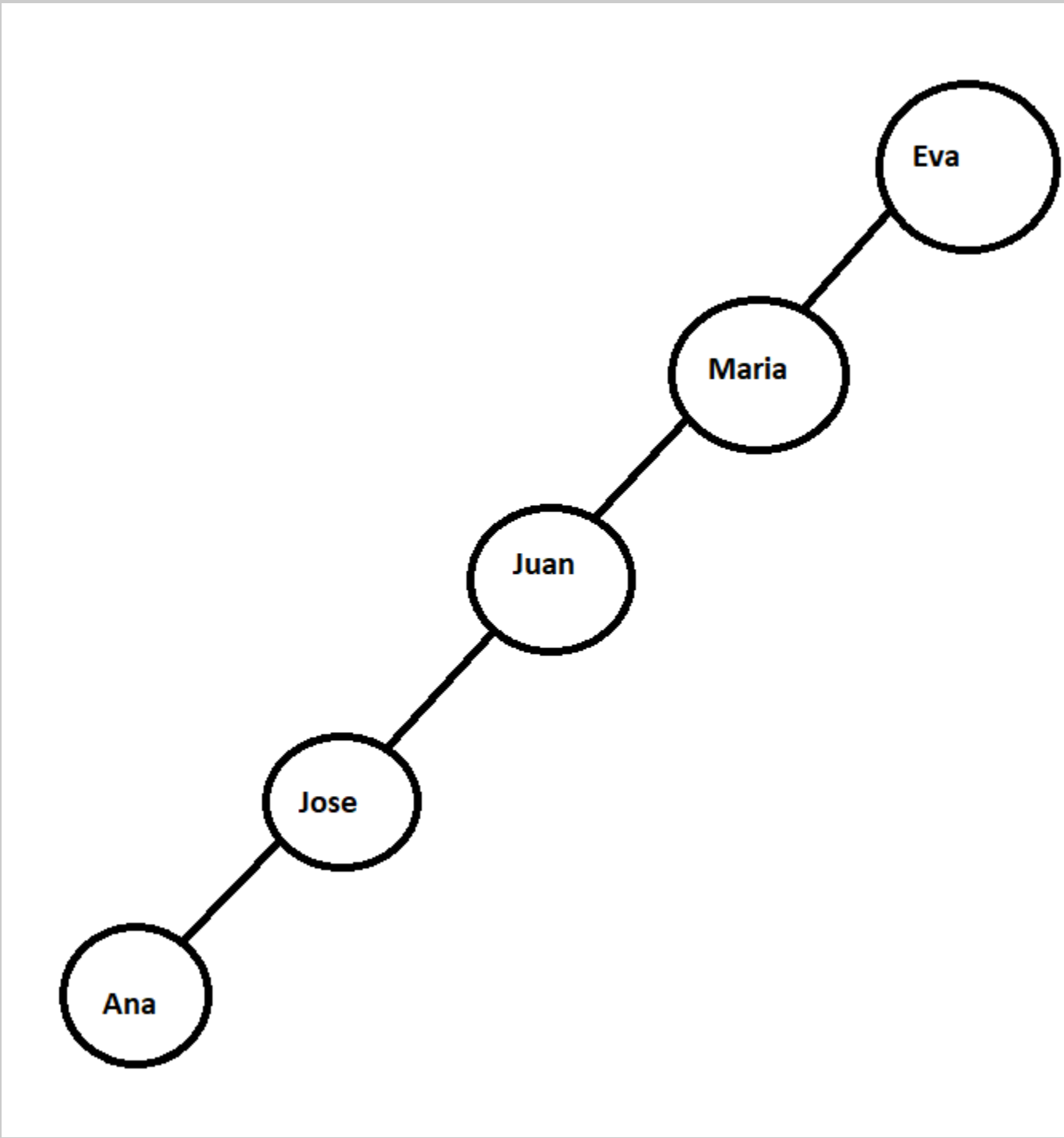
6. (Para alumnos libres)

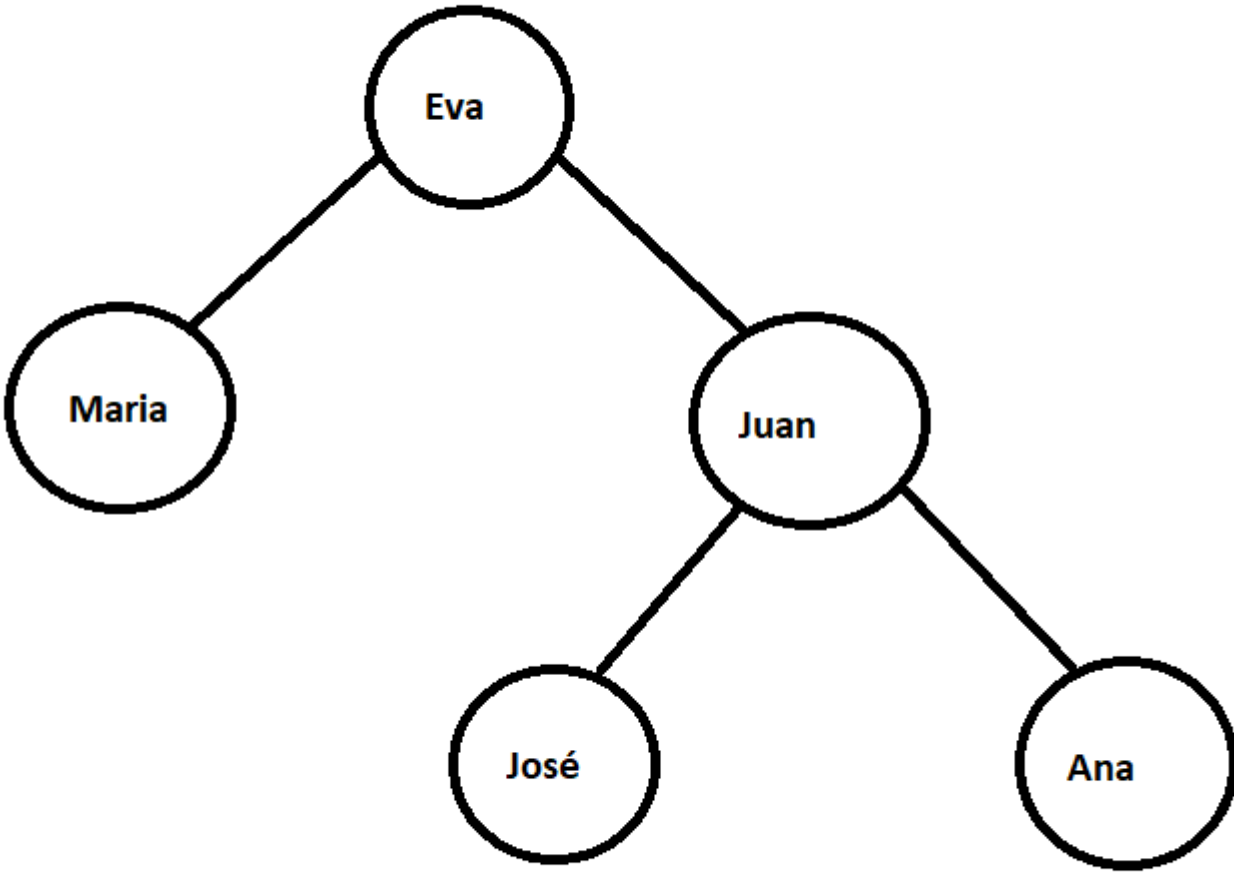
   (a) Graficá todos los árboles finitarios que al recorrerse en pre-orden el siguiente idéntico orden: eva, maría, juan, josé, ana.

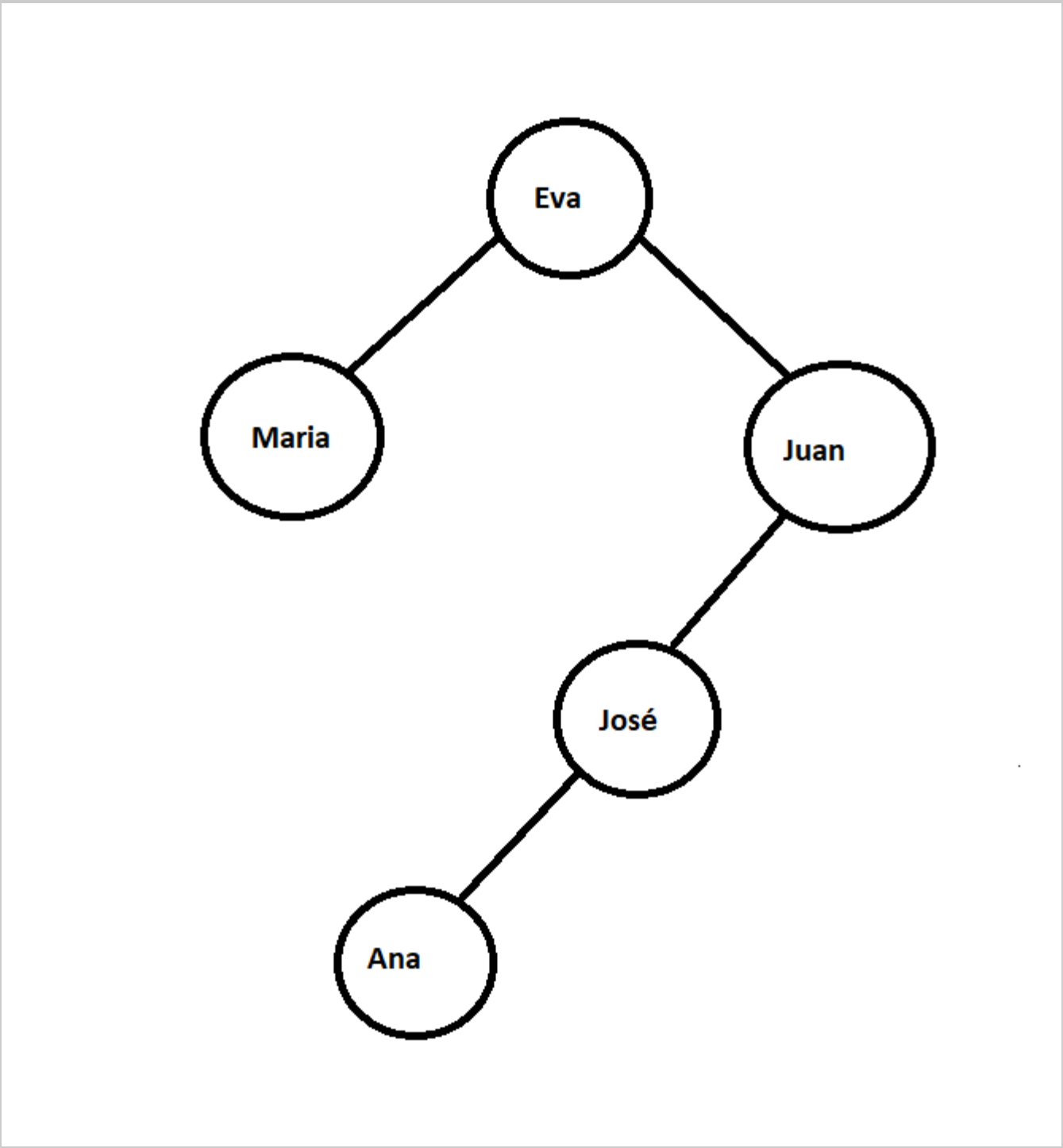
   (b) Graficá todos los árboles finitarios que al recorrerse en **pos-orden** resultan visitados en el siguiente orden: juan, josé, ana, eva; pero qu visitados en el siguiente orden: eva, juan, josé, ana.

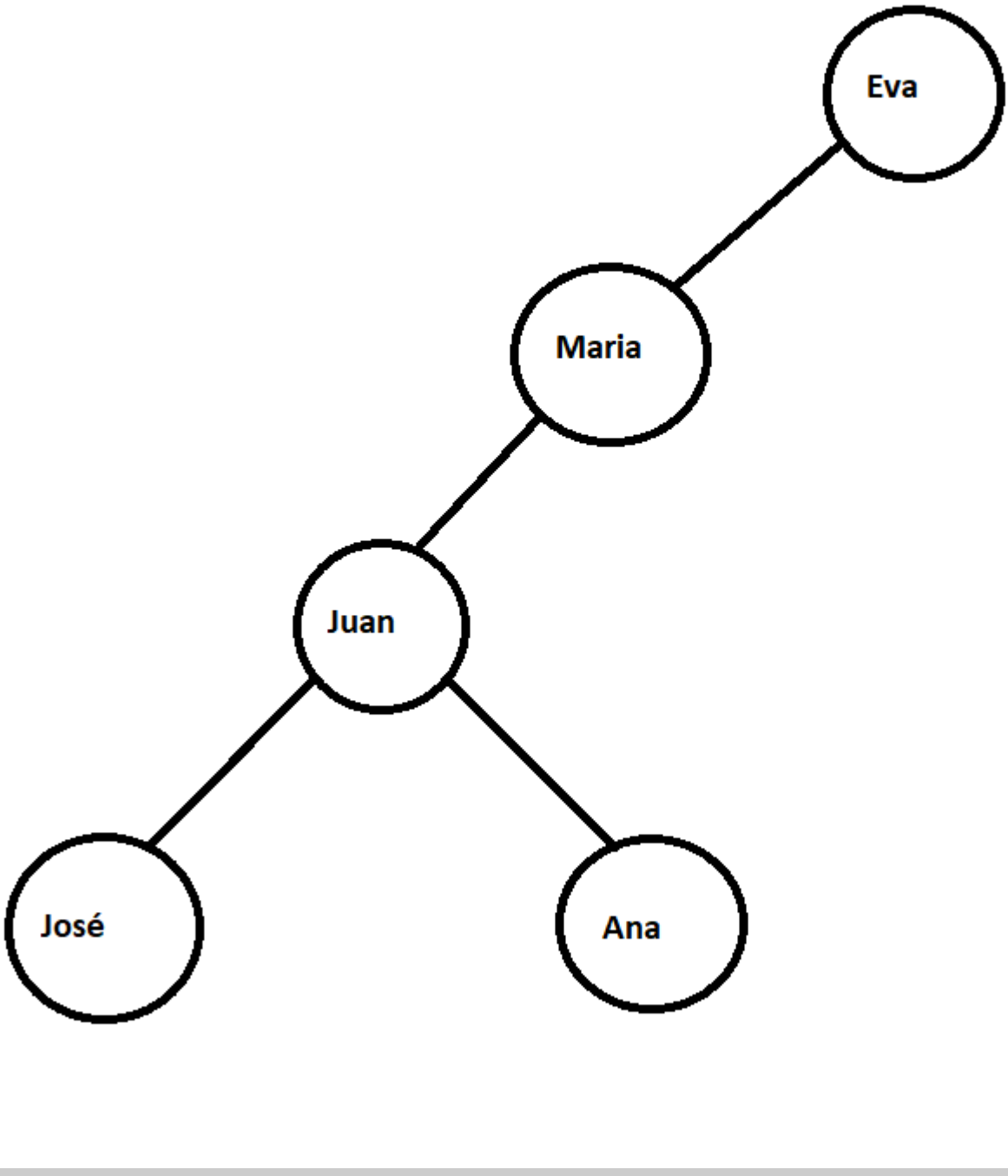
a)

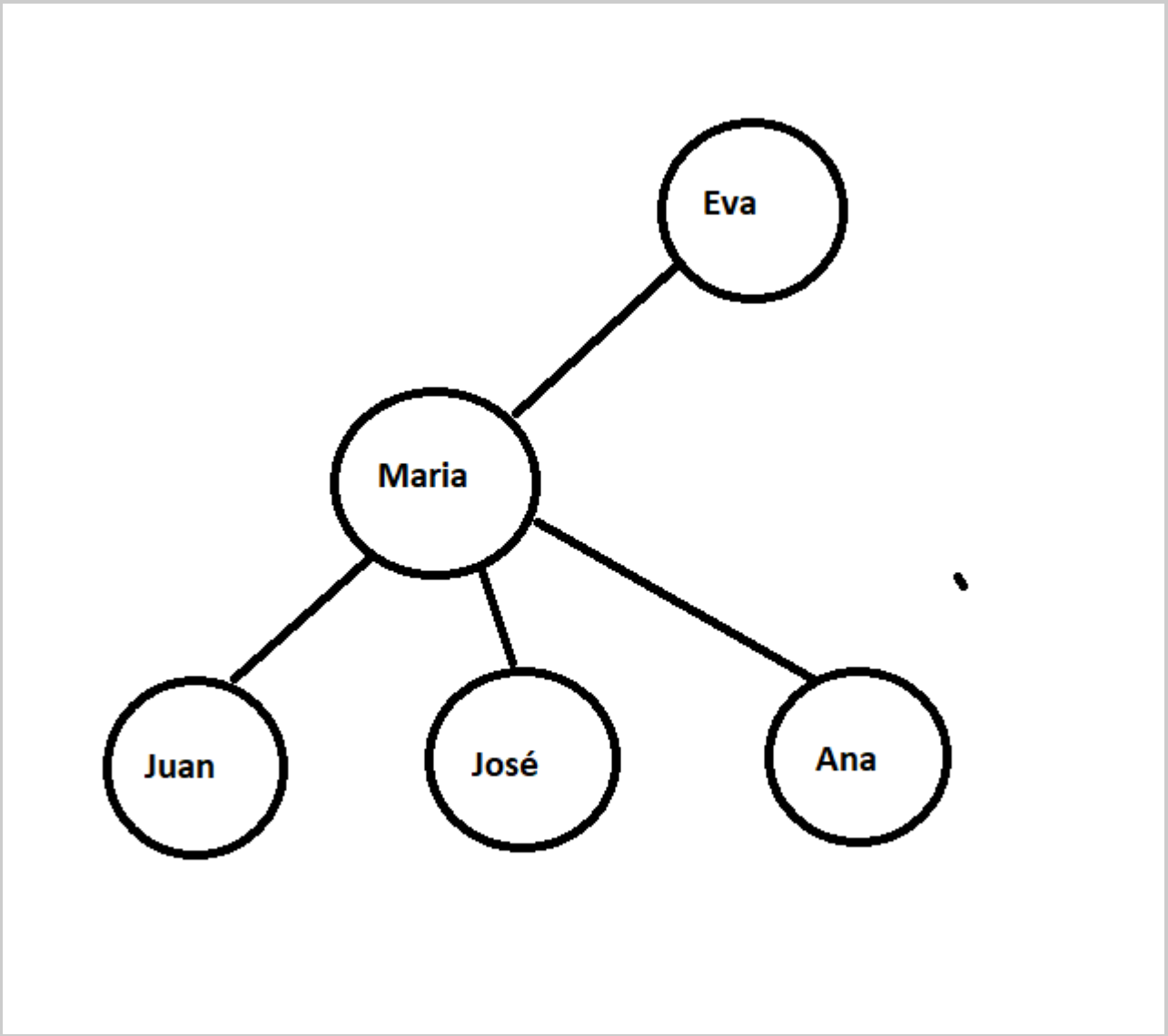



Eva
Maria
Juan
Jose
Ana

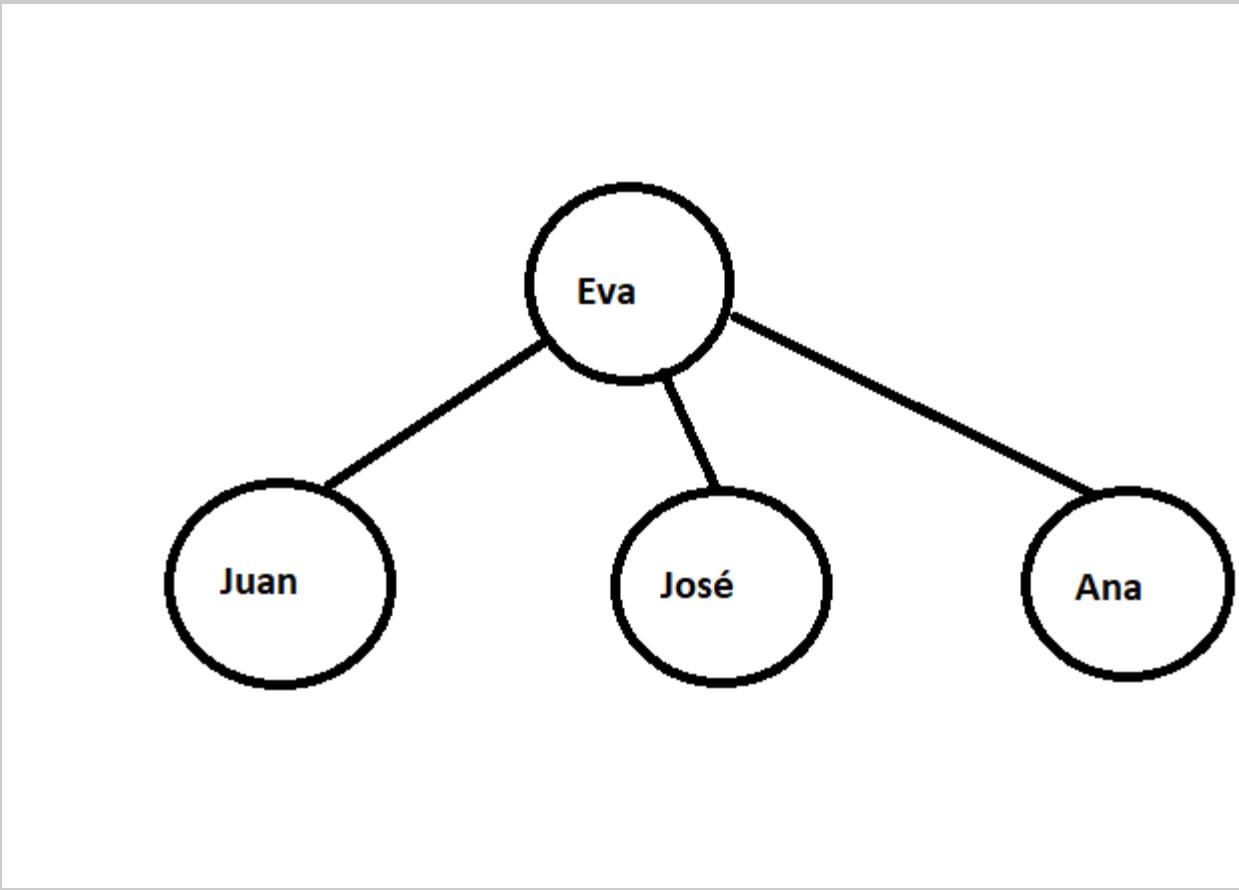

Eva
Maria
Juan
José
Ana


Eva
Maria
Juan
José
Ana


Eva
Maria
Juan
José
Ana



b)


Eva
Juan
José
Ana