

## DP23a - Funkcionális programozás, 3. gyakorlat

### Konvenciók, jelölések

A gyakorlatok anyaga szakaszokra van felosztva, minden szakaszban a bevezetés után néhány feladatot definiálunk, néha megoldott feladatokat is bemutatunk.

Halványzöld peremű, fekete háttérű cellában (a továbbiakban: specifikációs cella) van a szükséges „keretezéssel”, azaz a modul- és függvénydefinícióval együtt a megírandó függvény típusspecifikációja, valamint néhány tesztívás is. Ebbe a cellába nem lehet beleírni (csak szerkesztő módban), de a tartalmát ki lehet jelölni, lehet másolni.

Rózsaszín háttérű cellába írjuk az esetleges korlátozásokat: ne használja ezt, ne csinálja azt stb.

Halványzöld háttérű cellában jelennek meg a magyarázataink, illetve a javaslataink egyes feladatok megoldására. Az utóbbiak gyakran el vannak rejtve: a Súly felíratra kattintva jelennek meg.

Az egymást kölcsönösen kizáró minták használata...

Súly

Ezt és ezt javasoljuk a függvény megírásához.

A feladatot megoldó függvényt, kifejezést egy Elixir-cellába írja be: a felugró menüben a + Elixir felíratra kattintva hozzon létre egy új cellát, másolja be a specifikációs cella tartalmát, majd írja meg és értékelje ki a specifikált függvényt vagy a kért kifejezést.

### Elágazó rekurzió

Elágazó rekurzióról akkor beszélünk, ha egy rekurzív függvény *ugyanabban a klózban legalább kétszer* hívja meg saját magát.

Elágazóan rekurzív adatstruktúrák, pl. bináris fák bejárásának, feldolgozásának nyilvánvalóan az a természetes módja, ha elágazóan rekurzív algoritmusokat írunk rájuk, de vannak olyan számítási feladatok is, amelyekre sokkal könnyebb és érthetőbb elágazóan rekurzív algoritmust készíteni. Az utóbbiak között vannak olyanok, amelyeket egy kis fejtörés után érdemes sokkal hatékonyabban, azaz lineárisan rekurzív, akkumulátoros segédfüggvény segítségével megoldani, és vannak olyanok, amelyekre sok fejtörés után lehetne ugyan lineárisan rekurzív algoritmust írni, de nem érdemes.

### Számítások elágazó rekurzióval

Írjon függvényeket (lehetőleg többféle változatban) az alábbi számítási feladatok megoldására, először akkumulátor használata nélkül, majd, ha érdemes, (egy vagy több) akkumulátorral. Mindig törekedjen elegáns, tömör, érthető és hatékony függvények írására.

## Fibonacci-számok elágazó és lineáris rekurzióval Sűgő

A Fibonacci-számok matematikai definíciója:

$$F_0 = 0 \quad F_1 = 1 \quad F_n = F_{n-2} + F_{n-1}, \text{ ha } n > 1$$

Írjon elágazóan rekurzív függvényt `fib` néven az  $n$ -edik Fibonacci-szám kiszámítására a matematikai definíciót követve!

```
defmodule FibTree do
  @spec fib(n :: integer()) :: f :: integer()
  # Az n-edik Fibonacci-szám f
  def fib(n) do
    ...
  end
end
IO.puts(FibTree.fib(0) == 0)
IO.puts(FibTree.fib(1) == 1)
IO.puts(FibTree.fib(2) == 1)
IO.puts(FibTree.fib(5) == 5)
IO.puts(FibTree.fib(7) == 13)
IO.puts(FibTree.fib(43))
```

```
defmodule FibTree do
  @spec fib(n :: integer()) :: f :: integer()
  # Az n-edik Fibonacci-szám f
  def fib(0), do: 0
  def fib(1), do: 1
  def fib(n), do: fib(n - 2) + fib(n - 1)
end
```

```
IO.puts(FibTree.fib(0) == 0)
IO.puts(FibTree.fib(1) == 1)
IO.puts(FibTree.fib(2) == 1)
IO.puts(FibTree.fib(5) == 5)
IO.puts(FibTree.fib(7) == 13)
IO.puts(FibTree.fib(43))
```

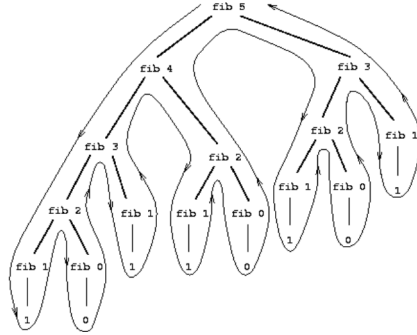
```
true
true
true
true
true
433494437
```

```
:ok
```

Az  $n$ -dik Fibonacci-szám meghatározása elágazó rekurzióval nagyon rossz hatékonyságú, mert a két elágazó ágat minden egyes rekurzív lépésben újra meg újra teljesen be kell járni, azaz az  $n$ -nél kisebb Fibonacci-számokat újra és

újra ki kell számolni, ráadásul a részeredményeket az egyre mélyülő veremben kell tárolni.

FibTree.fib(5) kiszámításának folyamata



fib 5-öt fib 4 és fib 3, fib 4-et fib 3 és fib 2 kiszámításával stb. kapjuk.

*Drámai hatékonyságnövelést* érünk el, ha lineárisan rekurzív megoldást írunk.

Írjon lineárisan rekurzív függvényt fib néven az  $n$ -edik Fibonacci-szám kiszámítására!

```
defmodule FibLin do
  @spec fib(n :: integer()) :: f :: integer()
  # Az n-edik Fibonacci-szám f
  def fib(n) do
    ...
  end
end
IO.puts(FibLin.fib(0) == 0)
IO.puts(FibLin.fib(1) == 1)
IO.puts(FibLin.fib(2) == 1)
IO.puts(FibLin.fib(5) == 5)
IO.puts(FibLin.fib(7) == 13)
IO.puts(FibLin.fib(43))
IO.puts(FibLin.fib(193))
```

Súgó

Az  $n$  meghatározásához szükséges két megelőző értéket,  $(n - 2)$ -t és  $(n - 1)$ -t adjuk át plusz paraméterként egy segédfüggvénynek: fib(n, n\_2, n\_1), amit az egyparaméteres verzióból hívunk meg, megfelelően inicializálva a két plusz paramétert.

```
@spec fib(n :: integer(), n_2 :: integer(), n_1 :: integer()) :: f :: integer()
defp fib(n, n_2, n_1) do
  ...
end
```

```

end

defmodule FibLin do
  @spec fib(n :: integer()) :: f :: integer()
  # Az n-edik Fibonacci-szám f
  def fib(n), do: fib(n, 0, 1)

  @spec fib(n :: integer(), n_2 :: integer(), n_1 :: integer()) :: f :: integer()
  # Az n-edik Fibonacci-szám ???
  defp fib(0, n_2, _) , do: n_2
  defp fib(n, n_2, n_1), do: fib(n - 1, n_1, n_2 + n_1)
end

```

```

IO.puts(FibLin.fib(0) == 0)
IO.puts(FibLin.fib(1) == 1)
IO.puts(FibLin.fib(2) == 1)
IO.puts(FibLin.fib(5) == 5)
IO.puts(FibLin.fib(7) == 13)
IO.puts(FibLin.fib(43))
# IO.puts(FibLin.fib(193))

```

```

true
true
true
true
true
433494437

```

:ok

### Pénzváltások száma (Szorgalmi feladat haladóknak)

A következő feladatot *elég könnyű elágazó rekurzióval* megoldani; *komoly fejtorést* okozna *iteratív programot* írni rá.

Határozzuk meg, hányféleképpen lehet felváltani egy adott összeget adott érmékkel, pl. 1000 forintot 200, 100, 50, 20, 10 és 5 forintos érmékkel?

Írjon elágazóan rekurzív függvényt `count_of_changes` néven az összes váltás számának meghatározására! Használjon segédfüggvényt!

```

defmodule CountOfChanges do
  @spec count_of_changes(amount :: integer()) :: count :: integer()
  # Az amount összeg összes lehetséges felváltásának száma count
  def count_of_changes(amount), do: count_of_changes(amount, 6)

  @spec count_of_changes(amount :: integer(), index :: integer()) :: count :: integer()
  # Az amount összeg összes lehetséges felváltásának száma a coin_id-nél nem nagyobb
  # indexű érmékkel count

```

```

defp count_of_changes(amount, coin_id) do
  ...
end
end
IO.puts(CountOfChanges.count_of_changes(5))
IO.puts(CountOfChanges.count_of_changes(10))
IO.puts(CountOfChanges.count_of_changes(100))
IO.puts(CountOfChanges.count_of_changes(1000))

```

A címleteket a coins függvénnyel kérdezheti le:

```

@spec coins(n :: integer()) :: c :: integer()
# Az n kulcsú címlet c
defp coins(6), do: 200
defp coins(5), do: 100
defp coins(4), do: 50
defp coins(3), do: 20
defp coins(2), do: 10
defp coins(1), do: 5

```

Súgó (rekurzió)

Tegyük föl, hogy a  $n$ -féle érménk van valamilyen, pl. nagyság szerint csökkenő sorrendben. Az a összeg lehetséges felváltásainak számát úgy kapjuk meg, hogy kiszámoljuk, hogy az a összeg hányféleképpen váltható fel a soron következő  $d$  értékű érmét kivéve a többi érmével (más szóval úgy, hogy a  $d$  érmét nem használjuk fel), és ehhez

hozzáadjuk, hogy az a -  $d$  összeg hányféleképpen váltható fel az összes érmével, a  $d$ -t is beleértve (más szóval úgy, hogy a  $d$  érmét is felhasználjuk legalább egyszer).

Súgó (esetszétválasztás)

A feladat megoldható rekurzióval, hiszen redukálható úgy, hogy minden lépésben vagy kisebb összeget kell felváltani, vagy kevesebb érmét kell felhasználni. A következő eseteket érdemes megkülönböztetni:

Ha  $a = 0$ , a felváltások száma 1. (Ui. ha az összeg 0, csak egyféleképpen, 0 db érmével lehet „felváltani”.)

Ha  $a < 0$ , a felváltások száma 0. (Ui. a soron következő érme nagyobb a még felváltandó összegnél.)

Ha  $n = 0$ , a felváltások száma 0. (Ui. elfogytak a címletek.)

Egyébként az előző bekezdésben leírt módon két rekurzív hívással számítjuk ki a még lehetséges váltások számát.

```

defmodule CountOfChanges do
  @spec count_of_changes(amount :: integer()) :: count :: integer()

```

```

# Az amount összeg összes lehetséges felváltásának száma count
def count_of_changes(amount), do: count_of_changes(amount, 6)

@spec count_of_changes(amount :: integer(), index :: integer()) :: count :: integer()
# Az amount összeg összes lehetséges felváltásának száma a coin_id-nél nem nagyobb
# indexű érmeikkel count
defp count_of_changes(amount, _) when amount < 0, do: 0
defp count_of_changes(0, _), do: 1
defp count_of_changes(_amount, 0), do: 0

defp count_of_changes(amount, coin_id) do
  count_of_changes(amount, coin_id - 1) +
  count_of_changes(amount - coins(coin_id), coin_id)
end

@spec coins(n :: integer()) :: c :: integer()
# Az n kulcsú címlet c
defp coins(6), do: 200
defp coins(5), do: 100
defp coins(4), do: 50
defp coins(3), do: 20
defp coins(2), do: 10
defp coins(1), do: 5
end

IO.puts(CountOfChanges.count_of_changes(5))
IO.puts(CountOfChanges.count_of_changes(10))
IO.puts(CountOfChanges.count_of_changes(100))
IO.puts(CountOfChanges.count_of_changes(1000))

```

```

1
2
50
98411

:ok

```

### Bináris fák feldolgozása elágazó rekurzióval

A következő feladatokhoz az alábbi tree és inttree adattípusokat definiáljuk:

```

@type tree() :: :leaf | {any(), tree(), tree()}
@type inttree() :: :leaf | {integer(), inttree(), inttree()}

```

Tehát egy tree() típusú Elixir-kifejezés

- vagy egy adatot (a továbbiakban *címkének* nevezzük) tartalmazó *csomópont*, amely további két tree() típusú értéket tartalmaz: az első a

- bal, a második a jobb részfa;
- vagy egy címke nélküli `:leaf` (azaz *levél*) atom.

Egy `inttree()` olyan `tree()`, amelynek minden címkéje egész szám. A példákban felhasznált változók és értékük:

```
t1 = {4,
      {3,:leaf,:leaf},
      {6,
       {5,:leaf,:leaf},
       {7,:leaf,:leaf}}
     }
t2 = {:a,
      {:b, {:v,:leaf,:leaf}, :leaf},
      {:c,
       :leaf,
       {:d,
        {w, {x,:leaf,:leaf}, :leaf},
        {f, {x,:leaf,:leaf}, {y,:leaf,:leaf}}}}
     }
```

A típusokat és a két változó helyett a két függvényt definiáljuk a `T` modulban, hogy más modulokból hivatkozhatunk rájuk:

```
defmodule T do
  @type tree() :: :leaf | {any(), tree(), tree()}
  @type inttree() :: :leaf | {integer(), inttree(), inttree()}
  def t1() do
    {4, {3, :leaf, :leaf}, {6, {5, :leaf, :leaf}, {7, :leaf, :leaf}}}
  end

  def t2() do
    {:a, {:b, {:v, :leaf, :leaf}, :leaf},
     {:c, :leaf,
      {:d, {w, {x, :leaf, :leaf}, :leaf}, {f, {x, :leaf, :leaf}, {y, :leaf, :leaf}}}}}
  end
end
```

```
IO.puts("t1 =" <> inspect(T.t1()))
IO.puts("t2 =" <> inspect(T.t2()))
```

```
defmodule T do
  @type tree() :: :leaf | {any(), tree(), tree()}
  @type inttree() :: :leaf | {integer(), inttree(), inttree()}
  def t1() do
```

```

    {4, {3, :leaf, :leaf}, {6, {5, :leaf, :leaf}, {7, :leaf, :leaf}}}
end

def t2() do
  {a, {b, {v, :leaf, :leaf}, :leaf},
   {c, :leaf,
    {d, {w, {x, :leaf, :leaf}, :leaf}, {f, {x, :leaf, :leaf}, {y, :leaf, :leaf}}}}}
end
end

IO.puts("t1 =" <> inspect(T.t1()))
IO.puts("t2 =" <> inspect(T.t2()))

t1 = {4, {3, :leaf, :leaf}, {6, {5, :leaf, :leaf}, {7, :leaf, :leaf}}}
t2 = {a, {b, {v, :leaf, :leaf}, :leaf}, {c, :leaf, {d, {w, {x, :leaf, :leaf}, :leaf}, {f, {x, :leaf, :leaf}, {y, :leaf, :leaf}}}}}

:ok

```

Bináris egészfa minden címkéjének megnövelése 1-gyel

```

defmodule IncTree do
  @spec fa_noveltje(f0::T.inttree()) :: f::T.inttree()
  # Az f fa minden címkéje eggyel nagyobb az f0 fa azonos helyen lévő címkéjénél
  def fa_noveltje(f0) do
    ...
  end
end

IncTree.fanoveltje(T.t1()) == {5, {4, :leaf, :leaf}, {7, {6, :leaf, :leaf}, {8, :leaf, :leaf}}}

defmodule IncTree do
  @spec fa_noveltje(f0 :: T.inttree()) :: f :: T.inttree()
  # Az f fa minden címkéje eggyel nagyobb az f0 fa azonos helyen lévő címkéjénél
  def fa_noveltje(:leaf), do: :leaf
  def fa_noveltje({c, bfa, jfa}), do: {c + 1, fa_noveltje(bfa), fa_noveltje(jfa)}
end

IncTree.fanoveltje(T.t1()) == {5, {4, :leaf, :leaf}, {7, {6, :leaf, :leaf}, {8, :leaf, :leaf}}}

true

```

Bináris fa tükörképe

```

defmodule Mirtree do
  @spec fa_tukorkepe(f0::T.tree()) :: f::T.tree()
  # f az f0 fa tükörképe
  def fa_tukorkepe(f0) do
    ...
  end
end

```



```

end
Mirtree.fa_tukorkepe(T.t1()) === {4,{6,{7,:level,:level},{5,:level,:level}},{3,:level,:level}}

defmodule Mirtree do
  @spec fa_tukorkepe(f0 :: T.tree()) :: f :: T.tree()
  # f az f0 fa tükörképe
  def fa_tukorkepe(:leaf), do: :leaf
  def fa_tukorkepe({c, bfa, jfa}), do: {c, fa_tukorkepe(jfa), fa_tukorkepe(bfa)}
end

Mirtree.fa_tukorkepe(T.t1()) === {4, {6, {7, :leaf, :leaf}, {5, :leaf, :leaf}}, {3, :leaf, :leaf}}

true

```

### Bináris fa inorder, preorder és postorder bejárása

```

defmodule TravTree do
  @spec inorder(f::T.tree()) :: ls::[any()]
  # ls az f fa elemeinek a fa inorder bejárásával létrejövő listája
  def inorder(f) do
    ...
  end
  @spec preorder(f::T.tree()) :: ls::[any()]
  # ls az f fa elemeinek a fa preorder bejárásával létrejövő listája
  def preorder(f) do
    ...
  end
  @spec postorder(f::T.tree()) :: ls::[any()]
  # ls az f fa elemeinek a fa postorder bejárásával létrejövő listája
  def postorder(f) do
    ...
  end
end

(TravTree.inorder(T.t1()) === [3,4,5,6,7]) |> IO.inspect()
(TravTree.preorder(T.t1()) === [4,3,6,5,7]) |> IO.inspect()
(TravTree.postorder(T.t1()) === [3,5,7,6,4]) |> IO.inspect()

defmodule TravTree do
  @spec inorder(f :: T.tree()) :: ls :: [any()]
  # ls az f fa elemeinek a fa inorder bejárásával létrejövő listája
  def inorder(:leaf), do: []
  def inorder({c, bfa, jfa}), do: inorder(bfa) ++ [c | inorder(jfa)]
  @spec preorder(f :: T.tree()) :: ls :: [any()]
  # ls az f fa elemeinek a fa preorder bejárásával létrejövő listája
  def preorder(:leaf), do: []
  def preorder({c, bfa, jfa}), do: [c | preorder(bfa) ++ preorder(jfa)]
  @spec postorder(f :: T.tree()) :: ls :: [any()]

```

```

# ls az ffa elemeinek a fa postorder bejárásával létrejövő listája
def postorder(:leaf), do: []
def postorder({c, bfa, jfa}), do: postorder(bfa) ++ postorder(jfa) ++ [c]
end

```

```

(TravTree.inorder(T.t1()) === [3, 4, 5, 6, 7]) |> IO.inspect()
(TravTree.preorder(T.t1()) === [4, 3, 6, 5, 7]) |> IO.inspect()
(TravTree.postorder(T.t1()) === [3, 5, 7, 6, 4]) |> IO.inspect()

```

```

true
true
true
true

```

### Címke előfordulása (rendezetlen) bináris fában

```

defmodule Contains do
  @spec tartalmaz(f::T.tree(), c::any()) :: b::boolean()
  # b igaz, ha c az ffa valamely címkéje
  def tartalmaz(f0) do
    ...
  end
end

(Contains.tartalmaz(T.t1(), :x) === false) |> IO.inspect()
(Contains.tartalmaz(T.t2(), :x) === true) |> IO.inspect()

```

```

defmodule Contains do
  @spec tartalmaz(f :: T.tree(), c :: any()) :: b :: boolean()
  # b igaz, ha c az ffa valamely címkéje
  def tartalmaz(:leaf, _) do: false
  def tartalmaz({c, _, _}, c) do: true
  def tartalmaz({_, bfa, jfa}, c) do: tartalmaz(bfa, c) or tartalmaz(jfa, c)
end

```

```

(Contains.tartalmaz(T.t1(), :x) === false) |> IO.inspect()
(Contains.tartalmaz(T.t2(), :x) === true) |> IO.inspect()

```

```

true
true
true

```

### Címke összes előfordulásának száma bináris fában

```

defmodule Occurs do
  @spec elofordul(f::T.tree, c::any()) :: n::integer()
  # A c címke az f fában n-szer fordul elő
  def elofordul(f0) do

```

```

...
end
end
(Occurs.elofordul(T.t1(), :x) === 0) |> IO.inspect()
(Occurs.elofordul(T.t2(), :x) === 2) |> IO.inspect()

defmodule Occurs do
  @spec elofordul(f :: T.tree(), c :: any()) :: n :: integer()
  # A c címke az f fában n-szer fordul elő
  def elofordul(:leaf, _), do: 0

  def elofordul({r, bfa, jfa}, c) do
    if r === c, do: 1, else: 0 + elofordul(bfa, c) + elofordul(jfa, c)
    end
  end
end

(Occurs.elofordul(T.t1(), :x) === 0) |> IO.inspect()
(Occurs.elofordul(T.t2(), :x) === 2) |> IO.inspect()

true
true
true

```

**Címkék felsorolása** írjon hatékony, lineáris időigényű algoritmust! Ehhez segédfüggvény használatát javasoljuk.

```

defmodule Labels do
  @spec cimkek(f::T.tree()) :: ls::[any()]
  # ls az f fá címkéinek listája inorder sorrendben
  def cimkek(f) do
    ...
  end

  @spec cimkek(f::T.tree(), zs::[any()]) :: ls::[any()]
  # ls az f fá címkéinek listája inorder sorrendben a zs elé fűzve
  defp cimkek(f, zs) do
    ...
  end
end

(Labels.cimkek(T.t1()) === [3,4,5,6,7]) |> IO.inspect()
(Labels.cimkek(T.t2()) === [:v,:b,:a,:c,:x,:w,:d,:x,:f,:y]) |> IO.inspect()

defmodule Labels do
  @spec cimkek(f :: T.tree()) :: ls :: [any()]
  # ls az f fá címkéinek listája inorder sorrendben
  def cimkek(fa), do: cimkek(fa, [])

```

```

@spec cimkek(f :: T.tree(), zs :: [any()]) :: ls :: [any()]
# ls az f fa címkéinek listája inorder sorrendben a zs elé fűzve
defp cimkek(:leaf, zs), do: zs
defp cimkek({r, bfa, jfa}, zs), do: cimkek(bfa, [r | cimkek(jfa, zs)])
end

(Labels.cimkek(T.t1()) === [3, 4, 5, 6, 7]) |> IO.inspect()
(Labels.cimkek(T.t2()) === [:v, :b, :a, :c, :x, :w, :d, :x, :f, :y]) |> IO.inspect()

true
true
true

```

**Bal és jobb szélső címkék visszaadása** Írjon egy-egy függvényt egy bináris fa bal, ill. jobb szélső címkéjének visszaadására!

```

defmodule MostLeftRight do
  @spec fa_balerteke(f::T.tree()) :: {:ok, c::any()} | :error
  # Egy nemüres f fa bal oldali szélső címkéje c (minden
  # felmenőjére is igaz, hogy bal oldali gyermek)
  # Ha nincs bal oldali szélső érték, az :error atomot adja vissza
  def fa_balerteke(f) do
    ...
  end

  @spec fa_jobberteke(f::T.tree()) :: {:ok, c::any()} | :error
  # Egy nemüres f fa jobb oldali szélső címkéje c (minden
  # felmenőjére is igaz, hogy jobb oldali gyermek)
  # Ha nincs jobb oldali szélső érték, az :error atomot adja vissza
  def fa_jobberteke(f) do
    ...
  end

  end

  (MostLeftRight.fa_balerteke(T.t1()) === {:ok, 3}) |> IO.inspect()
  (MostLeftRight.fa_balerteke(:leaf) === :error) |> IO.inspect()
  (MostLeftRight.fa_jobberteke(T.t1()) === {:ok, 7}) |> IO.inspect()
  (MostLeftRight.fa_jobberteke(:leaf) === :error) |> IO.inspect()

  defmodule MostLeftRight do
    @spec fa_balerteke(f :: T.tree()) :: {:ok, c :: any()} | :error
    # Egy nemüres f fa bal oldali szélső címkéje c (minden
    # felmenőjére is igaz, hogy bal oldali gyermek)
    # Ha nincs bal oldali szélső érték, az :error atomot adja vissza
    def fa_balerteke(:leaf), do: :error
    def fa_balerteke({c, :leaf, _}), do: {:ok, c}
    def fa_balerteke({_, bfa, _}), do: fa_balerteke(bfa)
  end

```

```

@spec fa_jobberteke(f :: T.tree()) :: {:ok, c :: any()} | :error
# Egy nemüres f fa jobb oldali szélső címkéje c (minden
# felmenőjére is igaz, hogy jobb oldali gyermek)
# Ha nincs jobb oldali szélső érték, az :error atomot adja vissza
def fa_jobberteke(:leaf), do: :error
def fa_jobberteke({c, _ :leaf}), do: {:ok, c}
def fa_jobberteke({_, _ :jfa}), do: fa_jobberteke(jfa)
end

```

```

(MostLeftRight.fa_balerteke(T.t1()) === {:ok, 3}) |> IO.inspect()
(MostLeftRight.fa_balerteke(:leaf) === :error) |> IO.inspect()
(MostLeftRight.fa_jobberteke(T.t1()) === {:ok, 7}) |> IO.inspect()
(MostLeftRight.fa_jobberteke(:leaf) === :error) |> IO.inspect()

```

```

true
true
true
true
true

```

**Bináris fa rendezettsége** Egy bináris fa rendezett, ha *inorder* bejárásakor a címkéi *szigorúan monoton növekednek*, azaz a csomópontjai kielégítik a keresőfa-tulajdonságot: minden egyes csomópont címkéje nagyobb a bal oldali gyermekei címkéjénél és kisebb a jobb oldali gyermekei címkéjénél. Oldja meg a feladatot

1. a `MostLeftRight.fa_balerteke/1` és a `MostLeftRight.fa_jobberteke/1` függvényekkel;
2. a `Labels.cimkek/1`, valamint az `Enum.sort/1` vagy más függvényekkel (pl. saját segédfüggénnyel).

```

defmodule Ordered do
@spec rendezett(f::T.tree()) :: b::boolean()
# b igaz, ha az f fa rendezett
def rendezett(f) do
...
end
end
(Ordered.rendezett(T.t1()) === true) |> IO.inspect()
(Ordered.rendezett(T.t2()) === false) |> IO.inspect()

```

```

defmodule Ordered do
@spec rendezett(f :: T.tree()) :: b :: boolean()
# b igaz, ha az f fa rendezett
def rendezett(:leaf), do: true

def rendezett({c, bfa, jfa}) do

```

```

case MostLeftRight.fa_jobberteke(bfa) do
  :error -> true
  {:ok, j} -> j < c
end and
rendezett(bfa) and
case MostLeftRight.fa_balerteke(jfa) do
  :error -> true
  {:ok, b} -> c < b
end and
rendezett(jfa)
end
end

```

```

(Ordered.rendezett(T.t1()) === true) |> IO.inspect()
(Ordered.rendezett(T.t2()) === false) |> IO.inspect()

```

true

true

true

```

defmodule Ordered2 do
  @spec rendezett(f :: T.tree()) :: b :: boolean()
  # b igaz, ha az f fa rendezett
  def rendezett(f) do
    Labels.cimkek(f) |> sorted?()
  end

  defp sorted?([x1 | [x2 | _xs] = xxs]) when x1 < x2, do: sorted?(xxs)
  defp sorted?([_x1, _x2 | _xs]), do: false
  defp sorted?(_), do: true
end

```

```

(Ordered2.rendezett(T.t1()) === true) |> IO.inspect()
(Ordered2.rendezett(T.t2()) === false) |> IO.inspect()

```

true

true

true

**Bináris fa összes címkéjének útvonala** Egy adott csomópont útvonalának nevezzük azon csomópontok címkéinek listáját, amelyeken át a fa gyökerétől az adott csomópontig el lehet jutni.

Javasoljuk a `Route.utak/2` segédfüggvény definiálását és használatát.

```

defmodule Route do
  @type route() :: [any()]

```

```

@spec utak(f::T.tree()) :: cimkezett_utak::[{c::any(), cu::route()}]
# A cimkezett_utak lista az f fa minden csomópontjához egy {c,cu} párt
# társít, ahol c az adott csomópont címkéje, cu pedig az adott
# csomóponthoz vezető útvonal
def utak(f) do
...
end

@spec utak(f::T.tree(), eddigi_ut::route()) :: cimkezett_utak::[{c::any(), cu::route()}]
# A cimkezett_utak lista az f fa minden csomópontjához egy {c,cu} párt
# társít, ahol c az adott csomópont címkéje, cu pedig az adott
# csomóponthoz vezető útvonal az eddigi_ut eddigi útvonal elé fűzve
def utak(f) do
...
end
end

(Route.utak(T.t1()) === [{4,[]},{3,[4]},{6,[4]},{5,[4,6]},{7,[4,6]}]) |> IO.inspect()
(Route.utak(T.t2()) === [{:a,[]},
{:b,[:a]},
{:v,[:a,:b]},
{:c,[:a]},
{:d,[:a,:c]},
{:w,[:a,:c,:d]},
{:x,[:a,:c,:d,:w]},
{:f,[:a,:c,:d]},
{:x,[:a,:c,:d,:f]},
{:y,[:a,:c,:d,:f]}
]) |> IO.inspect()

defmodule Route do
@type route() :: [any()]
@spec utak(f :: T.tree()) :: cimkezett_utak :: [{c :: any(), cu :: route()}]
# A cimkezett_utak lista az f fa minden csomópontjához egy {c,cu} párt
# társít, ahol c az adott csomópont címkéje, cu pedig az adott
# csomóponthoz vezető útvonal
def utak(fa), do: utak(fa, [])

@spec utak(f :: T.tree(), eddigi_ut :: route()) ::
cimkezett_utak :: [{c :: any(), cu :: route()}]
# A cimkezett_utak lista az f fa minden csomópontjához egy {c,cu} párt
# társít, ahol c az adott csomópont címkéje, cu pedig az adott
# csomóponthoz vezető útvonal az eddigi_ut eddigi útvonal elé fűzve
def utak(:leaf, _), do: []

def utak({c, bfa, jfa}, eddigi) do
# eddigi1 = eddigi ++ [c] # Költséges művelet: O(n2)!

```

```

# Kévszám:  $O(2n)$ 
eddig1 = Enum.reverse([c | Enum.reverse(eddigi)])
[ {c, eddigi} | utak(bfa, eddig1) ] ++ utak(jfa, eddig1)
end
end

(Route.utak(T.t1()) === [{4, []}, {3, [4]}, {6, [4]}, {5, [4, 6]}, {7, [4, 6]}] |> IO.inspect()

(Route.utak(T.t2()) === [
  {a, []},
  {b, [a]},
  {v, [a, b]},
  {c, [a]},
  {d, [a, c]},
  {w, [a, c, d]},
  {x, [a, c, d, w]},
  {f, [a, c, d]},
  {x, [a, c, d, f]},
  {y, [a, c, d, f]}
])
|> IO.inspect()

true
true
true

```

**Adott címke összes előfordulása bináris fában útvonallal** Oldja meg a feladatot

1. for-jelöléssel és a Route.utak/1 függvény felhasználásával;
2. takarékoskodva a memóriával, azaz az összes útvonal helyett csak a keresett útvonalakat tárolja el. (A megoldásban a Route.utak/2 függvényhez hasonló segédfüggvényt használjon, de a gyökér címkéjét csak egy bizonyos feltétel teljesülésekor tárolja el.)

```

defmodule Occurrences do
  @spec cutak(f::T.tree(), c::any()) :: utak::[{c::any(), cu::Route.route()}]
  # utak azon csomópontok útvonalainak listája f-ben, amelyek címkéje c
  def cutak(f) do
    ...
  end
end

(Occurrences.cutak(T.t1(), :x) === []) |> IO.inspect()
(Occurrences.cutak(T.t2(), :x) === [{:x,[a,c,d,w]},{:x,[a,c,d,f]}]) |> IO.inspect()

defmodule Occurrences do
  @spec cutak(f :: T.tree(), c :: any()) :: utak :: [{c :: any(), cu :: Route.route()}]

```



```

# utak azon csomópontok útvonalainak listája f-ben, amelyek címkéje c
def cutak(fa, c), do: for({c0, _} = cu <- Route.utak(fa), c0 === c, do: cu)
end

(Occurrences.cutak(T.t1(), :x) === []) |> IO.inspect()

(Occurrences.cutak(T.t2(), :x) === [{:x, [:a, :c, :d, :w]}, {:x, [:a, :c, :d, :f]}])
|> IO.inspect()

true
true
true

defmodule Occurrences2 do
  @spec cutak(f :: T.tree(), c :: any()) :: utak :: [{c :: any(), cu :: Route.route()}]
  # utak azon csomópontok útvonalainak listája f-ben, amelyek címkéje c
  def cutak(fa, c), do: cutak(fa, c, [])

  @spec cutak(f :: T.tree(), c :: any(), eddigi :: Route.route()) ::
    cimkezett_utak :: [{c :: any(), u :: Route.route()}]
  # A cimkezett_utak lista az f fa minden c címkéjű csomópontjához egy {c,u}
  # párt társít, ahol c az adott csomópont címkéje, u pedig az adott
  # csomóponthoz vezető útvonal az eddigi eddigi útvonal elé fűzve
  def cutak(:leaf, _, _), do: []

  def cutak({r, bfa, jfa}, c, eddigi) do
    eddigi1 = eddigi ++ [r]
    cutak = cutak(bfa, c, eddigi1) ++ cutak(jfa, c, eddigi1)
    if r === c, do: [{c, eddigi} | cutak], else: cutak
  end
end

(Occurrences2.cutak(T.t1(), :x) === []) |> IO.inspect()

(Occurrences2.cutak(T.t2(), :x) === [{:x, [:a, :c, :d, :w]}, {:x, [:a, :c, :d, :f]}])
|> IO.inspect()

true
true
true

```

## Binárisok kezelése

### Változó hosszúságú bájt sorozat dekódolása egész számmá

A tetszőleges hosszúságú nemnegatív egész számok bájtok formájában történő leírásának egyik módja, hogy minden bájt első bitjével jelezzük, folytatódik-e a

bájtsorozat, a maradék 7 bitben pedig a tényleges szám egy 7 bites szegmensét tároljuk. A bájtnak kezdő 0-s bitje jelzi, hogy vége van a sorozatnak, az ezt követő 7 bit pedig a tárolt szám utolsó 7 bitje; a kezdő 1-es bit pedig azt jelzi, hogy a következő 7 bit a szám következő legmagasabb helyiértékű 7 bitje, és a szám alacsonyabb helyiértékű bitjei a következő bájtnak vannak. Néhány példa bites reprezentációja:

00000001 -> 0b0000001 = 1 (a kezdő 0 bit jelzi, hogy a szám 1 bájtnak hosszú, értéke 0000001, vagyis 1)

00000010 -> 0b0000010 = 2 (szintén 1 bájtnak szám, értéke 0000010, vagyis 2)

1000000100000001 -> 0b000000100000001 = 129 (az első 7 bit 0000001, a kezdő 1-es bit jelzi,

hogyan folytatódik, a második 7 bit 0000001)

A bit shift az Elixirben nem beépített operátor, az `import Bitwise` paranccsal lehet aktiválni a megfelelő `<<<` és `>>>` operátorokat. (A `Bitwise` modul betöltése eltart egy ideig az első futtatáskor)

Érdekes segédfüggvényt használni a részeredmények tárolására külön paraméterben. A bit shift operátorok precedenciája alacsony, figyeljen a zárójelre!

A binárisok szintaxisáról és használatáról bővebb leírás érhető el a <https://elixir-lang.org/getting-started/binaries-strings-and-char-lists.html> oldalon.

```
defmodule Varlen do
```

```
  import Bitwise
```

```
  @spec decode(bin::binary()) :: int::integer()
```

```
  # bin decimális értéke int
```

```
  def decode(bin) do
```

```
    ...
```

```
  end
```

```
end
```

```
IO.puts(Varlen.decode(<<<0x01>>>) === 1)
```

```
IO.puts(Varlen.decode(<<<0x7F>>>) === 127)
```

```
IO.puts(Varlen.decode(<<<0xFF, 0x7F>>>) === 16383)
```

```
IO.puts(Varlen.decode(<<<0x81, 0x80, 0x01>>>) === 16385)
```

```
defmodule Varlen do
```

```
  import Bitwise
```

```
  @spec decode(bin :: binary()) :: int :: integer()
```

```
  # bin decimális értéke int
```

```
  def decode(bin) do
```

```
    decode(bin, 0)
```

```
  end
```

```
  defp decode(<<<0::1, bytes::7>>>, result) do
```

```
    (result <<< 7) + bytes
```

```
  end
```

```

defp decode(<<1::1, bytes::7, rest::binary>>, result) do
  decode(rest, (result <<< 7) + bytes)
end
end

IO.puts(Varlen.decode(<<0x01>>) === 1)
IO.puts(Varlen.decode(<<0x7F>>) === 127)
IO.puts(Varlen.decode(<<0xFF, 0x7F>>) === 16383)
IO.puts(Varlen.decode(<<0x81, 0x80, 0x01>>) === 16385)

true
true
true
true

:ok

```

## Közelítő számítások lineáris rekurzióval

Írjon lineárisan rekurzív függvényeket az alábbi számítási feladatok megoldására. Írjon többféle függvényváltozatot, először direkt rekurzióval, majd esetleg könyvtári függvények használatával. Mindig törekedjen elegáns, tömör, érthető és hatékony függvények írására.

### A Ludolph-féle szám ( $\pi$ ) közelítése a Leibniz-féle sorral

A Leibniz-féle sor:  $\pi/4 = 1 - 1/3 + 1/5 - 1/7 + \dots$

```

defmodule Pi do
  @spec pi(i :: integer()) :: v :: float()
  # A  $\pi$  i-edik közelítő értéke v
  def pi(i) do
    ...
  end
end

IO.puts(Pi.pi(9))
IO.puts(Pi.pi(10_000_000))
IO.puts(:math.pi())
IO.puts(abs(Pi.pi(10_000_000) - :math.pi()) < 1.0e-6)

```

Haladjon a 0. közelítéstől az i. közelítésig!

Súgó

Írjon két, akkumulátort használó, kölcsönösen rekurzív segédfüggvényt, az egyiket az  $1/2k + 1$  tört hozzáadására ( $k \geq 0$ ), a másikat a kivonására.

```

defmodule Pi1 do
  @spec pi(i :: integer()) :: v :: float()

```

```

# A  $\pi$  i-edik közelítő értéke v
def pi(i), do: pi_p(i, 0, 0)

@spec pi_p(i :: integer(), k :: integer(), a :: float()) :: v :: float()
# i az elvárt, k az aktuális lépésszám
# a az akkumulátor, melyhez a következő törtet hozzá kell adni
# v a (rész)eredmény
defp pi_p(i, k, a) when k <= i, do: pi_m(i, k + 1, a + 1 / (2 * k + 1))
defp pi_p(i, k, a) when k > i, do: 4 * a

@spec pi_m(i :: integer(), k :: integer(), a :: float()) :: v :: float()
# i az elvárt, k az aktuális lépésszám
# a az akkumulátor, melyből a következő törtet le kell vonni
# v a (rész)eredmény
defp pi_m(i, k, a) when k <= i, do: pi_p(i, k + 1, a - 1 / (2 * k + 1))
defp pi_m(i, k, a) when k > i, do: 4 * a
end

```

```

IO.puts(Pi1.pi(9))
IO.puts(Pi1.pi(10_000_000))
IO.puts(:math.pi())
IO.puts(abs(Pi1.pi(10_000_000) - :math.pi()) < 1.0e-6)

```

```

3.0418396189294032
3.1415927535897814
3.141592653589793

```

true

:ok

Sűgő

A két kölcsönösen rekurzív segédfüggvény helyett egy is elég, ha az előjelváltást egy további paraméter vezéreli.

```

defmodule Pi2 do
  @spec pi(i :: integer()) :: v :: float()
  # A  $\pi$  i-edik közelítő értéke v
  def pi(i), do: pi_s(i, 0, :p, 0)

  @spec pi_s(i :: integer(), k :: integer(), c :: atom(), a :: float()) :: v :: float()
  # i az elvárt, k az aktuális lépésszám
  # c a hozzáadást vagy kivonást jelző érték
  # a a közelítő értékek akkumulátora
  # v a (rész)eredmény
  defp pi_s(i, k, :p, a) when k <= i, do: pi_s(i, k + 1, :m, a + 1 / (2 * k + 1))
  defp pi_s(i, k, :m, a) when k <= i, do: pi_s(i, k + 1, :p, a - 1 / (2 * k + 1))
  defp pi_s(i, k, _, a) when k > i, do: 4 * a
end

```

**end**

```
IO.puts(Pi2.pi(9))
IO.puts(Pi2.pi(10_000_000))
IO.puts(:math.pi())
IO.puts(abs(Pi2.pi(10_000_000) - :math.pi()) < 1.0e-6)
```

```
3.0418396189294032
```

```
3.1415927535897814
```

```
3.141592653589793
```

```
true
```

```
:ok
```

Mindkét megoldás gyengéje, hogy a változatlan  $i$ -t, a közelítő lépések elvart számát minden lépésben át kell adnunk a segédfüggvény(ek)nek, hogy  $k$ -t, az aktuális lépésszámot legyen mivel összehasonlíthatunk. Ha nem 0-tól  $i$ -ig, hanem  $i$ -től 0-ig haladunk, akkor  $i$ -t paraméterként nem kell minden lépésben átadnunk a segédfüggvény(ek)nek, az  $\text{ör}$  vagy a feléltelvizsgálat helyett pedig mintaillesztést használhatunk. Azt, hogy összeadást vagy kivonást kell-e végeznünk,  $k$  páros vagy páratlan volta szabja meg.

**defmodule Pi3 do**

```
@spec pi(i :: integer()) :: v :: float()
```

```
# A  $\pi$   $i$ -edik közelítő értéke  $v$ 
```

```
def pi(i), do: pi_s(i, 1)
```

```
@spec pi_s(k :: integer(), a :: float()) :: v :: float()
```

```
# k az aktuális lépésszám
```

```
# a a közelítő értékek akkumulátora
```

```
# v a (rész)eredmény
```

```
defp pi_s(0, a), do: 4 * a
```

```
defp pi_s(k, a) do
```

```
  # 1 frac = 1 / (2*k+1)
```

```
  # 1 pi_s(k-1, if(rem(k, 2) == 0, do: a + frac, else: a - frac))
```

```
  # 2 m = if(rem(k, 2) == 0, do: &Kernel.+ /2, else: &:erlang.- /2)
```

```
  # 2 pi_s(k-1, m.(a, 1 / (2*k+1)))
```

```
  pi_s(k - 1, if(rem(k, 2) == 0, do: &Kernel.+ /2, else: &:erlang.- /2).(a, 1 / (2 * k + 1)))
```

```
end
```

**end**

```
IO.puts(Pi3.pi(9))
IO.puts(Pi3.pi(2))
IO.puts(:math.pi())
IO.puts(abs(Pi3.pi(10_000_000) - :math.pi()) < 1.0e-6)
```

```
3.0418396189294024
```

```
3.466666666666667
3.141592653589793
true
```

```
:ok
```

Érdekes dolgokról olvashat itt: <https://matekarcok.hu/pi-a-ludolph-fele-szam/>

Szorgalmi feladat: írja meg a közelítést az Euler-féle sorral is.

```
defmodule Euler do
```

```
  @spec pi(i :: integer()) :: pi :: float()
```

```
  # A  $\pi$  i-edik közelítő értéke pi
```

```
  def pi(i) do
```

```
    ...
```

```
  end
```

```
end
```

```
IO.puts(abs(Euler.pi(10_000_000)))
```

```
IO.puts(:math.pi())
```

```
IO.puts(abs(Euler.pi(10_000_000) - :math.pi()) < 1.0e-6)
```

```
defmodule Euler do
```

```
  @spec pi(i :: integer()) :: pi :: float()
```

```
  # A  $\pi$  i-edik közelítő értéke pi
```

```
end
```

```
IO.puts(abs(Euler.pi(10_000_000)))
```

```
IO.puts(:math.pi())
```

```
IO.puts(abs(Euler.pi(10_000_000) - :math.pi()) < 1.0e-6)
```

```
error: spec for undefined function pi/1
```

```
dp23a-fpgy.livemd#cell:wkq2ggerjddtr46ilqzqs6j5mm4ccmhf:2
```