R1. Sorteio ponderado

Enunciado
A partir de uma lista de pares peso-valor, sortear um valor com probabilidade proporcional ao peso.

Exemplo de uso
```
(rand-weighted [[3 :a] [2 :b] [6 :c]])
;=> :b
```

No exemplo, o valor `:a` tem 3/(3+2+6) * 100% de chances de ser sorteado, `:b` tem 2/(3+2+6) * 100% e `:c`, 6/(3+2+6) * 100%.

Projeto do fluxo de dados
1) Separar os pesos
2) Acumular os pesos (somar todos os pesos até cada valor: 3, 3 + 2, 3 + 2 + 6)
    2.1) Para isso, gerar a listas de pesos parciais [[3], [3,2], [3,2,6]]
3) Sortear um valor entre 0 e o maior valor acumulado
4) Encontrar a posição do valor sorteado no vetor de pesos acumulados
5) Usar essa posição para pegar o valor correspondente

Sessão de REPL
```
[[3 :a] [2 :b] [6 :c]]
;=> [[3 :a] [2 :b] [6 :c]]
(map first *1)  ;passo 1
;=> (3 2 6)
(range (count *1)) ;começa o passo 2.1
;=> (0 1 2)
(map inc *1)
;=> (1 2 3)
(map #(take % *3) *1) ;termina o passo 2.1
;=> ((3) (3 2) (3 2 6))
(map #(reduce + %) *1) ;termina o passo 2
;=> (3 5 11)
(rand-int (last *1)) ;passo 3
;=> 10
(take-while #(<= % *1) *2) ;começa ompasso 4
;=> (3 5)
(count *1) ;termina  o passo 4
;=> 2
(nth [[3 :a] [2 :b] [6 :c]] *1) ;começa o passo 5
;=> [6 :c]
(second *1) ; termina o passo 5
;=> :c
```

Função
```
(defn rand-weighted
  [w]
```

Testes
É fácil executar a função muitas vezes e contar os resultados para conferir se as proporções dos pesos foi atendida. A função `repeatedly` exige uma função sem parâmetros, por isso cria-se `para-teste`, na qual embute-se o parâmetro. O que se segue é bem básico: sorteiam-se 10000 valores, que são ordenados, separados e contados. As proporções esperadas são 3/11, 2/11 e 6/11, ou seja 27%, 18% e 54%.
```
(defn para-teste [] (rand-weighted [[3 :a] [2 :b] [6 :c]]))
(map count (partition-by identity (sort (repeatedly 10000 para-teste))))
```

R2. Gera grafo aleatoriamente (etapa 1 : sorteia camadas)

Enunciado
Todos os exercícios que se seguem, referem-se à geração aleatória de um grafo de 3 camadas. São sorteados:

- a quantidade de nós em cada camada, e
- a quantidade de arestas que saem de cada nó

Nessa etapa, é gerada uma lista como essa: ((1 2) (2 2 2) (1 1 1)) que significa: a primeira camada do grafo tem dois nós, com uma e duas arestas de saída respectivamente, a segunda camada tem três nós, todos com duas arestas e saídas, e a terceira camada tem 3 nós, todos com uma aresta de saída.

Uma condição importante que a primeira camada tenha arestas de saída suficientes para todos os nós da segunda, e, da mesma forma, a segunda em relação à terceira.

Exemplo de uso
```
(rand-graph-layout)
;=> ((1 2) (2 2 2) (1 1 1))
```

Projeto do fluxo de dados
Precondições: a) Função para sortear número de nós, b) Função para sortear arestas, c) Função para testar condição de camadas, d) Função para testar todo o grafo
1) Criar as camadas do grafo com a) e b)
2) Repetir 1 enquanto d) não for atendida.

Sessão de REPL
```
(defn aux-nodes [] (rand-weighted [[2 2] [3 3] [2 4]])) ;a)camada c/ 2 3 ou 4 nós
;=> #'user/aux-nodes
(defn aux-edges [] (rand-weighted [[7 1] [3 2]])) ;b) nó com 1 ou 2 arestas
;=> #'user/aux-edges
(defn not-enough-edges [out in] (< (reduce + out) (count in)))
;=> #'user/not-enough-edges
(defn bad-levels [gr] (or (not-enough-edges (first gr) (second gr))
                          (not-enough-edges (second gr) (nth gr 2))))
;=> #'user/bad-levels
(defn rand-level  [] (repeatedly (aux-nodes) aux-edges)) ; prepara passo 1
;=> #'user/rand-level
(defn rand-3-levels [] (repeatedly 3 rand-level)) ; prepara passo 1
;=> #'user/rand-3-levels
(first (drop-while bad-leves (repeatedly rand-3-levels))) ; passos 1 e 2
```

Função
(Este exercício é trivial. Para complicar, considere criar as funções auxiliares como internas da função.)

```
(defn rand-graph-layout []
  (let [aux-nodes (fn [] (rand-weighted [[2 2] [3 3] [2 4]]))
```

Testes

R3. Gera grafo aleatoriamente (etapa 2 : cria nós)

Enunciado
A partir de um leiaute de camadas (aqui chamado de *level-map*), cria rótulos para os nós e os particiona por camadas. Os nós são rotulados com números inteiros; zero é o primeiro.

Exemplo de uso
```
(graph-nodes '((1 2) (2 2 2) (1 1 1)))
;=> ((0 1) (2 3 4) (5 6 7))
```

Projeto de fluxo de dados
Cada número em *level-map* corresponde às saídas de um nó.
```
((1 2) (2 2 2) (1 1 1))    = level-map
```
O primeiro passo é obter todos esses valores em uma lista.
```
(1 2 2 2 2 1 1 1)      = outs
```
Com ela é possível contar o total de nós. Geram-se os nós com *range*.
```
(0 1 2 3 4 5 6 7)        = node-labels
```
Reagrupam-se os nós usando *partition-by*. Antes disso é preciso gerar uma lista casada com os rótulos dos grupos (camadas).
```
(0 0 1 1 1 2 2 2)         = node-groups
((0 1) (2 3 4) (5 6 7))
```

Sessão de REPL
```
[[1 2] [2 2 2] [1 1 1]]
;=> [[1 2] [2 2 2] [1 1 1]]
(def level-map *1)   ; nesta sessão, valores são armazenados em variáveis
;=> #'user/level-map
(flatten level-map)
;=> (1 2 2 2 2 1 1 1)
(def outs *1)        ; crie as variáveis como locais em let
;=> #'user/outs
(range (count outs))
;=> (0 1 2 3 4 5 6 7)
(def node-labels *1)
;=> #'user/node-labels
(map-indexed (fn [i x] (map (fn [y] i) x)) level-map)
;=> ((0 0) (1 1 1) (2 2 2))
(flatten *1)
;=> (0 0 1 1 1 2 2 2)
(def node-groups *1)
;=> #'user/node-groups
(map #(vector % %2) node-groups node-labels)
;=> ([0 0] [0 1] [1 2] [1 3] [1 4] [2 5] [2 6] [2 7])
(partition-by first *1)
;=> (([0 0] [0 1]) ([1 2] [1 3] [1 4]) ([2 5] [2 6] [2 7]))
(map #(map second %) *1)
;=> ((0 1) (2 3 4) (5 6 7))
```

Função
(Use a *form* `let` até a definição de `node-groups`. A partir daí, use uma *threading macro*.)

```
(defn graph-nodes
```