* PROBLEMA DA MOCHILA 0-1 * (Cormen, §16.2, exercícios)

0. DEFINIÇÃO:

   a) ENTRADA: vetores v[1..n] e p[1..n], o primeiro de reais positivos e o
               segundo de naturais positivos, e um natural P.

   b) SAÍDA: na versão completa, um vetor e[1..n] de 0's e 1's que maximize
             SOMATÓRIO_{i=1}^n v[i]*e[i] e que respeite a restrição
             SOMATÓRIO_{i=1}^n p[i]*e[i] <= P.
             Na versão simplificada, não é necessário retornar o vetor "e", mas
             sim apenas o valor de SOMATÓRIO_{i=1}^n v[i]*e[i].

1. EXERCÍCIO: uma heurística "gulosa" para o problema em questão é a seguinte:

   1. ENQUANTO houver espaço livre na mochila E itens ainda não considerados
   2. | Seja "i" um item, dentre aqueles ainda não considerados, com a melhor
      | relação custo-benefício (isto é, a maior razão v[i]/p[i]).
   3. | Se ainda houver espaço para "i" na mochila, coloque-o nela.

Mostre que a heurística acima nem sempre leva a uma solução ótima.

2. Um algoritmo de divisão-e-conquista (ideia: testar todas as possibilidades):

```
============================================================
Algoritmo: moch_dq
Entrada: vetores v[1..n] e p[1..n], e P, como descrito acima
Saída: um número real
------------------------------------------------------------
01. SE P = 0
02. | RETORNE 0.
03. SE n = 1
04. | SE p[1] <= P
05. | | RETORNE v[1].
06. | SENÃO
07. | | RETORNE 0.
08. VN := moch_dq( v[1..n-1], p[1..n-1], P )
09. SE p[n] <= P
10. | VS := moch_dq( v[1..n-1], p[1..n-1], P - p[n] ) + v[n]
11. | RETORNE máx { VN , VS }.
12. RETORNE VN.
============================================================
```

3. Um algoritmo de programação dinâmica (ideia: evitar cálculos repetidos):

```
============================================================
Algoritmo: moch_pd
Entrada: vetores v[1..n] e p[1..n], e P, como descrito acima
Saída: um número real
------------------------------------------------------------
01. Obter uma matriz M[0..P][1..n] de números reais.
02. PARA i DE 1 A n
03. | M[0][i] := 0
04. PARA peso DE 1 A P
05. | SE p[1] <= peso
06. | | M[peso][1] := v[1]
07. | SENÃO
08. | | M[peso][1] := 0
09. | PARA i DE 2 A n
10. | | VN := M[peso][i-1]
11. | | SE p[i] <= peso
12. | | | VS := M[peso - p[i]][i-1] + v[i]
13. | | | M[peso][i] := máx{ VN , VS }
```

```
14. | | SENÃO
15. | | | M[peso][i] := VN
16. RETORNE M[P][n].
============================================================
```

Observe que o algoritmo acima executa em tempo $\theta(n*P)$ e utilizando $\theta(n*P)$ de memória auxiliar.

4. EXERCÍCIO:

a) SOLUÇÃO COMPLETA: o algoritmo acima retorna o valor de uma solução ótima, mas não retorna a solução propriamente dita, isto é, o vetor e[1..n] de 0's e 1's que informa que itens são escolhidos. Estenda então o algoritmo, de forma que ele passe a receber como entrada também o vetor e[1..n], e então preencha o vetor com os valores correspondentes à solução ótima encontrada.

b) PESOS DESNECESSÁRIOS: suponha que, numa certa entrada do problema da mochila, tanto o valor P quanto os pesos dos objetos são todos números pares. Nesse caso, é completamente desnecessário calcular os elementos M[i][j] para valores ímpares de "i". Isso significa que aproximadamente a metade do tempo gasto pelo algoritmo "moch_pd" é em vão, no caso dessa entrada. Além disso, se todos os pesos forem múltiplos de 3, ao invés de 2, então a situação é ainda mais aguda: apenas aproximadamente 1/3 dos cálculos realizados pelo algoritmo são necessários.
ESCREVA ENTÃO uma variação do algoritmo acima que evita esse problema, e calcule a complexidade dessa variação.

c) PESOS RACIONAIS: acima foi possível construir uma solução de programação dinâmica porque nós sabíamos que, em todas as chamadas recursivas, a capacidade P da mochila era necessariamente um número inteiro. Numa aplicação mais prática desse problema, porém, pode muito bem ser o caso de os pesos dos objetos serem números racionais, não necessariamente inteiros. GENERALIZE, então, o algoritmo acima para o caso de pesos racionais (DICA: relacione esse caso com o caso do item "b" acima), e calcule a complexidade dessa generalização.