# Introdução à Análise de Algoritmos - Lista 2

## Mirela Mei - nº USP: 11208392

### Questão 1: Escreva um algoritmo guloso que selecione objetos em ordem do maior valor para o de menor valor que não excedam a capacidade W. Mostre com um exemplo que este algoritmo não resolve o problema da mochila.

O problema da mochila não pode ser solucionado por algoritmos gulosos, tendo em vista que neste caso a solução localmente ótima não é necessariamente aquela em que serão adicionados os itens de maior rentabilidade. Dessa forma, o algoritmo iria dividir a mochila de tamanho W em n mochilas de tamanho (W-1), (W-2), (W-3),...,1 e buscar a solução mais eficiente para cada uma delas. No caso, a solução mais eficiente é aquela envolvendo os itens com o maior resultado da divisão de seu valor pelo seu peso, porém, como mostrado no exemplo a seguir, as soluções parciais representam uma verdade para aquela parcialidade apenas.
Se a capacidade da mochila é 10, e os objetos: 1, de valor 11.000 e peso 10, 2 de valor 10.000 e peso 5 e 3 de valor 10.000 e peso 4. Como o algoritmo irá escolher o objeto de maior valor, será o objeto 1, ocupando toda a capacidade da mochila. Porém a melhor solução seria o O2 + O3 + O1/10, resultando em um valor de 21.100.

Considerando que os arrays “weight” e “value” estão ordenados:

```
Knapsack (weight, value, capacity)
i <- 0
para i<-0 até n
        profit[0][i] <- value[i] / weight[i]
        profit[1][i] <- weight[i]
enquanto capacity > 0
        items[i] <- profit[0][n]
        i++
        n–
        capacity <- capacity - profit[1][n]
retorna items
```

**Questão 2: Escreva um algoritmo de programação dinâmica que resolva o problema da mochila. Em função de W e n, assintoticamente qual o tempo de processamento de pior caso deste algoritmo? Também em função das mesmas variáveis, assintoticamente qual é o espaço de memória ocupado no pior caso?**

A lógica do algoritmo baseia-se em construir uma matriz que armazenará o valor máximo para cada máxima capacidade de carga da mochila.

```
DynamicKnapsack (numItems, capacity)
se items[numItems][capacity] != undefined
        retorna items[numItems][capacity]
se numItems == 0 ou capacity == 0
        result <- 0
senão se weight[numItems] > capacity
        result <- DynamicKnapsack(numItems-1, capacity)
senão
        aux1 <- DynamicKnapsack(numItems-1, capacity)
        aux2 <- value[numItems] + DynamicKnapsack(numItems-1,
capacity-weight[numItems])
        result <- max(aux1, aux2)
items[numItems][capacity] <- result
retorna result
```

O tempo de execução será θ(numItems * capacity), visto que a função pode ser chamada recursivamente por no máximo numItems*capacity vezes, e dependerá do tamanho da matriz.
O espaço de memória utilizado será proporcional ao produto de numItems e capacity, sendo representados pelas dimensões da matriz que armazena os resultados.

**Questão 3: Considere um arranjo de k bits A representando um número natural em notação binária. Esse arranjo começa com todas posições com 0 e é incrementado de um em um utilizando o seguinte algoritmo. Mostre, utilizando a técnica do contador para análise amortizada, que o tempo total e n operações de incremento tomam tempo total O(n).**

Associando o tempo de uma alteração de bit à uma moeda, pode-se dizer que antes da operação há um estoque de moedas que foi previamente acumulado ao longo das iterações. Para iniciar a operação, é necessário que haja o acréscimo de duas moedas, que serão utilizadas para realizar a atribuição “A[i] <- 1” na linha 6 e “A[i] <- 0” na linha 3.
Portanto, a invariante será: Antes de cada operação, o número de moedas no estoque é igual ao número de bits de A com valor igual a 1.
Então, k bits valem 1 e o k+1-ésimo bit vale 0. A invariante garante a existência de ao menos k moedas no estoque no início da execução, além da adição de duas moedas ao início de cada operação.
Entre as linhas 2 e 4, é gasta uma quantidade k de moedas para atribuir 0 a cada bit contendo 1. Posteriormente, o k+1-ésimo bit assume valor 1, consumindo a segunda moeda que foi adicionada. Pode-se concluir que o número de moedas permanece igual ao número de bits com o valor 1, portanto, a propriedade se mostra válida.
Assim, conclui-se que o bit A[0] é alterado em todas as execuções da operação. O bit A[1] é alterado a cada duas execuções, ou seja, [n/2] vezes. O bit A[2] é alterado [n/22] vezes e  o bit A[i] é alterado [n/2i] vezes. Portanto, o custo total é menor que $n + n/2 + n/22 + \cdots + n/2i + \cdots$ E essa soma não passa de 2n. É possível, portanto, verificar que o custo dessa operação é da ordem de O(n).