Lista 2 - IAA - Breno da Silva Nogueira - 12400392

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.

```
// A é uma matriz bidimensional de w x v
1 MochilaGulosa (A[w, v], W)
2   indices ← []
3   contador ← 0
4   OrdenarPorMaiorValor(A)
5   para i ← 1 até n
6       se (contador + A[i][0] ≤ W) // A[i][0] é o índice do peso do item
7           indices ← i
8           contador ← contador + A[i][0]
9
10  retorna indices
```

Análise:

É fácil perceber como este algoritmo não resolve o problema da mochila. Peguemos a seguinte situação:

Queremos achar a mochila ótima de capacidade de W=50, se utilizando seguinte grupo de itens:

Peso (w)	40	30	20	10
Valor (v)	840	600	400	100

Analisando as combinações, chegamos à conclusão que a mochila ótima para esse grupo de itens é a mochila ([30, 600], [20, 400]) que resulta no peso 50, e no maior valor de 1000.

Contudo, como o algoritmo considera o peso e o valor separadamente, (o valor como índice para ordenação, e o peso na ordem em que aparece na sequência ordenada), temos a seguinte saída:

```python

# p é igual a w no contexto do problema
def guloso (A, w):
    i = []
    tW = 0
    valor = 0
    A.sort(reverse=True, key=lambda x: x[1])

    for item in A:
        if(tW + item[0] <= w):
            i.append(item)
            tW += item[0]
            valor += item[1]

    print("A mochila está com os itens: " + str(i))
    print("O valor da mochila é: " + str(valor))


if __name__ == "__main__":
    # Matriz de cada elemento (w, v)
    matrix = [[40, 840],
    [30, 600],
    [20, 400],
    [10,100]]
    guloso(matrix, 50)

```

```
PROBLEMAS   SAÍDA   CONSOLE DE DEPURAÇÃO   TERMINAL

Windows PowerShell
Copyright (C) Microsoft Corporation. Todos os direitos reservados.

Experimente a nova plataforma cruzada PowerShell https://aka.ms/pscore6

PS C:\Users\Admin\Desktop\aed> & C:/Python39/python.exe c:/Users/Admin/Desktop/aed/iaa/guloso.py
A mochila está com os itens: [[40, 840], [10, 100]]
O valor da mochila é: 940
PS C:\Users\Admin\Desktop\aed>
```

Apesar de não ter excedido a capacidade da mochila, o algoritmo não acha a mochila ótima, por não ter a maior soma de valor pelo seu peso. Isso só seria possível nesta implementação caso os itens ótimos estejam no começo da sequência ordenada. Portanto, não resolve o problema da mochila para todo caso.

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 ´e o espa¸co de mem´oria ocupado no pior caso?

```
1 MochilaDinamica (w, v, n, W)
2   t[n, W] ← 0
3
4   para i ← 1 até n
5     para j ← 0 até W
6       t[i, j] ← t[i - 1, j]
7       aux ← t[i - 1, (j - w[i - 1])] + v[i - 1]
8       se j ≥ p[i - 1] e t[i, j] < aux
9         t[i, j] ← aux
10
11  retorna t[n, W]
```

Análise:

1) Como o algoritmo vai criar uma tabela de opções possíveis de combinações, o algoritmo vai rodar até que uma matriz de W x n seja criada, iterando por todos as combinações, guardando todos os cálculos já feitos para a resolução do problema maior. Neste caso, em razão de n e W, o algoritmo vai em seu pior caso, rodar em O(nW) para descobrir o valor que a mochila ótima carregará.

2) Considerando que no pior caso do algoritmo, ele criará uma matriz de soluções com as dimensões de (W + 1) x (n + 1), podemos generalizar o consumo de memória em O(nW).

Extra:

```python
def dinamico(p, v, n, w):
    t = [[0 for x in range(w + 1)] for y in range(n + 1)]

    for i in range(1, n + 1):
        t[0][i] = 0
        for j in range(0, w + 1):
            t[i][j] = t[i - 1][j]

            aux = t[i - 1][j - p[i - 1]] + v[i - 1]
            if j >= p[i - 1] and t[i][j] < aux:
                t[i][j] = aux

    print("O maior valor possível é: " + str(t[n][w]))

    index = []
    while n != 0:
        if(t[n][w] != t[n - 1][w]):
            index.append(n)
            w -= p[n - 1]
        n -= 1

    return index

if __name__ == "__main__":
    # p é o w do enunciado que se refere a sequência de pesos
    p = [40, 30, 20, 10]
    v = [840, 600, 400, 100]
    print("Os pacotes que foram escolhidos são: " + str(dinamico(p, v, len(p), 50)))
```

```
PROBLEMAS   SAÍDA   CONSOLE DE DEPURAÇÃO   TERMINAL

Windows PowerShell
Copyright (C) Microsoft Corporation. Todos os direitos reservados.

Experimente a nova plataforma cruzada PowerShell https://aka.ms/pscore6

PS C:\Users\Admin\Desktop\aed> & C:/Python39/python.exe c:/Users/Admin/Desktop/aed/iaa/dinamico.py
O maior valor possível é: 1000
Os pacotes que foram escolhidos são: [3, 2]
PS C:\Users\Admin\Desktop\aed>
```

Uma implementação do algoritmo mostrado em python, resolvendo o problema apresentado no exercício anterior, onde, de fato, resolve o problema da mochila, retornando o itens ([30, 600], [20, 400]) (aqui representados pelos números na sequência), e o seu valor correto de 1000 unidades de valor.

3) Mostre, utilizando a técnica do contador para análise amortizada, que o tempo total e *n* operações de incremento tomam tempo total O(n).

```
Incrementa(A)
1  i ← 1
2  while i < k e A[i] = 1
3        do A[i] ← 0
4                 i ← i + 1
5 if i < k
6        then A[i] ← 1
```

Análise:

Se utilizando da técnica do contador, podemos criar a seguinte proposição.

Vamos considerar que o comando para mudar o bit para 1 custe uma "unidade de execução" rodar no algoritmo. Toda vez que quisermos usar tal comando para atualizar o bit teremos que cobrar 2 unidades, uma unidade é cobrada para mudar o bit para 1 e outra fica de "crédito" para quando tivermos que atualizar o bit de novo para 0, em outra chamada da função. Em qualquer momento a quantidade de unidades de execução da função para cada bit será igual a quantidade de bits com valor 1 no arranjo.

Com essa ideia em mente, podemos afirmar que durante a execução da função, sempre que formos incrementar o arranjo, a linha 6 roda apenas uma vez por iteração, então o valor por incremento vai ser 2 unidades, pois para transformar os 0 em 1, vamos utilizar as unidades acumuladas das operações anteriores. Como a quantidade de bits com valor 1 no arranjo sempre será positiva, pois estamos aumentando a quantidade de um 1's no arranjo por ser uma função de incremento, garantimos que durante as execuções da função sempre teremos crédito disponível.

Dessa forma, ainda na analogia de "unidades de execução" podemos inferir que com o custo para cada incremento sendo de 2 unidades, teremos então para *n* incrementos o gasto de *2n* unidades.

Traduzindo isso em termos de notação de tempo, a função roda em tempo polinomial. Assim, chegamos a conclusão que o tempo de execução de n incrementos é O(n). Ou seja, em seu pior caso, toma tempo linear de execução