UNIFEI - UNIVERSIDADE FEDERAL DE ITAJUBÁ

CURSO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

SIN110 - ALGORITMOS E GRAFOS

RESOLUÇÃO DOS EXERCÍCIOS E11 DO DIA 06/11/2015

ITAJUBÁ

2015

Exercícios E11 – 06/11/15

Aluna: Karen Dantas **Número de matrícula:** 31243

1) Algoritmo:

```
Encontra_quantidade_maçãs (x, y, n, m, k, MT, Vet)
 1. se (x = k+1 ou y = 3+1)
 2.       devolve cont_maça
 3. Vet = encontra_adjacentes (MT[x][y])
 4. para i ← 1 até i ≤ num_de_adjacentes_encontrados faça
 5.        se (Vet[i] = tempo + 1)
 6.                cont_maça← cont_maça + 1
 7.                tempo ← tempo + 1
 8.                descobre_posiçao (Vet[i], x, y)
 9.  Encontra_quantidade_maçãs (x, y, n, m, k, MT, Vet)
```

O algoritmo descobre a quantidade de maçãs que Rafael pegou através dos tempos dos adjacentes do índice [x, y] da matriz. Na linha 8, as variáveis ‘x’ e ’y’ são atualizadas para a coordenada do adjacente que possui o tempo de caída da maçã correspondente ao tempo que Rafael estará naquela posição. Para se verificar o tempo de seus adjacentes são realizadas chamadas recursivas.

É utilizada a programação dinâmica, pois, o problema maior foi quebrado em subproblemas para se descobrir quantas maçãs Rafael conseguiu pegar dentre todas as 'k' maçãs derrubadas por seu primo.

2) Algoritmo:

```
Numero_Mínimo_Tripla (D, f)
1.  c ← 1
2.  para i ← 1 até i ≤ numero_de_palavras faça
3.      palavra ← D[i]
4.      para j ← 1 até j < tamanho_frase faça
5.              para k ← 1 até k < tamanho_palavra faça
6.                      se (f[j] = palavra[k] e f[j+1] = palavra[k+1])
7.                              então   triplas[c] ← monta_tripla(i, k, k+1)
8.                                      c ← c + 1
9.                                      exclui_silaba(f, j, j+1)
10.                                     j ← j + 1
11. se (existe_silaba (f) = verdadeiro)
12.   para i ← 1 até i ≤ numero_de_palavras faça
13.             palavra ← D[i]
14.             para j ← 1 até j ≤ tamanho_frase faça
15.                     para k ← 1 até k ≤ tamanho_palavra  faça
16.                             se (f[j] = palavra[k])
17.                                     então   triplas[c] ← monta_tripla(i, k, k)
18.                                             c ← c + 1
19.                                             exclui_silaba(f, j, j)
20.                                             j ← j + 1
21.                             se (existe_silaba (f) = falso)
22.                                     devolve triplas
23. devolve triplas
```

Correção: Na linha 1 tem um laço ‘para’ que fará com que sejam percorridas todas as palavras do dicionário. Na linha 3 tem um laço ‘para’ que irá percorrer os caracteres da frase e na linha 4 o laço ‘para’ irá percorrer os caracteres das palavras do dicionário. Na linha 5 é verificado se existe uma sílaba na palavra que tenha na frase, se sim, ele monta a tripla e exclui a sílaba da frase para se manter o controle e não se correr o risco de se verificar novamente se outra palavra possui essa sílaba já encontrada. Na linha 9 é verificado se existe alguma sílaba que não foi encontrada, se sim, ele faz o mesmo procedimento dos laços que foram explicados anteriormente, com a diferença de que é verificado apenas se a palavra tem o caractere igual a um dos caracteres das sílabas da frase. O algoritmo, por ter três laços alinhados um dentro do outro, possui complexidade O (n^3).

O algoritmo utiliza o método guloso, pois, sempre buscará, primeiramente, uma palavra que tenha uma sílaba da frase e, posteriormente, caso necessário, que possua um caractere da sílaba da frase. E, no fim de sua execução, ele retorna o menor número de triplas possível.

---

3) Algoritmo:

```
Min_Env (p, n)
1.  num_min_envolp ← 0, j ← n
2.  MergeSort (p, n)
3.  para i ← 1 até i ≤ j faça
4.          se (p[i]+ p[j] ≤ 1)
5.                  então i ← i + 1
6.          j ← j – 1
7.          num_min_envolp ← num_min_envolp + 1
8.  devolve num_min_envolp
```

O consumo de tempo do algoritmo é O (nlgn) como pode ser visto abaixo:

Linhas	Consumo
1	O (1)
2	O (nlgn)
3	O (n)
4	O (n)
5	0
6	O (n)
7	O (n)
8	O (1)

Considerei como pior caso que todos os envelopes conterão apenas um livro, ou seja, todos os livros conterão pesos muito altos não tornando possível colocá-los junto com outros dentro dos envelopes. Logo, a linha 5 nunca será executada.

Cálculo: 2.O(1) + O (nlgn) + 4.O(n) + 0 = O (nlgn)

Correção: A linha 3 garante que todos os pesos dos livros serão verificados e as linhas 5 e 6

garantem que o algoritmo tem parada.

A linha 4 sempre verificará se é possível colocar o livro de peso[i] com o livro de peso[j] em um mesmo envelope de forma que não se exceda o peso 1 e obtenha-se o número mínimo de envelopes. E a linha 7 contabilizará o número mínimo de envelopes que serão precisos o qual é retornado pela função na linha 8.

O algoritmo é do tipo guloso, pois, ele sempre vai colocar no envelope o livro de peso p[j] junto ao livro de peso[i] caso a soma de seus pesos for menor ou igual a 1. Caso contrário, o livro de p[j] ficará sozinho no envelope e a execução continua comparando-se o peso de p[i] com p[j-1]. Logo, no fim de sua execução, ele retorna o menor número possível de envelopes.