Exercícios de Programação Dinâmica

PARTE I

1)

```
corte_de_barras(p, n)
1.  r[0..n] = new array
2.  for i = 0 até n
3.      r[i] = -infinito
4.  return corte_de_barras_aux(p, n, r)

corte_de_barras_aux(p, n, r)
1.  if r[n] => 0
2.      return r[n]
3.   if n == 0
4.      q = 0
5.   else q = −infinito
6.   for i = 1 até n
7.      q = max(q, p[i]+corte_de_barras_aux(p, n−i, r))
8.   r[n] = q
9.  return q
```

2)

```
subsequencia_mais_longa(X, Y)
1.  m = length(X)
2.  n = length(Y)
3.  for i= 0 até m
4.      for j= 0 até n
5.              c[i, j] = infinito
6.  return subsequencia_mais_longa_aux(X, Y, m, n)

subsequencia_mais_longa_aux(X, Y, i, j)
1.  if c[i, j] == infinito
2.      if i == 0 OU j == 0
3.              c[i, j] = 0
4.      elseif X[i] == Y[j]
5.              c[i, j] = subsequencia_mais_longa_aux (X, Y, i − 1, j − 1) + 1
6.      else
7.              a = subsequencia_mais_longa_aux(X, Y, i − 1, j)
8.              b = subsequencia_mais_longa_aux(X,Y , i, j − 1)
9.              c[i, j] = max(a, b)
10.  return c[i,j]
```

3)

```
subsequencia_continua(S, p, r)
```

```
1.  F[p] = A[p]
2.  para q = p + 1 até r
3.      se F[q − 1] > 0
4.          F[q] ← F[q−1] + S[q]
5.      senão
6.          F[q] ← S[q]
7.          x = q
8.  para q = p + 1 até r
9.      se F[q] > x
10.         x = F[q]
11. return x
```

4)

```
estoque_moeda(x, v)
inicialize t
for i = 1 até v
        t[i] = false
        for j = 1 até length(x)
                if i >= moeda[j] AND t[i – moeda[j]]
                        t[i] = true
return t[v]
```

5)

```
3_particao(A)
1.  m = somatorio dos elemntos de A
2.  if m mod 3 != 0
3.      return false
4.  aux = m/3
5.  a, b, c = 1
6.  for l = 0 até n:
7.      enquanto sum(I) != aux AND sum(j) != aux AND sum(k) != aux
8.          if  sum(I) != aux AND aux >= sum(I) + A[l]
9.              I[a] = A[l]
10.             a++
11.         else if  sum(J) != aux AND aux >= sum(J) + A[l]
12.             J[b] = A[l]
13.             b++
14.         else if  sum(K) != aux AND aux >= sum(K) + A[l]
15.             K[c] = A[l]
16.             c++
17.         else
18.             return false
19.  return true
```

PARTE II

1)

```
Memoized-Cut-Rod(p, n)
1.  r[0..n] = new array
2.  for i = 0 até n
3.      r[i] = -infinito
4.  return Memoized-Cut-Rod-Aux(p, n, r)

Memoized-Cut-Rod-Aux(p, n, r)
1.  if r[n] => 0
2.      return r[n]
3.   if n == 0
4.      q = 0
5.   else q = −infinito
6.   for i = 1 até n
7.      q = max(q - c, p[i])
8.   r[n] = q
9.  return q
```

2)

```
palindromo(palavra)
1.  n = tamanho da palavra
2.  [1..n] são as letras da palavra dada
3.  dividir a palavra de entrada em alguma posição i
4.  resolver o mais longo problema de subseqüência comum [1..i] e [i+ 1..n]
```

Primeiro note que para um palíndromo ser uma subsequência, devemos ser capazes de dividir a palavra de entrada em alguma posição i e, em seguida, resolver o mais longo problema de subseqüência comum [1..i] e [i+ 1..n]. Já que existem n lugares em que poderíamos dividir o palavra de entrada e o problema leva tempo O(n^2), resolver o palíndromo problema no tempo O(n3).

3)

a) Vamos indexar nossos subproblemas por dois inteiros, $1 \leq i \leq m$, que indicará um elemento de x e $1 \leq j \leq n$, que indicará um elemento de y.

```
edicao_distancia(x, y, i, j)
1.   m = último elemento de x
2.   n = último elemento de y
3.   if i = m
4.      return (n-j)cost(insert)
5.   end
```

```
6.  if j = n
7.     return min{(m−i)cost(delete), cost(kill)}
8.  end
9.  inicialização de a1, a2, a3, a4, a5
10. if x[i] = y[j]
11.    a1 = cost(copy) + edicao_distancia(x, y, i+ 1, j+ 1)
12. end
13. a2 = cost(replace) + edicao_distancia(x, y, i+ 1, j+ 1)
14. a3 = cost(delete) + edicao_distancia(x, y, i+ 1, j)
15. a4 = cost(insert) + edicao_distancia(x, y, i, j+ 1)
16. if i < m−1 e j < n−1
17.    if x[i] = y[j+ 1] e x[i+ 1] = y[j]
18.           a5 = cost(twiddle) + edicao_distancia(x, y, i+ 2, j+ 2)
19.    end if
20. end if
21. return min{a1, a2, a3, a4, a5}
```

b) Uma tradução de custo mínimo da primeira cadeia na segunda corresponde a um alinhamento. Uma “copy” ou “replace” incrementando um ponteiro para as duas cadeias. Uma “insert” como um espaço na posição atual do ponteiro na primeira string. Uma operação de “delete” significa colocar um espaço na posição atual da segundo cadeia. Como os “twiddle” e “kill” têm custos infinitos, não teremos uma solução de custo mínimo.