Entrega 3

Daniel Brito

1) [OPCIONAL] Em uma heap, sabemos que o filho esquerdo de um nó é indexado por $\lfloor \frac{n}{2} \rfloor + 1$, e também que heaps são árvores quase completas, ou seja, não é adicionado um filho à direita sem que antes seja adicionado o filho à esquerda. Assim, temos que:

$$\begin{aligned} ESQ(\left\lfloor \frac{n}{2} \right\rfloor + 1) &= 2(\left\lfloor \frac{n}{2} \right\rfloor + 1) \\ &> 2(\frac{n}{2} - 1) + 2 \\ &= n - 2 + 2 \\ &= n \end{aligned}$$

Uma vez que o elemento do índice da esquerda representa um número maior que a quantidade de elementos da heap, o nó não possui filhos, logo, é uma folha. O mesmo acontece com todos os nós com índices maiores.

2) [OPCIONAL] Note que, para qualquer $n > 0$, o número de folhas de uma árvore quase completa é $\lceil \frac{n}{2} \rceil$.

Seja n_h o número de nós em uma altura h. O *upper bound* é válido para o caso base, uma vez que $n_0 = \lceil \frac{n}{2} \rceil$ é exatamente o número de folhas em uma heap de tamanho n.

Passo indutivo: Suponha que é verdade para $h - 1$. Assim, temos que provar que também é válido para h.

Note que se n_{h-1} é par, cada nó de uma árvore de altura h tem exatamente dois filhos, o que implica $n_h = \frac{n_{h-1}}{2} = \lfloor \frac{n_{h-1}}{2} \rfloor$.

Se n_{h-1} é ímpar, um nó na altura h tem um filho, e o restante tem dois filhos, o que implica $n_h = \lfloor \frac{n_{h-1}}{2} \rfloor + 1 = \lceil \frac{n_{h-1}}{2} \rceil$.

Assim, temos que:

$$
\begin{aligned}
n_h &= \left\lceil \frac{n_{h-1}}{2} \right\rceil \\
&\leq \left\lceil \frac{1}{2} \cdot \left\lceil \frac{n}{2^{(h-1)+1}} \right\rceil \right\rceil \\
&= \left\lceil \frac{1}{2} \cdot \left\lceil \frac{n}{2^h} \right\rceil \right\rceil \\
&= \left\lceil \frac{n}{2^{h+1}} \right\rceil
\end{aligned}
$$

3) [OPCIONAL] Abaixo, o algoritmo que determina os valores que representam o infixo de soma máxima em uma sequência de números:

```
from sys import maxsize as MAX

def somaMaxima(a, n):
  maximo = - MAX
  somaAtual = 0
  inicio = 0
  fim = 0
  s = 0

  for i in range(n):
    somaAtual += a[i]

    if maximo < somaAtual:
      maximo = somaAtual
      inicio = s
      fim = i

    if somaAtual < 0:
      somaAtual = 0
      s = i + 1

  return a[inicio:fim+1]
```

4) Abaixo, o algoritmo que determina em que hotéis deve-se parar, de forma que a soma das penalidades diárias seja minimizada:

```
def hoteis(a):
  n = len(a)
  penalidades = [0]*n
  paradas = [0]*n

  for i in range(n):
    penalidades[i] = int(math.pow(200 - a[i], 2))
    paradas[i] = 0

    for j in range(i):
      if penalidades[j] + math.pow((200 - (a[i] - a[j])), 2) < penalidades[i]:
        penalidades[i] = int(penalidades[j] + math.pow(200 - (a[i] - a[j]), 2))
        paradas[i] = j + 1

  resultado = []
  index = len(paradas)-1

  while index >= 0:
    resultado.append(index + 1)
    index = paradas[index] - 1

  return resultado[::-1]
```

5) Abaixo, o algoritmo que decide se uma dada string `s[1..n]` pode ser segmentada em uma sequência de palavras de um dicionário `d`:

```
def consulta(d, s):
  if not s:
    return True

  for i in range(1, len(s) + 1):
    prefixo = s[:i]

    if prefixo in d and consulta(d, s[i:]):
      return True

  return False
```

6) Abaixo, o algoritmo que encontra a subsequência palindrômica máxima de $a_1, a_2, ..., a_n$:

```
def palindromo(a):
  dp = [[False for i in range(len(a))] for i in range(len(a))]

  for i in range(len(a)):
     dp[i][i] = True

  max = 1
  inicio = 0

  for i in range(2, len(a) + 1):
    for j in range(len(a)- i + 1):
      fim = j + i

      if i == 2:
        if a[j] == a[fim-1]:
          dp[j][fim-1] = True
          max = i
          inicio = j
      else:
        if a[j] == a[fim-1] and dp[j+1][fim-2]:
          dp[j][fim-1] = True
          max = i
          inicio = j

  return a[inicio:inicio+max]
```

7) Abaixo, o algoritmo que encontra o maior comprimento de uma substring comum entre duas strings `x[1..m]` e `y[1..n]`:

```
def comprimentoSubstring(x, y):
  m = len(x)
  n = len(y)

  comum = [[0 for i in range(n+1)] for j in range(m+1)]

  maiorComprimento = 0

  for i in range(m + 1):
    for j in range(n + 1):
      if i == 0 or j == 0:
        comum[i][j] = 0
      elif (x[i-1] == y[j-1]):
        comum[i][j] = comum[i-1][j-1] + 1
        maiorComprimento = max(maiorComprimento, comum[i][j])
      else:
        comum[i][j] = 0

  return maiorComprimento
```

8) Abaixo, o algoritmo para calcular a probabilidade de se obter exatamente `k` caras quando cada uma das `n` moedas é arremessada uma vez:

```
from math import log2

def probabilidade(k, n):
  resposta = 0

  for i in range(k, n+1):
    exp = e[n] - e[i] - e[n-i] - n
    resposta += pow(2.0, exp)

  return resposta
```

Para otimizar o cálculo do fatorial, é realizada uma pré-computação de `e[]`, utilizando o módulo `log2` da biblioteca `math`. Caso contrário, corre-se o risco de ocorrer overflow para valores muito altos:

```
N = 1000001
e = [0] * N

for i in range(2, N):
  e[i] = log2(i) + e[i-1]
```

9) TO-DO

10) Abaixo, o algoritmo para determinar o número mínimo de moedas necessárias para representar um certo valor v:

```
def minMoedas(x, v):
   if v == 0:
      return 0

   if v < 0:
      # Caso nao seja possivel representar v:
      return float('INF')

   moedas = float('INF')
   n = len(x)

   for i in range(n):
      m = minMoedas(x, v - x[i])

      if m != float('INF'):
         moedas = min(moedas, m + 1)

   # Caso seja possivel representar v:
   return moedas
```

11) TO-DO