



**Aluna:** Daniela América da Silva  
**Disciplina:** CT208 - Matemática Computacional  
**Prof. Ney Soma**  
**Orientação:** Prof. Tasinaffo  
**Co-orientação:** Prof. Johnny  
**Data:** 23/Junho/2020

## **Exercício: Dedução do caso médio do maior, menor elemento e autômato celular**

*Dedicado à memória de Heitor Rafael Silva, celebraria 15 anos em 21-Junho-20  
(21/06/2005-09/05/2019)*

### **1- Introdução**

Este relatório analisa os algoritmos:

- Dedução do caso médio do maior
- Tempo de processamento do maior e menor elemento
- Tempo de processamento dos dois menores

Adicionalmente implementa:

- heap do máximo
- heap do mínimo
- heap do mínimo para uma amostra de 100 elementos (com valores de 2 à 10000)

Quanto ao autômato celular é apresentado:

- regra 90 começando com uma única célula central
- uma formulação matemática que não faz parte do jardim do Éden

Na seção anexo estão os slides utilizados como referência a partir da CT-234.

Os códigos estão armazenados no gitHub

[https://github.com/dasamerica/ct208/tree/master/aula\\_10jun\\_maiormenor\\_CA](https://github.com/dasamerica/ct208/tree/master/aula_10jun_maiormenor_CA)

### **2- Dedução do caso médio do maior**

#### Questão 01

⇒ Se você tivesse que determinar  $\bar{m}_n$  para  $n = 10^{10}$ , não seria possível utilizar encontrar o valor numérico a partir de  $T_{n,k}$ , certo? (Explique seu “sim” ou o seu “não”). ⇐

Não é possível encontrar o valor numérico a partir de  $T_{n,k}$ , para  $n = 10^{10}$  pois o valor recursivo de trocas será muito grande.

Por exemplo:

$$\begin{aligned} T_{10^{10},k} &= T_{10^{10}-1,k-1} + (10^{10} - 1) T_{10^{10}-1,k} \\ T_{10^{10}-1,k-1} &= T_{10^{10}-2,k-2} + (10^{10} - 2) T_{10^{10}-2,k-1} \\ T_{10^{10}-1,k} &= T_{10^{10}-2,k-1} + (10^{10} - 2) T_{10^{10}-2,k} \end{aligned}$$

E assim por diante, gerando um número massivo de recursões.

## Questão 02

⇒ A seguinte afirmação está correta? Explique.

“O cálculo de  $\bar{m}_n$  a partir de  $T_{n,k}$  é inviável pois,

$$T_{n,k} = T_{(n-1),(k-1)} + (n-1)T_{(n-1),k},$$

e em termos de tempo de processamento (chamadas recursivas), sejam elas  $c(T_{n,k})$ , temos que  $c(T_{(n-1),(k-1)}) = \mathcal{O}(c(T_{(n-1),k}))$

$$c(T_{n,k}) = 2c(T_{n-1}), \quad \text{e} \quad c(T_{2,0}) = c(T_{2,1}) = 1.$$

Logo,  $c(T_{n,k}) = \mathcal{O}(2^n)$ . Como é preciso ter  $n$  valores para o cálculo de  $\bar{m}_n$ , temos que todo o processo será limitado por  $\mathcal{O}(n2^n)$  o que faz com que o cálculo a partir da recorrência  $T_{n,k}$  seja impraticável mesmo para valores de  $n$  ‘pequenos’. Como exemplo, mesmo para  $n = 30$  já não é possível tal cálculo.” ⇐

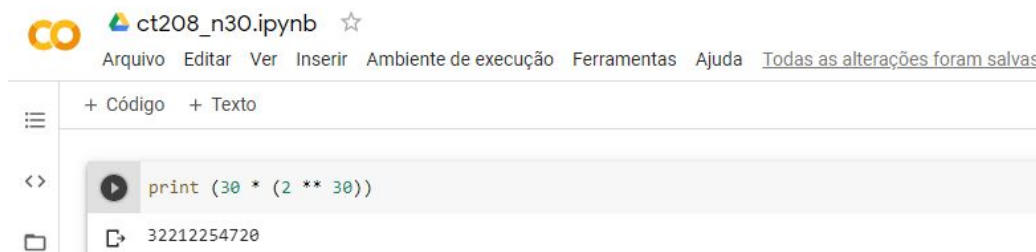
A definição de  $c(T_{n,k}) = \mathcal{O}(2^n)$  está correta conforme demonstrado, pois por definição, em comandos consecutivos somam-se os tempos (Capítulo 01 CE-234, páginas 31 e 34).

Adicionalmente, na hierarquia de funções da notação  $\mathcal{O}$ , a função exponencial é a com maior ordem, conforme definições apresentadas nas figuras 3 e 4 do anexo.

E o cálculo impraticável mesmo para valores pequenos de  $n$ , com trocas acima de 32 milhões para  $n=30$ , conforme código python a seguir.

Fonte no Github:

[https://github.com/dasamerica/ct208/blob/master/aula\\_10jun\\_maiormenor\\_CA/ct208\\_n30.py](https://github.com/dasamerica/ct208/blob/master/aula_10jun_maiormenor_CA/ct208_n30.py)



The screenshot shows a Jupyter Notebook window titled 'ct208\_n30.ipynb'. The interface includes a menu bar with options like 'Arquivo', 'Editar', 'Ver', 'Inserir', 'Ambiente de execução', 'Ferramentas', 'Ajuda', and a status bar indicating 'Todas as alterações foram salvas'. Below the menu, there is a code editor with a single cell containing the Python code `print(30 * (2 ** 30))`. The output of this cell is displayed as `32212254720`.

## 3- Maior e o Menor

### Questão 03

⇒ A seguinte afirmação está correta? Explique.

**Explicação Maior e menor:** “Ao determinar o maior e o menor elemento em tempo limitado por  $2n - 3 = 2n + \mathcal{O}(1) = \mathcal{O}(n)$ , é claro que o algoritmo será ótimo. Sei que não é possível fazer em um tempo menor que  $\Omega(n)$ . Logo como os dois limitantes são iguais, o algoritmo que primeiro determina o maior dentre  $n$ -elementos e depois o que encontra o menor dentre  $n-1$ -elementos é o algoritmo ótimo.” ⇐

Conforme premissa da Figura 1, uma constante é  $\mathcal{O}(1)$  e uma função linear  $2n$  é  $\mathcal{O}(n)$ . E convém utilizar a menor ordem portanto o tempo estará limitado por  $\mathcal{O}(n)$  (Capítulo 01 CE-234, página 26). Adicionalmente os dois algoritmos têm a mesma taxa de crescimento. Ou seja o mesmo tempo gasto para encontrar o maior é o tempo gasto para encontrar o menor elemento (Capítulo 01 CE-234, página 24). Definições apresentadas nas figuras 5 e 6 do anexo. Entretanto este não é o algoritmo ótimo pois é possível utilizar uma otimização alternativa apresentada na questão 4.

### Questão 04

⇒ A explicação a seguir está correta? Explique.

**Explicação Maior e menor alternativa:** “Ao determinar o maior e o menor elemento procurando-os nas posições pares e ímpares do vetor  $v$  gastaremos somente  $n$  comparações e com mais  $n/2$  comparações para separar os maiores dos menores. O tempo total limitado por  $3n/2$  é muito melhor que o determinado na ‘*Explicação Maior e menor*’ dada antes, que era  $2n + \mathcal{O}(1)$  é claro que este algoritmo é que será ótimo. Sei que não é possível fazer em um tempo menor que  $\Omega(n)$ . Logo como os dois limitantes são iguais e o algoritmo aqui é o ótimo.” ⇐

Correto, conforme apresentado em aula, realizando as operações separando por números pares e ímpares, é possível diminuir o tempo de busca, por  $3n/2$  e portanto não é possível executar em um tempo menor que  $\Omega(n)$ . Por definição, quando uma função pertence simultaneamente ao limite superior e inferior dizemos que é ótima. Referências figuras 7 e 8 do anexo.

### Questão 05

⇒ A explicação a seguir está correta? Explique.

“Nenhuma das duas explicações anteriores, para a determinação do maior e do menor, é correta. É possível que a menor quantidade de operações necessárias para encontrar o Maior e o menor seja limitado no máximo por  $7n/5$ . Assim, nenhuma das duas conclusões é correta.” ⇐

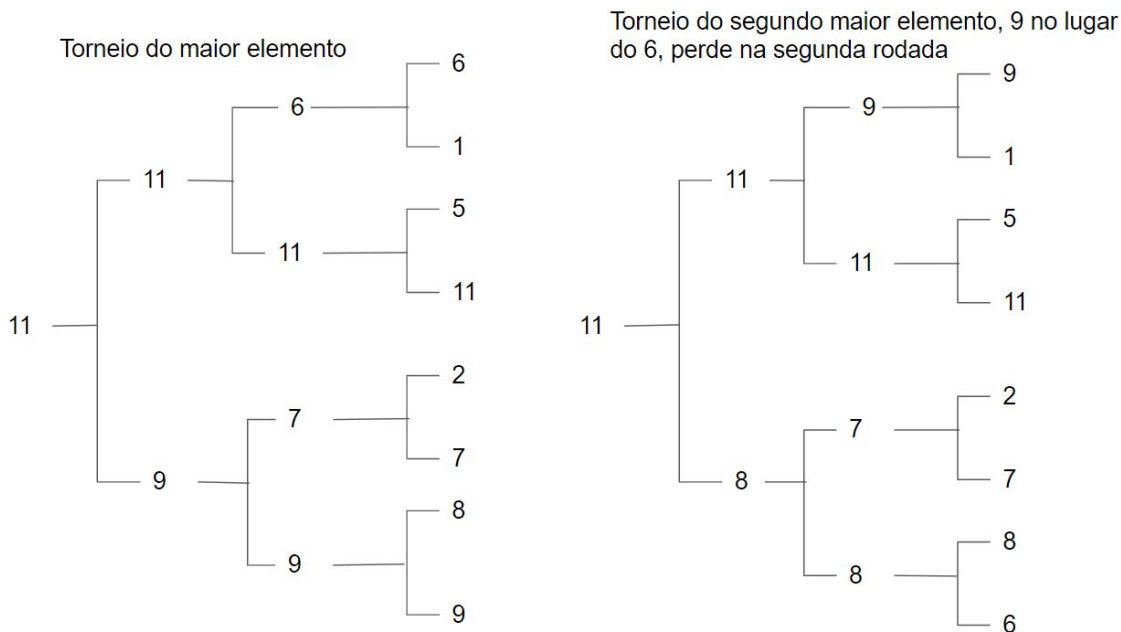
Incorreto, eu pesquisei outras técnicas como a média das médias, porém não é adequada para a determinação do maior e do menor, sendo o tempo total  $3n/2$  apresentado na questão 04 o algoritmo ótimo.

### 3- Os dois menores

#### Questão 06

⇒ Tanto o algoritmo de encontrar o Maior e o menor quanto o dos dois menores são algoritmos ótimos. Para o caso do Maior e do menor o limitante superior foi apresentado, mas a explicação de porquê ele é ótimo não. Faça a explicação do limitante inferior com a matéria apresentada em sala. ⇐

Utilizando a mesma representação da aula, é realizado a seguir a representação do torneio para identificar o maior elemento. Assim como no torneio do menor elemento também é necessário guardar uma lista de quem perdeu para o maior para que seja processado o segundo maior elemento. Dito isto o torneio para o maior elemento é também  $O(\lg n)$ .



### Questão 07

Por outro lado, o dos dois menores o limitante inferior tem a explicação dada aqui, mas o algoritmo (limitante superior) não. Para o algoritmo usa-se um Heap de mínimo e o tempo é de  $\mathcal{O}(n)$ . Estude o capítulo 6 do livro do Cormen.

Antes de implementar o heap de mínimo estudei o heap de máximo utilizando o código apresentado na CT-234. Dito isto, segue exemplo de implementação via código Python do heap máximo, calculando o maior e o menor, o segundo maior e o segundo menor e, apresentando os resultados no formato de lista e no formato de árvore. Neste código, para o cálculo do segundo maior e o segundo menor, é utilizada a propriedade de extração do máximo e do mínimo, e verificado novamente o máximo e o mínimo na nova estrutura da árvore. Tanto o heap de máximo quanto o heap de mínimo são  $\mathcal{O}(n)$ . Definições apresentadas nas figuras 9,10 e 11 do anexo.

Fonte no Github:

[https://github.com/dasamerica/ct208/blob/master/aula\\_10jun\\_maiormenor\\_CA/ct208\\_heap\\_maximo\\_maior\\_menor.py](https://github.com/dasamerica/ct208/blob/master/aula_10jun_maiormenor_CA/ct208_heap_maximo_maior_menor.py)

## Código Python

```
# Daniela America da Silva
# Exemplos de implementação heap maximo utilizando exemplo Prof. Alonso CT234

def Sift(i, n):
    esq = 2*i
    dir = (2*i)+1
    maior = i
    if (esq <= n and v[esq] > v[i]):
        maior = esq
    if (dir <= n and v[dir] > v[maior]):
        maior = dir
    if (maior != i):
        aux = v[i]
        v[i] = v[maior]
        v[maior] = aux
        Sift(maior, n)

def Build(v):
    n = len(v)-1
    x = int(n/2)
    for i in range (x,0,-1):
        Sift(i, n)

def Max():
    return v[1]

def Min():
    return v[len(v)-1]

def ExtractMax():
    size = len(v)-1
    if (size < 1):
        print ("heap underflow")
    else:
        max = v[1]
        v[1] = v[size]
        del v[size]
        size = len(v)-1
        #print (1,size)
        Sift(1,size)
        return max

def ExtractMin():
    size = len(v)-1
    if (size < 1):
        print ("heap underflow")
```

```

    else:
        min = v[size]
        del v[size]
        size=len(v)-1
        #print (1,size)
        Sift(1,size)
        return min

# Imprime a arvore
# exemplo do link
https://www.w3resource.com/python-exercises/heap-queue-algorithm/python-heapq-exercise-19.ph
p
import math
from io import StringIO
def show_tree(tree, total_width=60, fill=' '):
    """Pretty-print a tree.
    total_width depends on your input size"""
    output = StringIO()
    last_row = -1
    y=[] + tree
    del y[0]
    for i, n in enumerate(y):
        if i:
            row = int(math.floor(math.log(i+1, 2)))
        else:
            row = 0
        if row != last_row:
            output.write('\n')
        columns = 2**row
        col_width = int(math.floor((total_width * 1.0) / columns))
        output.write(str(n).center(col_width, fill))
        last_row = row
    print (output.getvalue())
    print ('-' * total_width)
    return

v = ["",4,1,3,2,16,9,10,14,8,7]
v[0] = "Lista"
print (v)
Build(v)
v[0]= "Heap"

print(v)
print("Max:",Max(), "Min:",Min())

show_tree(v)

print("Extract Max", ExtractMax())

```

```

print("Extract Min", ExtractMin())

v[0]= "Heap sem max e sem min"
print(v)
print("2o Max:",Max(), "2o Min:",Min())

show_tree(v)

```

## Resultado da execução

```
['Lista', 4, 1, 3, 2, 16, 9, 10, 14, 8, 7]
```

```
['Heap', 16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
```

**Max: 16 Min: 1**

```

              16
            14      10
          8        7        9        3
        2    4    1
-----

```

Extract Max 16

Extract Min 1

```
['Heap sem max e sem min', 14, 8, 10, 4, 7, 9, 3, 2]
```

**2o Max: 14 2o Min: 2**

```

              14
            8      10
          4        7        9        3
        2
-----

```



```
ct208_heap_maximo_maior_menor.ipynb
File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

show_tree(v)

['Lista', 4, 1, 3, 2, 16, 9, 10, 14, 8, 7]
['Heap', 16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
Max: 16 Min: 1

      16
     /  \
    8    14
   /  \  /  \
  2  4 1  7  9  10 3
-----
Extract Max 16
Extract Min 1
['Heap sem max e sem min', 14, 8, 10, 4, 7, 9, 3, 2]
2o Max: 14 2o Min: 2

      14
     /  \
    4    8
   /  \  /  \
  2  7 1  9  10 3
-----
```

A implementação do heap de mínimo é similar ao heap de máximo e portanto conforme apresentado por Cormen no capítulo 6, o código para o heap de mínimo foi escrito substituindo no procedimento do Sift o cálculo do maior pelo cálculo do menor, e portanto continua sendo  $O(n)$ .

Fonte no Github:

[https://github.com/dasamerica/ct208/blob/master/aula\\_10jun\\_maiormenor\\_CA/ct208\\_heap\\_minimo\\_maior\\_menor.py](https://github.com/dasamerica/ct208/blob/master/aula_10jun_maiormenor_CA/ct208_heap_minimo_maior_menor.py)

## Código Python

```
# Daniela America da Silva
# Exemplos de implementação heap minimo utilizando exemplo Prof. Alonso CT234

def Sift(i, n):
    esq = (2*i)
    dir = (2*i)+1
    menor = i
    if (esq <= n and v[esq] < v[i]):
        menor = esq
    if (dir <= n and v[dir] < v[menor]):
        menor = dir
    if (menor != i):
        aux = v[i]
        v[i] = v[menor]
        v[menor] = aux
        Sift(menor, n)
```

```

def Build(v):
    n = len(v)-1
    x = int(n/2)
    for i in range(x,0,-1):
        Sift(i, n)

def Max():
    return v[len(v)-1]

def Min():
    return v[1]

def ExtractMin():
    size = len(v)-1
    if (size < 1):
        print ("heap underflow")
    else:
        min = v[1]
        v[1]= v[size]
        del v[size]
        size=len(v)-1
        #print (1,size)
        Sift(1,size)
        return min

def ExtractMax():
    size = len(v)-1
    if (size < 1):
        print ("heap underflow")
    else:
        max = v[size]
        del v[size]
        size=len(v)-1
        #print (1,size)
        Sift(1,size)
        return max

# Imprime a arvore
# exemplo do link
https://www.w3resource.com/python-exercises/heap-queue-algorithm/python-heapq-exercise-19.php
p
import math
from io import StringIO
def show_tree(tree, total_width=60, fill=' '):
    """Pretty-print a tree.
    total_width depends on your input size"""
    output = StringIO()

```

```

last_row = -1
y=[] + tree
del y[0]
for i, n in enumerate(y):
    if i:
        row = int(math.floor(math.log(i+1, 2)))
    else:
        row = 0
    if row != last_row:
        output.write('\n')
    columns = 2**row
    col_width = int(math.floor((total_width * 1.0) / columns))
    output.write(str(n).center(col_width, fill))
    last_row = row
print (output.getvalue())
print ('-' * total_width)
return

v = ["",4,1,3,2,16,9,10,14,8,7]
v[0] = "Lista"
print (v)
Build(v)
v[0]= "Heap"

print(v)
print("Max:",Max(), "Min:",Min())

show_tree(v)

print("Extract Max", ExtractMax())
print("Extract Min", ExtractMin())

v[0]= "Heap sem max e sem min"
print(v)
print("2o Max:",Max(), "2o Min:",Min())

show_tree(v)

```

## Resultado da execução

```

['Lista', 4, 1, 3, 2, 16, 9, 10, 14, 8, 7]
['Heap', 1, 2, 3, 4, 7, 9, 10, 14, 8, 16]
Max: 16 Min: 1

```

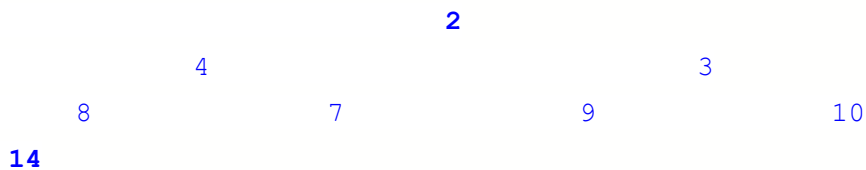


Extract Max 16

Extract Min 1

['Heap sem max e sem min', 2, 4, 3, 8, 7, 9, 10, 14]

2o Max: 14 2o Min: 2



```

print("2o Max:",Max(), "2o Min:",Min())
show_tree(v)

['Lista', 4, 1, 3, 2, 16, 9, 10, 14, 8, 7]
['Heap', 1, 2, 3, 4, 7, 9, 10, 14, 8, 16]
Max: 16 Min: 1

      1
     / \
    4   3
   / \ / \
  14 8 16 7 9 10

-----
Extract Max 16
Extract Min 1
['Heap sem max e sem min', 2, 4, 3, 8, 7, 9, 10, 14]
2o Max: 14 2o Min: 2

      2
     / \
    4   3
   / \ / \
  8  7 9 10
 14
  
```

Considere a seguinte lista:  $\{2, 6, 3, 4, 1, 1, 3, 3, -1, 3, 3\}$ . Usando um heap de mínimo, a montagem das 11 árvores (implícitas no vetor) é:

Pode realmente parecer que o *menor limitante e o errado* é  $\mathcal{O}(n \lg n)$ , pois cada inserção terá altura máxima de  $\lg n$  e há  $n$  números.  $\Leftarrow \odot \Rightarrow$

Segue o ajuste do código de heap mínimo para apresentar cada uma das 11 árvores (um instrução `show_tree` foi adicionada no início da função `sift`. O algoritmo continua sendo  $\mathcal{O}(n)$ ).

Fonte disponível no github:

[https://github.com/dasamerica/ct208/blob/master/aula\\_10jun\\_maiormenor\\_CA/ct208\\_heap\\_minimo\\_11\\_arvores.py](https://github.com/dasamerica/ct208/blob/master/aula_10jun_maiormenor_CA/ct208_heap_minimo_11_arvores.py)

```

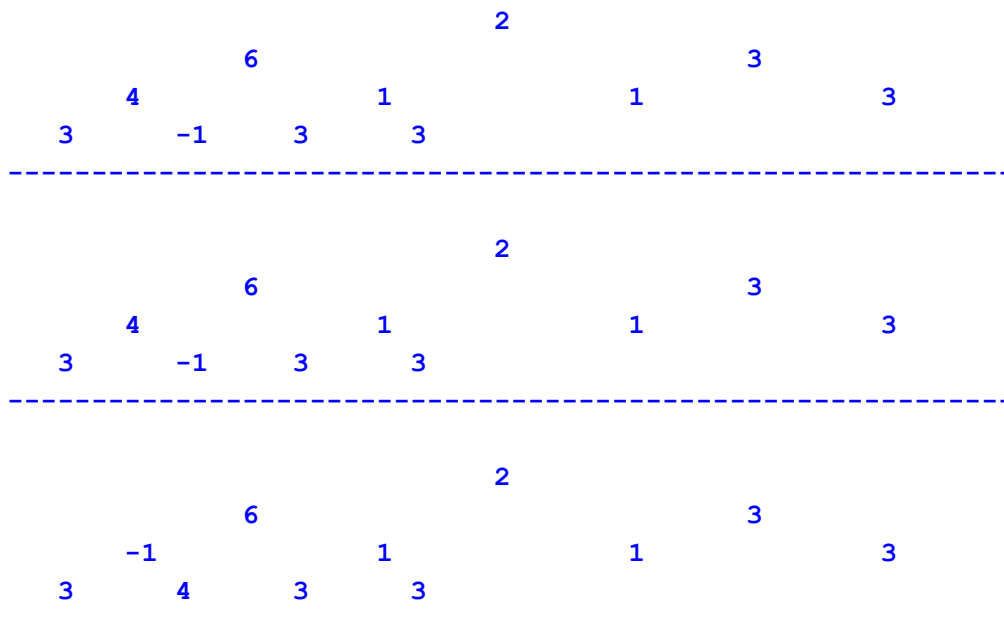
def Sift(i, n):
    show_tree(v)
    esq = (2*i)
    dir = (2*i)+1
    menor = i
    if (esq <= n and v[esq] < v[i]):
        menor = esq
    if (dir <= n and v[dir] < v[menor]):
        menor = dir
    if (menor != i):
        aux = v[i];
        v[i] = v[menor]
        v[menor] = aux
        Sift(menor, n)

v = ["", 2, 6, 3, 4, 1, 1, 3, 3, -1, 3, 3]
v[0] = "Lista"
print (v)
Build(v)

```

Resultados de cada árvore a cada processamento

```
['Lista', 2, 6, 3, 4, 1, 1, 3, 3, -1, 3, 3]
```



2  
6 3  
-1 1 1 3  
3 4 3 3

---

2  
6 1 3  
-1 1 3 3  
3 4 3 3

---

2  
6 1 3  
-1 1 3 3  
3 4 3 3

---

2  
-1 1 3  
6 3 3 3  
3 4 3 3

---

2  
-1 1 3  
3 3 3 3  
6 4 3 3

---

2  
-1 1 3  
3 3 3 3  
6 4 3 3

---

-1  
2 1 3  
3 3 3 3  
6 4 3 3

---

1 -1 1



### Questão 08

⇒ Faça experimentos para estabelecer o tempo de processamento do maior e menor. Faça outro para os dois menores. Para tanto, considere  $2 \leq n \leq 10000$  e uma amostra de tamanho 100 para cada um dos valores de  $n$ . Apresente em figuras como aquela da página 3 da aula de hoje. ⇐

Utilizou-se o mesmo código apresentado anteriormente para o heap do mínimo, adicionando uma função para gerar a amostra de 100 números entre 2 e 10000, e outra para imprimir o tempo de processamento. O mesmo programa realiza também o cálculo do maior e menor e os dois menores. O valor do menor está correto, porém como não estou utilizando heapsort, o maior elemento não é necessariamente o maior da lista, porém é a última folha no último nível de profundidade da árvore.

### Resultados

Tamanho da lista: 100

```
['Lista', 8091, 9081, 9819, 7785, 7172, 3071, 3010, 213, 5747, 5789, 9950,
1726, 4635, 4107, 9157, 934, 3616, 8649, 4528, 7403, 2998, 3450, 1895,
315, 5968, 2597, 5335, 1683, 9985, 9833, 7591, 9609, 9017, 6741, 879,
7790, 4950, 1129, 4564, 954, 8875, 1328, 6805, 3024, 4115, 6584, 3567,
6815, 85, 5872, 9447, 1161, 5914, 9516, 9600, 9941, 5342, 7327, 9278,
8715, 1403, 4228, 8018, 6070, 4032, 1459, 2513, 5265, 9513, 8192, 6980,
1395, 7231, 2741, 5015, 3202, 8581, 948, 6725, 3996, 4949, 7611, 4956,
8216, 3471, 8901, 1127, 1246, 2860, 7865, 5003, 4288, 4204, 4929, 3632,
7040, 8768, 496, 3933]
```

\*\*\*\* Performance extraindo o maior e o menor \*\*\*\*

```
['Heap', 85, 213, 315, 879, 954, 496, 1403, 934, 948, 1127, 1246, 1726,
1161, 1683, 3010, 1459, 3616, 1395, 1129, 3996, 1328, 2860, 1895, 3071,
5872, 2597, 5335, 4107, 7327, 8715, 4228, 4032, 2513, 5265, 6980, 7231,
2741, 3202, 4528, 4949, 4956, 3471, 2998, 3024, 4115, 4204, 3567, 6815,
3933, 5968, 9447, 4635, 5914, 9516, 9600, 9941, 5342, 9985, 9278, 9157,
9833, 7591, 8018, 6070, 9609, 9017, 9081, 6741, 9513, 8192, 7785, 7790,
8649, 4950, 5015, 5747, 8581, 4564, 6725, 7403, 5789, 7611, 8875, 8216,
7172, 8901, 6805, 9950, 3450, 7865, 5003, 4288, 6584, 4929, 3632, 7040,
8768, 9819, 8091]
```

**Max: 8091 Min: 85**

\*\*\*\* Performance extraindo o segundo maior e o segundo menor \*\*\*\*

Extract Max 8091

Extract Min 85

```
['Heap sem max e sem min', 213, 879, 315, 934, 954, 496, 1403, 1459, 948,
1127, 1246, 1726, 1161, 1683, 3010, 2513, 3616, 1395, 1129, 3996, 1328,
2860, 1895, 3071, 5872, 2597, 5335, 4107, 7327, 8715, 4228, 4032, 9017,
5265, 6980, 7231, 2741, 3202, 4528, 4949, 4956, 3471, 2998, 3024, 4115,
4204, 3567, 6815, 3933, 5968, 9447, 4635, 5914, 9516, 9600, 9941, 5342,
9985, 9278, 9157, 9833, 7591, 8018, 6070, 9609, 9819, 9081, 6741, 9513,
8192, 7785, 7790, 8649, 4950, 5015, 5747, 8581, 4564, 6725, 7403, 5789,
7611, 8875, 8216, 7172, 8901, 6805, 9950, 3450, 7865, 5003, 4288, 6584,
4929, 3632, 7040, 8768]
```

**2o Max: 8768 2o Min: 213**

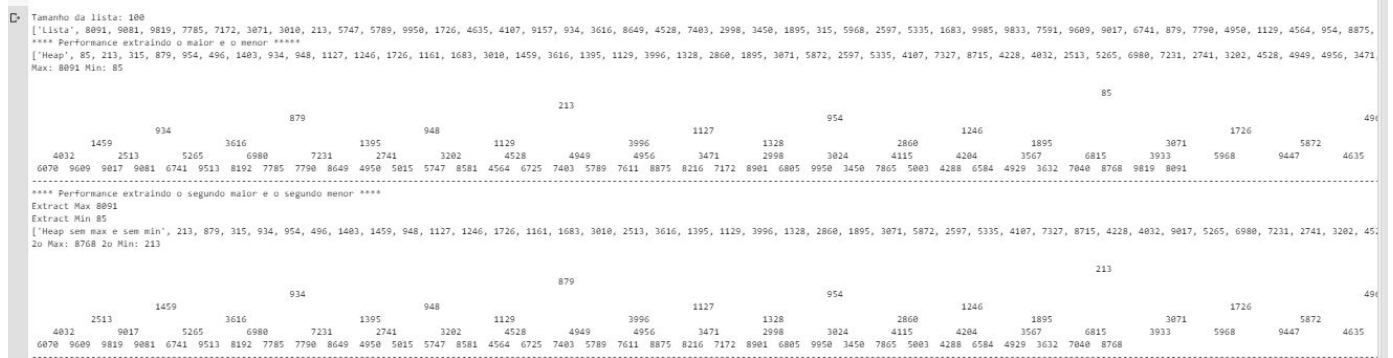
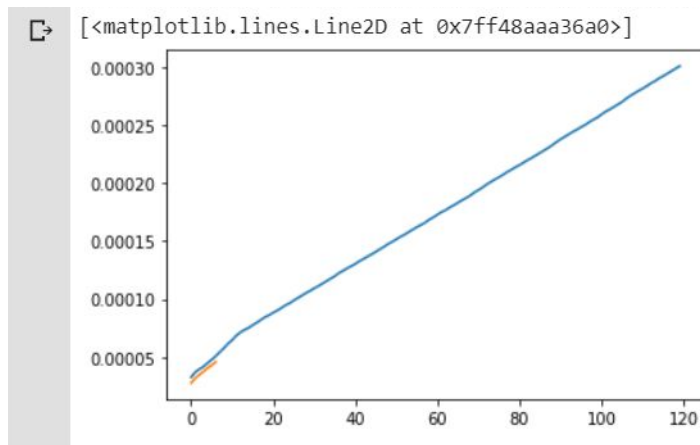


Gráfico demonstrando performance linear do programa de heap.

Linha azul execução do heap.

Linha abóbora execução do max e min.



Fonte disponível no Github:

[https://github.com/dasamerica/ct208/blob/master/aula\\_10jun\\_maiormenor\\_CA/ct208\\_heap\\_minimo\\_amostra100.py](https://github.com/dasamerica/ct208/blob/master/aula_10jun_maiormenor_CA/ct208_heap_minimo_amostra100.py)

**Código Python**



```

#@title
# Daniela America da Silva
# Exemplos de implementação heap mínimo utilizando exemplo Prof. Alonso CT234

import pandas as pd
import numpy as np
import time, datetime
import matplotlib.pyplot as plt

def Sift(i, n):
    counters.append(counter)
    ttime=str(datetime.timedelta(seconds=time.time() - t0))
    times.append(time.time() - t0)

    esq = (2*i)
    dir = (2*i)+1
    menor = i
    if (esq <= n and v[esq] < v[i]):
        menor = esq
    if (dir <= n and v[dir] < v[menor]):
        menor = dir
    if (menor != i):
        aux = v[i];
        v[i] = v[menor]
        v[menor] = aux
        Sift(menor, n)

def Build(v):
    n = len(v)-1
    x = int(n/2)
    for i in range(x,0,-1):
        Sift(i, n)

def Max():
    return v[len(v)-1]

def Min():
    return v[1]

def ExtractMin():
    size = len(v)-1
    if (size < 1):
        print ("heap underflow")
    else:
        min = v[1]
        v[1]= v[size]
        del v[size]

```

```

        size=len(v)-1
        #print (l,size)
        Sift(1,size)
        return min

def ExtractMax():
    size = len(v)-1
    if (size < 1):
        print ("heap underflow")
    else:
        max = v[size]
        del v[size]
        size=len(v)-1
        #print (l,size)
        Sift(1,size)
        return max

# Imprime a arvore
# exemplo do link
https://www.w3resource.com/python-exercises/heap-queue-algorithm/python-heapq-exercise-19.ph

p


import math
from io import StringIO
def show_tree(tree, total_width=400, fill=' '):
    """Pretty-print a tree.
    total_width depends on your input size"""
    output = StringIO()
    last_row = -1
    y=[] + tree
    del y[0]
    for i, n in enumerate(y):
        if i:
            row = int(math.floor(math.log(i+1, 2)))
        else:
            row = 0
        if row != last_row:
            output.write('\n')
        columns = 2**row
        col_width = int(math.floor((total_width * 1.0) / columns))
        output.write(str(n).center(col_width, fill))
        last_row = row
    print (output.getvalue())
    print ('-' * total_width)
    return

# Gerar 100 numeros aleatórios de 2 a 10000
import random

```

```

v = random.sample(range(2,10000), 100)

v[0] = "Lista"
print ("Tamanho da lista:", len(v))
print (v)

print("**** Performance extraindo o maior e o menor ****")
times=[]
counters=[]
counter=0
t0=time.time()

Build(v)
v[0]= "Heap"
print(v)
print("Max:",Max(), "Min:",Min())
show_tree(v)

plt.plot(times)

print("**** Performance extraindo o segundo maior e o segundo menor ****")

times=[]
counters=[]
counter=0
t0=time.time()

print("Extract Max", ExtractMax())

times=[]
counters=[]
counter=0
t0=time.time()
print("Extract Min", ExtractMin())

v[0]= "Heap sem max e sem min"
print(v)
print("2o Max:",Max(), "2o Min:",Min())
show_tree(v)

plt.plot(times)

```

## 4- Autômato celular

### Questão 09

⇒ Repita a Figura 6 usando a regra 90, com uma única célula central.⇐

Exemplo de código coletado a partir de

[https://rosettacode.org/wiki/Elementary\\_cellular\\_automaton#Python](https://rosettacode.org/wiki/Elementary_cellular_automaton#Python)

Ajustado os parametros de execução para apresentar 63 linhas, uma única célula central e aplicar a regra 90 apenas.

```
lines, start, rules = 63, '1', (90,)
```

Fonte disponível no Github:

[https://github.com/dasamerica/ct208/blob/master/aula\\_10jun\\_maiormenor\\_CA/ct208\\_CA\\_rule90\\_infinite.py](https://github.com/dasamerica/ct208/blob/master/aula_10jun_maiormenor_CA/ct208_CA_rule90_infinite.py)

### Código python

```
#https://rosettacode.org/wiki/Elementary_cellular_automaton#Python
```

```
def _notcell(c):
    return '0' if c == '1' else '1'

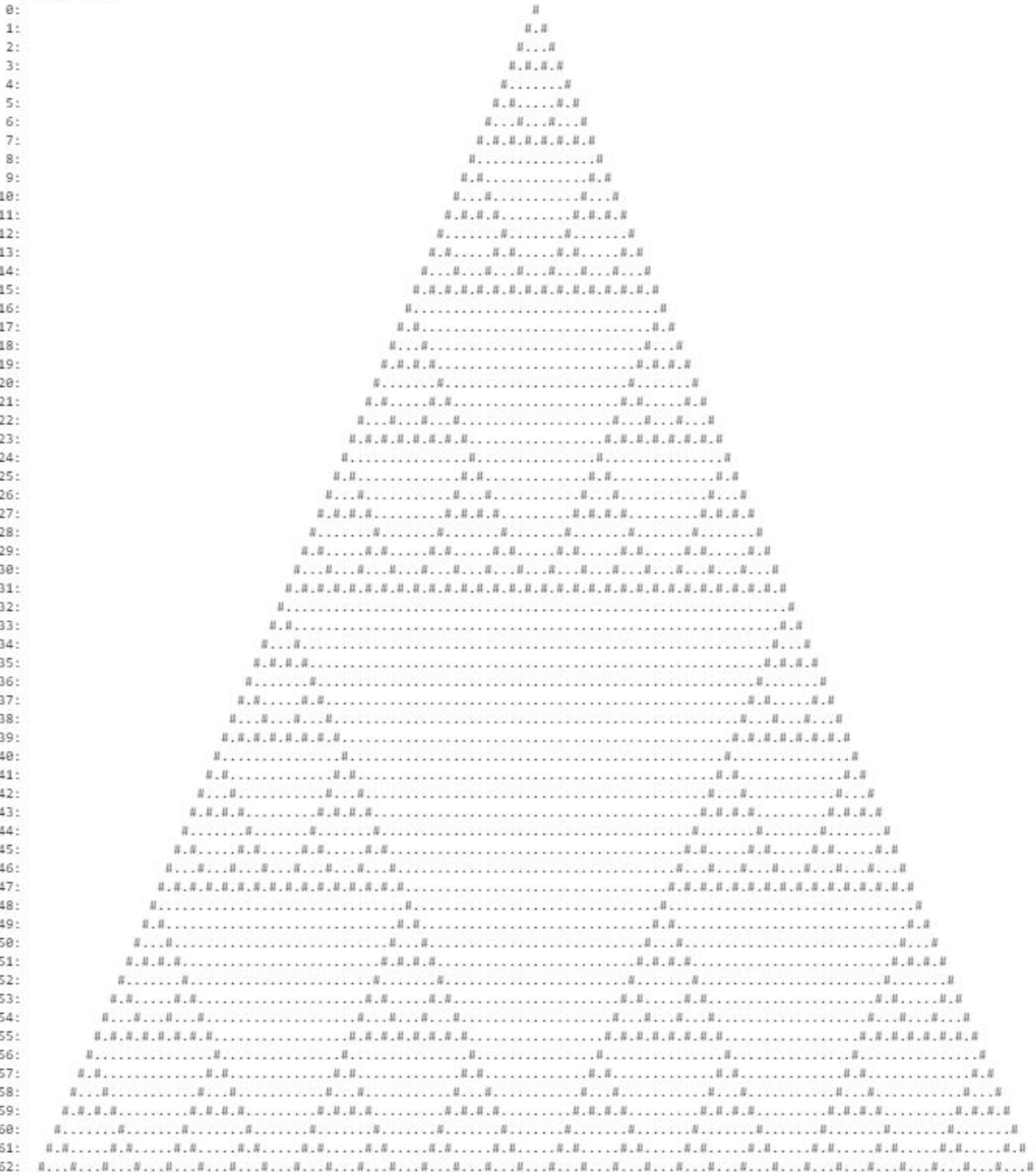
def eca_infinite(cells, rule):
    lencells = len(cells)
    rulebits = '{0:08b}'.format(rule)
    neighbours2next = {'{0:03b}'.format(n):rulebits[::-1][n] for n in range(8)}
    c = cells
    while True:
        yield c
        c = _notcell(c[0])*2 + c + _notcell(c[-1])*2    # Extend and pad the ends

        c = ''.join(neighbours2next[c[i-1:i+2]] for i in range(1,len(c) - 1))
        #yield c[1:-1]

if __name__ == '__main__':
    lines, start, rules = 63, '1', (90,)
    zipped = [range(lines)] + [eca_infinite(start, rule) for rule in rules]
    print('\n Rules: %r' % (rules,))
    for data in zip(*zipped):
        i = data[0]
        cells = ['%s%s%s' % (' '*(lines - i), c, ' '*(lines - i)) for c in data[1:]]
        print('%2i: %s' % (i, ' '.join(cells).replace('0', '.').replace('1', '#')))
```

00000000 (700)

```
0:
1:
2:
3:
4:
5:
6:
7:
8:
9:
10:
11:
12:
13:
14:
15:
16:
17:
18:
19:
20:
21:
22:
23:
24:
25:
26:
27:
28:
29:
30:
31:
32:
33:
34:
35:
36:
37:
38:
39:
40:
41:
42:
43:
44:
45:
46:
47:
48:
49:
50:
51:
52:
53:
54:
55:
56:
57:
58:
59:
60:
61:
62:
```



### Questão 10

É possível fazer uma formulação matemática simples relacionando os estados de em AC e o tempo. Antes, entretanto, vamos adotar as seguintes notações:  $c_{k,j}$  é o estado da célula  $j$  no tempo  $k$ , com alfabeto  $\Sigma = \{0, 1\}$ , e  $c_0$  dado. Temos então:

$$c_{k,j} = \begin{cases} (c_{k-1,1} + c_{k-1,2}) \mod 2; & j = 1; \\ (c_{k-1,j-1} + c_{k-1,j} + c_{k-1,j+1}) \mod 2, & j = 2, 3, \dots, n-1; \\ (c_{k-1,n-1} + c_{k-1,n}) \mod 2, & j = n. \end{cases}$$

Por exemplo, considere que  $n = 5$ , com o autômato anterior e  $c_0 = \{0, 0, 1, 1, 0\}$ . Para o instante  $t = 1$  os cálculos são:

$$\begin{cases} c_{1,1} = (c_{0,0} + c_{0,1}) \mod 2 = 0 + 0 \mod 2 = 0; \\ c_{1,2} = (c_{0,1} + c_{0,2} + c_{0,3}) \mod 2 = 0 + 0 + 1 \mod 2 = 1; \\ c_{1,3} = (c_{0,2} + c_{0,3} + c_{0,4}) \mod 2 = 0 + 1 + 1 \mod 2 = 0; \\ c_{1,4} = (c_{0,3} + c_{0,4} + c_{0,5}) \mod 2 = 1 + 1 + 0 \mod 2 = 0; \\ c_{1,5} = (c_{0,4} + c_{0,5}) \mod 2 = 1 + 0 \mod 2 = 1. \end{cases}$$

Enquanto que para o tempo  $t = 2$ :

$$\begin{cases} c_{2,1} = (c_{1,0} + c_{1,1}) \mod 2 = 0 + 1 \mod 2 = 1; \\ c_{2,2} = (c_{1,1} + c_{1,2} + c_{1,3}) \mod 2 = 0 + 1 + 0 \mod 2 = 1; \\ c_{2,3} = (c_{1,2} + c_{1,3} + c_{1,4}) \mod 2 = 1 + 0 + 0 \mod 2 = 1; \\ c_{2,4} = (c_{1,3} + c_{1,4} + c_{1,5}) \mod 2 = 0 + 0 + 1 \mod 2 = 1; \\ c_{2,5} = (c_{1,4} + c_{1,5}) \mod 2 = 0 + 1 \mod 2 = 1. \end{cases}$$

⇒ Apresente a formulação matricial para o caso acima. Utilizando sua formulação matricial, sob que condições não se tem um jardim do Éden? ⇐

Conforme apresentado via programa não é possível através da formulação matricial proposta, a geração de um jardim do Éden pois a partir de  $t_4$  são gerados os mesmos resultados.

Fonte disponível no Github:

[https://github.com/dasamerica/ct208/blob/master/aula\\_10jun\\_maiormenor\\_CA/ct208\\_CA\\_formulacao\\_math.py](https://github.com/dasamerica/ct208/blob/master/aula_10jun_maiormenor_CA/ct208_CA_formulacao_math.py)

```

+ Code + Text

#start = '{0:05b}'.format(6)
start= [0,0,1,1,0]
v = [] + start
print ("t",0,start)
next=[0,0,0,0,0]
for i in range(10):
    for n in range(5):
        if (n==0):
            next[n] = ((int(v[n]) + int(v[n+1]))%2)
        else:
            if (n==4):
                next[n] = ((int(v[n-1]) + int(v[n]))%2)
            else:
                next[n] = ((int(v[n-1]) + int(v[n]) + int(v[n+1]))%2)

    print("t",i+1, next)
    v=[] + next

t 0 [0, 0, 1, 1, 0]
t 1 [0, 1, 0, 0, 1]
t 2 [1, 1, 1, 1, 1]
t 3 [0, 1, 1, 1, 0]
t 4 [1, 0, 1, 0, 1]
t 5 [1, 0, 1, 0, 1]
t 6 [1, 0, 1, 0, 1]
t 7 [1, 0, 1, 0, 1]
t 8 [1, 0, 1, 0, 1]
t 9 [1, 0, 1, 0, 1]
t 10 [1, 0, 1, 0, 1]

```

## Anexos

### Comparações entre funções

- A partir da notação  $O$ , é possível estabelecer uma hierarquia entre as funções:

Constante	$O(1)$
Logarítmica	$O(\log n)$
Linear	$O(n)$
$n \cdot \log n$	$O(n \cdot \log n)$
Quadrática	$O(n^2)$
Cúbica	$O(n^3)$
Polinomial	$O(n^k)$ , com $k \geq 4$
Exponencial	$O(k^n)$ , com $k > 1$

Maior  
ordem

Evidentemente, as funções lineares, quadráticas e cúbicas também são polinomiais ...

Figura 3 - Comparações entre funções (Capítulo 01 CE-234, página 34)

### Notação $O$ na análise de algoritmos

- Em comandos consecutivos, somam-se os tempos:  

```
for (int x=1; x <= n; x++)  
    <operação primitiva qualquer>;
```

 }  $O(n)$   

```
for (int x=1; x <= n; x++)  
    for (int y=1; y <= n; y++)  
        <operação primitiva qualquer>;
```

 }  $O(n^2)$
- Tempo total:  $O(n) + O(n^2) = O(n^2)$
- E se o laço `for` mais interno começasse com `y=x`?
- Qual o tempo total gasto pelo laço abaixo?  

```
for (int x=1, int y=1; y <= n; x++) {  
    <operação primitiva qualquer>;  
    if (x==n) { y++; x=1; }  
}
```

 }  $O(n^2)$

Figura 4 - A Notação  $O$  na análise dos algoritmos (Capítulo 01 CE-234, página 31)

### Algumas dicas sobre a notação $O$

- No uso da notação  $O$ , consideramos apenas valores suficientemente grandes de  $n$ , ou seja,  $n \rightarrow \infty$
- Se  $p(n)$  é um polinômio de grau  $k$ , então  $p(n)$  é  $O(n^k)$ 
  - Pode-se descartar seus termos de menor ordem, inclusive as constantes.
- Convém utilizar a menor ordem:
  - " $2n$  é  $O(n)$ " é preferível a " $2n$  é  $O(n^2)$ "
  - " $3n + 5$  é  $O(n)$ " é preferível a " $3n + 5$  é  $O(3n)$ "

Figura 5 - A Notação  $O$  e a menor ordem (Capítulo 01 CE-234, página 26).



## A notação $O$ e a taxa de crescimento

- A notação  $O$  fornece um *limite superior* para a taxa de crescimento de uma determinada função.
- A afirmação " $f(n)$  é  $O(g(n))$ " significa que a taxa de crescimento de  $f(n)$  não é maior que a de  $g(n)$ .
- A notação  $O$  permite ordenar as funções de acordo com as suas correspondentes taxas de crescimento.

	$f(n)$ é $O(g(n))$ ?	$g(n)$ é $O(f(n))$ ?
Se $g(n)$ cresce mais que $f(n)$ :	Sim	Não
Se $f(n)$ cresce mais que $g(n)$ :	Não	Sim
Se $f(n)$ e $g(n)$ têm a mesma taxa:	Sim	Sim

Figura 6 - A Notação  $O$  e a taxa de crescimento (Capítulo 01 CE-234, página 24)

## Limites inferiores

- Enquanto a notação  $O$  fornece limites superiores para o crescimento das funções, também há outras notações que oferecem mais informações interessantes.
- Seja  $\Omega(g(n))$  o conjunto de funções  $f(n)$  para as quais existem constantes positivas  $c$  e  $n_0$  tais que  $f(n) \geq c \cdot g(n)$ ,  $\forall n \geq n_0$ .
- A notação  $\Omega$  fornece um limite inferior para o crescimento das funções.

Figura 7 - Limites inferiores (Capítulo 01 CE-234, página 36)

## Na prática...

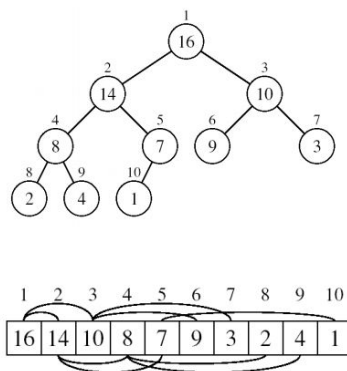
- Na notação  $\Omega$ , convém utilizar a maior função possível:

É correto dizer que  $f(n) = 3n^2 + 10$  é  $\Omega(1)$ ,  
mas representa pouca coisa sobre  $f(n)$ ...

- Analogamente,  $f(n) = \Omega(g(n))$  significa que  
 $f(n) \in \Omega(g(n))$ .

Figura 8 - Limites inferiores (Capítulo 01 CE-234, página 38)

## Representação de *heap* com vetor



- Armazenamento de um *heap* com  $n$  elementos em um vetor  $v$ :
  - A raiz está em  $v[1]$
  - O filho esquerdo de  $v[i]$  é  $v[2i]$
  - O filho direito de  $v[i]$  é  $v[2i+1]$
- O pai de  $v[i]$  será  $v[\lfloor i/2 \rfloor]$ .
- Os elementos do subvetor  $v[\lfloor n/2 \rfloor + 1 .. n]$  são as folhas.
- É fácil constatar que a altura do *heap* é  $\Theta(\log n)$ .

Figura 9 - Representação do heap (Capítulo 06 CE-234, página 4)

## Complexidade de tempo de *Build*

Tempo de pior caso, correspondente a uma árvore completa com  $n$  nós e altura  $h$ :

$$T(n) = \sum_{i=0}^h 2^i (h-i)$$

Multiplicando o numerador e o denominador por  $2^h$ :

$$= \sum_{i=0}^h \frac{h-i}{2^{h-i}} 2^h$$

Troca de variáveis ( $k = h - i$ ):

$$= 2^h \sum_{k=0}^h \frac{k}{2^k}$$

Sabemos que  $h = \lg n$  e que essa somatória é menor que a correspondente somatória até  $\infty$ :

$$\leq n \sum_{k=0}^{\infty} \frac{k}{2^k}$$

Sabemos também que essa somatória é menor que 2:

$$= \Theta(n)$$

Figura 10 - Tempo linear do Build (Capítulo 06 CE-234, página 09)

## Sumário

- A tabela abaixo indica as complexidades de tempo das operações de uma fila de prioridades implementada com um *heap*:

<u>Operação</u>	<u>Tempo</u>
<i>Build</i>	Linear
<i>Max</i> (ou <i>Min</i> )	Constante
<i>ExtractMax</i> (ou <i>ExtractMin</i> )	Logarítmico
<i>Modify</i>	Logarítmico
<i>Insert</i>	Logarítmico

Figura 11 - Resumo de complexidade de tempo para as operações do heap (Capítulo 06 CE-234, página 16)

## Referências

1. Knuth, D, Stanford, Stanford Lecture - Don Knuth: The Analysis of Algorithms (2015, recreating 1969), 2015, acessado de <https://www.youtube.com/watch?v=vkUNH9r6UCI> em 08 Junho 2020
2. Knuth, Donald Ervin. *The art of computer programming*. Vol. 3. Pearson Education, 1997, páginas de 96-104
3. Soma, N.Y., ITA, Matemática Computacional, CT208 Notas da Aula de 10 de Junho 2020 - Maior/Menor elemento e Autômatos Celulares, 2020
4. Sanches, C.A.A, ITA, Estruturas de Dados, Análise de Algoritmos e Complexidade Estrutural, 2020, acessado de <http://www.comp.ita.br/~alonso/ensino/CT234/CT234-Cap01.pdf> em 16 Junho 2020.

5. Sanches, C.A.A, ITA, Estruturas de Dados, Análise de Algoritmos e Complexidade Estrutural, 2020, acessado de <http://www.comp.ita.br/~alonso/ensino/CT234/CT234-Cap06.pdf> em 16 Junho 2020.
6. Araujo,A, Celani, G, Cadernos ProArq29, Interpretações Arquitetônicas dos Autômatos Celulares: conceitos e aplicações recentes, 2017, acessado de <http://cadernos.proarq.fau.ufrj.br/public/docs/Proarq29%20ART%2008.pdf> em 16 Junho 2020.
7. Saraswat, A, Kapoor, N, Cornell University, CS2110 Lecture Median Finding Algorithm, acessado de <http://www.cs.cornell.edu/courses/cs2110/2009su/Lectures/examples/MedianFinding.pdf> em 20 Junho 2020
8. Wikipedia, 2020, Média das médias, acessado de [https://en.wikipedia.org/wiki/Median\\_of\\_medians](https://en.wikipedia.org/wiki/Median_of_medians) em 20 Junho 2020.
9. Cornell, Média das médias, acessado de <http://www.cs.cornell.edu/courses/cs2110/2009su/Lectures/examples/MedianFinding.pdf>. Em 22 Junho 2020