INSTITUTO INFNET

ESCOLA SUPERIOR DE TECNOLOGIA

GRADUAÇÃO EM ENGENHARIA DE SOFTWARE

Projeto de Bloco: Ciência da Computação

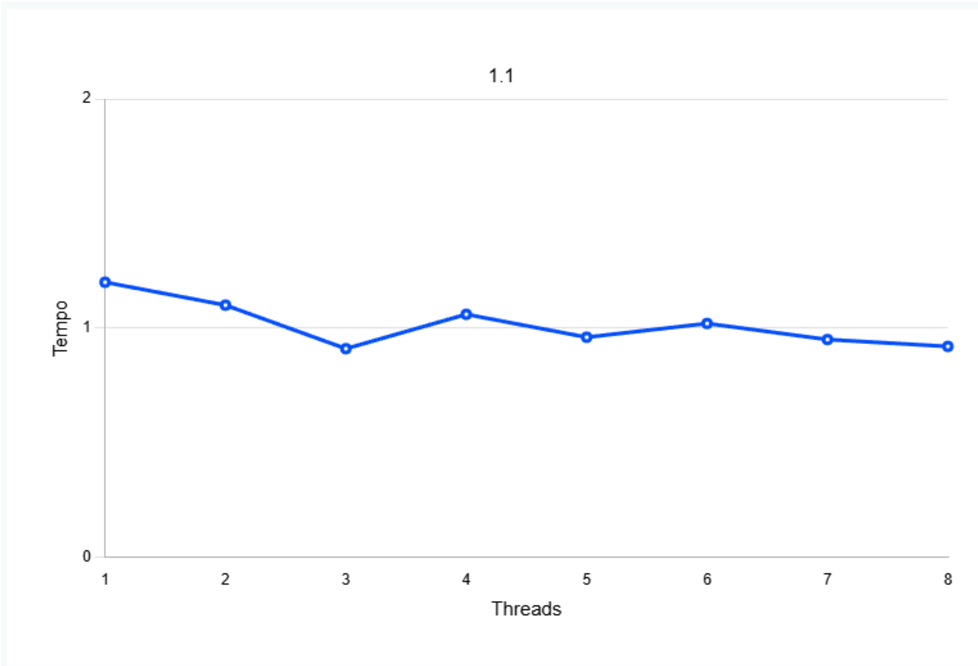
TP2

Daniel Gomes Lipkin

2 de fev. de 2025

1.1

```
============ RESTART: C
1.20 segundos - 1 threads
1.10 segundos - 2 threads
0.91 segundos - 3 threads
1.06 segundos - 4 threads
0.96 segundos - 5 threads
1.02 segundos - 6 threads
0.95 segundos - 7 threads
0.92 segundos - 8 threads
```



O algoritmo particiona a lista de URLs pela quantidade de threads e usa o asyncio para executar eles em um a thread assincronamente.
Parece que a complexidade de tempo é O(1). Mesmo testando com quantidades variadas de URLs.

1.2

```
============ RESTART: C:\Users\donke\Documents\Infnet\cienciacomp.py ===========
0.01 segundos - linear
0.87 segundos - 2 threads
0.86 segundos - 3 threads
0.84 segundos - 4 threads
0.85 segundos - 5 threads

============ RESTART: C:\Users\donke\Documents\Infnet\cienciacomp.py ===========
0.96 segundos - linear
1.56 segundos - 2 threads
1.55 segundos - 3 threads
1.55 segundos - 4 threads
1.54 segundos - 5 threads
```

```
0.89 segundos - linear
0.93 segundos - 2 threads
0.92 segundos - 3 threads
0.93 segundos - 4 threads
0.92 segundos - 5 threads
```

A versão paralela simplesmente executa a versão linear em partições da lista entrada usando o ThreadPoolExecutor.
Parece que não faz diferença alguma por mais que divida a tarefa em múltiplas threads.

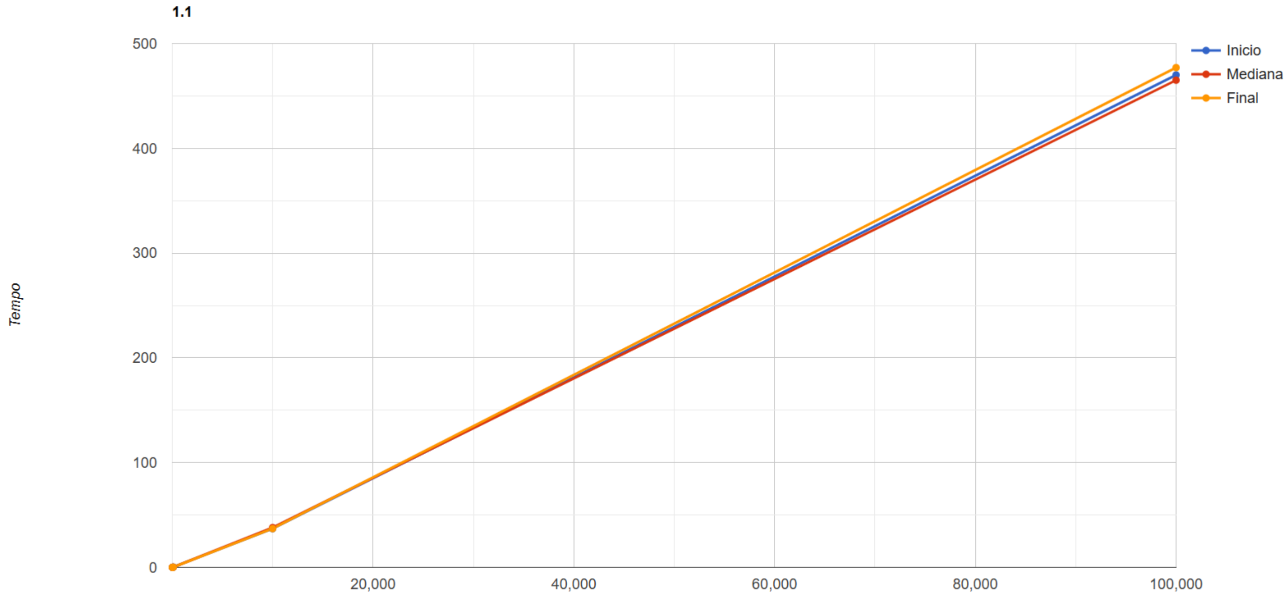
1.3

```
============ RESTART: C
3.22 segundos - 1 threads
3.09 segundos - 2 threads
3.08 segundos - 3 threads
3.08 segundos - 4 threads
3.10 segundos - 5 threads
3.06 segundos - 6 threads
3.06 segundos - 7 threads
3.07 segundos - 8 threads
```

O algoritmo pega todos arquivos de imagem da pasta de entrada e particiona a lista de imagens pela quantidade de threads e usa o asyncio para processá-las em uma thread. O(1) de tempo.

2.1

```
============ RESTART: C:\Users\donke\Documents
Inicio do array - 0.030825001886114478 segundos
Mediana do array - 0.02680000034160912 segundos
Final do array - 0.026749999960884452 segundos
Inicio do array - 0.3031250089406967 segundos
Mediana do array - 0.23202500597108155 segundos
Final do array - 0.2215249987784773 segundos
Inicio do array - 37.37610000825953 segundos
Mediana do array - 38.9487999927951 segundos
Final do array - 37.59909998916555 segundos
Inicio do array - 470.22892501263414 segundos
Mediana do array - 465.4085750080412 segundos
Final do array - 477.50197497953195 segundos
```



Dado uma lista e um pivo, a lista é dividida em duas sub-listas, uma com valores menor que o pivo e outra com o valor maior, depois para cada sub-lista é executado recursivamente o algoritimo com um pivô aleatório dentro da sub-lista. O tempo de O(n*log(n)) coincide com o grafico.
Foram testados n = 10,100,10000 e 100000. Parece que o pivô na posição final tem o melhor desempenho em média. Tempo multiplicado por 1000.

2.2

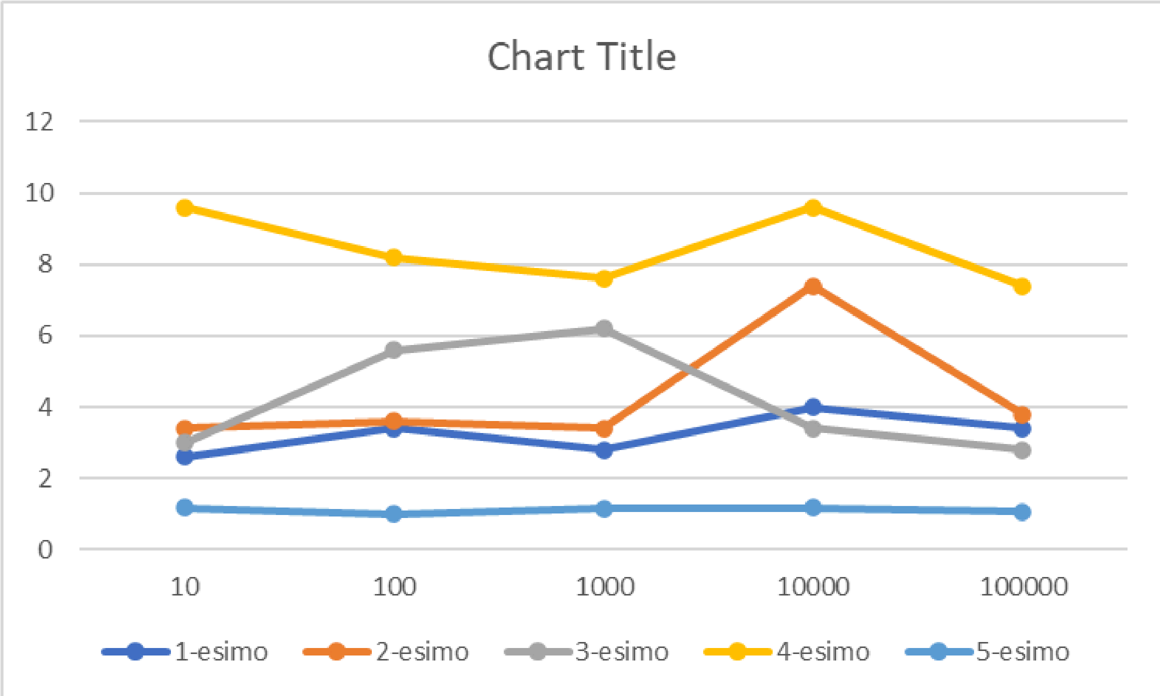
```
#Exercicio 2.2
class Student:
   def __init__(s, nome, nota):
      s.nome = nome
      s.nota = nota
   def __str__(s):
      return s.nome + " - " + str(s.nota)

def quicksortStudent(arr):
         if len(arr) < 2:
                  return arr
         pivot = random.randint(0, len(arr)-1)
         biggie = []
         smalls = []
         eq = []
         nota_t = arr[pivot].nota
         for x in arr:
                  if x.nota > nota_t:
                           biggie.append(x)
                  elif x.nota < nota_t:
                           smalls.append(x)
                  else:
                     eq.append(x)
         return quicksortStudent( smalls ) + eq + quicksortStudent( biggie )
```

```
Aluno76 - 0.0
Aluno25 - 0.1
Aluno49 - 0.1
Aluno9 - 0.3
Aluno70 - 0.3
Aluno60 - 0.4
Aluno6 - 0.5
Aluno39 - 0.7
Aluno63 - 0.8
Aluno11 - 0.9
Aluno0 - 1.0
Aluno30 - 1.4
Aluno57 - 1.4
Aluno5 - 1.6
Aluno2 - 1.8
Aluno36 - 1.8
Aluno66 - 1.8
Aluno29 - 2.1
Aluno47 - 2.4
Aluno40 - 2.5
Aluno53 - 2.6
```

A unica diferença aqui é que o quicksort foi ajustado para acomodar a classe Aluno.

2.3

```
1-esimo menor elemento - 3.399909473955631e-07 segundos - 10 elementos
2-esimo menor elemento - 4.00003045797348e-07 segundos - 10 elementos
3-esimo menor elemento - 3.799912519752979e-07 segundos - 10 elementos
4-esimo menor elemento - 3.2000243663787844e-07 segundos - 10 elementos
5-esimo menor elemento - 2.799904905259609e-07 segundos - 10 elementos
1-esimo menor elemento - 3.800028935074806e-07 segundos - 100 elementos
2-esimo menor elemento - 3.6000274121761323e-07 segundos - 100 elementos
3-esimo menor elemento - 3.199907951056957e-07 segundos - 100 elementos
4-esimo menor elemento - 2.800021320581436e-07 segundos - 100 elementos
5-esimo menor elemento - 3.00002284348011e-07 segundos - 100 elementos
1-esimo menor elemento - 3.00002284348011e-07 segundos - 1000 elementos
2-esimo menor elemento - 4.2000319808721544e-07 segundos - 1000 elementos
3-esimo menor elemento - 3.400025889277458e-07 segundos - 1000 elementos
4-esimo menor elemento - 2.600019797682762e-07 segundos - 1000 elementos
5-esimo menor elemento - 3.400025889277458e-07 segundos - 1000 elementos
1-esimo menor elemento - 7.400056347250939e-07 segundos - 10000 elementos
2-esimo menor elemento - 7.400056347250939e-07 segundos - 10000 elementos
3-esimo menor elemento - 7.199938409030437e-07 segundos - 10000 elementos
4-esimo menor elemento - 1.079996582120657e-06 segundos - 10000 elementos
5-esimo menor elemento - 1.3199984095990659e-06 segundos - 10000 elementos
1-esimo menor elemento - 1.1399853974580764e-06 segundos - 100000 elementos
2-esimo menor elemento - 1.300009898841381e-06 segundos - 100000 elementos
3-esimo menor elemento - 1.0800082236528397e-06 segundos - 100000 elementos
4-esimo menor elemento - 1.179997343569994e-06 segundos - 100000 elementos
5-esimo menor elemento - 1.0999850928783416e-06 segundos - 100000 elementos
```



Não há um padrão discernível que possa se observar dos resultados. Tempo multiplicado por 1000 no grafo (eixo-Y). A complexidade de espaço será O(log n) nos maiores dos casos por particionar a entrada de elementos em entradas menores para as chamadas recursivas.

O quickselect faz a mesma coisa que o quicksort, com a divisão da lista sendo feito com a função partition em torno do valor K (n-esimo elemento). Se o pivô for igual a K (indice) da lista então o valor é retornado.

```
#Exercicio 2.3
def partition(arr, l, r, pivot):
    pivot_v = arr[pivot]
    arr[pivot], arr[r] = arr[r], arr[pivot]
    l_i = l #valor R futuro
    for i in range(l, r):
        if arr[i] < pivot_v:
            arr[l_i], arr[i] = arr[i], arr[l_i]
            l_i += 1
    arr[r], arr[l_i] = arr[l_i], arr[r] #agora R=L e R=R futuro
    return l_i
```

2.4

```
Mediana - 4
Mediana - 56
Mediana - 7500
Mediana - 50000
```

Foram colocadas ranges com os valores 9, 112, 15000 e 100000 respectivamente.

3.1

```
2 elementos
10
10, 5
5


4 elementos
10
20, 10
20, 10, 5
20, 10, 5, 10
20, 5, 10
5, 10
10


6 elementos
10
20, 10
30, 20, 10
30, 20, 10, 5
30, 20, 10, 5, 10
30, 20, 10, 5, 10, 15
30, 20, 5, 10, 15
30, 5, 10, 15
5, 10, 15
10, 15
15
```

Nos testes são adicionados n/2 elementos em cada lado da lista e depois removidos por valor.
O tempo da implementação sera O(1) em ambas inserções pois a estrutura de dado supõe que não precisam percorrer a lista, e a remoção sera O(n) pois potencialmente percorre toda a lista até achar o item com o valor para remover.

3.2

```
4 elementos
10
20 -> 10
20 -> 10 -> 5
20 -> 10 -> 5 -> 10
10 -> 5 -> 10
10 -> 10
10 -> 10
10 <- 10

6 elementos
10
20 -> 10
30 -> 20 -> 10
30 -> 20 -> 10 -> 5
30 -> 20 -> 10 -> 5 -> 10
30 -> 20 -> 10 -> 5 -> 10 -> 15
20 -> 10 -> 5 -> 10 -> 15
20 -> 5 -> 10 -> 15
20 -> 5 -> 15
20 -> 5 -> 15
15 <- 5 <- 20

8 elementos
10
20 -> 10
30 -> 20 -> 10
40 -> 30 -> 20 -> 10
40 -> 30 -> 20 -> 10 -> 5
40 -> 30 -> 20 -> 10 -> 5 -> 10
40 -> 30 -> 20 -> 10 -> 5 -> 10 -> 15
40 -> 30 -> 20 -> 10 -> 5 -> 10 -> 15 -> 20
30 -> 20 -> 10 -> 5 -> 10 -> 15 -> 20
30 -> 10 -> 5 -> 10 -> 15 -> 20
30 -> 10 -> 10 -> 15 -> 20
30 -> 10 -> 10 -> 20
30 -> 10 -> 10 -> 20
20 <- 10 <- 10 <- 30
```

Os elementos são adicionados como anteriormente e até metade dos elementos são removidos em base na posição, com a lista invertida no final da saida.
Remover em um index especifico nessa estrutura de dados exige percorrer a lista com um iterador aditivo para corresponder a posição especificada pois não é uma lista que funciona a base de ponteiros numerados e sim referencias. Potencialmente chegando a O(n)

3.3

```
2 elementos
10, 5
Procurando 5: 1
5, 10

4 elementos
20, 10, 5, 10
Procurando 5: 2
Procurando 10: 1
10, 5, 10, 20

6 elementos
30, 20, 10, 5, 10, 15
Procurando 5: 3
Procurando 10: 2
Procurando 15: 5
15, 10, 5, 10, 20, 30
```

Denovo a mesma coisa com a inserção. Buscamos somente metade dos elementos, com um tempo potencial de O(n) pois percorre a todos elementos da lista.

3.4

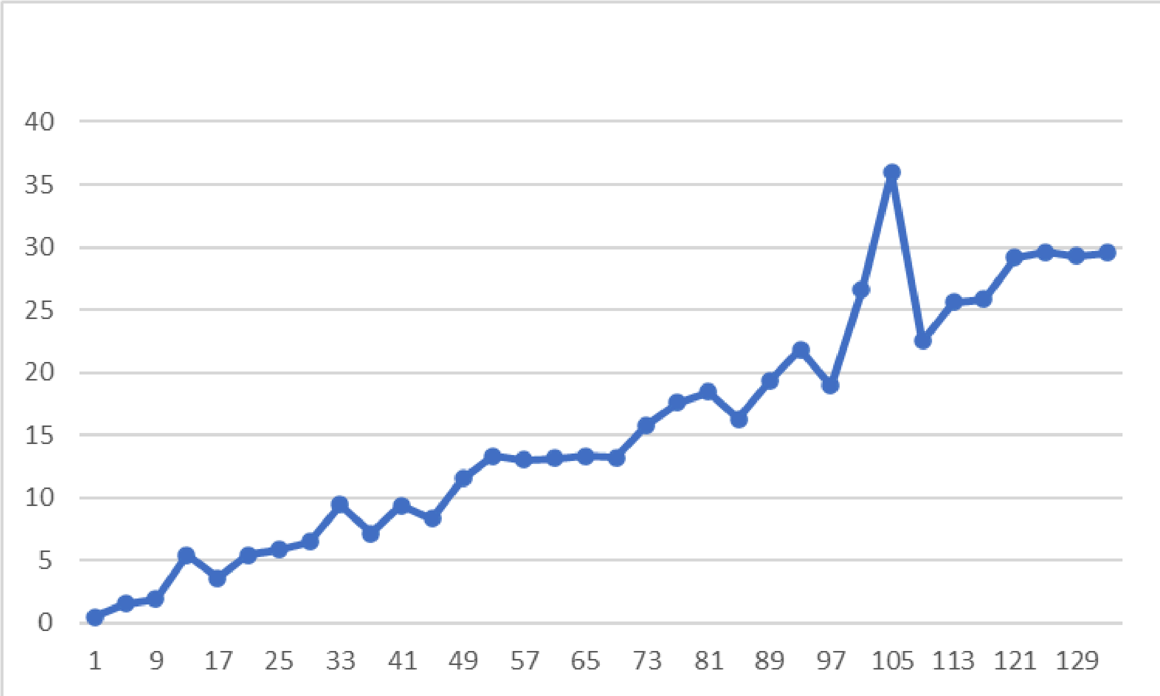
```
2 elementos
5 -> 10
4 -> 25
4 -> 5 -> 10 -> 25

4 elementos
5 -> 10 -> 10 -> 20
8 -> 4 -> 25 -> 28
4 -> 5 -> 8 -> 10 -> 10 -> 20 -> 25 -> 28

6 elementos
5 -> 10 -> 10 -> 15 -> 20 -> 30
12 -> 8 -> 4 -> 25 -> 28 -> 31
4 -> 5 -> 8 -> 10 -> 10 -> 12 -> 15 -> 20 -> 25 -> 28 -> 30 -> 31
```

Mesclando as listas da 1a e 2a linha é inicialmente simples com a troca de ponteiros do item inicial, porem como é feito a ordenação com Insertion Sort, um algoritimo que demoraria O(1) de tempo se torna O(n^2) pois percorre cada elemento, e em cada elemento percorre os elementos anteriores para reoordena-los. Pelo menos a complexidade de espaço se mantem in-place.

4.1

```
Fatorial de 1 - 0.004200031980872154
Fatorial de 5 - 0.021400046534836292
Fatorial de 9 - 0.04299997817724943
Fatorial de 13 - 0.029199989512562752
Fatorial de 17 - 0.038800062611699104
Fatorial de 21 - 0.061400001868605614
Fatorial de 25 - 0.0681999372318387
Fatorial de 29 - 0.08260004688054323
Fatorial de 33 - 0.07720000576227903
Fatorial de 37 - 0.1420000335201621
Fatorial de 41 - 0.11340004857629538
Fatorial de 45 - 0.09979994501918554
Fatorial de 49 - 0.1708000199869275
Fatorial de 53 - 0.12079998850822449
Fatorial de 57 - 0.2845999551936984
Fatorial de 61 - 0.1420000335201621
Fatorial de 65 - 0.2981999423354864
Fatorial de 69 - 0.1700000138953328
Fatorial de 73 - 0.16959989443421364
Fatorial de 77 - 0.26839994825422764
Fatorial de 81 - 0.23699994198977947
Fatorial de 85 - 0.19799999427050352
Fatorial de 89 - 0.22139993961900473
Fatorial de 93 - 0.26000000070780516
Fatorial de 97 - 0.26000000070780516
Fatorial de 101 - 0.2847999567165971
Fatorial de 105 - 0.28820009902119637
Fatorial de 109 - 0.29960006941109896
Fatorial de 113 - 0.33739989157766104
```

```
Fatorial de 117 - 0.28979999478906393
Fatorial de 121 - 0.3287999425083399
Fatorial de 125 - 0.3467999631538987
Fatorial de 129 - 0.3591999411582947
Fatorial de 133 - 0.36099995486438274
Fatorial de 137 - 0.32700004521757364
Fatorial de 141 - 0.3980000037699938
Fatorial de 145 - 0.34380005672574043
Fatorial de 149 - 0.34760008566081524
Fatorial de 153 - 0.34460006281733513
Fatorial de 157 - 0.3700000233948231
Fatorial de 161 - 0.3712000325322151
Fatorial de 165 - 0.40600006468594074
Fatorial de 169 - 0.4008000250905752
Fatorial de 173 - 0.47420000191777945
Fatorial de 177 - 0.5097999237477779
Fatorial de 181 - 0.4162000259384513
Fatorial de 185 - 0.4403999773785472
Fatorial de 189 - 0.44499989598989487
Fatorial de 193 - 0.48640009481459856
Fatorial de 197 - 0.5477999802678823
Fatorial de 201 - 0.43839996214956045
```



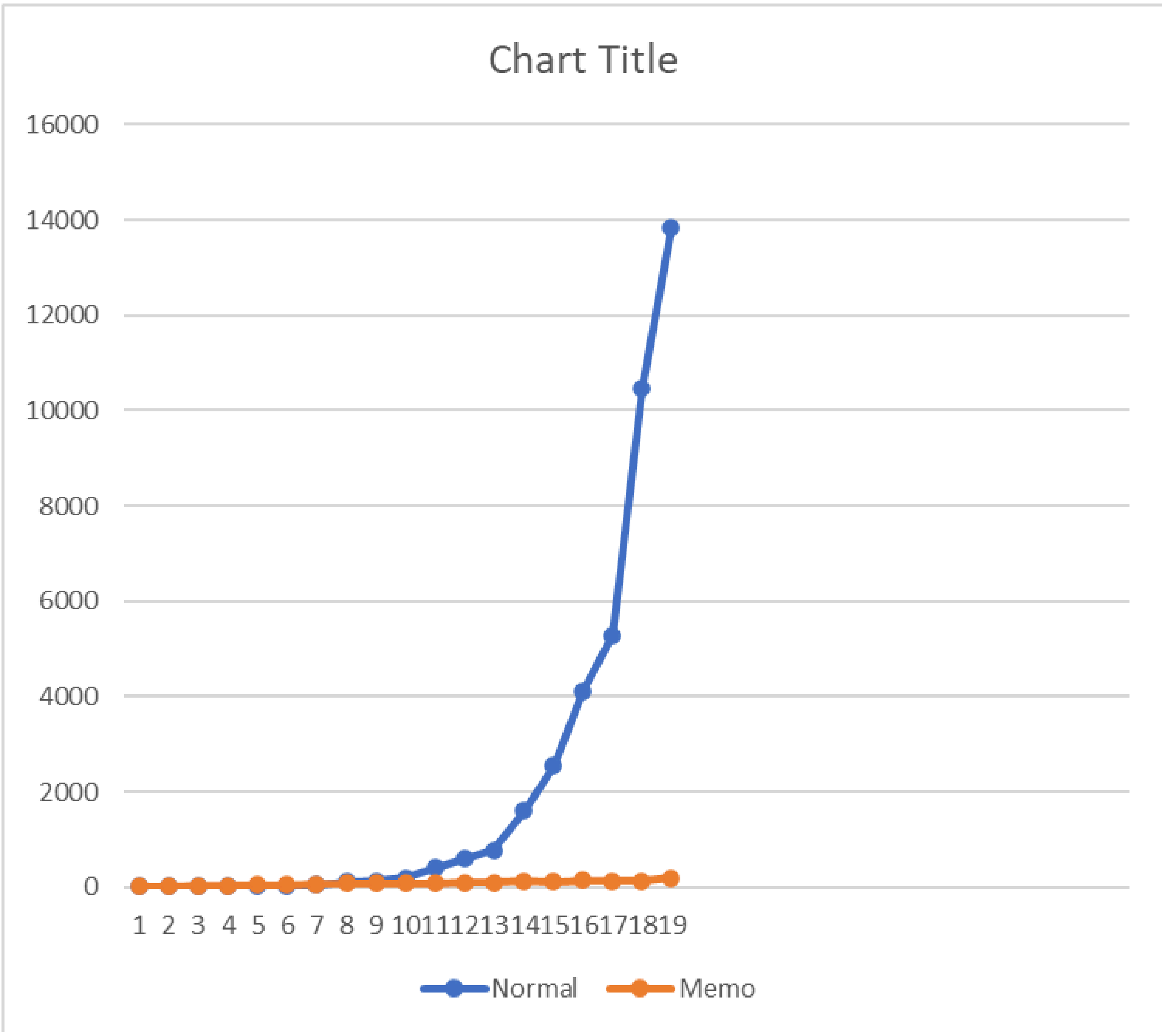
Tempo multiplicado por 10000. A complexidade de tempo parece ser O(n) e a de espaço será O(n) também pois cria uma instância de recursão sequencialmente até n chegar a 1 ou menos. Tempo multiplicado por 100 para o grafo.

4.2

```
Fibonacci na posição 1 - 0.005199923180043697
Fibonacci na posição 2 - 0.012599979527294636
Fibonacci na posição 3 - 0.014999997802078724
Fibonacci na posição 4 - 0.021000043489038944
Fibonacci na posição 5 - 0.02839986700564623
Fibonacci na posição 6 - 0.041199964471161366
Fibonacci na posição 7 - 0.08100003469735384
Fibonacci na posição 8 - 0.09640003554522991
Fibonacci na posição 9 - 0.14899997040629387
Fibonacci na posição 10 - 0.22960000205785036
Fibonacci na posição 11 - 0.4564000992104411
Fibonacci na posição 12 - 0.6209999555721879
Fibonacci na posição 13 - 0.9536000434309244
Fibonacci na posição 14 - 1.5280001098290086
Fibonacci na posição 15 - 2.474999986588955
Fibonacci na posição 16 - 4.002599976956844
Fibonacci na posição 17 - 6.418599979951978
Fibonacci na posição 18 - 10.511600063182414
Fibonacci na posição 19 - 16.93759998306632
```

```
Fibonacci com memorização na posição 1 - 0.0077999429777264595
Fibonacci com memorização na posição 2 - 0.014999997802078724
Fibonacci com memorização na posição 3 - 0.027599997732937336
Fibonacci com memorização na posição 4 - 0.035400036722421646
Fibonacci com memorização na posição 5 - 0.03620004281401634
Fibonacci com memorização na posição 6 - 0.04980002995580435
Fibonacci com memorização na posição 7 - 0.05719985347241163
Fibonacci com memorização na posição 8 - 0.06739993114024401
Fibonacci com memorização na posição 9 - 0.06340001709759235
Fibonacci com memorização na posição 10 - 0.0887999776750803
Fibonacci com memorização na posição 11 - 0.07619999814778566
Fibonacci com memorização na posição 12 - 0.1047999830916524
Fibonacci com memorização na posição 13 - 0.09200011845678091
Fibonacci com memorização na posição 14 - 0.10180007666349411
Fibonacci com memorização na posição 15 - 0.13699999544769526
Fibonacci com memorização na posição 16 - 0.11500006075948477
Fibonacci com memorização na posição 17 - 0.1455999445170164
Fibonacci com memorização na posição 18 - 0.14999997802078724
Fibonacci com memorização na posição 19 - 0.13020006008446217
```

Tempo multiplicado por 10000.



Tempo multiplicado por 1000 para o grafo. Observamos que o normal se aproxima de O(2^n), enquanto o com memorização cresce ligeiramente com O(n) de tempo pois o resultado das execuções anteriores são salvos e retornam eles para evitar o desencadeamento de multiplas recursões.

5.1

```
 10001 elementos
0.00257489993236959 segundos -  Soma paralela
50005000
0.00036310008727014065 segundos -  Soma linear
50005000

 100001 elementos
0.0027073000092059374 segundos -  Soma paralela
5000050000
0.0036442000418901443 segundos -  Soma linear
5000050000

 1000001 elementos
0.03692439990118146 segundos -  Soma paralela
500000500000
0.04978410014882684 segundos -  Soma linear
500000500000

 10000001 elementos
0.4280488998629153 segundos -  Soma paralela
50000005000000
0.4880907000042498 segundos -  Soma linear
50000005000000

 100000001 elementos
9.731650700094178 segundos -  Soma paralela
5000000050000000
5.343587799929082 segundos -  Soma linear
5000000050000000
```

A versão paralela particiona a entrada em uma quantidade de threads igual a contagem de cores do CPU, usando a função linear para executar a soma.
A versão paralela demonstra um pouco mais de velocidade com entradas maiores.

5.2

```
2 Nodes - 0.5054787799948827 segundos
4 Nodes - 0.4975977399852127 segundos
8 Nodes - 0.49206802002154293 segundos
16 Nodes - 0.4882247399888001 segundos
32 Nodes - 0.5069768800050951 segundos
64 Nodes - 0.49252170000690965 segundos
128 Nodes - 0.5074204600183293 segundos
256 Nodes - 0.5011023400002159 segundos
```

Para cada sub-arvore criada pelo algoritimo de busca, um processo é assinalado para buscar o elemento dentro dela, e dentro desses processos a mesma coisa é feito até não sobrar mais sub-arvores. Um objeto Value, ou uma variavel compartilhada entre esses processos, atualiza com o termino dos processos individuais.
A execução em paralelo parece ter O(1) de complexidade de tempo pois não varia com

o número de nódulos na arvore.

5.3

```
Linear - 0.043833460006862876 segundos
Paralelo - 0.46790697999531405 segundos
Linear - 0.5299815000034869 segundos
Paralelo - 0.7715007600025274 segundos
Linear - 6.8945374800008725 segundos
Paralelo - 4.797826699982397 segundos
```

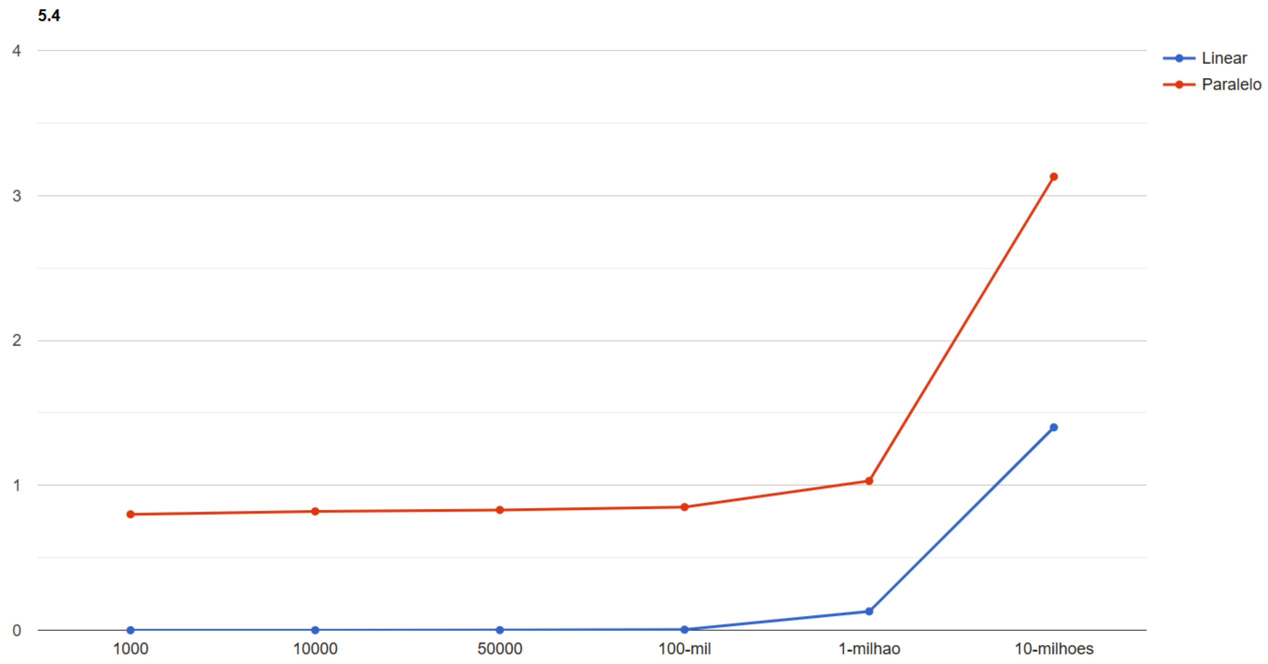
As entradas foram 10000, 100000 e 1000000 respectivamente. Ambas são O(n*log(n)), a paralela parece demorar mais com entradas menores, mas supera a versão serial com entradas maiores já que coloca cada lista dividida em um thread próprio.

5.4

```
Maximo linear: 999 - 3.3240008633583784e-05 segundos
Maximo paralelo: 999 - 0.826763539982494 segundos
Maximo linear: 9999 - 0.00031619999790564177 segundos
Maximo paralelo: 9999 - 0.8180296999984421 segundos
Maximo linear: 49999 - 0.0015216000145301222 segundos
Maximo paralelo: 49999 - 0.8371893600095064 segundos
Maximo linear: 99999 - 0.004298439994454384 segundos
Maximo paralelo: 99999 - 0.8517226599855349 segundos
```

```
=================== RESTART: C:\Users\donke\Docume
Maximo linear: 999999 - 0.13352262001717463 segundos
Maximo paralelo: 999999 - 1.034208860003855 segundos

=================== RESTART: C:\Users\donke\Docume
Maximo linear: 9999999 - 1.472459119989071 segundos
Maximo paralelo: 9999999 - 3.1369474000181072 segundos
```

Foi usado um equivalente de threads a contagem de cores do CPU
As entradas foram 1000, 10000, 50000, 100000, 1 milhão e 10 milhões respectivamente.



A complexidade de tempo para ambas parece ser O(n^2), com a paralela perdendo para a linear.