Análise experimental de algoritmos usando Python

Patrícia Mariana Ramos Marcolino
pmrmarcolino@hotmail.com

Eduardo Pinheiro Barbosa
eduardptu@hotmail.com

Faculdade de Computação
Universidade Federal de Uberlândia

1 de julho de 2016

Lista de Figuras

Lista de Tabelas

Lista de Listagens

Sumário

Capítulo 1

Análise

O algoritmo Radix Sort ordena um vetor A de n números inteiros com um número constante d de dígitos, através de ordenações parciais dígito a dígito.
Podemos também ordenar os números ordenando-os segundo cada um de seus dígitos, começando pelo menos significativo.
O seguinte argumento indutivo garante a corretude do algoritmo:
Por hipótese de indução, assumimos que os números estão ordenados com relação aos $i-1$ dígitos menos significativos.
Ordenando os números com relação ao i-ésimo dígito, com um algoritmo estável, teremos então os números ordenados com relação aos i dígitos menos significativos, pois:
Para dois números com o i-ésimo dígito distintos, o de menor valor no dígito estará antes do de maior valor;
E se ambos possuem o i -ésimo dígito igual, então a ordem dos dois também estará correta pois utilizamos um método de ordenação estável e, por hipótese de indução, os dois elementos já estavam ordenados segundo os i-1dígitos menos significativos. A complexidade do Radix Sort depende da complexidade do algoritmo estável usado para ordenar cada dígito dos elementos. Se essa complexidade estiver em $\theta(f(n))$, obtemos uma complexidade total de $\theta(df(n))$ para o Radix Sort. Como supomos d constante, a complexidade reduz-se para $\theta(f(n))$. Se o algoritmo estável for, por exemplo, o Counting Sort, obtemos a complexidade $\theta(n+k)$. Supondo $k \in O(n)$, resulta numa complexidade linear em n. Em contraste, um algoritmo por comparação como o MergeSort teria complexidade $n\log(n)$. Assim, o Radix Sort é mais vantajoso que o MergeSort quando $d < \log(n)$, ou seja, o número de dígitos for menor que log n.
Se considerarmos que n também é um limite superior para o maior valor a ser ordenado, então $O(\log(n))$ é uma estimativa para o tamanho, em dígitos, desse número.
Veja que se o uso de memória auxiliar for muito limitado, então o melhor mesmo é usar um algoritmo de ordenação por comparação in-place.
Note que e possível usar o Radix Sort para ordenar outros tipos de elementos, como datas, palavras em ordem lexicográfica e qualquer outro tipo que possa ser visto como uma tupla ordenada de itens comparáveis

Capítulo 2

Resultados

2.1 Tabelas

n	comparações	tempo(s)
32	7	0.001504
64	7	0.002867
128	7	0.005680
256	7	0.011012
512	7	0.022024
1024	7	0.043471
2048	7	0.086574
4096	7	0.172890
8192	7	0.356761

Tabela 2.1: *Tabela com vetor teste aleatório: A linha te interesse analisada para este caso é a 16.*

n	comparações	tempo(s)
32	7	0.001576
64	7	0.002985
128	7	0.005884
256	7	0.011169
512	7	0.022276
1024	7	0.043654
2048	7	0.084924
4096	7	0.170405
8192	7	0.352600

Tabela 2.2: *Tabela com vetor teste crescente: A linha te interesse analisada para este caso é a 16.*

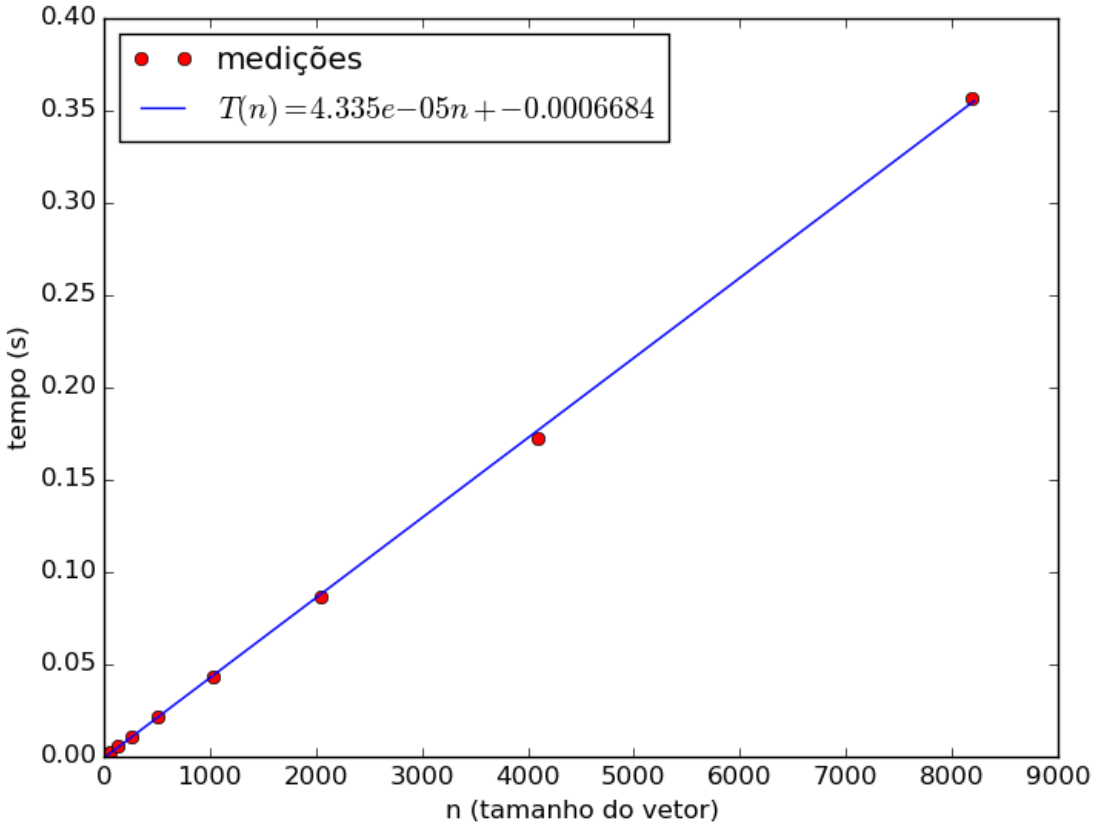

Figura 2.1: *A análise do grafico para* 2^{32} *segue abaixo para radixsort*
Tendo a função $T(n) = 4.335e-5 * n - 0.0006684$ e para o $n = 2^{32}$,
$T(2^{32}) = 186186.8316132$

n	comparações	tempo(s)
32	7	0.001467
64	7	0.002939
128	7	0.005533
256	7	0.010907
512	7	0.022306
1024	7	0.043331
2048	7	0.083704
4096	7	0.177603
8192	7	0.363321

Tabela 2.3: *Tabela com vetor teste decrescente: A linha te interesse analisada para este caso é a 16.*

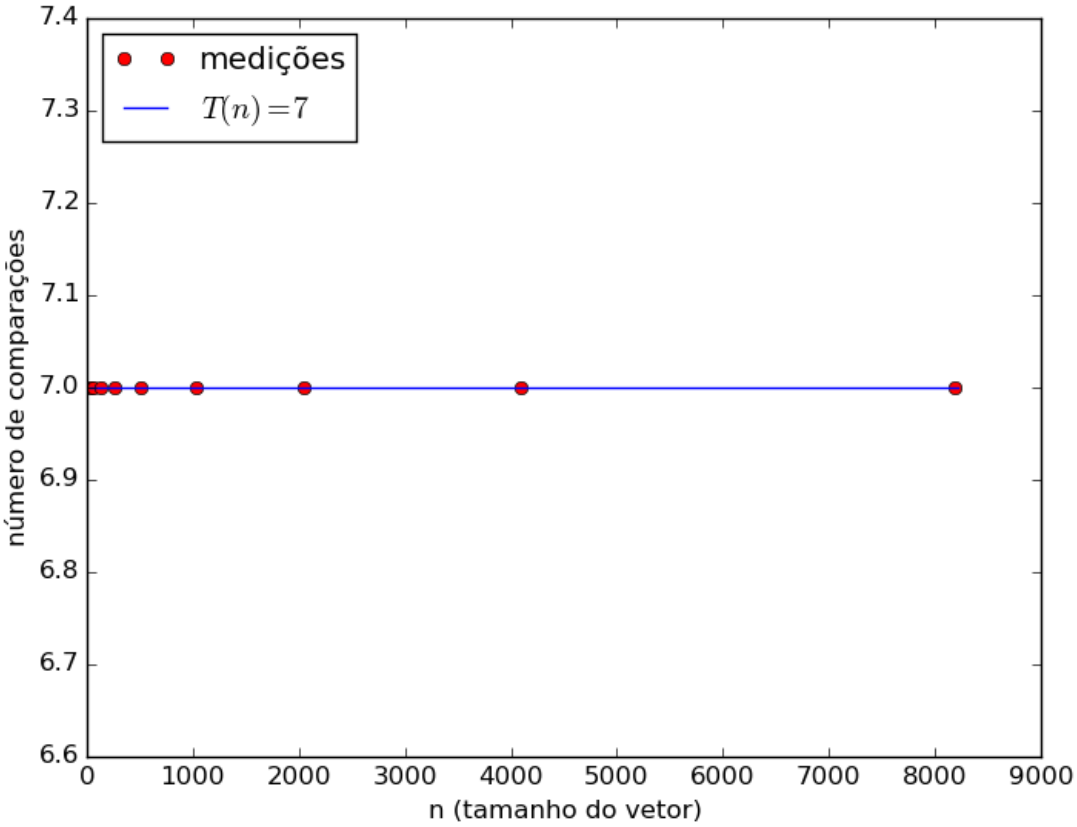

Figura 2.2: *A análise do grafico para* 2^{32} *segue abaixo para radixsort*
Tendo a função $T(n) = 7$ e para o $n = 2^{32}$, $T(2^{32}) = 7$

n	comparações	tempo(s)
32	7	0.001560
64	7	0.002925
128	7	0.005988
256	7	0.011510
512	7	0.024760
1024	7	0.042652
2048	7	0.085191
4096	7	0.177059
8192	7	0.370108

Tabela 2.4: *Tabela com vetor teste quase crecente 10%: A linha te interesse analisada para este caso é a 16.*

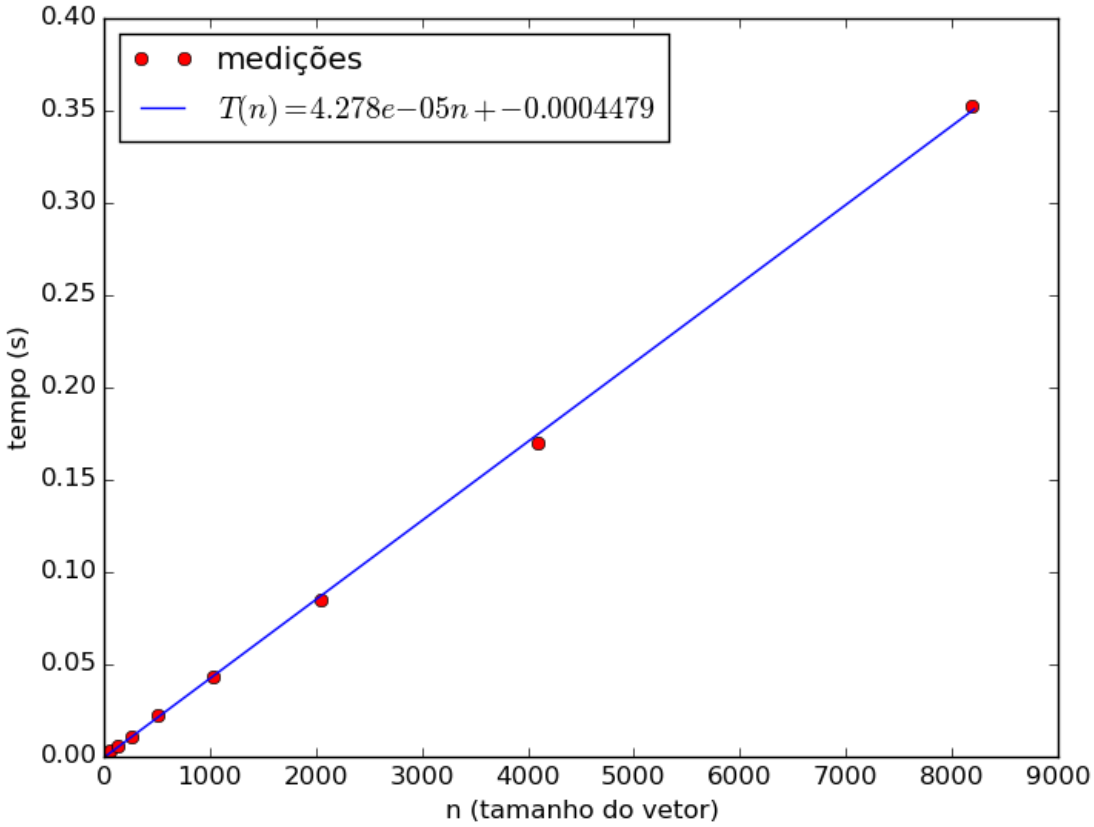

Figura 2.3: *A análise do grafico para* 2^{32} *segue abaixo para radixsort*
Tendo a função $T(n) = 4.278\text{e} - 5 * n - 0.0004479$ e para o $n = 2^{32}$,
$T(2^{32}) = 183738.70047498$

n	comparações	tempo(s)
32	7	0.001606
64	7	0.002909
128	7	0.005773
256	7	0.011253
512	7	0.022697
1024	7	0.045749
2048	7	0.087966
4096	7	0.176329
8192	7	0.352703

Tabela 2.5: *Tabela com vetor teste quase crecente 20%: A linha te interesse analisada para este caso é a 16.*

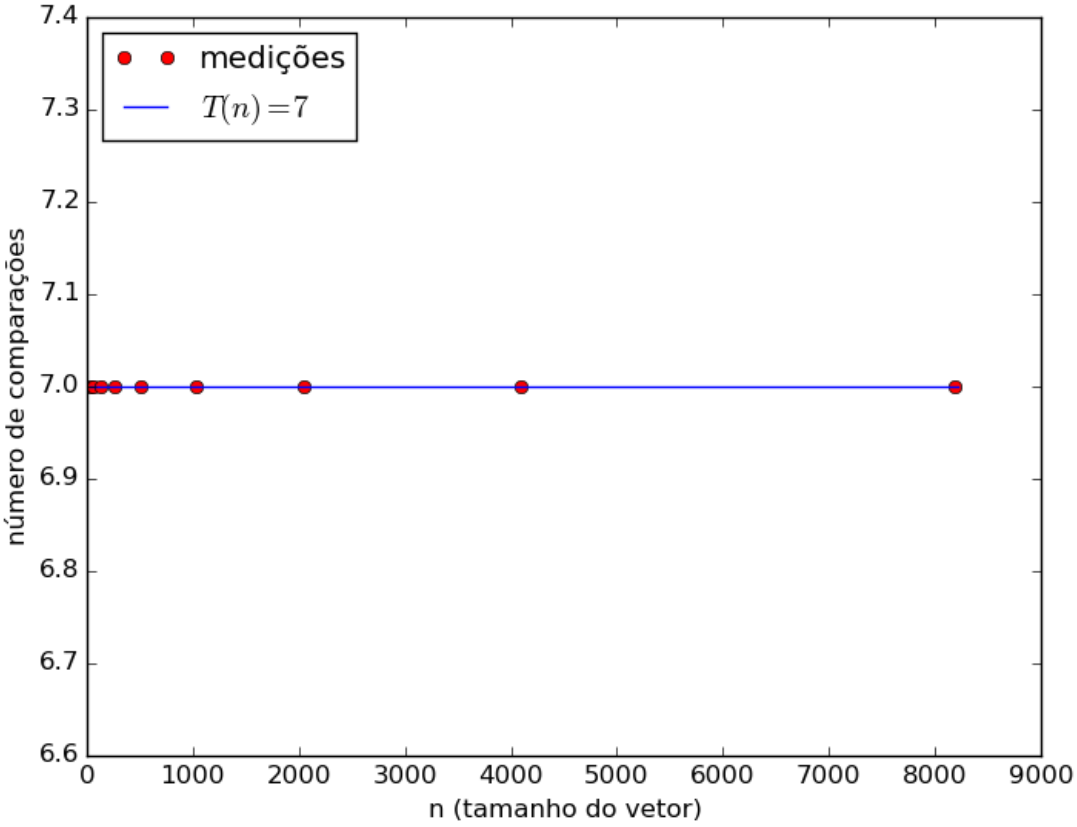

Figura 2.4: *A análise do grafico para* 2^{32} *segue abaixo para radixsort*
Tendo a função $T(n) = 7$ e para o $n = 2^{32}$, $T(2^{32}) = 7$

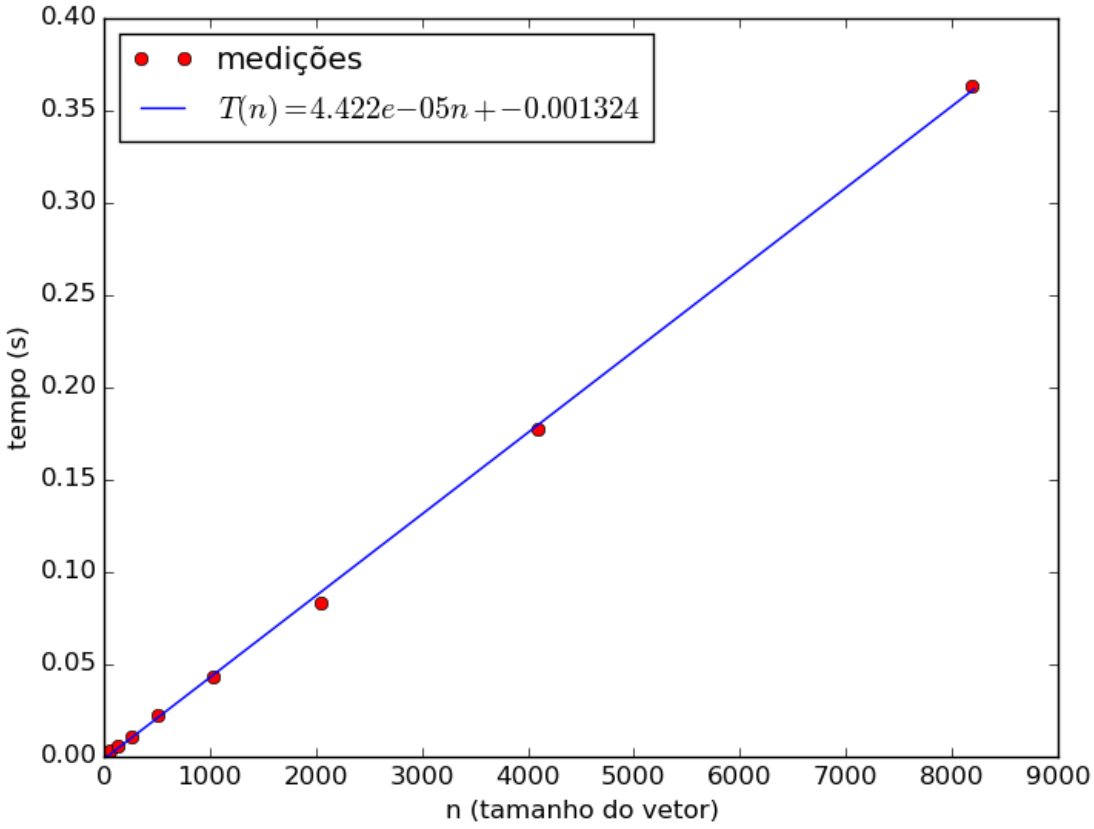

Figura 2.5: *A análise do grafico para* 2^{32} *segue abaixo para radixsort*
Tendo a função $T(n) = 4.422\text{e} - 5 * n - 0.0001324$ e para o $n = 2^{32}$,
$T(2^{32}) = 189923.45369672$

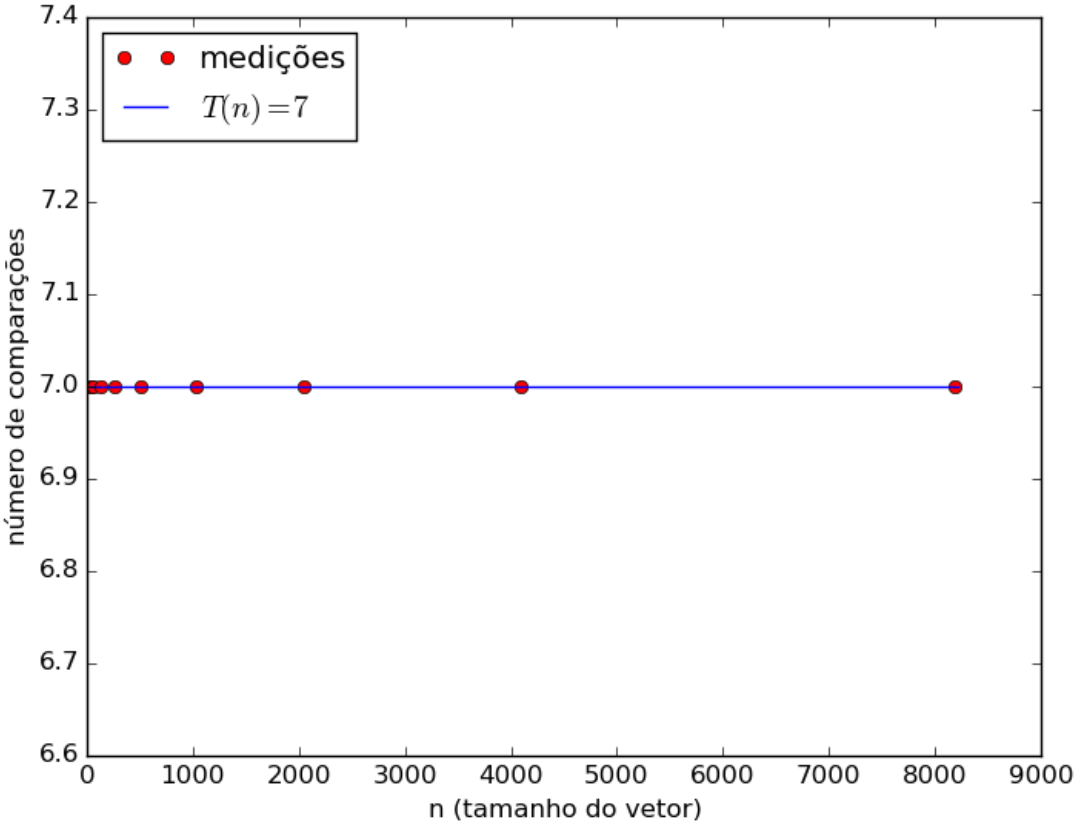

Figura 2.6: *A análise do grafico para* 2^{32} *segue abaixo para radixsort*
Tendo a função $T(n) = 7$ e para o $n = 2^{32}$, $T(2^{32}) = 7$

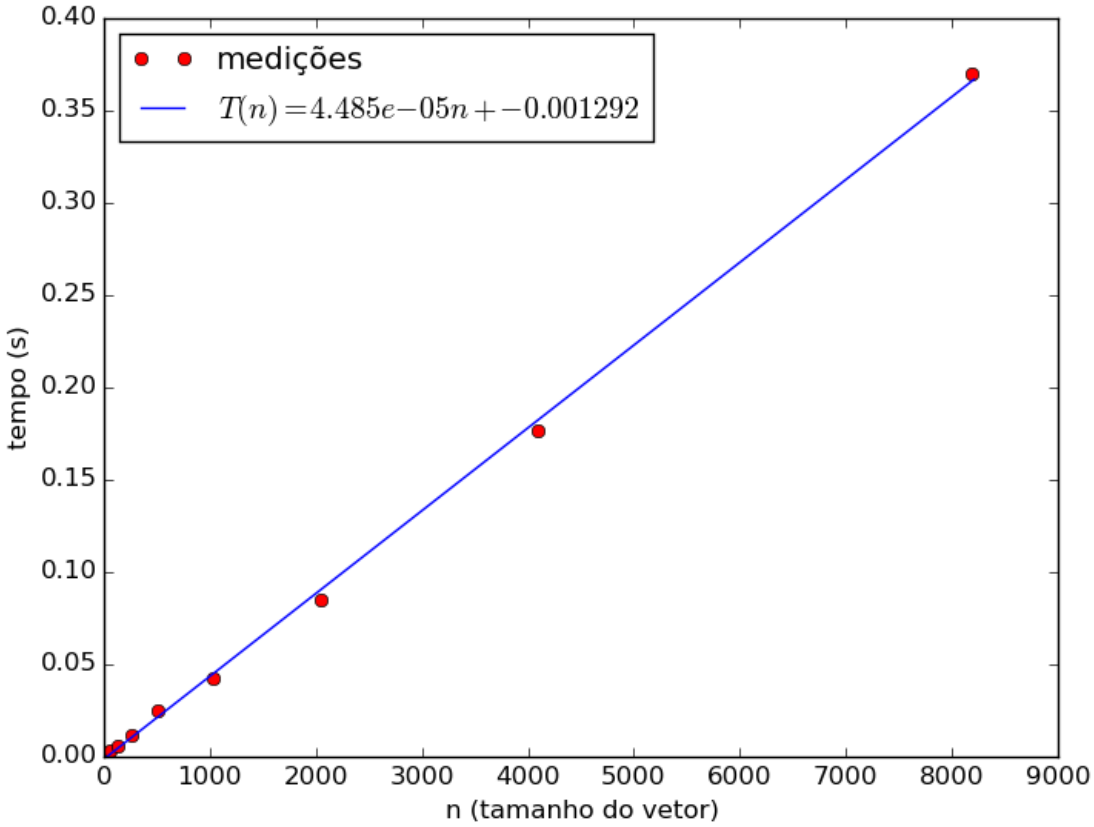

Figura 2.7: *A análise do grafico para* 2^{32} *segue abaixo para radixsort*
Tendo a função $T(n) = 4.485\text{e} - 5 * n - 0.0001292$ e para o $n = 2^{32}$,
$T(2^{32}) = 192629.2830964$

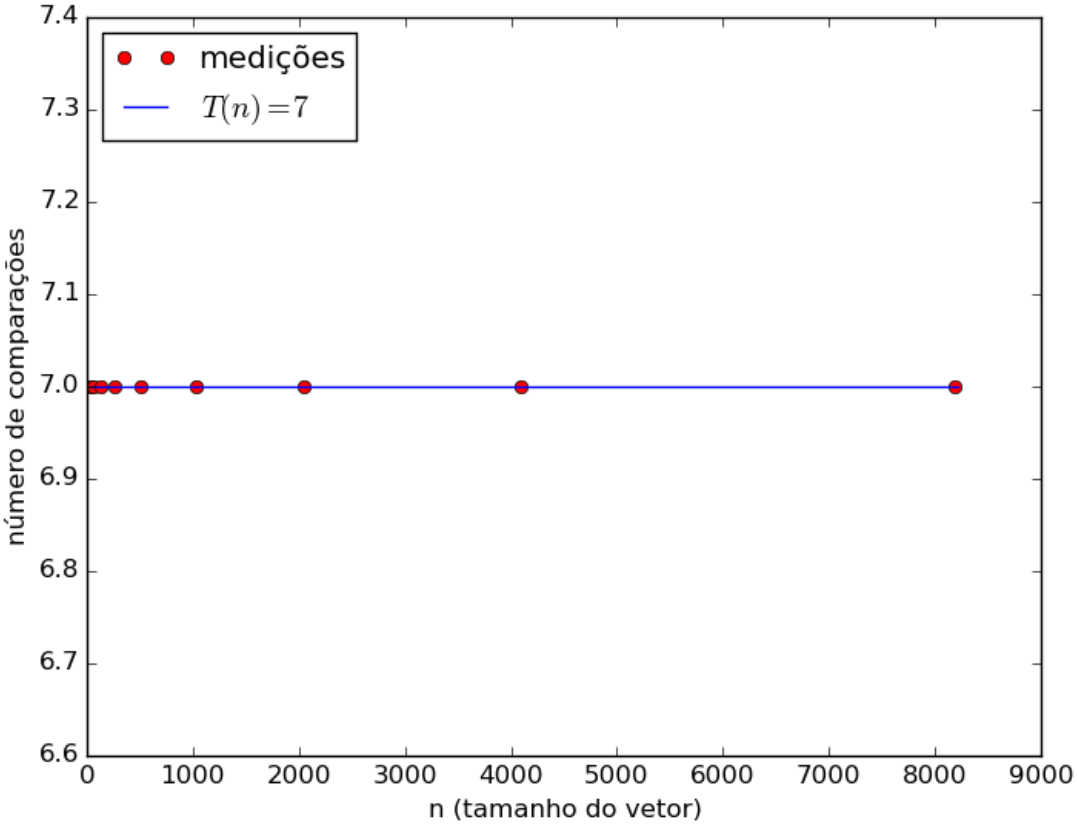

Figura 2.8: *A análise do grafico para* 2^{32} *segue abaixo para radixsort*
Tendo a função $T(n) = 7$ e para o $n = 2^{32}$, $T(2^{32}) = 7$

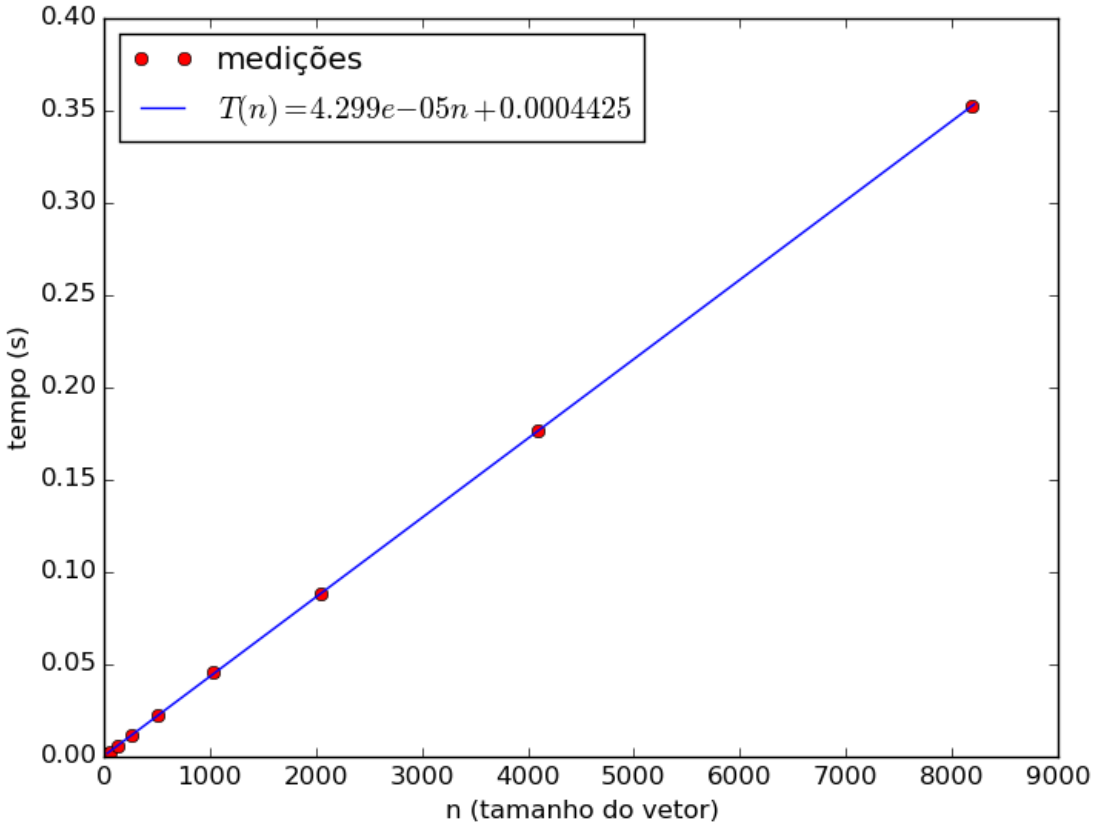

Figura 2.9: *A análise do grafico para* 2^{32} *segue abaixo para radixsort*
Tendo a função $T(n) = 4.299\text{e} - 5 * n - 0.0004425$ e para o $n = 2^{32}$,
$T(2^{32}) = 184640.64361254$

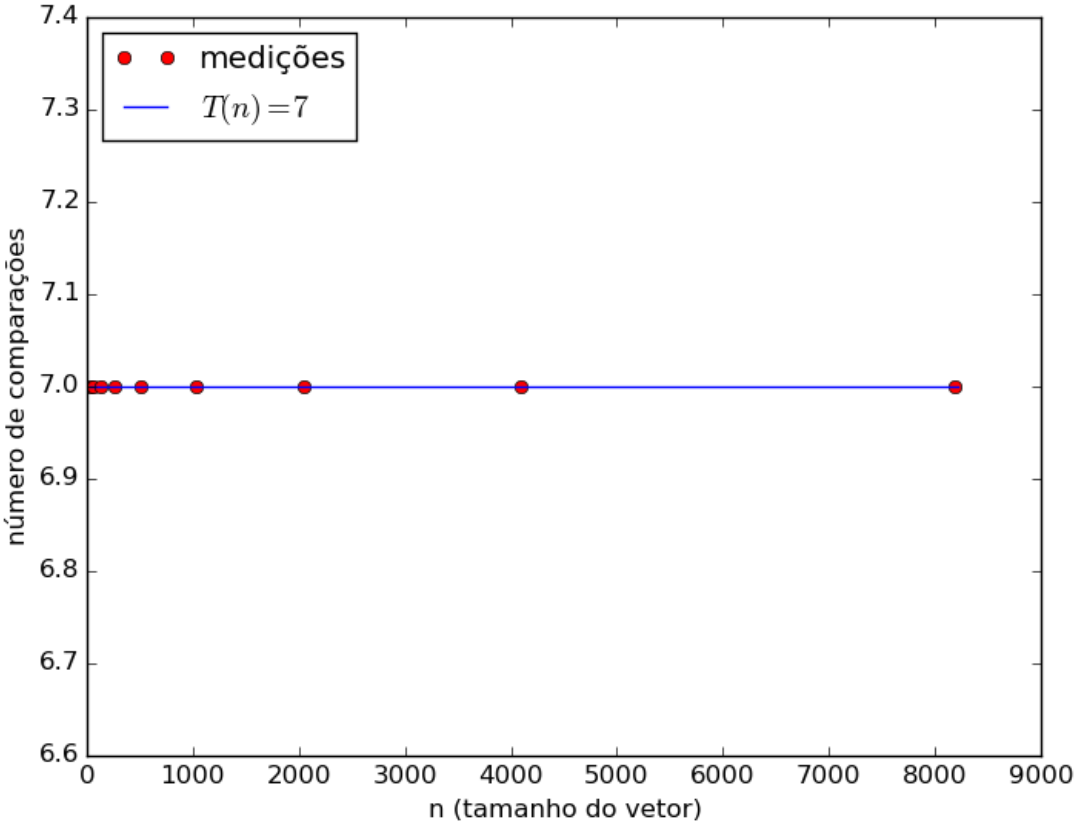

Figura 2.10: *A análise do grafico para* 2^{32} *segue abaixo para radixsort*
Tendo a função $T(n) = 7$ e para o $n = 2^{32}$, $T(2^{32}) = 7$

n	comparações	tempo(s)
32	7	0.001494
64	7	0.002827
128	7	0.005499
256	7	0.010844
512	7	0.022666
1024	7	0.042608
2048	7	0.084882
4096	7	0.170542
8192	7	0.344394

Tabela 2.6: *Tabela com vetor teste quase crecente 30%: A linha te interesse analisada para este caso é a 16.*

n	comparações	tempo(s)
32	7	0.001515
64	7	0.003014
128	7	0.005460
256	7	0.011347
512	7	0.022346
1024	7	0.043654
2048	7	0.088364
4096	7	0.173760
8192	7	0.364793

Tabela 2.7: *Tabela com vetor teste quase crecente 40%: A linha te interesse analisada para este caso é a 16.*

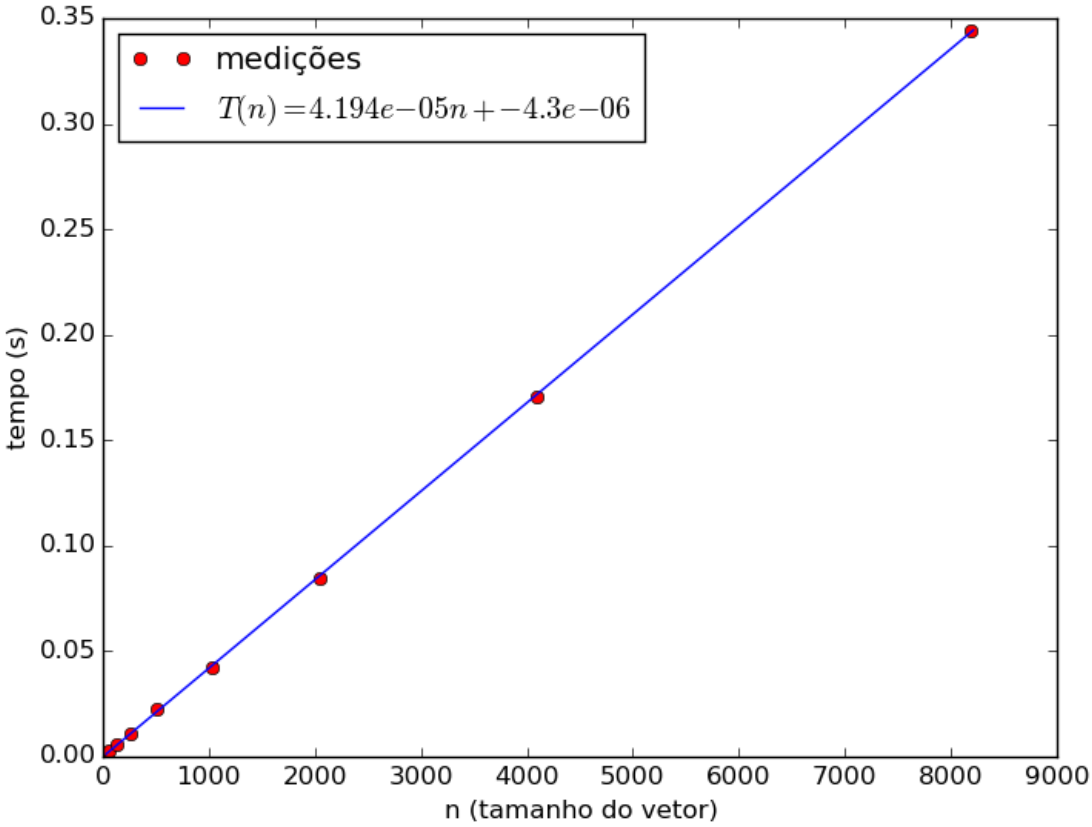

Figura 2.11: *A análise do grafico para* 2^{32} *segue abaixo para radixsort*
Tendo a função $T(n) = 4.194\text{e} - 5 * n - 4.3\text{e} - 6$ e para o $n = 2^{32}$,
$T(2^{32}) = 180130.92838994$

n	comparações	tempo(s)
32	7	0.001634
64	7	0.003030
128	7	0.006089
256	7	0.010584
512	7	0.023260
1024	7	0.045386
2048	7	0.091863
4096	7	0.170947
8192	7	0.371376

Tabela 2.8: *Tabela com vetor teste quase crecente 50%: A linha te interesse analisada para este caso é a 16.*

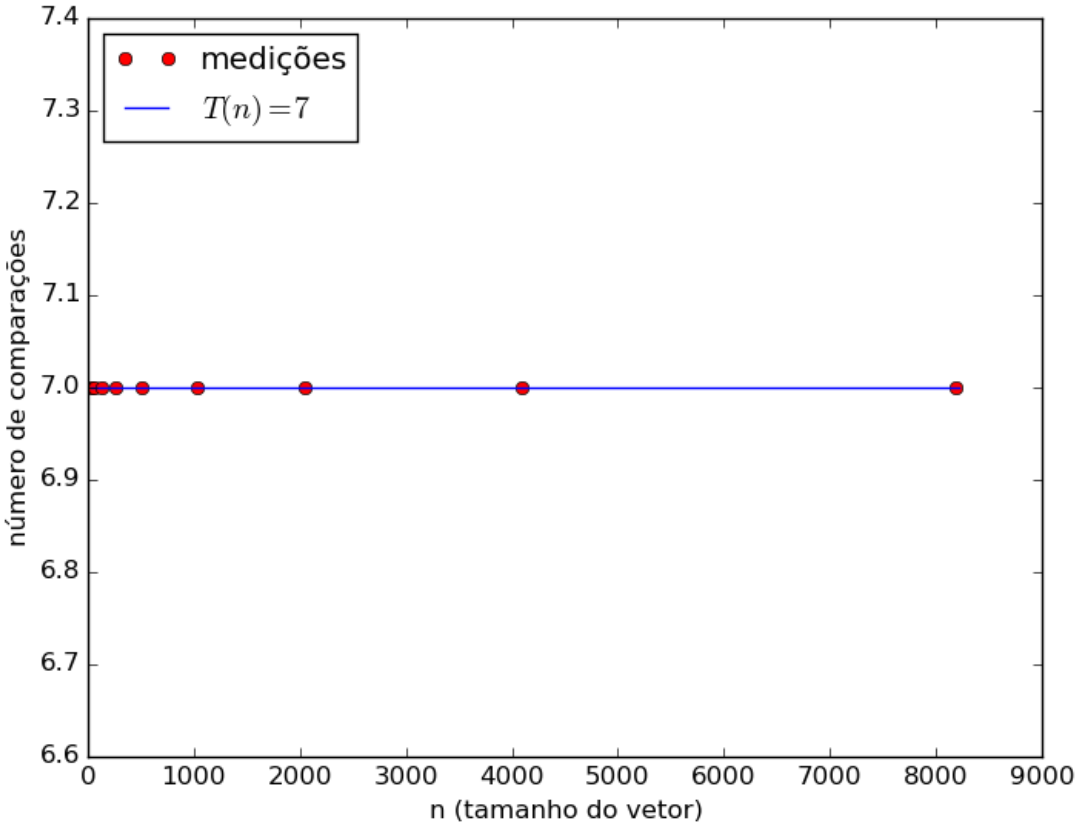

Figura 2.12: *EA análise do grafico para* 2^{32} *segue abaixo para radixsort*
Tendo a função $T(n) = 7$ e para o $n = 2^{32}$, $T(2^{32}) = 7$

n	comparações	tempo(s)
32	7	0.001529
64	7	0.003056
128	7	0.005462
256	7	0.011047
512	7	0.022181
1024	7	0.044950
2048	7	0.087427
4096	7	0.168300
8192	7	0.343764

Tabela 2.9: *Tabela com vetor teste quase decrecente 10%: A linha te interesse analisada para este caso é a 16.*

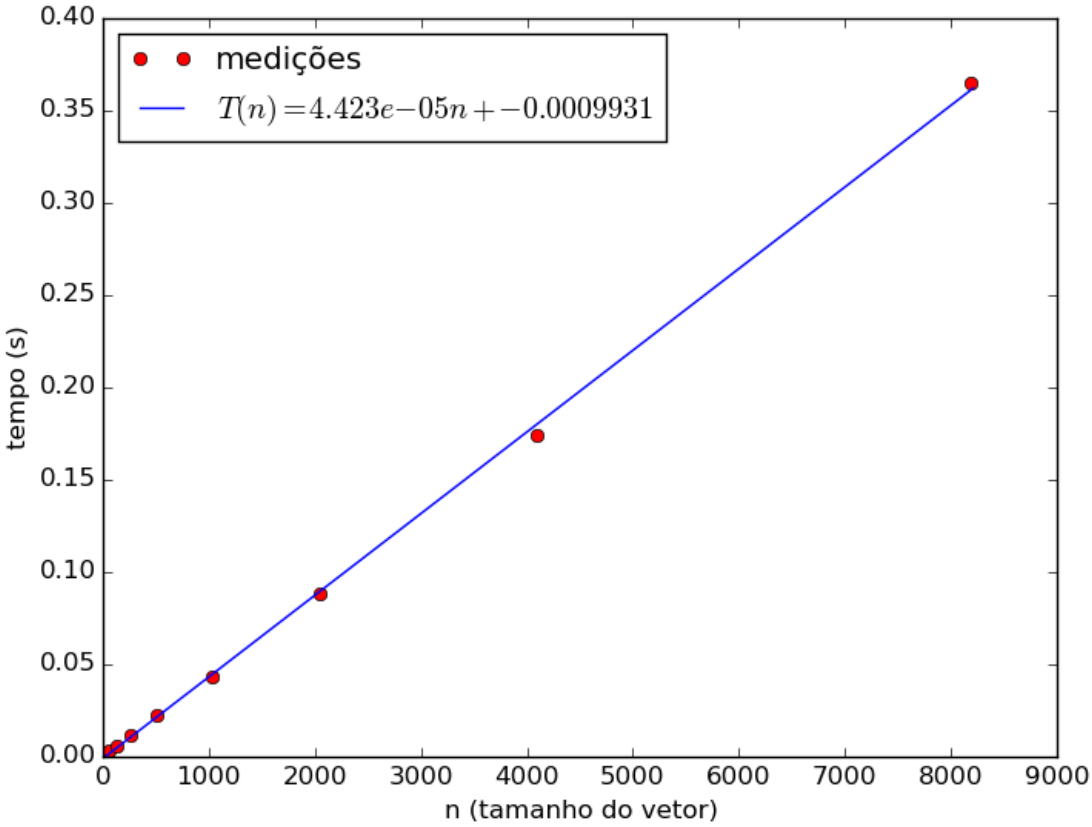

Figura 2.13: *A análise do grafico para* 2^{32} *segue abaixo para radixsort*
Tendo a função $T(n) = 4.423\text{e} - 5 * n - 0.0009931$ e para o $n = 2^{32}$,
$T(2^{32}) = 189966.40250898$

n	comparações	tempo(s)
32	7	0.001511
64	7	0.003001
128	7	0.005442
256	7	0.010978
512	7	0.021155
1024	7	0.043353
2048	7	0.086680
4096	7	0.171603
8192	7	0.343652

Tabela 2.10: *Tabela com vetor teste quase decrecente 20%: A linha te interesse analisada para este caso é a 16.*

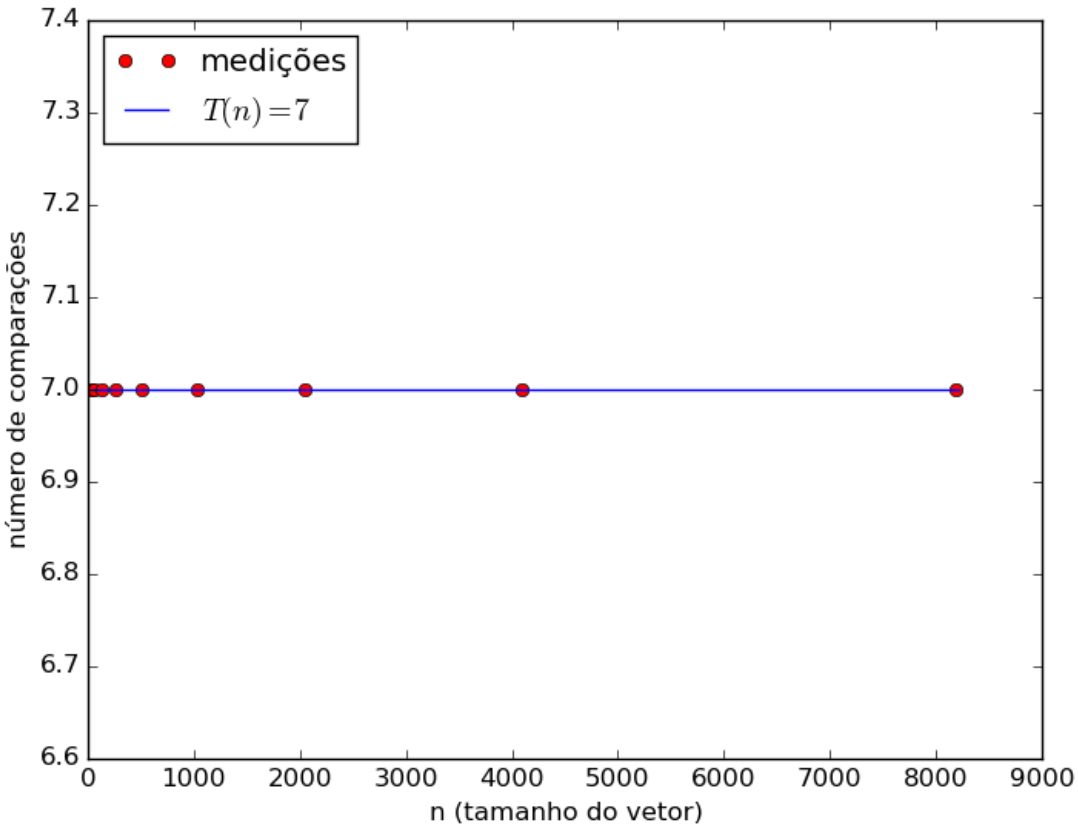

Figura 2.14: *A análise do grafico para* 2^{32} *segue abaixo para radixsort*
Tendo a função $T(n) = 7$ e para o $n = 2^{32}$, $T(2^{32}) = 7$

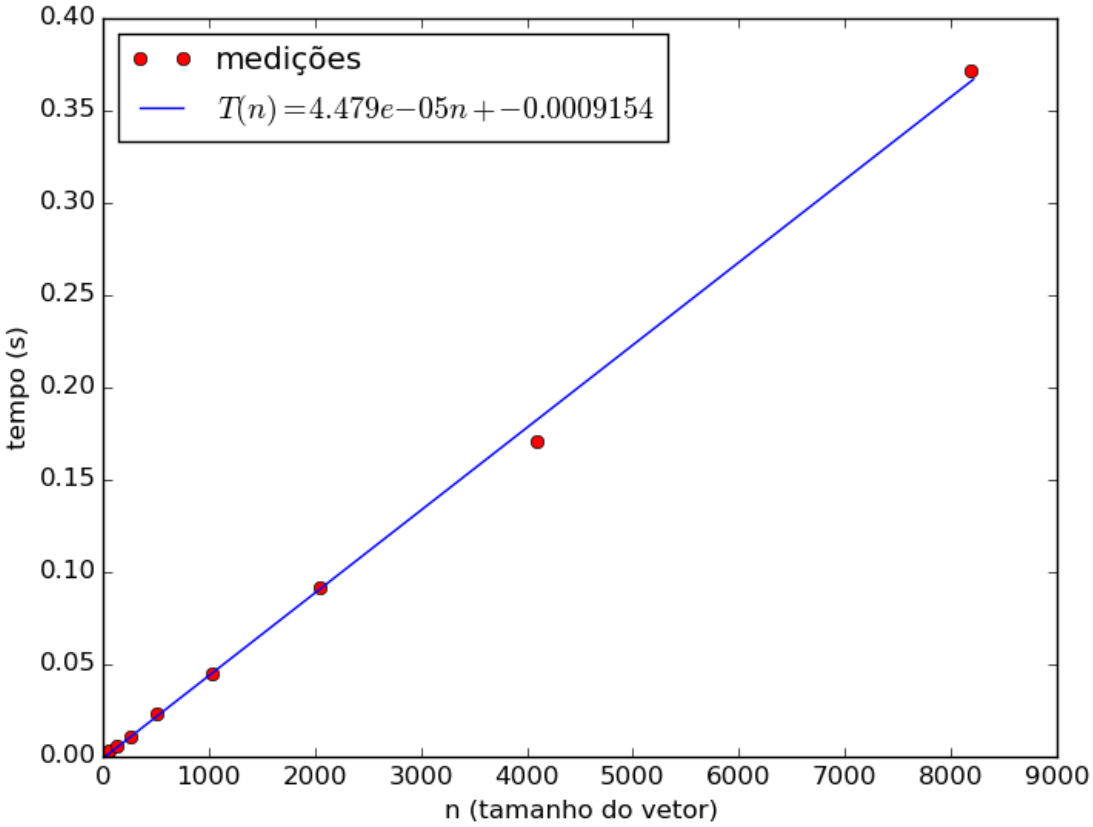

Figura 2.15: *A análise do grafico para* 2^{32} *segue abaixo para radixsort*
Tendo a função $T(n) = 4.479\text{e} - 5 * n - 0.0009154$ e para o $n = 2^{32}$,
$T(2^{32}) = 192371.58427244$

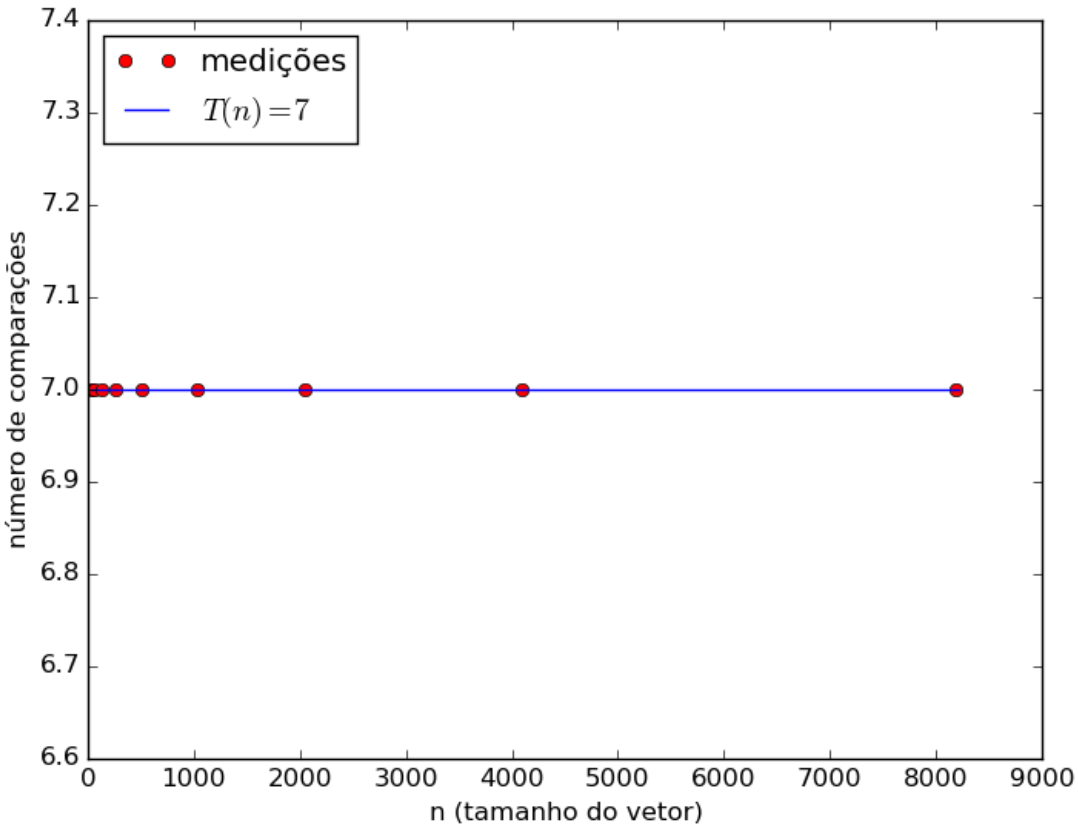

Figura 2.16: *A análise do grafico para* 2^{32} *segue abaixo para radixsort*
Tendo a função $T(n) = 7$ e para o $n = 2^{32}$, $T(2^{32}) = 7$

2.1

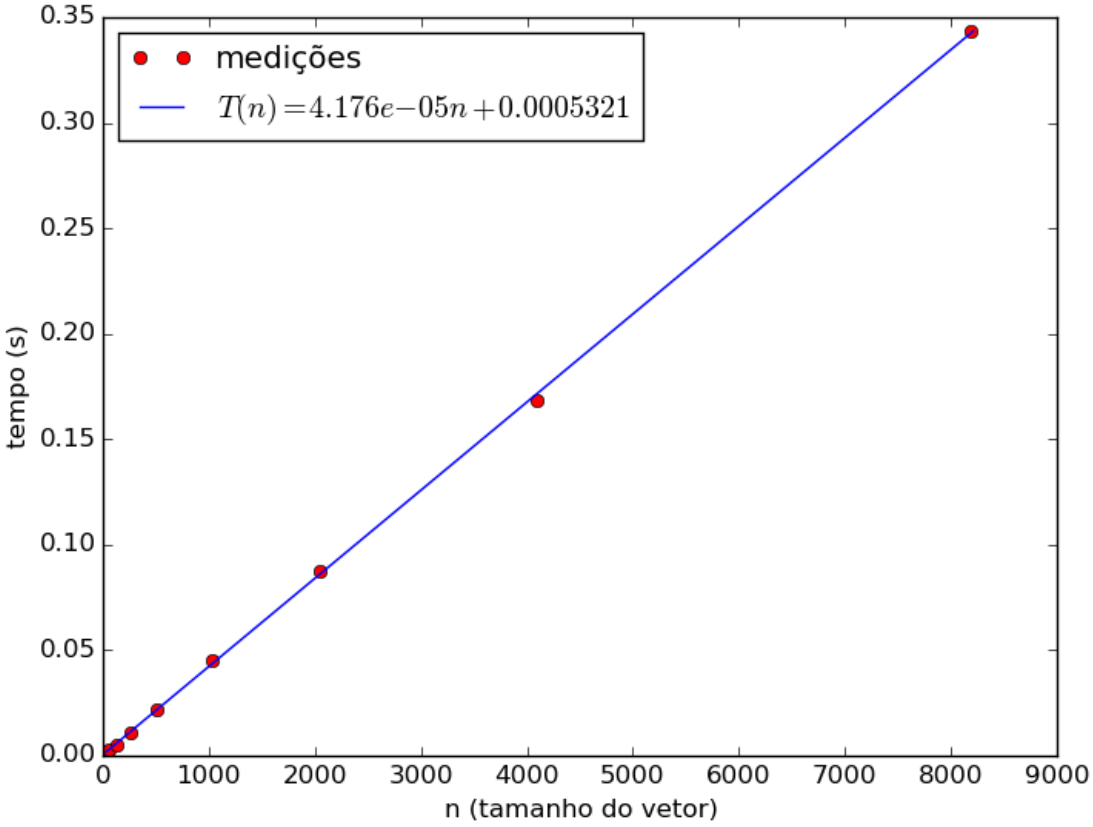

Figura 2.17: *Explique o gráfico: radixsortQuaseDecresc100.png*

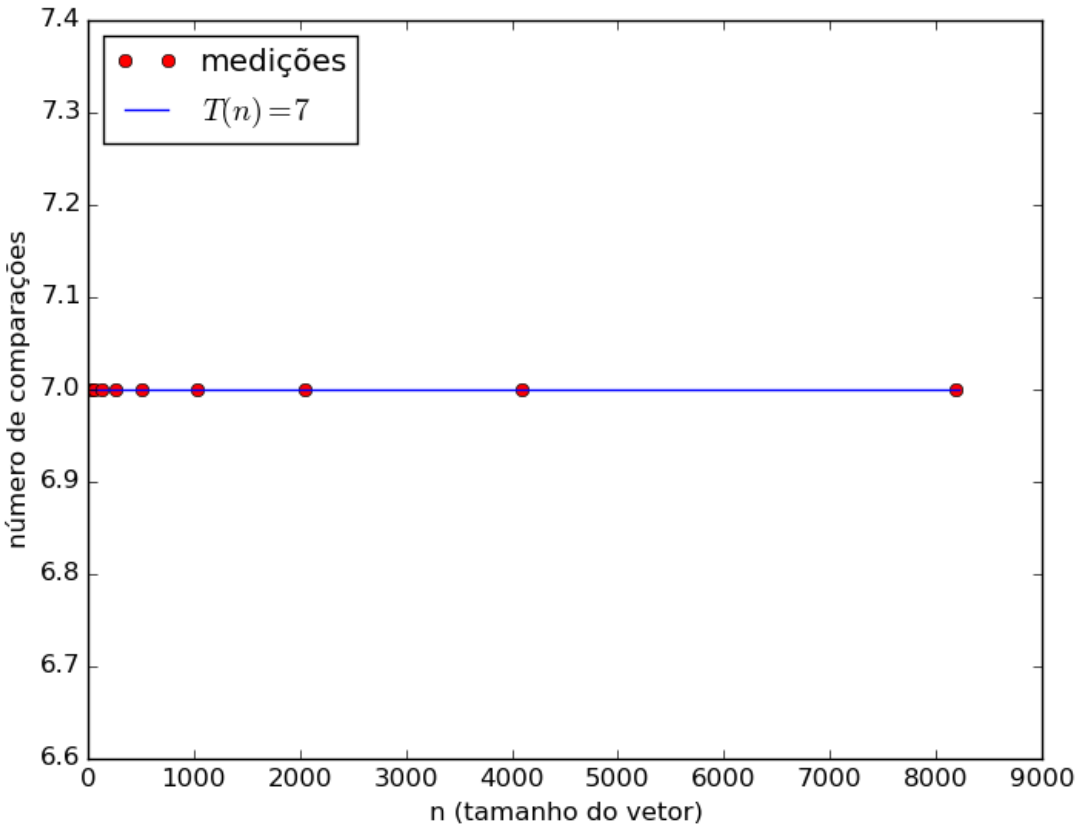

Figura 2.18: *Explique o gráfico: radixsortQuaseDecresc101.png*

2.1

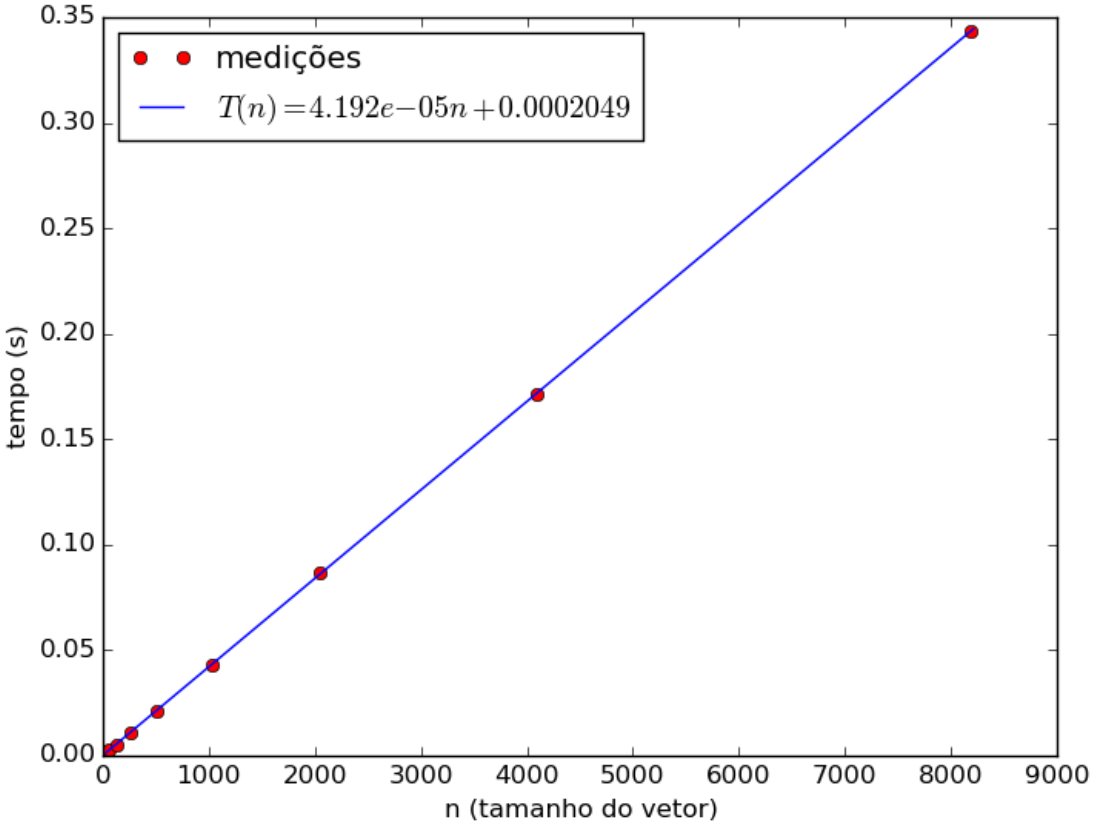

Figura 2.19: *Explique o gráfico: radixsortQuaseDecresc200.png*

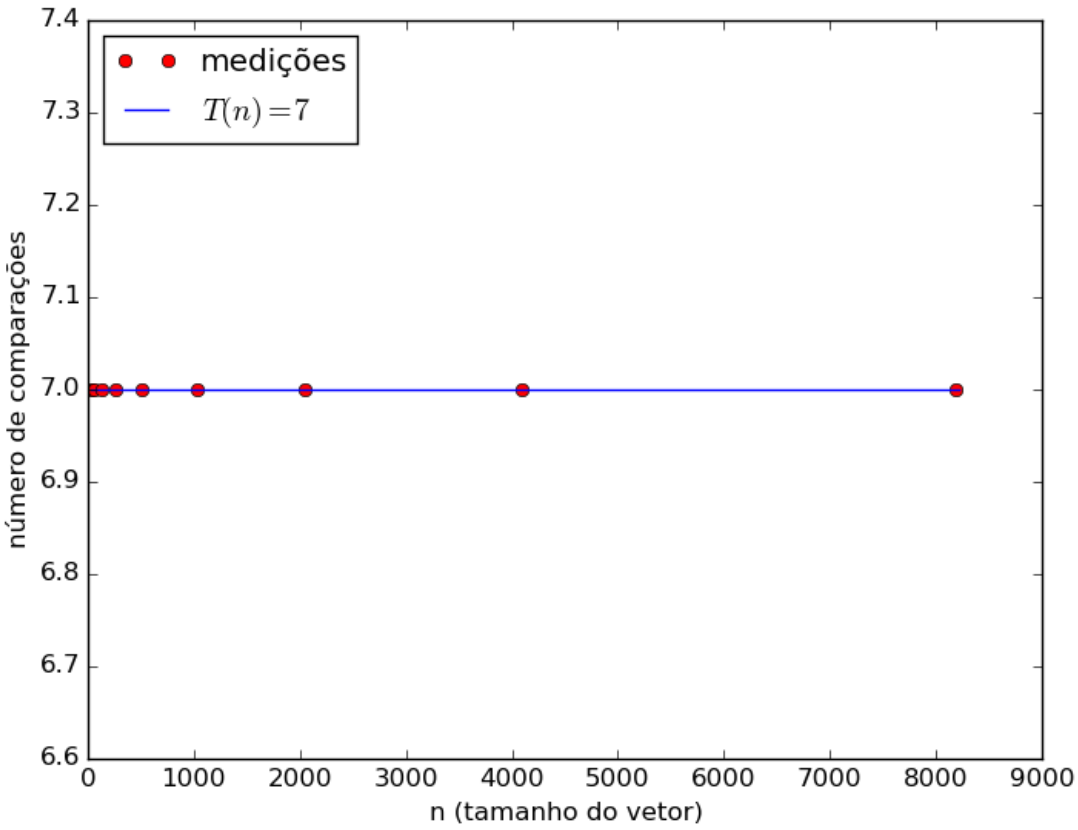

Figura 2.20: *Explique o gráfico: radixsortQuaseDecresc201.png*

n	comparações	tempo(s)
32	7	0.001571
64	7	0.003061
128	7	0.005747
256	7	0.011117
512	7	0.022279
1024	7	0.043290
2048	7	0.088795
4096	7	0.165978
8192	7	0.332857

Tabela 2.11: *Tabela com vetor teste quase decrecente 30%: A linha te interesse analisada para este caso é a 16.*

n	comparações	tempo(s)
32	7	0.001609
64	7	0.003027
128	7	0.005516
256	7	0.011404
512	7	0.021144
1024	7	0.045238
2048	7	0.088607
4096	7	0.174904
8192	7	0.365459

Tabela 2.12: *Tabela com vetor teste quase decrecente 40%: A linha te interesse analisada para este caso é a 16.*

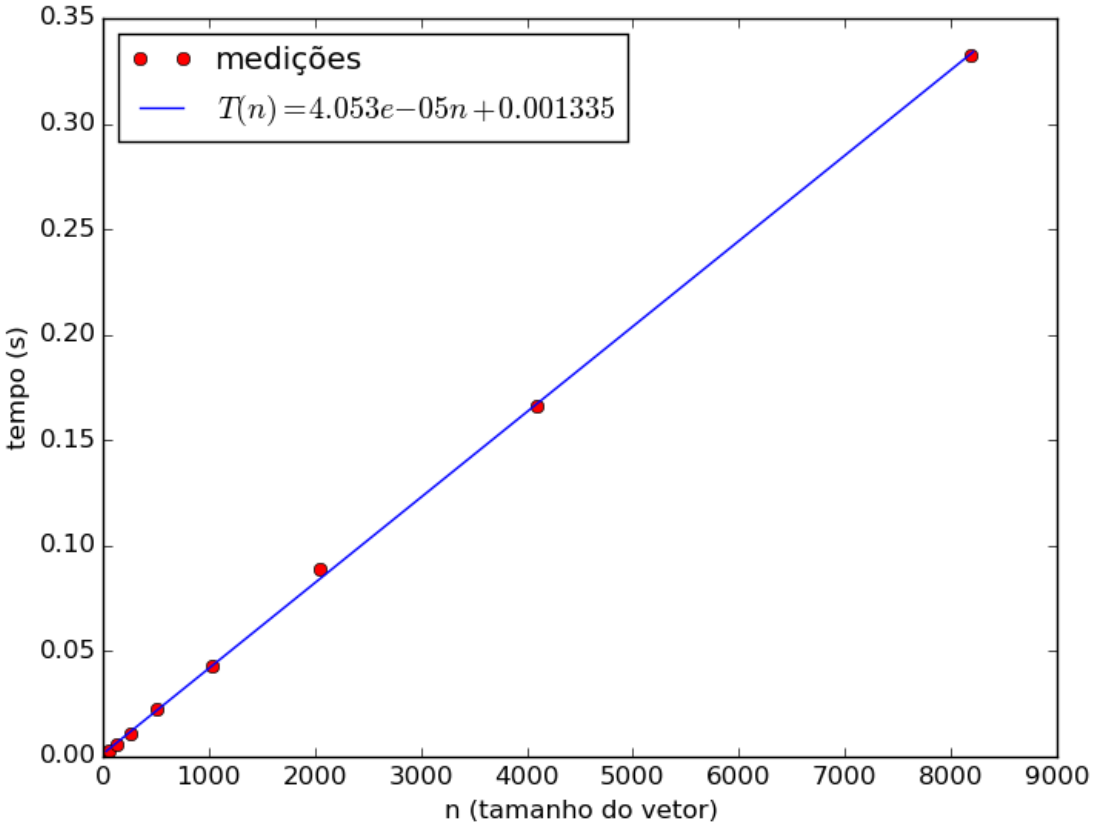

Figura 2.21: *Explique o gráfico: radixsortQuaseDecresc300.png*

n	comparações	tempo(s)
32	7	0.001490
64	7	0.002849
128	7	0.005595
256	7	0.010997
512	7	0.023139
1024	7	0.042961
2048	7	0.085150
4096	7	0.167020
8192	7	0.343861

Tabela 2.13: *Tabela com vetor teste quase decrecente 50%: A linha te interesse analisada para este caso é a 16.*

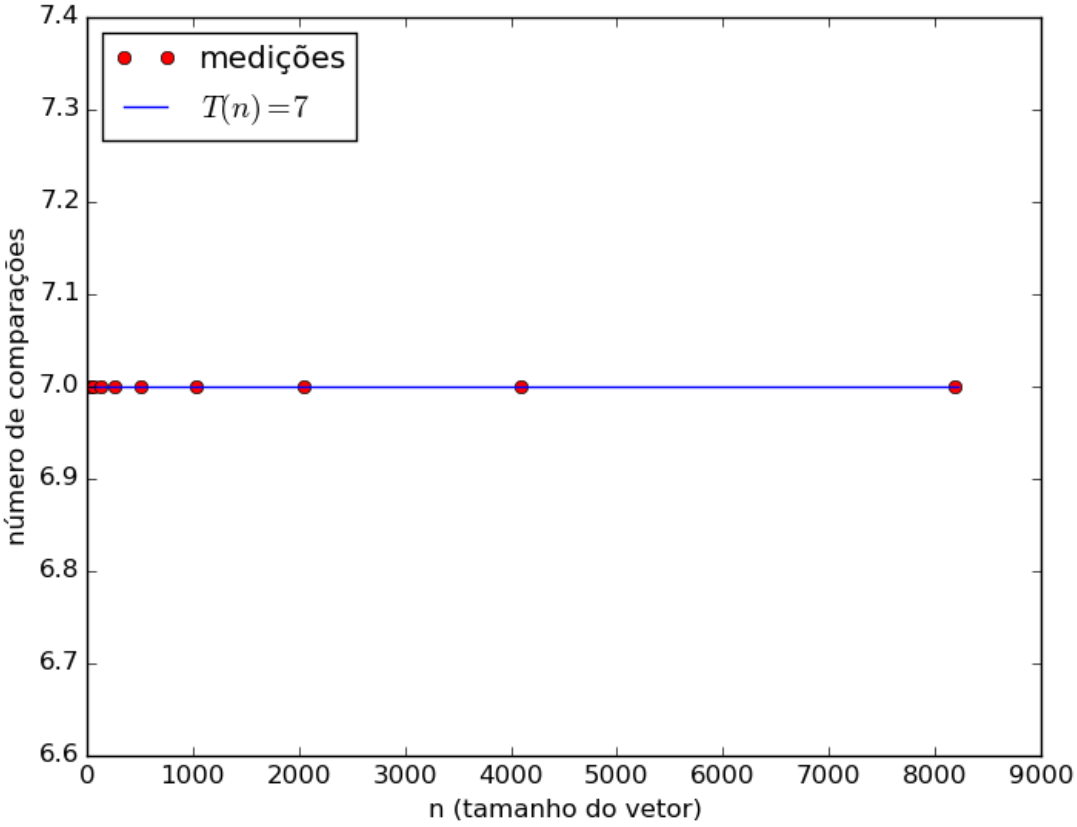

Figura 2.22: *Explique o gráfico: radixsortQuaseDecresc301.png*

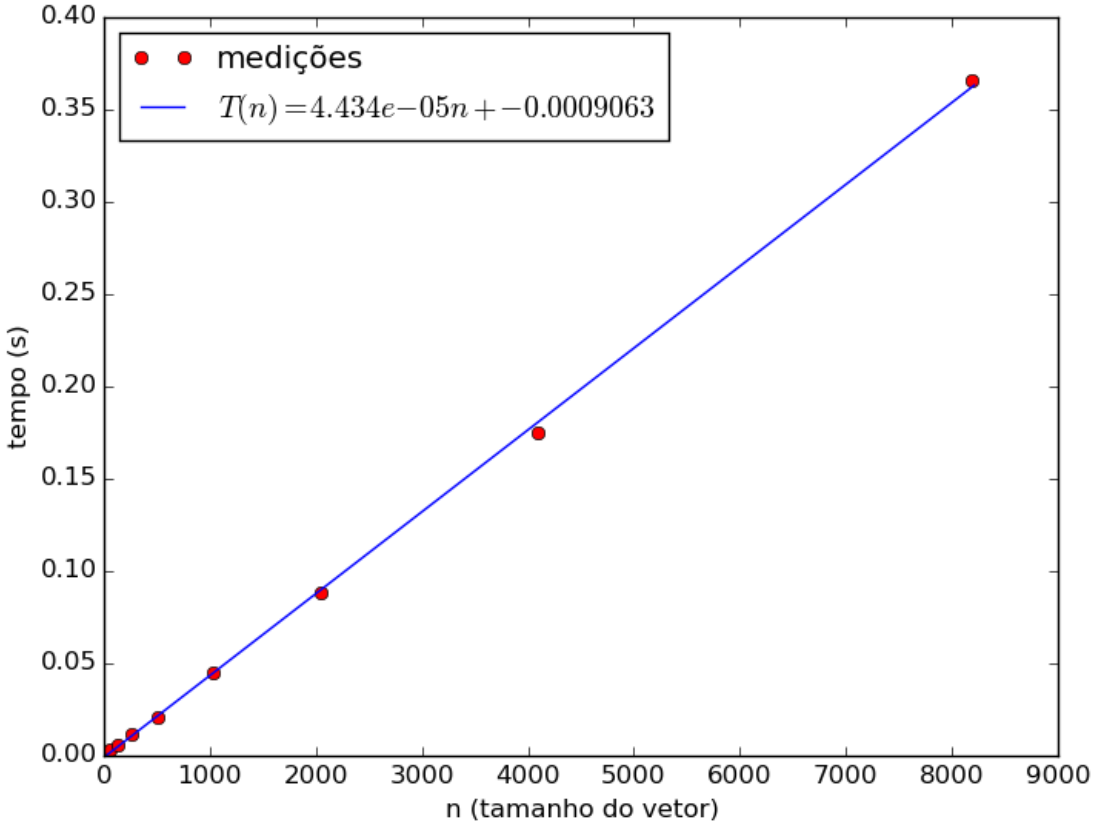

Figura 2.23: *Explique o gráfico: radixsortQuaseDecresc400.png*

2.1

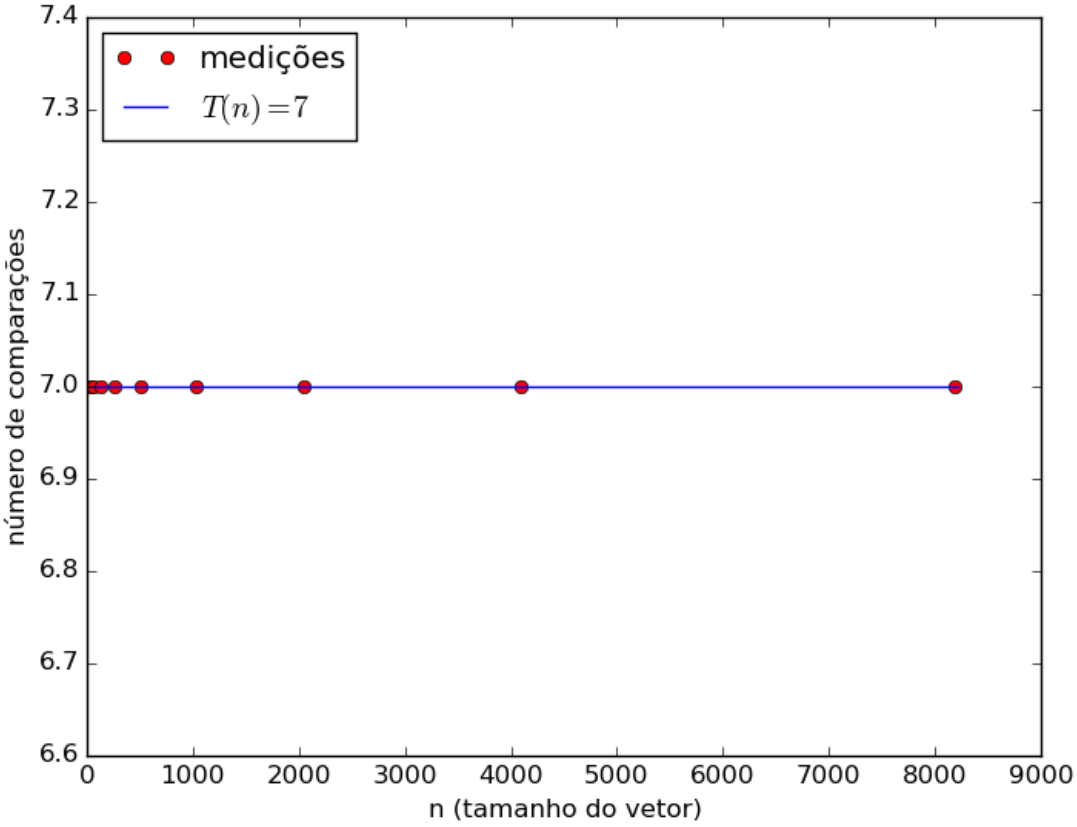

Figura 2.24: *Explique o gráfico: radixsortQuaseDecresc401.png*

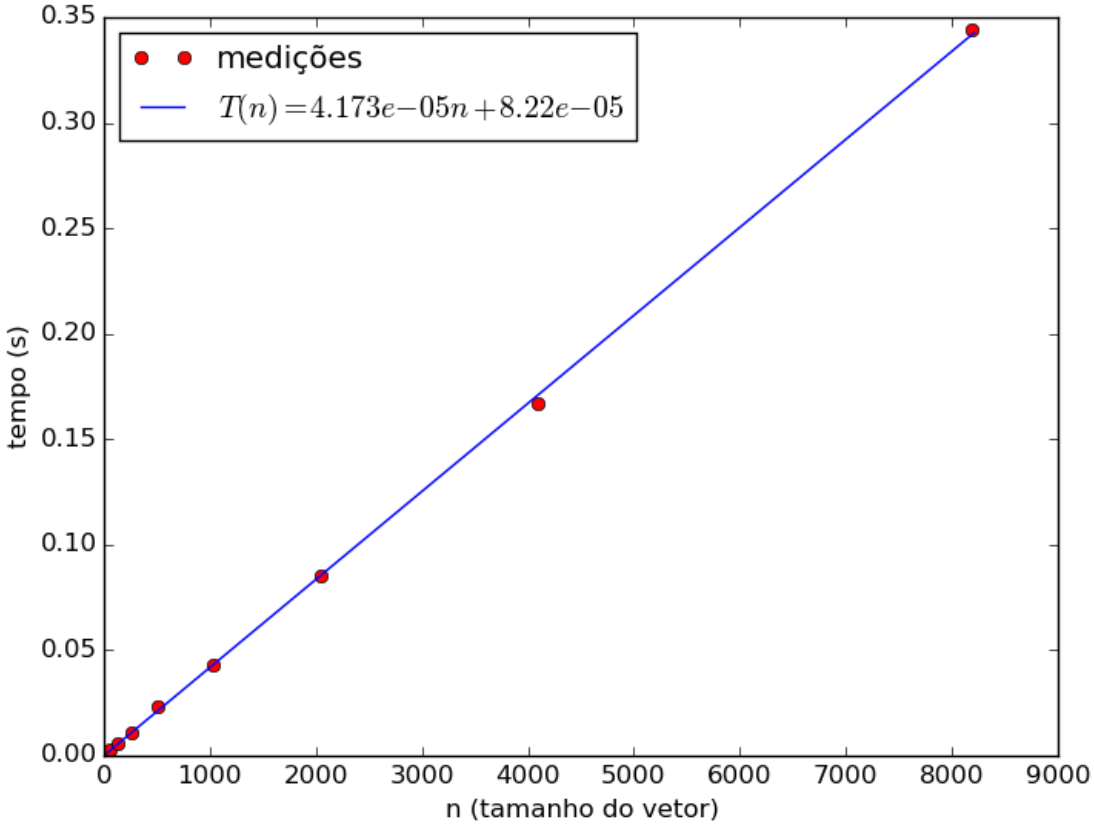

Figura 2.25: *Explique o gráfico: radixsortQuaseDecresc500.png*

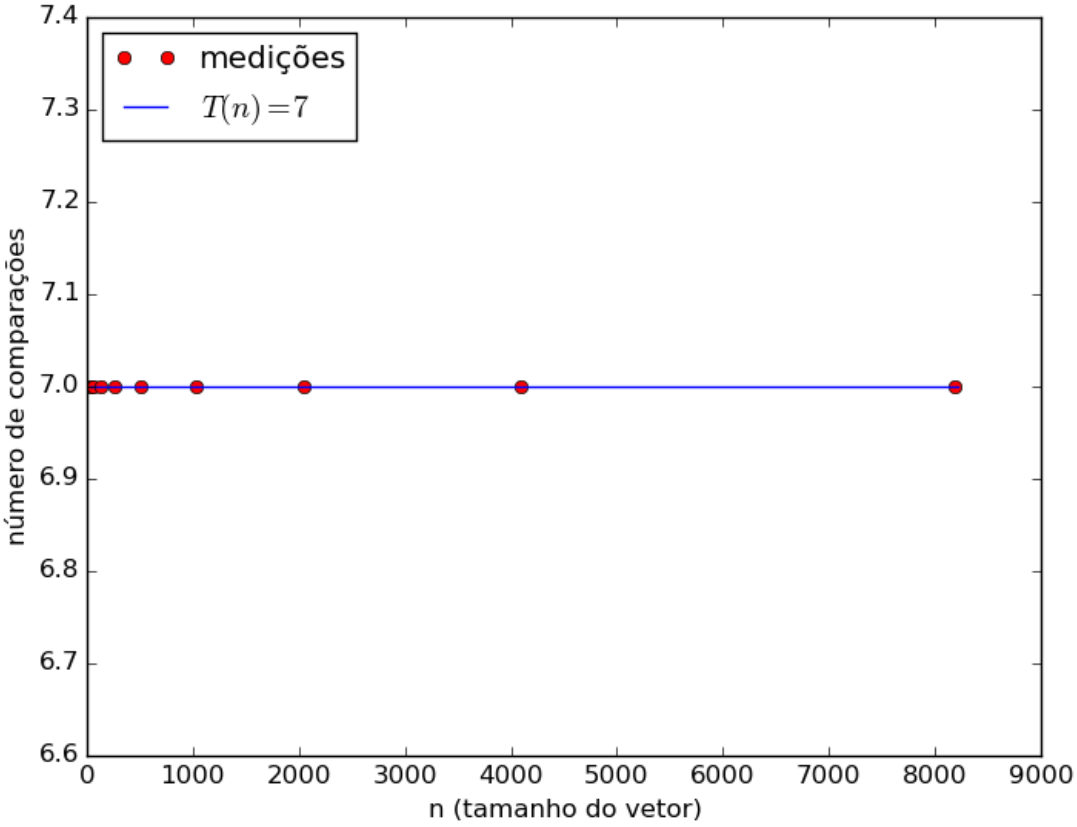

Figura 2.26: *Explique o gráfico: radixsortQuaseDecresc501.png*

Apêndice A

Arquivo ../radixsort/radixsort.py

Listagem A.1: ../radixsort/radixsort.py

```
@profile
def radixsort( aList ):
    RADIX = 4
    maxLength = False
    tmp , placement = -1, 1

    while not maxLength:
        maxLength = True
        # declare and initialize buckets
        buckets = [list() for _ in range( RADIX )]

        # split aList between lists
        for i in aList:
            tmp = i // placement
            buckets[int(int(tmp) % RADIX)].append( int(i) )
            if maxLength and tmp > 0:
                maxLength = False
        # empty lists into aList array
        a = 0
        for b in range( RADIX ):
            buck = buckets[b]
            for i in buck:
                aList[a] = i
                a += 1

        # move to next digit
        placement *= RADIX
    return aList
```

Apêndice B

Arquivo ../radixsort/ensaio.py

Listagem B.1: ../radixsort/ensaio.py

```
1  import numpy as np
2  import argparse
3
4  from radixsort import *
5
6  parser = argparse.ArgumentParser()
7  parser.add_argument("arq_vetor",
8                      help="nome do arquivo contendo o vetor de teste")
9  args = parser.parse_args()
10
11 # Lê o arquivo contendo o vetor e passado na linha de comando como um
12 # vetor do Numpy.
13
14 vet = np.loadtxt(args.arq_vetor)
15 radixsort(vet)
```