INSTITUTO INFNET

ESCOLA SUPERIOR DE TECNOLOGIA

GRADUAÇÃO EM ENGENHARIA DE SOFTWARE

Projeto de Bloco: Ciência da Computação

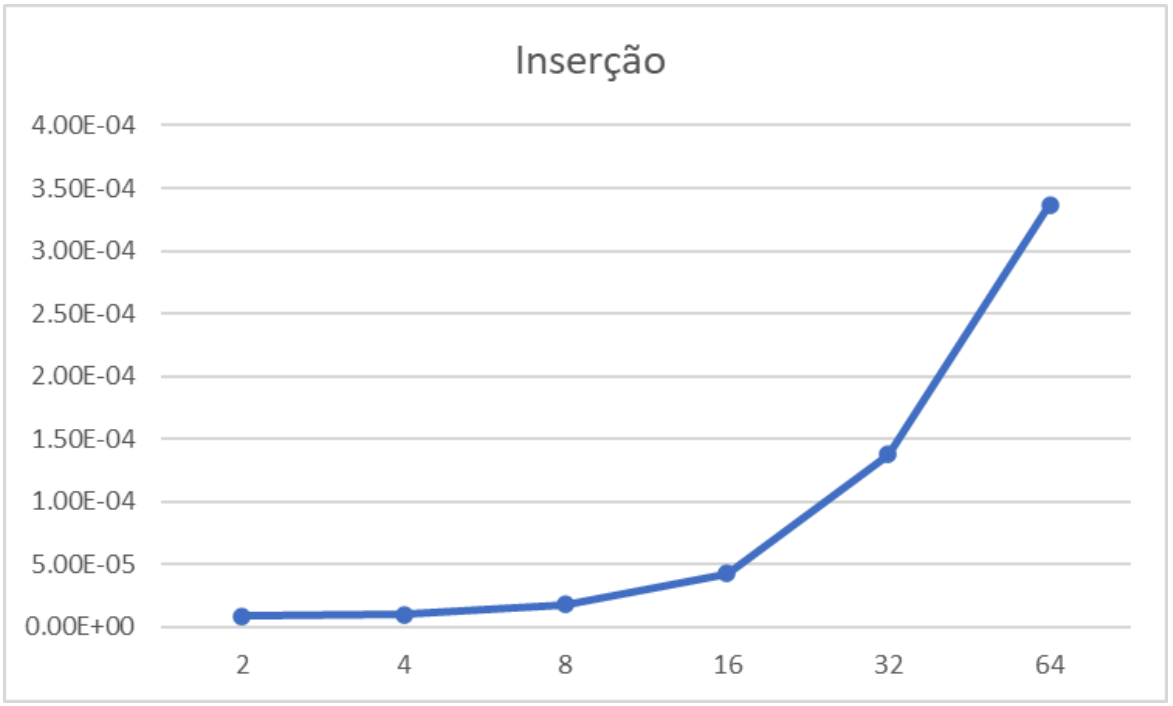
TP3

Daniel Gomes Lipkin

17 de mar. de 2025

1.1)

```
, 2 elementos
Inserção 8.899951353669167e-06 segundos
In order
1, 1, 2, Pre order
1, 1, 2, Post order
1, 2, 1,
 0.011071000015363097 segundos

, 4 elementos
Inserção 9.800074622035027e-06 segundos
In order
1, 2, 2, 3, 4, Pre order
2, 1, 2, 3, 4, Post order
2, 1, 4, 3, 2,
 0.016236199997365475 segundos

, 8 elementos
Inserção 1.810002140700817e-05 segundos
In order
1, 2, 3, 3, 4, 5, 6, 7, 8, Pre order
3, 1, 2, 3, 4, 5, 6, 7, 8, Post order
3, 2, 1, 8, 7, 6, 5, 4, 3,
 0.0320401000790298 segundos

, 16 elementos
Inserção 4.2499974370002747e-05 segundos
In order
1, 2, 3, 4, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, Pre order
4, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, Post order
4, 3, 2, 1, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4,
 0.06496679992415011 segundos

, 32 elementos
Inserção 0.00013759988360106945 segundos
In order
1, 2, 3, 4, 5, 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, Pre order
5, 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, Post order
5, 4, 3, 2, 1, 32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8,
7, 6, 5,
 0.12746119988150895 segundos

, 64 elementos
Inserção 0.0003370000049471855 segundos
In order
1, 2, 3, 4, 5, 6, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29
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,
, 59, 60, 61, 62, 63, 64, Pre order
6, 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
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,
, 59, 60, 61, 62, 63, 64, Post order
6, 5, 4, 3, 2, 1, 64, 63, 62, 61, 60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50, 49, 48, 47, 46, 45, 44, 43,
, 40, 39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15,
3, 12, 11, 10, 9, 8, 7, 6,
 0.32057500001974404 segundos
```



A inserção parece ser O(n^2), mas vemos que estamos considerando o fato de primeiro percorrer a lista de entradas e depois chamar a função de inserção para cada elemento, que no pior caso é O(n) para BSTs.
A função de inserção percorre toda a arvore recursivamente, indo para a esquerda ou a direita dependendo do valor até chegar a um nódulo sem filho que apropriaria o valor novo.

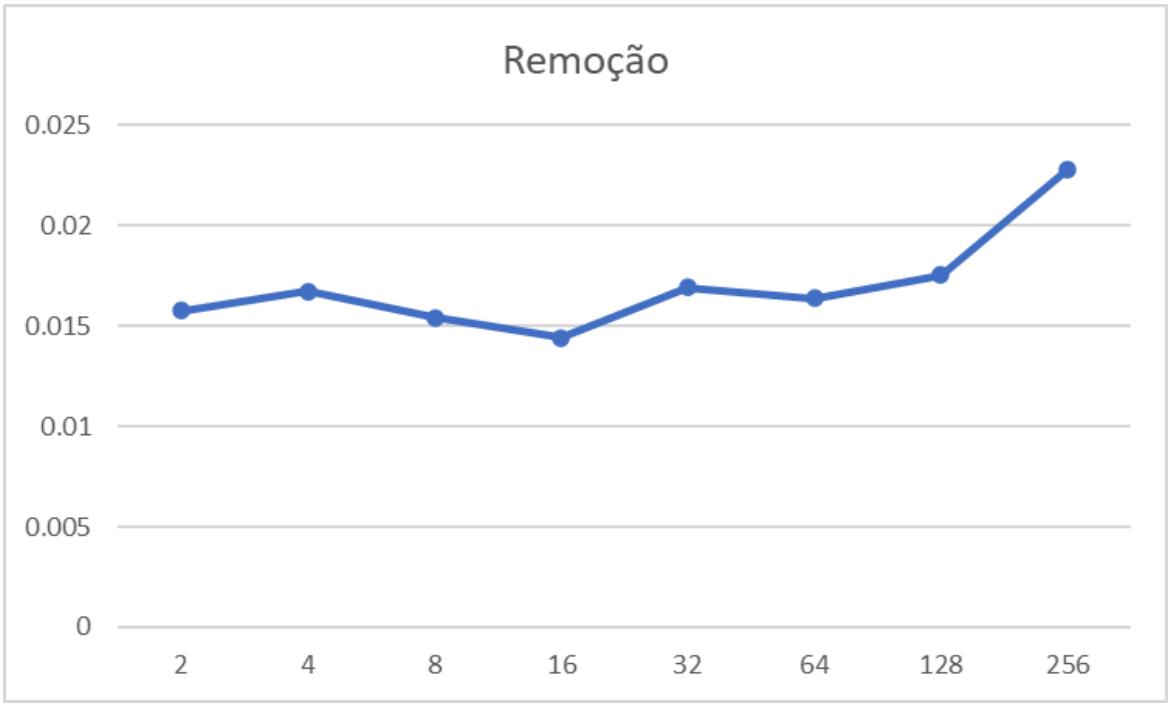
Os algoritmos de travessia são todos iguais em complexidade e parecem ser O(n^2) em complexidade de tempo.
As funções de travessia percorrem ambas a esquerda e a direita dos niveis da arvore, onde cada tipo de travessia só muda a ordem que é executado as recursões.

```
def inOrder(self, node):
    if node is not None:
        self.inOrder(node.node_l)
        print(node.val, end=", ")
        self.inOrder(node.node_r)

def preOrder(self, node):
    if node is not None:
        print(node.val, end=", ")
        self.preOrder(node.node_l)
        self.preOrder(node.node_r)

def postOrder(self, node):
    if node is not None:
        self.postOrder(node.node_l)
        self.postOrder(node.node_r)
        print(node.val, end=", ")
```

1.2)

```
, 2 elementos
1, 1, 2,
Removendo 1
0.01578649994917214 segundos
1, 2,


, 4 elementos
1, 2, 2, 3, 4,
Removendo 2
0.016715900041162968 segundos
1, 2, 3, 4,


, 8 elementos
1, 2, 3, 3, 4, 5, 6, 7, 8,
Removendo 4
0.015407599974423647 segundos
1, 2, 3, 3, 5, 6, 7, 8,


, 16 elementos
1, 2, 3, 4, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
Removendo 8
0.014423999935388565 segundos
1, 2, 3, 4, 4, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 16,


, 32 elementos
1, 2, 3, 4, 5, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26
31, 32,
Removendo 16
0.0169214999768883 segundos
1, 2, 3, 4, 5, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27
32,


, 64 elementos
1, 2, 3, 4, 5, 6, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26
31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 5
, 59, 60, 61, 62, 63, 64,
Removendo 32
0.016398699954152107 segundos
1, 2, 3, 4, 5, 6, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26
31, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 5
, 60, 61, 62, 63, 64,
```

```
, 128 elementos
1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 2
31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48,
, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76
6, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 1
10, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 12
Removendo 64
0.01754379994235933 segundos
1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 2
31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48,
, 59, 60, 61, 62, 63, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77
7, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103,
111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 1



, 256 elementos
1, 2, 3, 4, 5, 6, 7, 8, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 2
31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48,
, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76
6, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 1
10, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 12
 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 1
1, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164,
172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 1
2, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205,
213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 2
3, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246,
254, 255, 256,
Removendo 128
0.022779999999329448 segundos
```



Começa com um custo mais alto com menos elementos mas parece seguir um padrão linear O(n) o'que está coerente com o tempo visto em outras fontes.
O algoritimo percorre toda a arvore até achar o valor a remover, depois recursivamente troca o lugar com os filhos, começando com o menor valor da sub-arvore e escalando de volta até o filho original da direita.

1.3)

```
, 2 elementos
Procurando elemento 1
1
0.016917199827730656 segundos

, 4 elementos
Procurando elemento 2
2
0.016369800083339214 segundos

, 8 elementos
Procurando elemento 4
4
0.01677490002475679 segundos

, 16 elementos
Procurando elemento 8
8
0.01619850005954504 segundos

, 32 elementos
Procurando elemento 16
16
0.02471370017156005 segundos

, 64 elementos
Procurando elemento 32
32
0.016177500132471323 segundos

, 128 elementos
Procurando elemento 64
64
0.01681159995496273 segundos

, 256 elementos
Procurando elemento 128
128
0.019568199990317225 segundos
```

Incrivelmente a pesquisa dura somente O(1) se fomos considerar esses resultados, mas é possivel observar que com 128 e 256 elementos o tempo começa aumentar aos poucos, mas nunca escalando de forma linear, sugerindo O(log n). A complexidade de

espaço se fosse medida idealmente seria O(nivel da arvore) pois cria uma instancia de recursão para cada elemento em cada nivel da arvore até chegar ao fim ou o elemento. O algoritmo busca recursivamente sub-arvores da esquerda e direita até retornar o valor.

1.4)

```
, 2 elementos
Arvore válida?
False
0.0068252000492066145 segundos
Arvore invalidada válida?
False
0.00773399998433888 segundos

, 4 elementos
Arvore válida?
True
0.007571500027552247 segundos
Arvore invalidada válida?
False
0.006561699789017439 segundos

, 8 elementos
Arvore válida?
True
0.009081500116735697 segundos
Arvore invalidada válida?
False
0.008091000141575933 segundos

, 16 elementos
Arvore válida?
True
0.009005800122395158 segundos
Arvore invalidada válida?
False
0.007666700053960085 segundos

, 32 elementos
Arvore válida?
True
0.006942099891602993 segundos
Arvore invalidada válida?
False
0.00717880018055439 segundos

, 64 elementos
Arvore válida?
True
0.007833200041204691 segundos
Arvore invalidada válida?
False
```

Denovo vemos o mesmo caso que ocorreu na função de busca.

2.1

```
Soma Paralela vs Linear

 10001 elementos
0.002096200129017234 segundos -  Soma paralela
50005000
0.0003420999273657799 segundos -  Soma linear
50005000

 100001 elementos
0.0022305999882519245 segundos -  Soma paralela
5000050000
0.0036042998544871807 segundos -  Soma linear
5000050000

 1000001 elementos
0.03586840000934899 segundos -  Soma paralela
500000500000
0.03763739997521043 segundos -  Soma linear
500000500000

 10000001 elementos
0.37679379992187023 segundos -  Soma paralela
50000005000000
0.41180179989896715 segundos -  Soma linear
50000005000000
```

O tempo parece ser linear para ambos pois ao multiplicar os elementos por 10, o tempo faz o mesmo. A paralela demonstra uma branda superioridade comparada a linear. A quantidade de threads e partições da entrada na paralela depende da quantidade de cores do CPU.

2.2

Matriz
50 x 50
0.02410140004940331 segundos - Matriz Paralela

Squeezed text (197 lines).

Squeezed text (197 lines).

0.1015188000164926 segundos - Matriz Linear
100 x 100
0.14750790013931692 segundos - Matriz Paralela

Squeezed text (768 lines).

Squeezed text (768 lines).

0.6148151999805123 segundos - Matriz Linear
150 x 150
0.4974316000007093 segundos - Matriz Paralela

Squeezed text (1714 lines).

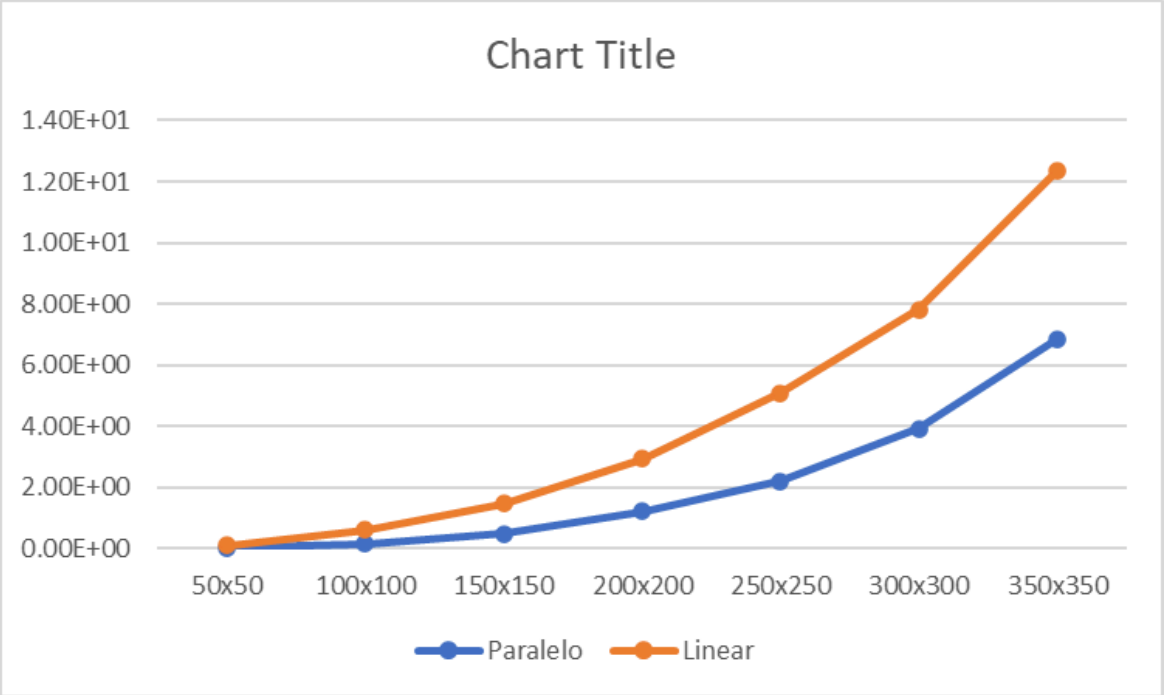
Squeezed text (1714 lines).

1.5182483000680804 segundos - Matriz Linear
200 x 200
1.144457699963823 segundos - Matriz Paralela

Squeezed text (3036 lines).

Squeezed text (3036 lines).

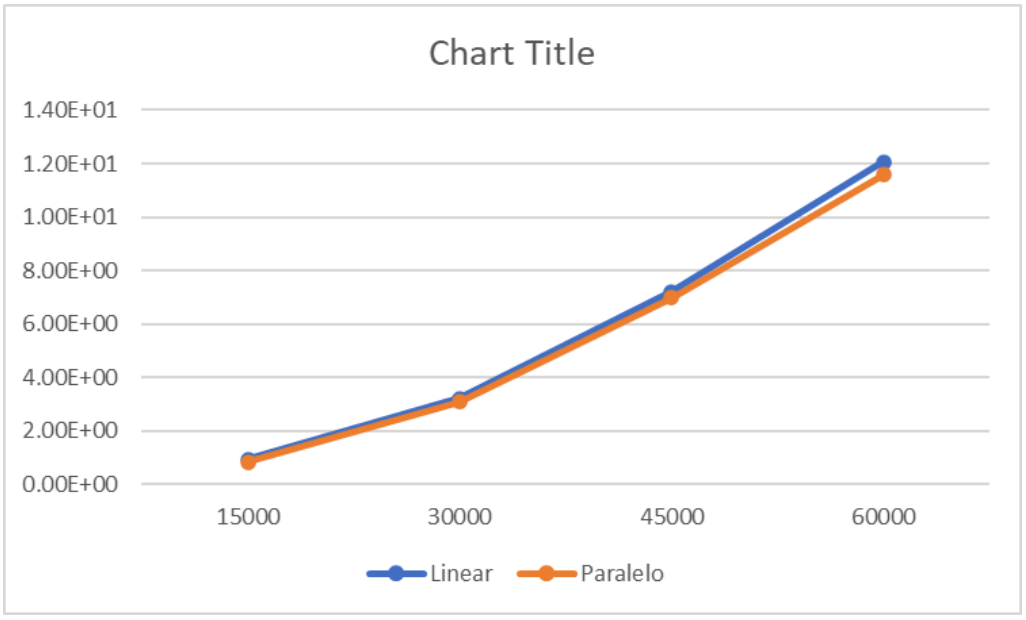
2.8089248999021947 segundos - Matriz Linear



O algoritimo que segue as etapas basicas de multiplicação de matriz chega a demorar O(n^3), pois tamanho da matriz (N*N) * a soma dos elementos multiplicados (N). Ambas parecem ter tempos exponenciais. Dividindo as matrizes em sub-matrizes na versão paralela tornou-a bem melhor que a linear.

2.3

```
Primos

 15000 primos
1755
Linear 15001 - 0.8060165999922901 segundos
Paralelo 15000 - 0.8101033999118954 segundos
1755

 30000 primos
3246
Linear 30001 - 3.17742699990049 segundos
Paralelo 30000 - 3.053691399982199 segundos
3246

 45000 primos
4676
Linear 45001 - 6.918279200093821 segundos
Paralelo 45000 - 6.883870800025761 segundos
4676

 60000 primos
6058
Linear 60001 - 12.0479767001234 segundos
Paralelo 60000 - 11.586665699956939 segundos
6058
```

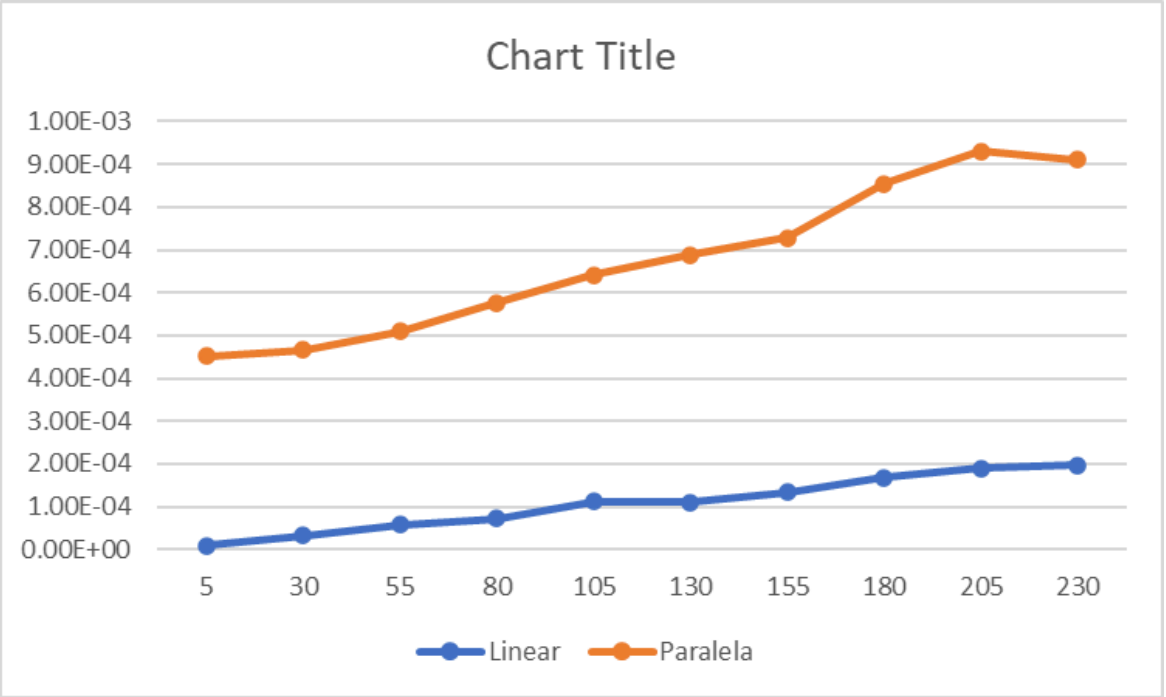


Para cada numero até n, todos numeros posteriores dividem o numero para verificar se é primo se nenhuma divisão ter resto 0.
Ambos parecem ter O(n) de complexidade de tempo, com a paralela tendo uma diferença ligeira de desempenho.

3.1

```
Arvore paralela

Buscando 5
9.600073099136353e-06  segundos - Linear
0.0004517000634223223  segundos - Paralela

Buscando 30
3.310013562440872e-05  segundos - Linear
0.00046559982001781464  segundos - Paralela

Buscando 55
5.8300094678997993e-05  segundos - Linear
0.0005099999252706766  segundos - Paralela

Buscando 80
7.24999699741602e-05  segundos - Linear
0.0005761000793427229  segundos - Paralela

Buscando 105
0.00011230004020035267  segundos - Linear
0.0006413001101464033  segundos - Paralela

Buscando 130
0.00011019990779459476  segundos - Linear
0.0006883000023663044  segundos - Paralela

Buscando 155
0.00013320008292794228  segundos - Linear
0.0007279000710695982  segundos - Paralela

Buscando 180
0.0001674999948590994  segundos - Linear
0.0008551001083105803  segundos - Paralela

Buscando 205
0.00018950016237795353  segundos - Linear
0.0009296000935137272  segundos - Paralela

Buscando 230
0.00019600009545683860  segundos - Linear
0.0009101999457925558  segundos - Paralela
```

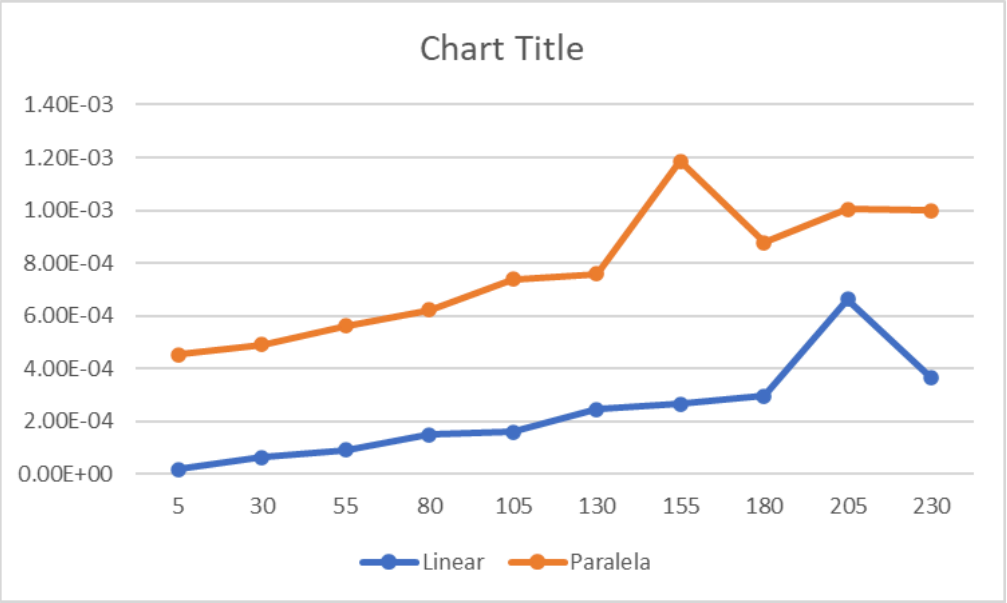
O valor buscado é metade da quantidade total de elementos. Para cada sub-arvore, é assinalado um thread com a função de busca paralela que vai recursivamente criando mais threads para as outras sub-arvores.



Vemos que ambos são O(n) e é possível teorizar que o processo de paralelização que ocorre no código deixa um pouco mais demorado.

3.2

```
Buscando (DFS) 5
1.7900019884109497e-05  segundos - Linear
0.00045219995081424713  segundos - Paralela

Buscando (DFS) 30
6.200000643730164e-05  segundos - Linear
0.0004913001321256161  segundos - Paralela

Buscando (DFS) 55
9.099999442696571e-05  segundos - Linear
0.0005620999727398157  segundos - Paralela

Buscando (DFS) 80
0.00015020021237432957  segundos - Linear
0.0006226000841706991  segundos - Paralela

Buscando (DFS) 105
0.0001596000511199236  segundos - Linear
0.0007395998109132051  segundos - Paralela

Buscando (DFS) 130
0.00024500000290572643  segundos - Linear
0.0007599000819027424  segundos - Paralela

Buscando (DFS) 155
0.00026510003954172134  segundos - Linear
0.0011855000630021095  segundos - Paralela

Buscando (DFS) 180
0.0002949000336229801  segundos - Linear
0.0008769000414758921  segundos - Paralela

Buscando (DFS) 205
0.0006625000387430191  segundos - Linear
0.0010041000787168741  segundos - Paralela

Buscando (DFS) 230
0.00036409986205399036  segundos - Linear
0.0009979000315070152  segundos - Paralela
```



O DFS percorre as sub-arvores direita e esquerda até chegar ao um beco sem fim, retornando ao nodulo anterior e fazendo a mesma coisa recursivamente. A versão paralela assinala uma thread para cada sub-arvore.
Parece igual a anterior, se não fosse pelo pico que apareceu em ambos e em entradas diferentes. Isso porque o valor foi achado prematuramente.

3.3

```
Valor Maximo 5
4
0.005157300038263202  segundos - Linear
4
0.006006599869579077  segundos - Paralela

Valor Maximo 25
24
0.00510590011253953  segundos - Linear
24
0.005396400112658739  segundos - Paralela

Valor Maximo 125
124
0.004616399994120002  segundos - Linear
124
0.005339399911463261  segundos - Paralela

Valor Maximo 625
624
0.005130999954417348  segundos - Linear
624
0.006373199867084622  segundos - Paralela
Grupo 4
```

O tempo parece ser O(1), oque coincide com o esperado. Ele executa a função max() para cada elemento na pilha de recursão. A complexidade de espaço sera O(niveis da arvore) pois o valor maximo obrigatoriamente estara no ultimo nivel.

4.1

```
def isInPrefix(ip, prefix):
    return ipmod.ip_network(ip, strict=False).prefixlen == prefix
```

```
192.168.1.1 em 24?
32
False
```

```
192.168.1.1/24 em 24?
24
True
```

Uma mascara de rede chamado prefixo é aplicado ao IP com uma operação bitwise para ser identificado e usado em sub-redes. Nesse exemplo a mascara possui 24 bits ativos. Com uma operação AND é possivel obter o endereço da sub-rede.

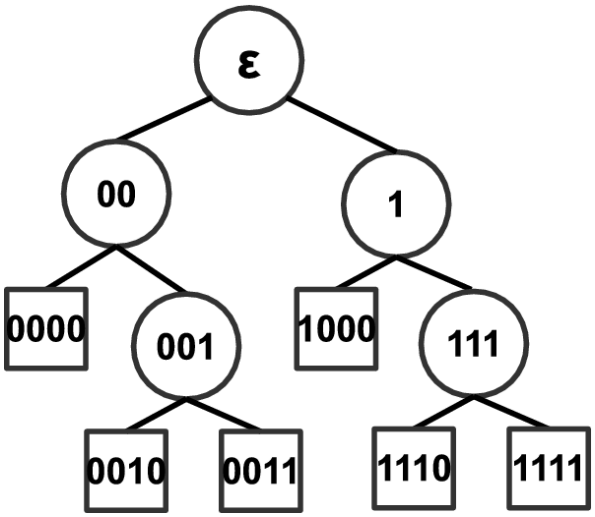
11000000.10101000.00000001.00000001 (192.168.1.1)
11111111.11111111.11111111.00000000 (255.255.255.0)

A parte com zeros da mascara indica os identificador do host, que nesse caso seria o final do nosso IP, mas como não especificamos o prefixo, ele pode pertencer tanto quanto para esse prefixo quanto outros. Por padrão ele coloca o prefixo 32 bits onde todos o bits da mascara estão ativos portanto somente o IP especificado seria o host dessa sub-rede. Mas quando especificamos o prefixo no segundo exemplo, ele retorna verdadeiro.

4.2

```
#Ex 4.2
print("\nIPV4 Trie")
trie = IPTrie()
trie.insert("192.168.1.0/24")
trie.insert("10.0.0.0/24")
```

```
IPV4 Trie
Buscando 192.168.1.100
24
Buscando 10.0.0.1
24
Buscando 172.16.0.1
None
```



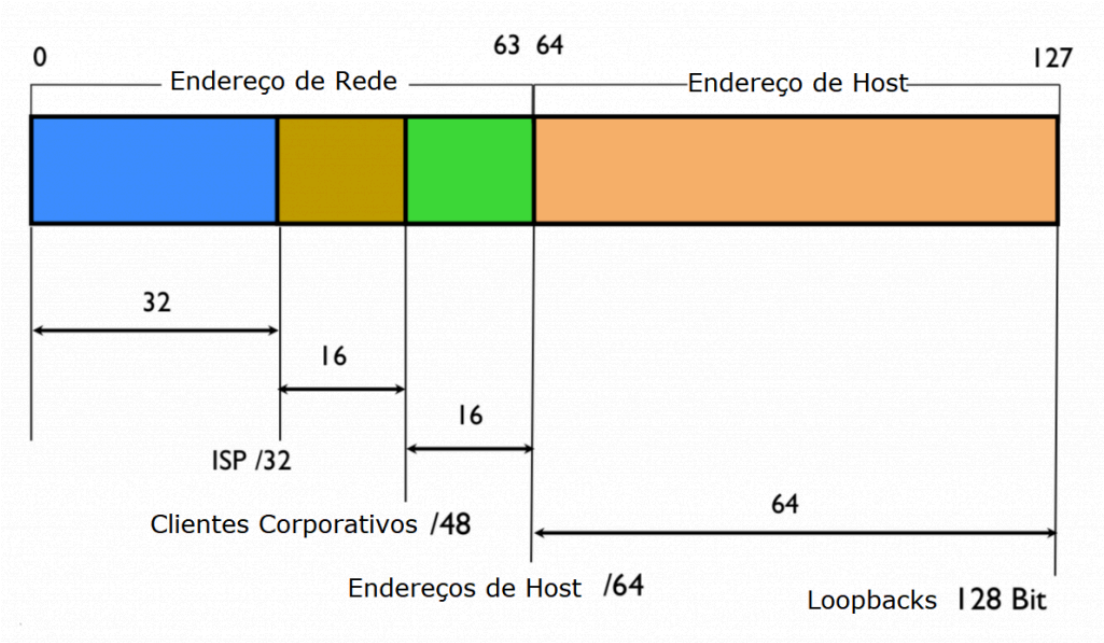
https://www.researchgate.net/figure/An-example-of-a-Patricia-trie-Leaves-are-represented-by-squares-and-internal-nodes-are_fig1_330855487

Implementamos o equivalente de uma patricia Trie, inserindo o endereço da sub-rede de uma entrada nova em formato de bits. Onde os bits tiverem um desvio de uma entrada para outro, o nodulo da arvore é dividido para cada entrada.
A função de busca tenta atravessar a trie dado um IP, e opcionalmente uma sub-rede, para achar o prefixo mais aproximado dele.
Aqui o tempo de busca, dada a implementação, pode variar linearmente com a quantidade elementos em um nivel da arvore e consequentemente com a quantidade de níveis na arvore.

4.3

```
print("\nIPV6 Trie")
trie = IPTrie(ipv6=True)
trie.insert("2001:0db8:85a3::/48")
trie.insert("2001:0db8:85a3:0000:0000:8a2e:0370:7334/64")
trie.insert("2606:4700:4700::1111/128")
trie.insert("::1/128")
trie.insert("fc00::/7")
trie.insert("fe80::/10")
```

```
IPV6 Trie
Buscando 2001:db8::1
None
Buscando 2001:0db8:85a3:0000:0000:8a2e:0370:7334
64
Buscando 2606:4700:4700::1111
128
Buscando ::1
128
Buscando fd12:3456:789a::1
7
Buscando fe80::1234
10
```

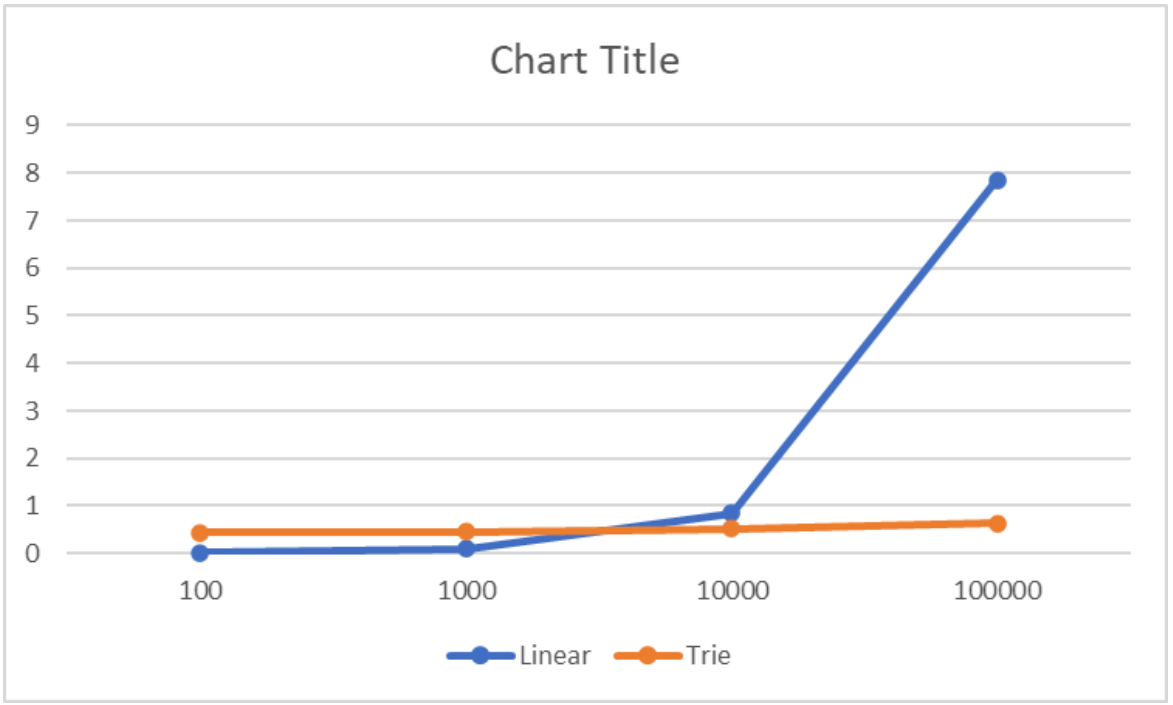


https://www.dltec.com.br/blog/redes/plano-de-enderecamento-ipv6/

A diferença entre o IPV6 e o IPV4 no aspecto de sub-redes é que inves da mascara conter 32 bits, o IPV6 contem 128 bits e é representado hexadecimalmente, alem de que inves de usar mascaras, a subrede é imbutida no proprio IP do host.

4.4

```
Linear vs Trie
Buscando 192.168.1.55...

 100 prefixos
Linear: None - 0.0010526999831199646 segundos
Trie: None - 4.3400097638368607e-05 segundos

 1000 prefixos
Linear: None - 0.009363000048324466 segundos
Trie: None - 4.5800115913152695e-05 segundos

 10000 prefixos
Linear: 192.213.23.115/9 - 0.0843021001201123 segundos
Trie: 9 - 5.179992876946926e-05 segundos

 100000 prefixos
Linear: 192.184.115.254/11 - 0.7854264001362026 segundos
Trie: 11 - 6.320001557469368e-05 segundos
```



Forma gerados n prefixos IPV4 aleatoriamente.
Multiplicando os valores para alinharem melhor no grafico, o linear, por mais que se chame busca linear, parece ter um aumento exponencial enquanto a busca em Trie parece ser O(n) ou O(log n) por mais minúsculo que seja a mudança no tempo, com o tempo variando baseado na quantidade de elementos por nivel da arvore e o total de niveis da arvore.