

AED1 - Aula 06

Crescimento de funções, busca sequencial e binária em vetores

"Busca binária está para algoritmos assim como a roda está para mecânica: ela é simples, elegante e imensamente importante"

- U. Manber, Introduction to Algorithms: a Creative Approach, 1989.

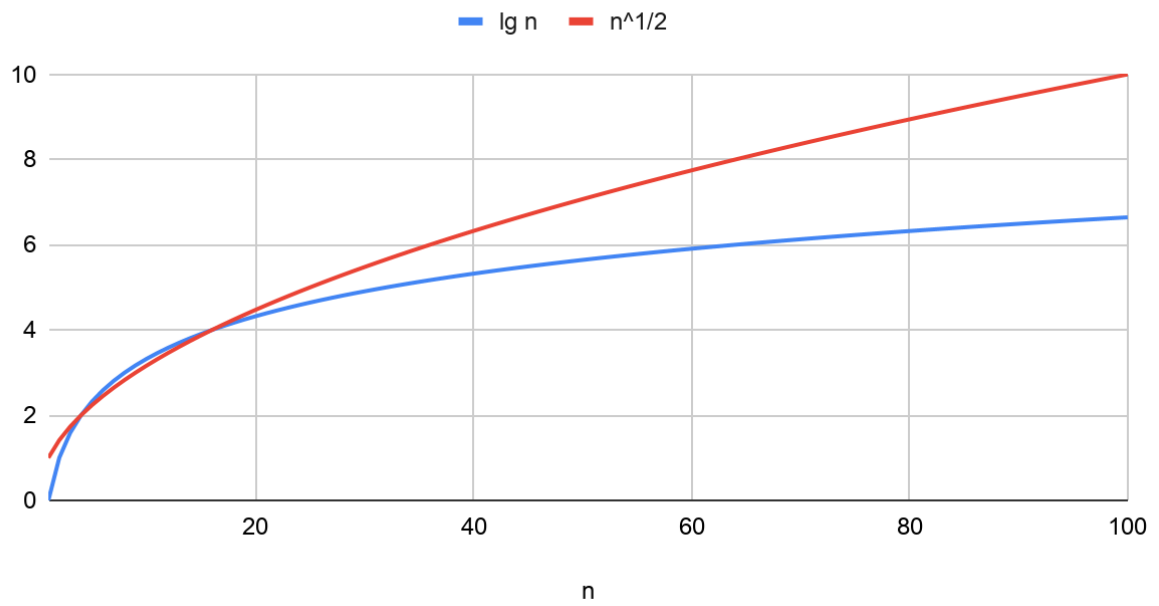
Crescimento de funções

n	10^3	10^6	10^9
$\log_2 n$	10	20	30
$n^{1/2}$	32	10^3	$3 \cdot 10^4$
n	10^3	10^6	10^9
$n \log_2 n$	10^4	$2 \cdot 10^7$	$3 \cdot 10^{10}$
n^2	10^6	10^{12}	10^{18}
n^3	10^9	10^{18}	10^{27}
2^n	10^{300}	10^{300000}	$10^{(3 \cdot 10^8)}$

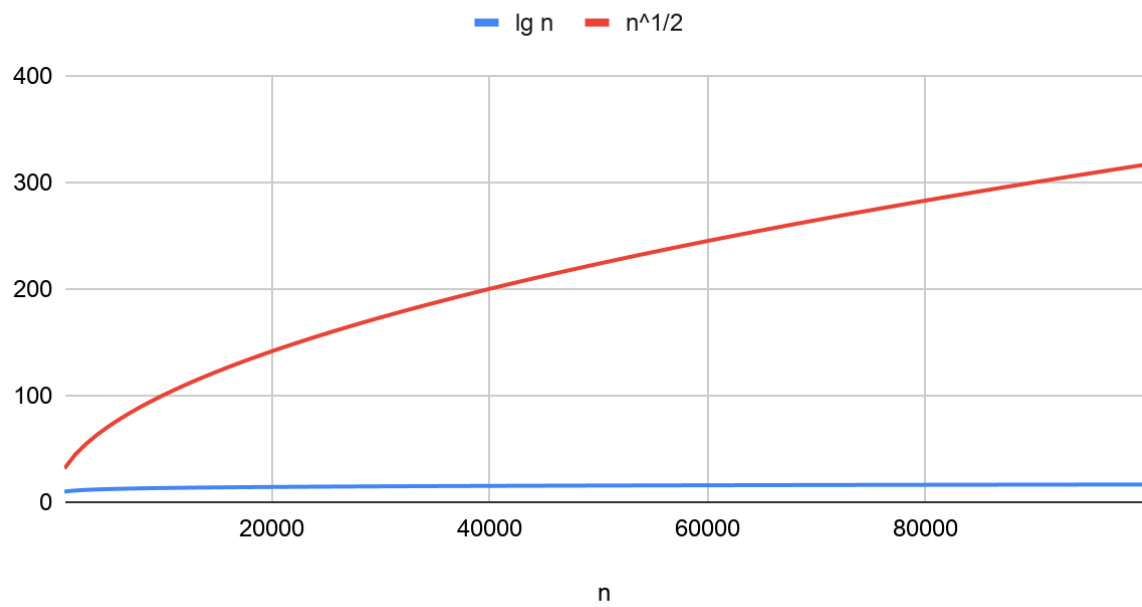
Interpretação temporal considerando um computador de 1GHz

n	10^3	10^6	10^9
$\log_2 n$	$\ll 1s$	$\ll 1s$	$\ll 1s$
$n^{1/2}$	$\ll 1s$	$\ll 1s$	$\ll 1s$
n	$\ll 1s$	$\ll 1s$	1s
$n \log_2 n$	$\ll 1s$	$\ll 1s$	30s
n^2	$\ll 1s$	16 min	31 anos
n^3	1s	31 anos	31709791 milênios
2^n	esquece...		

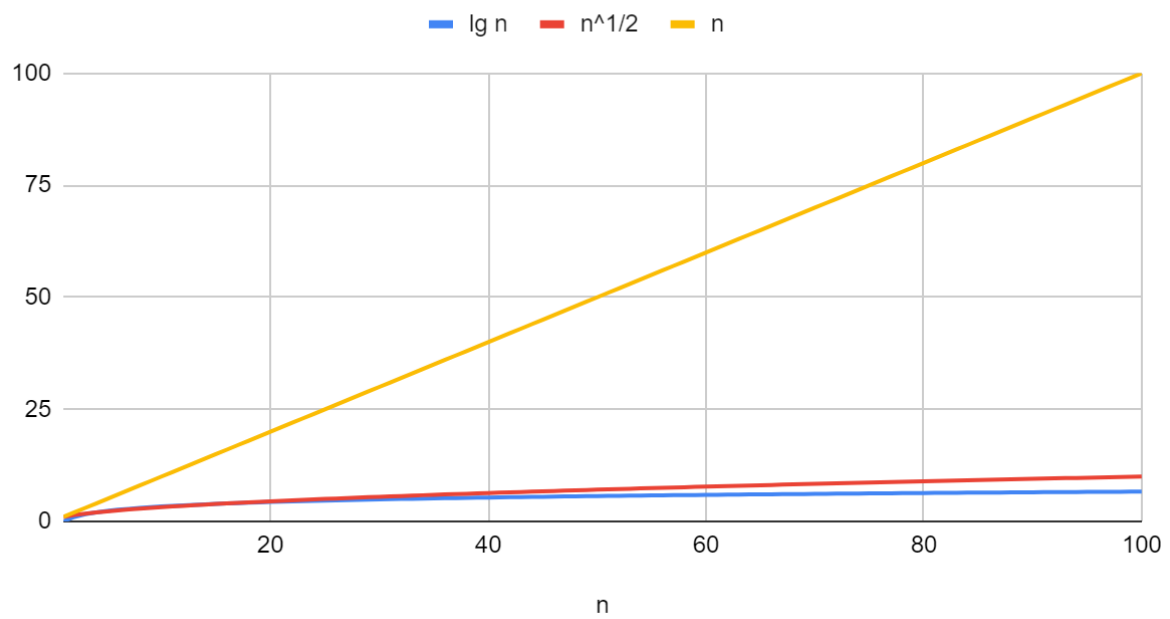
$\lg n \in n^{1/2}$



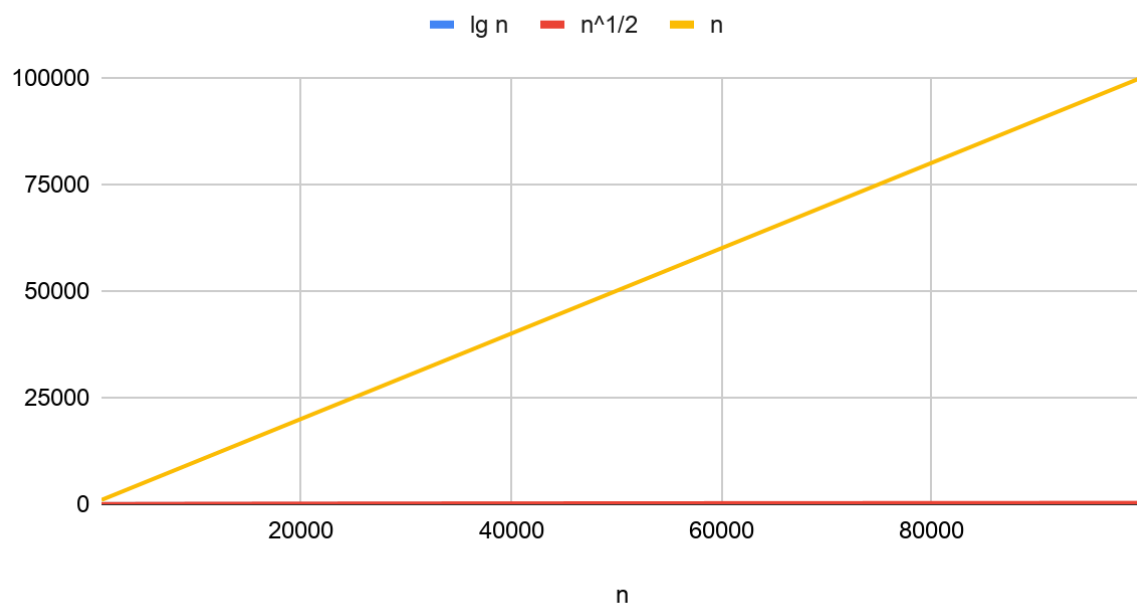
$\lg n \in n^{1/2}$



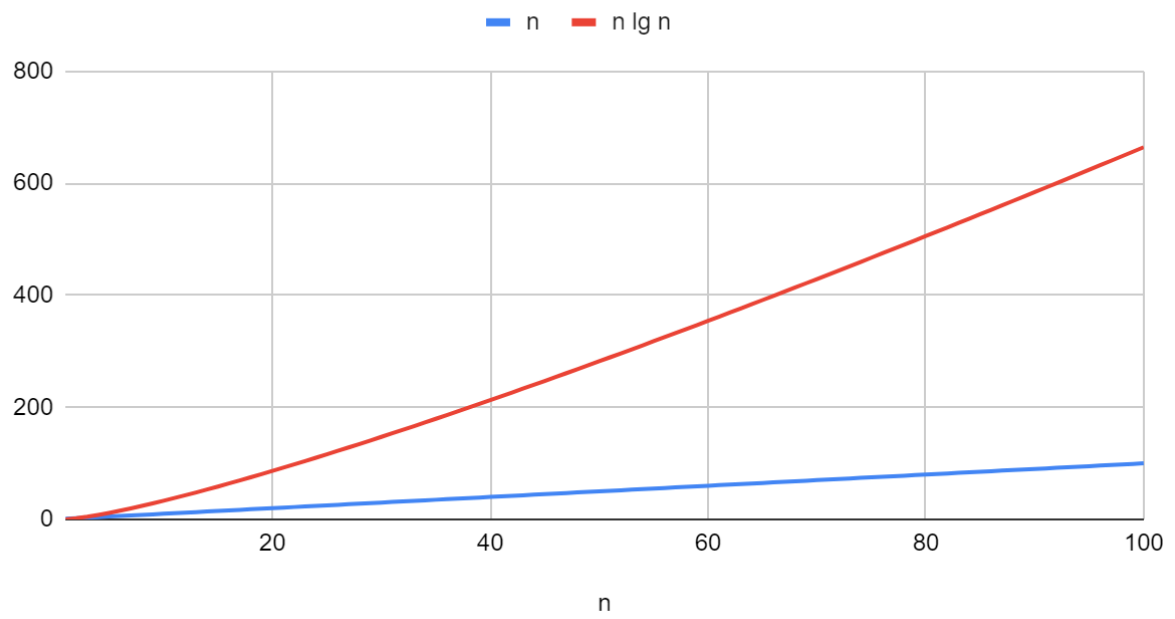
$\lg n$, $n^{1/2}$ e n



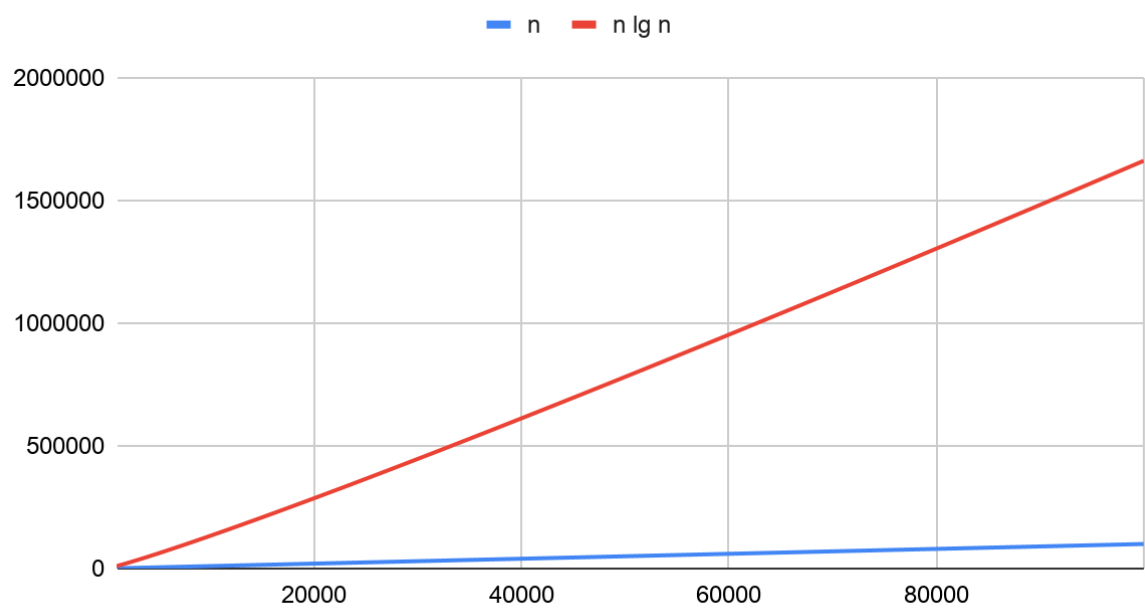
$\lg n$, $n^{1/2}$ e n



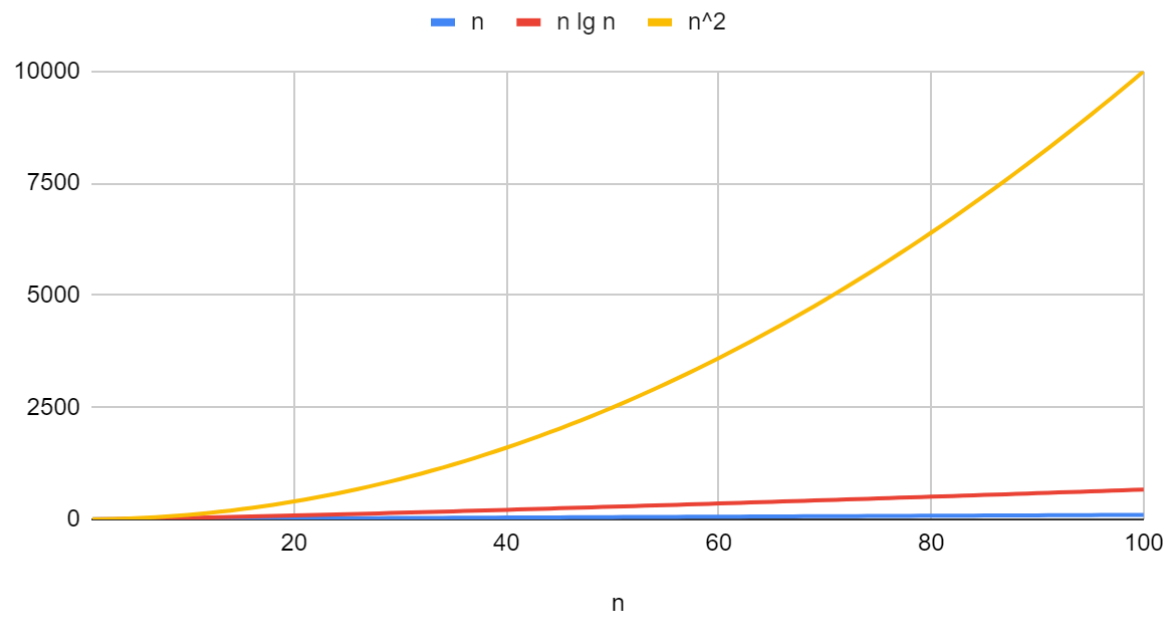
n $n \lg n$



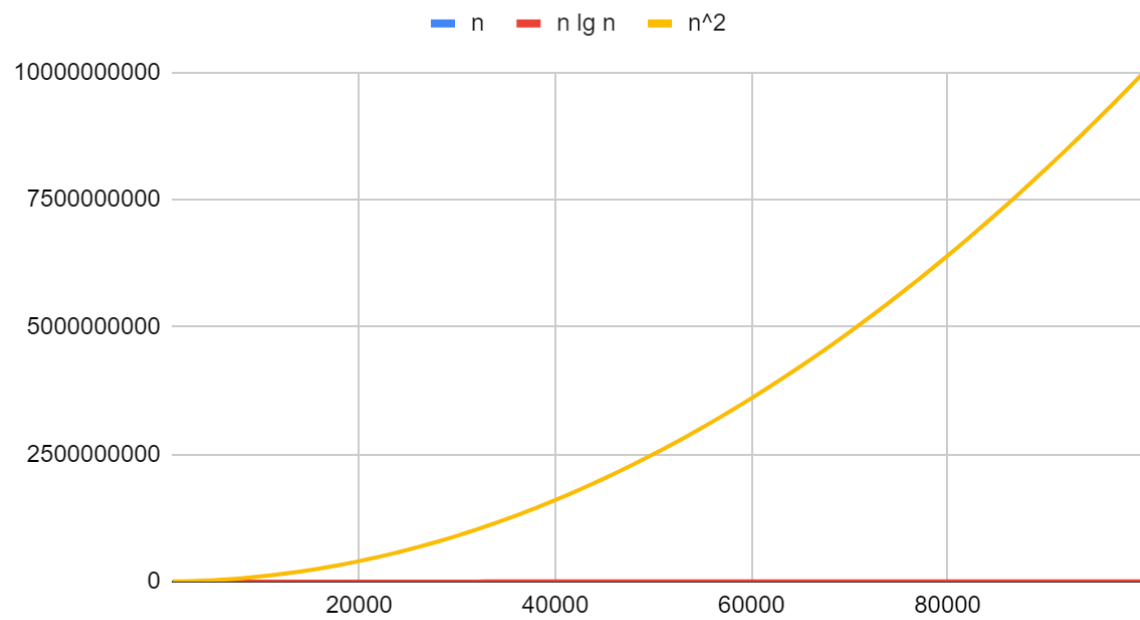
n $n \lg n$



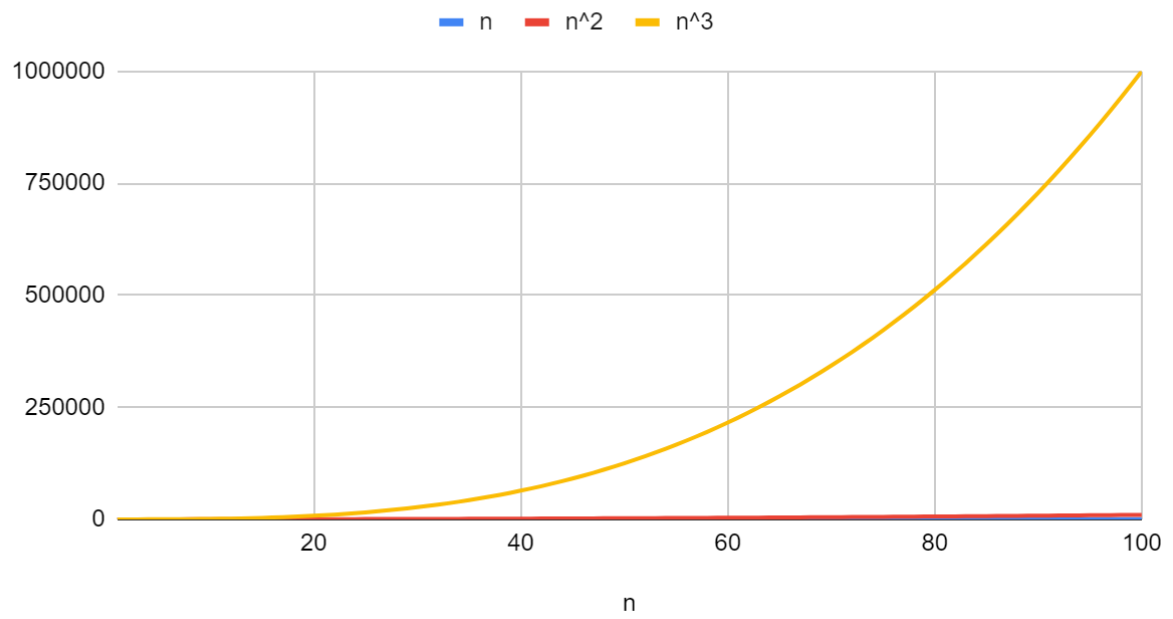
$n, n \lg n \in n^2$



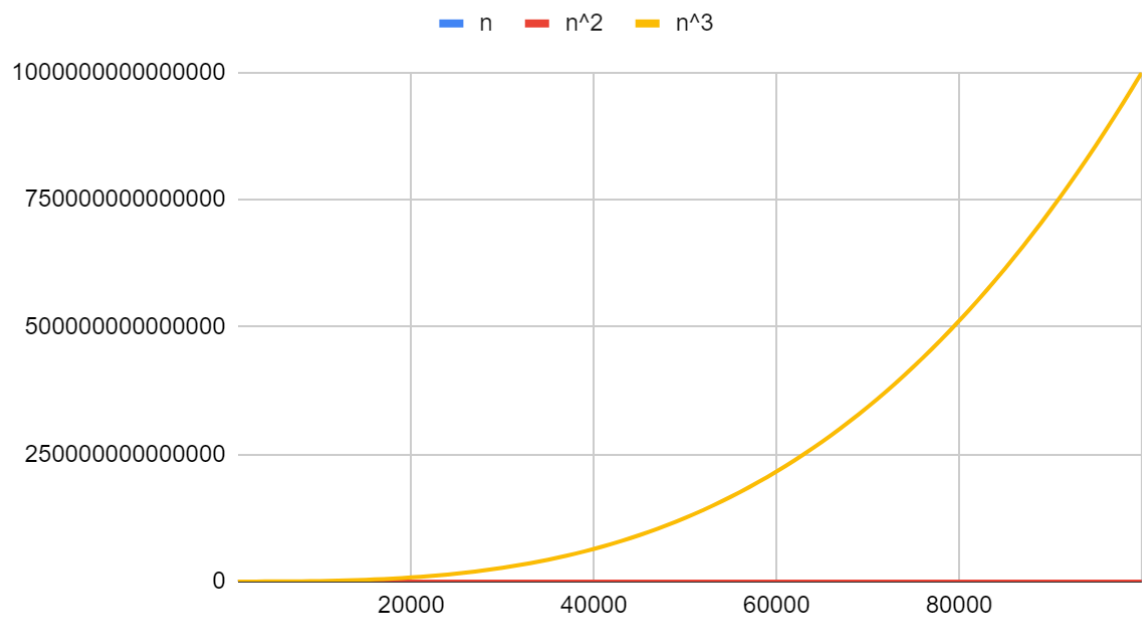
$n, n \lg n \in n^2$



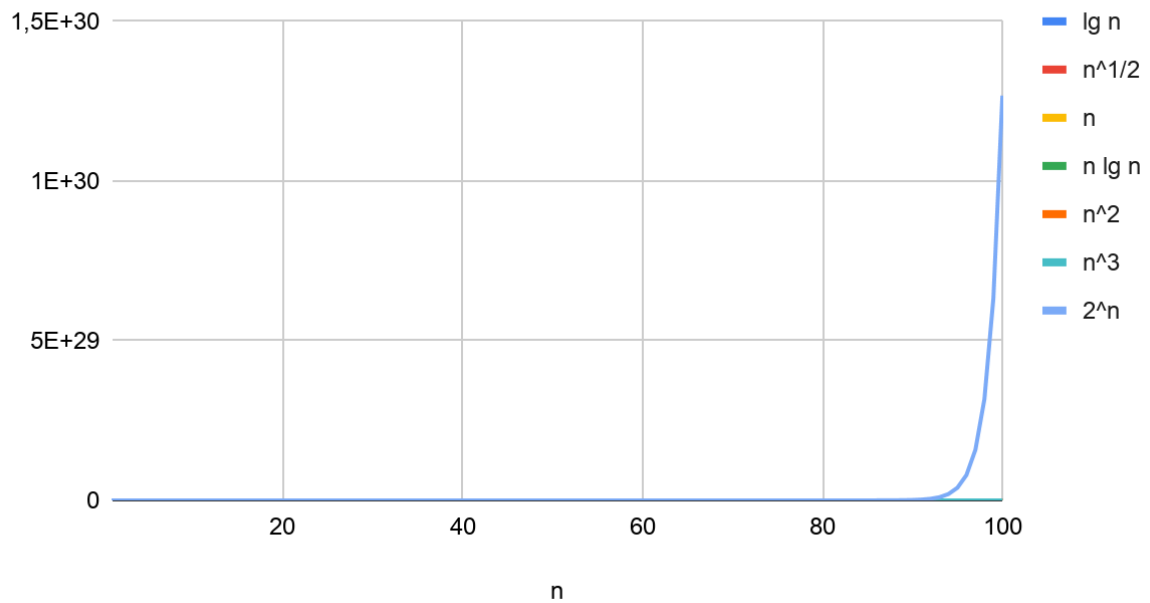
n , n^2 e n^3



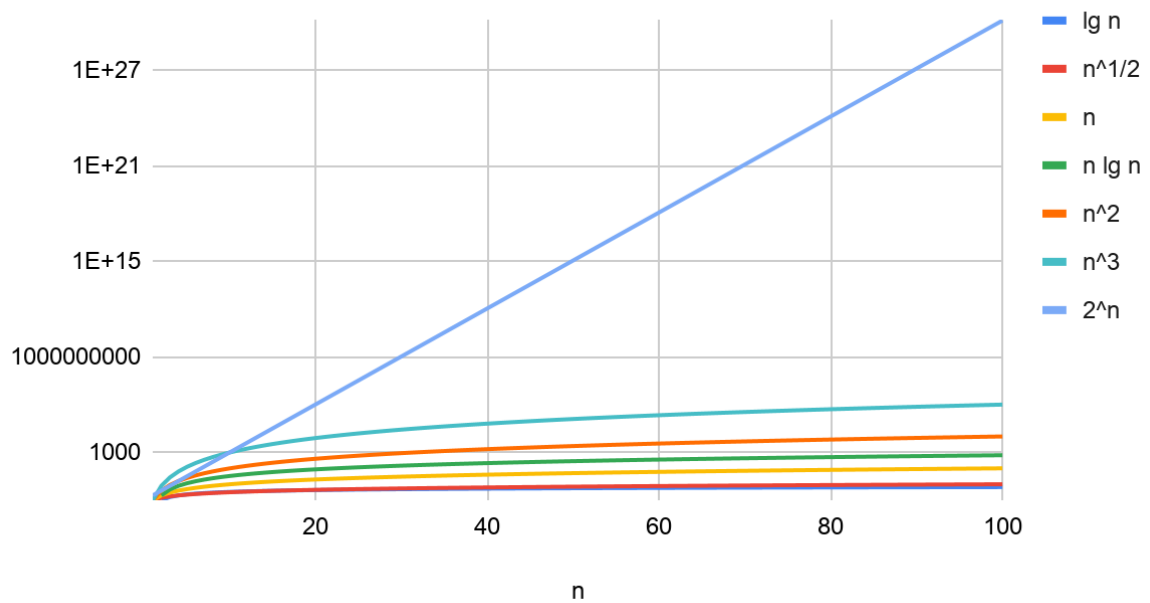
n , n^2 e n^3



$\lg n, n^{1/2}, n, n \lg n, n^2 \dots$



$\lg n, n^{1/2}, n, n \lg n, n^2 \dots$



Busca sequencial

Para resolver este problema, temos de

- devolver a posição de um determinado elemento x
 - em um vetor de inteiros v .

Uma ideia básica é percorrer o vetor verificando se x é o elemento de cada posição.

Algoritmo iterativo que busca um elemento x em um vetor v de tamanho n

```
int buscaSequencial1(int v[], int n, int x)
{
    int i = 0;
    while (i < n && v[i] != x)
        i++;
    if (i < n)
        return i;
    return -1;
}
```

Variações:

- Uma variação do algoritmo anterior é percorrer o vetor do fim para o início.
 - Neste caso, se o elemento não for encontrado i sai do laço valendo -1 ,
 - o que permite eliminar o `if`.
- Embora esta variante também resolva o problema,
 - ela pode devolver valor diferente numa situação específica. Qual?

Corretude e invariante:

- O invariante principal do algoritmo é que, no início de toda iteração do laço,
 - o vetor $v[0 \dots i - 1]$ não contém x .
- No início o invariante vale trivialmente,
 - pois $i = 0$ e $v[0 \dots i - 1]$ é vazio.
- O invariante se mantém de uma iteração para outra
 - pois i só é incrementado se $v[i] \neq x$.
- Se o algoritmo sair do laço por violar a primeira condição ($i < n$),
 - o invariante garante que x não está no vetor.
- Caso contrário, a violação da segunda condição ($v[i] \neq x$)
 - garante que o algoritmo devolve a posição correta de x .

Eficiência de tempo:

- No pior caso o algoritmo precisa percorrer o vetor inteiro,
 - realizando da ordem de n operações, i.e., $O(n)$.

Eficiência de espaço:

- $O(1)$, pois só usa uma pequena quantidade de variáveis auxiliares,
 - cujos tamanhos não dependem de n .

E se o vetor estiver ordenado, nossa busca sequencial pode ser melhorada?

- Supondo que o vetor está em ordem crescente e

- que estamos percorrendo-o do início ao fim,
- quando encontrarmos algum valor maior que x
 - sabemos que não adianta continuar buscando. Por que?
- O seguinte algoritmo utiliza essa ideia.

Algoritmo iterativo que realiza busca sequencial de um elemento x

- em um vetor v em ordem crescente de tamanho n.

```
int buscaSequencial2(int v[], int n, int x)
{
    int i = 0;
    while (i < n && v[i] < x)
        i++;
    if (i < n && v[i] == x)
        return i;
    return -1;
}
```

Convenções e variações:

- O algoritmo anterior devolve -1 se não encontrou o elemento.
- Outra convenção válida é devolver a posição em que
 - o elemento deveria ser inserido, de modo a manter a ordenação.
 - Como modificar o algoritmo para refletir esta convenção?

Corretude e invariante:

- O invariante principal do algoritmo é que, no início de toda iteração do laço,
 - temos $v[i - 1] < x$.
- Note que, como o vetor é ordenado, isso implica que
 - todo elemento em $v[0 .. i - 1]$ é menor que x.
- O invariante vale trivialmente no início, pois i começa valendo 0.
- Supondo que ele vale no início de uma iteração qualquer,
 - como o laço só incrementa i se $v[i] < x$,
 - ele continua valendo no início da próxima iteração.
- Quando o algoritmo sai do laço, temos que i indica onde x deveria estar,
 - pois todo elemento em $v[0..i - 1]$ é menor que x e
 - o algoritmo só sai do laço caso
 - o vetor tenha terminado, i.e., $i = n$,
 - ou $v[i] \geq x$.
- Se i é um índice válido do vetor e $x = v[i]$,
 - o algoritmo devolve i, indicando sucesso na busca.
- Senão, ele devolve -1, indicando que x não está no vetor.

Eficiência de tempo:

- o número de operações no pior caso é da ordem de n ,
 - ou, simplesmente, $O(n)$,
- mas vale notar que a constante é melhor no caso médio,
 - já que, em média, após percorrer metade do vetor
 - encontramos um elemento $\geq x$ e saímos do laço.

Eficiência de espaço:

- $O(1)$, pois só usa uma pequena quantidade de variáveis auxiliares,
 - cujos tamanhos não dependem de n .

Busca binária

A ideia da busca binária deriva da seguinte propriedade de vetores ordenados:

- Se o valor buscado x é menor que o valor na i -ésima posição do vetor v ,
 - i.e., $x < v[i]$,
- Então x é menor que todo valor em $v[i .. n - 1]$.
 - Portanto, x só pode ser encontrado
 - no subvetor complemento $v[0 .. i - 1]$.
- Caso contrário, i.e., $x > v[i]$, temos x maior que todo valor em $v[0 .. i]$.
 - Portanto, x só pode ser encontrado
 - no subvetor complemento $v[i+1 .. n-1]$.

Essa propriedade significa que,

- dependendo do índice i do valor $v[i]$ com o qual comparamos x ,
 - podemos descartar grandes pedaços do vetor.
- Por isso, devemos escolher sabiamente o índice i .

Note que, um índice i próximo dos extremos do vetor corrente

- pode resultar em descartes pequenos,
 - dependendo do resultado da comparação entre x e $v[i]$.

Assim, o valor que nos garante descartes significativos,

- independente de tal resultado é
 - i igual ao meio do vetor corrente.

Exemplo de busca binária:

$x = 3$ $[1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8]$ $x = 3 < 4 = v[i]$
 $[1 \ 2 \ 3] \ 4 \ 5 \ 6 \ 7 \ 8$ $x = 3 > 2 = v[i]$
 $1 \ 2 \ [3] \ 4 \ 5 \ 6 \ 7 \ 8$ $x = 3 = v[i]$
inversion

Algoritmo recursivo para busca binária de um elemento x

- em um vetor v em ordem crescente de tamanho n .

```
int buscaBinariaR(int v[], int e, int d, int x)
{
    int m;
    if (d < e)
        return -1;
    m = (e + d) / 2;
    if (v[m] == x)
        return m;
    if (v[m] < x)
        return buscaBinariaR(v, m + 1, d, x);
    return buscaBinariaR(v, e, m - 1, x);
}

int buscaBinaria2(int v[], int n, int x)
{
    return buscaBinariaR(v, 0, n - 1, x);
}
```

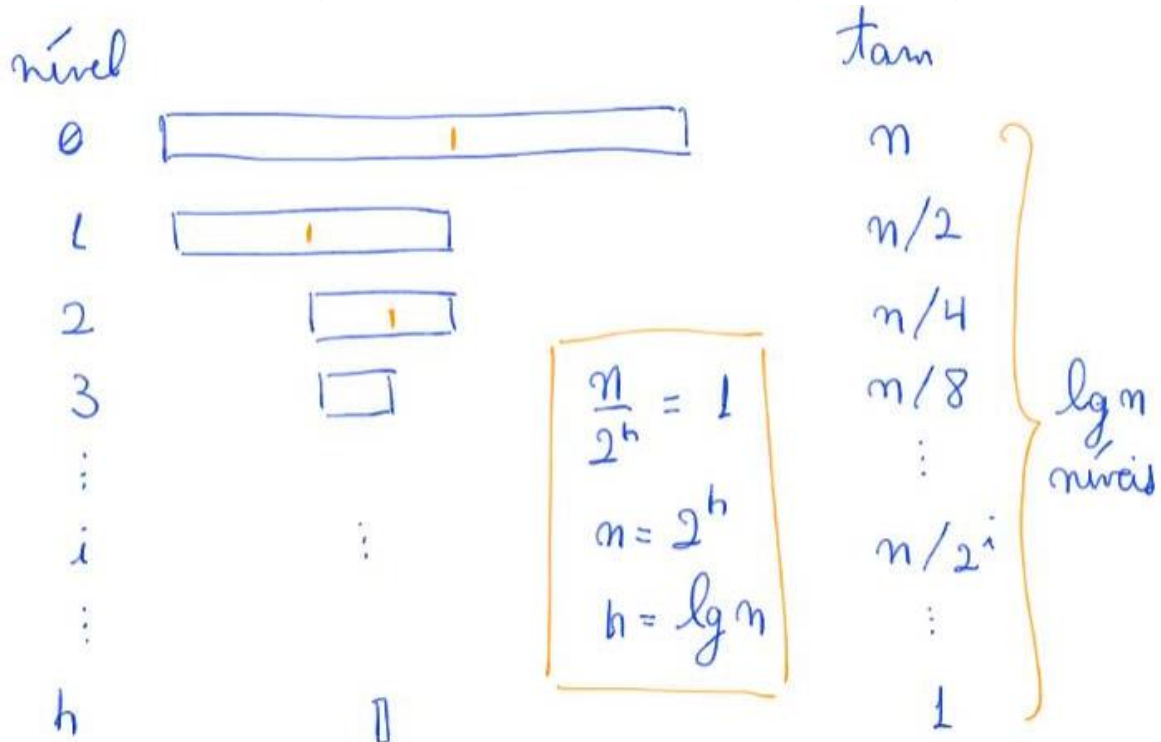
Corretude:

- Deriva diretamente da propriedade anterior
 - e pode ser demonstrada usando indução matemática.

Eficiência de tempo:

- Cada chamada da função buscaBinariaR

- desencadeia no máximo uma chamada recursiva,
 - na qual um dos extremos (e ou d) é atualizado com m (+ ou - 1),
- sendo que $m = (e + d) / 2$.
- Por isso, o vetor corrente (que começa em e e termina em d)
 - diminui de pelo menos metade a cada chamada recursiva.
- Intuitivamente, podemos pensar que a cada chamada recursiva
 - o vetor que começa com tamanho n é dividido pela metade.
- Assim, depois de aproximadamente $\lg n$ chamadas recursivas,
 - seu tamanho é reduzido a 0, e as chamadas terminam.
- Como o número de operações realizadas
 - localmente em cada chamada da função é constante,
 - o algoritmo leva tempo da ordem de $\lg n$, ou, $O(\lg n)$.



- Para calcular precisamente o número total de operações, usamos a recorrência
 - $T(n) = T(n / 2) + 1$,
 - $T(0) = 1$.
- Seguindo a recorrência
 - $T(n / 2) = T(n / 4) + 1$
 - $T(n / 4) = T(n / 8) + 1$
 - $T(n / 8) = T(n / 16) + 1$
 - ...
- Substituindo e simplificando
 - $T(n) = T(n / 2^1) + 1$
 - $= T(n / 2^2) + 2$
 - $= T(n / 2^3) + 3$
 - $= T(n / 2^4) + 4$

- ...
- Esta relação sugere a fórmula geral
 - $T(n) = T(n / 2^k) + k$
- Sabemos que a recorrência acaba
 - quando o vetor corrente tiver tamanho 0.
- Como as sucessivas divisões por 2 no tamanho do vetor
 - são arredondadas para baixo, isso ocorre para k que satisfaça
 - $n / 2^k < 1 \leq n / 2^{(k-1)}$.
- Portanto, $n / 2^k < 1 \leq n / 2^{(k-1)} \rightarrow n < 2^k \leq 2n$
 - $\rightarrow \lg n < k \leq \lg 2n = 1 + \lg n$
- Substituindo k por $1 + \lg n$ na recorrência
 - $T(n) = T(n / 2^k) + k = T(0) + \lg n + 1 = \lg n + 2$
- Assim, o número de operações no pior caso é da ordem de $\lg n$,
 - ou simplesmente $O(\lg n)$.

Eficiência de espaço:

- $O(\log n)$, devido ao tamanho da cadeia de chamadas recursivas.

Algoritmo iterativo para busca binária de um elemento x

- em um vetor v em ordem crescente de tamanho n.

```
int buscaBinaria(int v[], int n, int x)
{
    int e, m, d;
    e = 0;
    d = n - 1;
    while (e <= d)
    {
        m = (e + d) / 2;
        if (v[m] == x)
            return m;
        if (v[m] < x)
            e = m + 1;
        else
            d = m - 1;
    }
    return -1;
}
```

Corretude e invariante:

- O invariante principal é que no início de cada iteração

- $v[e - 1] < x < v[d + 1]$.
- Adotamos a convenção de que $v[-1] = -\infty$ e $v[n] = +\infty$.
 - Portanto, o invariante vale no início da primeira iteração.
- Supondo que o invariante vale no início de uma iteração qualquer,
 - como no laço só atualizamos o extremo (e ou d) que não contém x,
 - de acordo com o resultado da comparação $v[m] < x$,
 - o invariante continua valendo no início da iteração seguinte.
 - Ou seja, sempre descartamos o subvetor correto.
- Se em alguma iteração a comparação $v[m] == x$ for verdadeira,
 - devolve a posição de x.
- Caso contrário, sai do laço quando $d = e - 1$.
 - Pelo invariante, neste caso, $v[e - 1] < x < v[d + 1] = v[e]$.
 - Portanto, x não está no vetor e o algoritmo devolve -1.

Eficiência de tempo:

- $O(\log n)$, sendo que a demonstração
 - é igual aquela feita para o algoritmo recursivo,
 - substituindo chamada recursiva por iteração na argumentação.

Eficiência de espaço:

- $O(1)$, pois só usa uma pequena quantidade de variáveis auxiliares,
 - cujos tamanhos não dependem de n.

Convenções e variações:

- Nossos algoritmos para busca binária
 - devolvem -1 se não encontram o elemento.
- Outra convenção válida é devolver a posição em que
 - o elemento deveria ser inserido, de modo a manter a ordenação.
- Como modificar o algoritmo para refletir esta convenção?
 - Basta devolver o valor de e. Por que?
 - A resposta deriva do fato de, nos dois algoritmos
 - $v[0 .. e - 1] < x < v[d + 1 .. n - 1]$
 - No caso em que o elemento não é encontrado,
 - temos $e = d - 1$.
 - Portanto, $v[0 .. e - 1] < x < v[e .. n - 1]$,
 - ou seja, todo elemento até $e - 1 < x$
 - e todo elemento a partir de $e > x$.