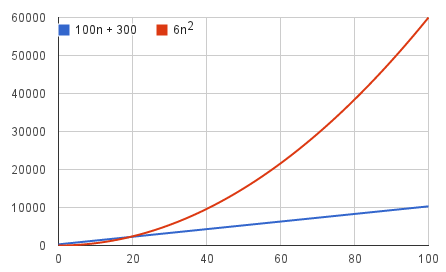
**Análise Assintótica de Algoritmos**

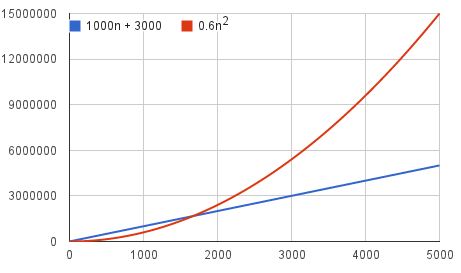
Analisamos a **busca linear e a busca binária** contando o número máximo de tentativas necessárias. Mas o que realmente **queremos saber é quanto tempo esses algoritmos demoram**.

**O tempo de execução de um algoritmo depende do quão demorado é para um computador executar as linhas de código do algoritmo**, e isto, depende da velocidade deste computador, da linguagem de programação, e do compilador que traduziu o programa da linguagem de programação para o código que executa diretamente no computador, entre outros.

**Precisamos determinar quanto tempo o algoritmo leva em termos do tamanho da entrada**. Já vimos que o máximo de tentativas necessárias em busca linear e em **busca binária aumenta conforme aumenta o tamanho do array de candidatos**.

A ideia é que precisamos focar em **quão rápido uma função cresce com o tamanho da entrada**, **chamamos isso de taxa de crescimento do tempo de execução**. Para manter as coisas tratáveis, **precisamos simplificar a função** até evidenciar a parte mais importante e deixar de lado as menos importantes. Por exemplo, suponha que um **algoritmo**, sendo executado com uma **entrada de tamanho n**, leve **6n2 + 100n + 300 instruções de máquina**. O termo **6n2 torna-se maior do que os outros termos**, 100n + 300, **uma vez que n torna-se grande o suficiente**. Abaixo temos um gráfico que mostra os valores de 6n2 e 100n + 300 para valores de n variando entre 0 e 100.

Podemos dizer que este **algoritmo cresce a uma taxa n2**, deixando de fora o coeficiente 06 e os termos restantes 100n + 300. Não é realmente importante ressaltar quais coeficientes usamos, já que o tempo de execução é an2 + bn + c, para alguns números a > 0, b e c, **sempre haverá um valor de n para o qual an2 é maior que bn + c**, e essa diferença aumenta juntamente com n. Por exemplo, aqui está um gráfico mostrando os valores de **0,6n2 e 1000n + 3000** de modo que **reduzimos o coeficiente de n2 por um fator de 10 e aumentamos as outras duas constantes por um fator de 10**:



O valor de n para o 0,6n2 é maior que 1000n + 3000, e sempre haverá um ponto de cruzamento, independentemente das constantes.

**Descartando os termos menos significativos e os coeficientes constantes**, podemos nos concentrar na **parte importante do tempo de execução de um algoritmo** — **sua taxa de crescimento** — sem sermos atrapalhados por detalhes que complicam sua compreensão. **Quando descartamos os coeficientes constantes e os termos menos significativos, usamos** **notação assintótica**. Vamos estudar suas **três formas: notação Θ, notação O, notação Ω**.

**Usamos a notação assintótica para expressar a taxa de crescimento do tempo de execução de um algoritmo, em termos do tamanho de entrada n**.

Suponha que um **algoritmo** demorou uma quantidade constante de **tempo**, independentemente do **tamanho da entrada**. Por exemplo, se você recebeu um **array** que já estava em **ordem crescente** e você tinha que **localizar o menor elemento**, levaria um **tempo constante**, uma vez que o **elemento** mínimo deve estar no **índice 0**. E já que nós gostamos de usar **funções de n em notação assintótica**, você poderia dizer que este **algoritmo é executado num tempo Θ(n0)**.

Por que? Porque no = 1 e o tempo de execução do algoritmo está contido em um fator constante de 01. Na prática, não escrevemos Θ (n0), ao invés disso, nós **escrevemos Θ(1)**.

Há uma **ordem** para as **funções** que vemos frequentemente quando **analisamos algoritmos** usando **notação assintótica**. Se **a e b são constantes e a < b**, então o **tempo de execução Θ (na) cresce mais lentamente do que um tempo de execução Θ(nb)**. Por exemplo, o **tempo de execução Θ(n), que é Θ(n1), cresce mais lentamente do que um tempo de execução Θ(n2)**. Por exemplo, o tempo de execução de Θ (n2) cresce mais lentamente do que um tempo de execução Θ (n2\*sqrt{n}), que é Θ (n2,5).

**Logaritmos crescem mais lentamente do que os polinômios**. Ou seja, Θ(lgn) cresce mais lentamente do que Θ(na) para qualquer constante positiva a. Mas, uma vez que o valor de lg n aumenta à medida que n aumenta, **Θ(lg n) cresce mais rápido do que Θ(1)**.

Aqui está uma **lista de funções na notação assintótica** com a qual nos deparamos muitas vezes ao analisar algoritmos, **listados do crescimento mais lento ao mais rápido**. Existem muitos algoritmos cujos tempos de execução não aparecem aqui: para n grande.

* Θ(1)
* Θ (lg n)
* Θ (n)
* Θ(nlgn)
* Θ (n^2)
* Θ(n^2lgn)
* Θ (n^3)
* Θ (2^n)

Note que uma função exponencial an, onde a > 1, cresce mais rápido do que qualquer função polinomial nb, onde b é qualquer constante.

**Análise Assintótica de Algoritmos T(n)**

**Exemplo de funções** com consumo de tempo de algoritmos: **n2 + 3n – 3; (3n2 +7n -8) / 2**; da ordem: c2n2 +c1n + c0.

Seja **T(n) o consumo de tempo** (no pior caso (em geral calculamos o pior caso), no melhor caso, caso médio) do algoritmo A, para instâncias de tamanho N. **Em geral calculamos o consumo de tempo no pior caso**.

**Precisamos ter um modo grosseiro de comparar funções**, que considere a **velocidade de crescimento das funções**. Pretendemos medir a ordem de grandeza da função de tempo do algoritmo.

No exemplo acima, ficamos satisfeitos em notar que, **no pior caso**, o **tempo cresce na proporção do quadrado do tamanho da sequência de entrada, no caso do exemplo n2**. Podemos formalizar estes conceitos com as **notações O, Ω (ômega) e Θ (teta)**, que **permitem** fazer uma **comparação assintótica de funções**.

**Comparações Assintóticas de Funções**

Exemplo de **consumo de tempo** de vários algoritmos, T(n): Para um n grande qual função cresce mais rápido.

* n6
* n + 10000
* (3n2 +7n -8) / 2
* 83n3 + 3n -3
* 3n5 + 8n -5
* log n
* 2n + 3n
* Sqrt(n)

Vamos ordenar: **1º as exponenciais, 2º os polinomiais, 3º os logaritmos e constantes**.

Comparação Assintótica de Funções, tempos três tipos de comparações assintóticas:

* Comparação com sabor de “<=”: O (pior caso)
* Comparação com sabor de “=”: Θ (caso médio)
* Comparação com sabor de “>=”: Ω (melhor caso)

**Notação O**

**O(f(n))** intuitivamente são **funções** que **não crescem mais rápido que f(n)**

**n2** + 3n -3, (3**n2** +7n -8) / 2 e c2**n2**+ c1n + c0, são O(**n2**), n2 + 3n -3 é O(**n2**), isto é, **n2**+3n-3 não cresce mais rápido que **n2**;

(3**n2** +7n -8) / 2 é O(**n2**), isto é, (3**n2**+7n-8)/2 não cresce mais rápido que **n2**;

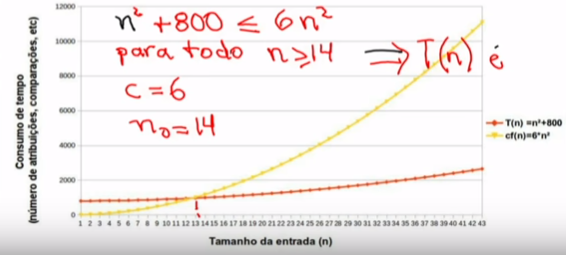
c2**n2**+ c1n + c0, é O(**n2**), isto é, c2**n2**+ c1n + c0, não cresce mais rápido que **n2**.

Sejam **T(n) e f(n)** funções dos inteiros. Dizemos que **T(n) é O(f(n))** se existem constantes positivas **c e n0**, tais que: **T(n) <= c.f(n)** para todo n >= n0.

**Lê-se**: **T(n) é O(f(n))** ou **T(n) é da ordem de f(n)** ou **T(n) ϵ O(f(n))** ou abuso da linguagem **T(n) = O(f(n)**.

**Exemplo 1: n2 + 800 é O(n2);**

n2 + 800 <= **6n2** para todo **n >= 14** => **T(n) é O(f(n))**. **c= 6** e **n0 = 14**.



**Exemplo 2: Demonstrar que n2 + 800 é O(n2); n2 + 800 <= c.f(n) => n2 + 800 <= cn2**

Prova: n2 + 800 <= n2 + 800n2 => **n2 + 800 <= 801n2** para todo n >=1, logo, **c = 801 e n0 = 1**.

**Prova Alternativa:**

Prova: n2 + 800 <= n2 + n\*n => n2 + 800 <= 2n2 para todo n >= 800, logo, c = 2 e n0 = 800.

Então, **n2 + 800 é O(n2)**, logo, n2 + 800 <= cf(n), pois, **n2 + 800 <= cn2**.

**Exemplo 3: Demonstrar que 100n2 é O(n3);**

100n2 <= n3 => 100n2 <= n \* n2, logo, para todo n >= 100 => **c = 1 e n0 = 100**.

**Exemplo 4: Demonstrar que 10n3-3n2+27 é O(n3);**

10n3-3n2+27 <= cf(n)

10n3-3n2+27 <= 10n3, para todo n >= 27 => **c = 10 e n0 = 3**.

**Exemplo 5: Demonstrar que n é O(2n)**

Será provado por indução que n <= 2n para todo n >= 1 => **c = 1 e n0 = 1**.

**Caso Base: Se n = 1 temos 1 <= 2**.

**Caso por Indução**: Para n >= 2 por **hipótese de indução temos que (n – 1)** <= 2n-1. Então **n <= n + ( n – 2 ) = ( n – 1 ) + ( n – 1 ) <= 2n -1 + 2n -1 <= 2(2n -1) = O(2n)**.

**Classes O**:

1. O(1) = constante
2. O (lg n) = logarítmica
3. O (n) = linear
4. O (nlgn) = n log n
5. O (n2) = quadrática
6. O (n3) = cúbica
7. O (nk) com k >= 1, polinomial
8. O (2n) exponencial
9. O (an) com a > 1, exponencial

**Ordenação por Inserção**

Ordena-por-inserção(A, n)

1. para j <- 2 até n faça O(n)
2. chave<-A[j] O(n)
3. i <- j – 1 O(n)
4. enquanto ( i >= 1 e A[i] > chave ) faça n O(n) = O(n2)
5. A[i+j] <- A[i] n O(n) = O(n2)
6. i <- i – 1 n O(n) = O(n2)
7. fim enquanto
8. A[i+1] <- chave O(n)
9. fim para

**Somatório => O(3n2 + 4n) = O(n2)**

A linha 04 é executada um número de vezes menor que n em cada iteração de j. Assim, ela consome nO(n). **O algoritmo consome O(n2) unidades de tempo**.

Portanto,

* nO(n) = O(n2)
* O(n) + O(n) + O(n) + O(n) = O(4n)
* O(n2) + O(n2) + O(n2) = O(3n2)
* O(3n2) + O(4n) = O(3n2+4n)
* O(3n2+4n) = O(n2)

Sejam T(n) e f(n) funções dos inteiros. Dizemos que **T(n) é O(f(n)) se existem constantes positivas c e n0**, tais que: **T(n) <= cf(n)** para todo **n >= n0**.

**Demonstrar que O(n2) + O(n2) = O(2n2)**, isso significa que se **T(n) é O(n2)**, então **T(n) + G(n) é O(2n2)**.

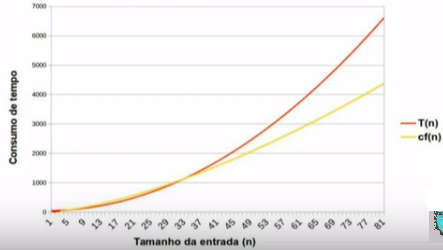
Prova: Existem constantes positivas c1 e n01 tais que: T(n) <= c1n2 para todo n >= n01 e existem constantes positivas c2 e n02 tais que: G(n) <= c2n2 para todo n >= n02.

T(n) + G(n) <= c2n2.

T(n) + G(n) <= c1n2 + c2n2 = 2n2(c1 + c2)/2 => **c = (c1 + c2)/2 e n0= max{n01, n02}**

**Notação Ω (Omega)**

Dizemos que **T(n) é Ω(f(n))** se existem constantes positivas c e n0, tais que: **T(n) >= cf(n)** para todo **n >= n0**.



**Exemplo 1:**

Demonstrar que **6n+5 é Ω(n)**

Prova: 6n+5 >= 6n; para todo **c = 6 e n >= 0, n0=0**.

**Exemplo 2:**

Demonstrar que **n2 - 2 é Ω(n2)**

Prova: n2 – 2 >= n2 / 2 + (n2/2 – 2) >= n2/2 se (n2/2-2) >= 0, isto é, para todo **n >=2 => c = ½ e n0 = 2**.

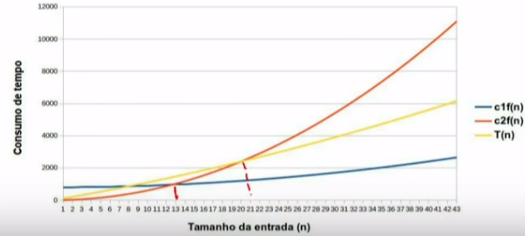
**Exemplo 3:**

Demonstrar que **nlogn é Ω(n)**

Prova: **logn >= 1** para todo **n >= 2**, portanto nlog n >= n => **c = 1 e n0 = 2**.

**Notação Θ (Teta)**

Dizemos que **T(n) é Θ(f(n))** se existem constantes positivas c1, c2 e n0, tais que: **c1f(n) <= T(n) <= c2f(n)** para todo **n >= n0**.



Dizemos que T(n) é **Θ(f(n)** se T(n) é O(f(n)) e T(n) é Ω(n).

**Exemplo 1:**

Demonstrar que **n2-2 é Θ(n2)**

Prova:

n2 – 2 é **O(n2)**

n2 – 2 <= n2, para n >= 0, tais que **c2 = 1, n02 = 0**;

n2 – 2 é **Ω(n2)**

n2 – 2 >= n2/2 + (n2/2-2) >= n2/2, se (n2/2-2) >= 0,

isto é, para todo n >= 2=> **c1 = 1/2, n01 = 2**;

Então,

n2/2 <= n2-2 <= n2, **c1 = ½, c2 = 1 e n0 = max{n01, n02} = 2**.

**Classes Θ**

Θ(1) constante

Θ(lg n) logarítmica

Θ(n) linear

Θ(nlgn) n log n

Θ(n2) quadrática

Θ(n3) cúbica

Θ(nk) com k >= 1 é polinomial

Θ(2n) exponencial

Θ(an) exponencial

Θ(1) **constante**, são algoritmos muito rápido, não depende do tamanho de entrada, realiza operações simples;

Θ(lg n) **logarítmica**, são algoritmos considerados muito rápido, exemplo de algoritmos, busca binária;

Θ(n) **linear**, n multiplicado por 10, tempo multiplicado por 10 algoritmos considerados muito rápidos;

Θ(n lg n) n log n, com constantes razoáveis, algoritmos considerados bem eficientes, exemplos, são os algoritmos de ordenação;

Θ(n2) **quadrático**, n multiplicado por 10, tempo multiplicado por 100, algumas vezes podem ser satisfatórios;

Θ(n3) **cúbica**, => n multiplicado por 10, tempo multiplicado por 1000, algumas vezes podem ser satisfatórios, exemplos, são algoritmos que realizam multiplicação de matrizes;

Θ(nk) **polinomial**, com k >= 1 exemplos: Θ(n), Θ(n2), Θ(n3), Θ(n50);

Θ(2n) **exponencial**, n multiplicado por 10, tempo multiplicados por 10, não úteis do ponto de vista prático, exemplos: Θ(2n),Θ(3n), Θ(4n), etc;

Θ(an) **exponencial**, com a > 1, exemplos: Θ(2n), Θ(3n), não úteis do ponto de vista prático.

**Propriedades – Cálculo de Consumo de Tempo**

As **propriedades são importantes**, às vezes, nos **permitem realizar cálculos de consumo de tempo dos algoritmos**.

**Transitiva**

T(n) é Θ(f(n)) e f(n) é Θ(g(n)) então T(n) é Θ(g(n))

T(n) é O(f(n)) e f(n) é O(g(n)) então T(n) é O(g(n))

T(n) é Ω(f(n)) e f(n) é Ω(g(n)) então T(n) é Θ(g(n))

**Reflexiva**

T(n) é Θ(T(n))

T(n) é O(T(n))

T(n) é Ω(T(n))

**Simétrica**

T(n) é Θ(f(n)), então f(n) é Θ(T(n))

**Antissimétrica**

T(n) é O(f(n)), então f(n) é Ω(T(n)

T(n) é Ω(f(n)), então f(n) é O(T(n))

**Revisão de Matemática**

**Expoentes**

Xn \* Xm = Xn+m

Xn / Xm = Xn-m

(Xn)m = Xnn\*m

Xn + Xn = 2Xn

**Logaritmos**

loga(b\*c) = logab + logac

loga(b/c) = logab - logac

logabm = mlogab

logab = logcb + logca (mudança de base)

**Somatórias**

**Soma de Quadrados**

**Série Aritmética – PA**

**Série Geométrica - PG**

Soma dos termos de uma PG.

, se 0 < q < 1

**Análise de Recorrência**

Uma **recorrência é uma equação ou desigualdade** que **descreve uma função em termos de seu valor em entradas menores**.

Para analisar o **consumo de tempo de um algoritmo recursivo** é **necessário resolver uma recorrência**. **Uma recorrência é uma expressão que dá o valor de uma função em termos dos valores anteriores da mesma função**. Por exemplo, **F(n) = F(n−1) + 3n + 2**.

É uma recorrência que dá o valor de F(n) em relação a F(n−1). Portanto, **recorrência pode ser vista como um algoritmo recursivo que calcula uma função a partir de um valor inicial**. No exemplo acima, podemos, por exemplo, tomar **F(1) = 1 como valor inicial**.

Uma recorrência é satisfeita por muitas funções diferentes, uma para cada valor inicial; mas todas essas funções são, em geral, do mesmo tipo.

**Resolver uma recorrência é encontrar uma fórmula fechada que dê o valor da função diretamente em termos do seu argumento**. Tipicamente, a fórmula fechada é uma combinação de polinômios, quocientes de polinômios, logaritmos, exponenciais, etc.

São funções recursivas definidas por termos de menor valor de entrada. Úteis para análise de algoritmos recursivos.

Exemplo: **T(n) = Ω(2n/2)**

**Exemplo 1**

Considere a **recorrência F(n) = F(n−1) + 3n + 2** e suponha que **n** **pertence ao conjunto {2,3,4,…}**. Há uma **infinidade de funções F** que **satisfazem** a **recorrência**. A tabela abaixo sugere uma dessas funções:

**n 1 2 3 4 5 6 …**

**F(n) 1 9 20 34 51 71 …**

A tabela seguinte sugere **outra função** que satisfaz a **recorrência**:

**n 1 2 3 4 5 6 …**

**F(n) 10 18 29 43 60 80 …**

**Exemplo 2 – Algoritmo Fibonacci**

f(n) = 1, se 1 <= n <= 2

f(n-1) + f(n-2), se n > 2

**Exemplo 3 – Algoritmo QSoft Merge**

f(n) = 1, se n = 1

2g(n/2) + n, se n > 1 (algoritmos menores)

**Técnicas para Resolução de Recorrências**

Obter uma fórmula fechada para T(n) que dê o valor da função diretamente em termos de seu argumento.

Resolver:

T(n) = 1, caso base.

T(n) = 2T(n/2) + 2, para n = 2, 4, 6, 8, ..., 2i.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **N** | **1** | **2** | **4** | **8** | **16** | **32** |
| T(n) | 1 | 4 | 10 | 22 | 46 | 94 |

**Resposta: T(n) = 3n -2** Será? Vamos validar pelos outros métodos...

**- Método de Iteração**

Consiste em expandir, isto é, **iterar a recorrência** e escrevê-la como uma **somatória de termos que dependem apenas de n**.

**Exemplo**

**T(n) = 1, caso base.**

**T(n) = 2T(n/2) + 2, para n = 2, 4, 6, 8, ..., 2i**.

T(n)

= 2\*T(n/2) + 22 -2 it1

= 2\*[T(n/4) + 2] +2 => 22 T(n/4) +6 => 22 T(n/22) + 23 -2 it2

= 2\*[2\*T(n/8) + 2] +2 +2] => 23 T(n/8) +14 => 23T(n/23)+24 -2 it3

= 2\*[2\*[2\*T(n/16) + 2] +2 +2 +2] => 24 T(n/16) + 30 => 24T(n/24) +25 -2 it4

e para a i-ésima iteração? = **2iT(n/2i) +2i+1 -2** iti

**Quando chegar no caso base**? Será quando vai parar.

**Quando T(n/2i)=T(1)**

n/2i = 1=> n = 2i => i = lgn, substituindo na fórmula da it i, temos:

T(n) = nT(1) + 2n – 2= 3n -2, logo, **T(n) = 3n -2, T(n) = Θ(n)**

*A validação: Será através do método de Árvore de Recorrência.*

**- Método de Substituição**

Trabalha com a ideia de **indução matemática**. Utiliza a indução matemática para **provar a recorrência**. O **método começa** com um **“chute”** para valor de **T(n)**. **Após isso, demonstrado por indução que o “chute” está certo.**

**- Método de Árvore de Recorrência**

Mais fácil de analisar, se a recorrência for fácil de analisar.

É útil para **estimar a solução de uma recorrência**. Representa em uma árvore o desenvolvimento da recorrência. **Permite visualizar melhor o que acontece quando a recorrência é iterada**. Cada **nó da árvore representa um subproblema**. Para calcular o consumo de tempo total, são somados todos os custos por nível.

Somamos os nós da árvore, que serão associados às chamadas do algoritmo analisado. **Para analisar nos casos médios é mais complicado**. **É mais para limite inferior e limite superior**.

**Exemplo**

**T(n) = 1, caso base.**

**T(n) = 2T(n/2) + 2, para n = 2, 4, 6, 8, ..., 2i**.

Resultado da árvore de recorrência: **T(n/2i)**

Quando vai chegar ao caso base? Quando vai parar? **Quando T(n/2i) = T(1)**, então, **n/2i = 1 => n = 2i => i = lgn**

|  |  |  |
| --- | --- | --- |
| **Nível** | **nós folhas** | **∑**  **T(n) = 2 + 22 + 23 + ... + 2i + 2i = 2i+=2i + (2i+1-1)/(2-1)-1** |
| 0 | 1 | 21 |
| 1 | 2 | 22  **=> n + 2n -1 -1 = 3n -2** |
| 2 | 22 | 23 |
| 3 | 23 | 24  **T(n) = 3n -2, logo o resultado encontrado pelo método de indução está correto.** |
| ... | ... | ... |
| i-1 | 2i-1 | 2i |
| i | 2i |  |

**- Método do Teorema Mestre**

O teorema Mestre utiliza **métodos específicos para resolver recorrências** **no formato**.

Sejam as constantes a>= 1 e b > 1, seja a função f(n) e seja a **recorrência** **T(n)** na forma, **T(n) = aT(n/b) + f(n)**.

**- Análise de algoritmo árvore de recursão**

**Ordem O**

f=O(g); f(n) <= cg(n)

**Ordem Θ**

O e Ω > Θ

**f=O(g), se f=O(g) e f= Ω(g)** números positivos c e d, tais que **c(g) <= f(n) <= dg(n)**

**f(n) >= cg(n)** (despreza o positivo) e **f(n) <= cg(n)** (despreza o negativo), para todo **n** suficientemente grande. **T(n) = c+T(n-1)+T(n-2)**

**Hierarquia de Funções**

|  |  |
| --- | --- |
| **Função** | **Estrutura de Dados** |
| 1 | Hash |
| log n | Árvore binária |
| N | Vetor |
| Nlogn | Ordenação por Comparação |
| n2 | Matriz |
| n3 | Algoritmos Grafos |
| n2logn | Algoritmos Grafos |
| 2i (Exponencial) |  |
| n! (Fatorial) |  |
| nn (Combinações) |  |

**Teorema Mestre**

O método mestre fornece uma receita para solução de recorrências da forma **T(n) = aT(n/b) + f(n)**, onde a >=1 e b >= 2 são constantes e f(n) é uma função assintótica positiva.

A **função T(n)** terá os seguintes limites:

* 1. Se f(n) = O(nloga - ϵ), para uma constante ϵ > 0, então T(n) = Θ(nlogba)
  2. Sef(n) = Θ(nlogba), então T(n) = Θ(nlogba \* lgbn)
  3. Se f(n) = Ω( nlog ba+ϵ), para uma constante ϵ > 0 e se af(n) <= cf(n) para alguma constante c > 1, então T(n) = Θ(f(n))

Considere a recorrência **T(n) = aT(n/b) + f(n)** onde a>= 1 e b >= 2, são constantes, f(n) é uma função assintótica positiva, e n/b pode ser [n/b].

Então: **T(n) = aT(n/b) + f(n)**

|  |  |
| --- | --- |
| T(n) | Se |
| Θ(nlogba) | f(n) = O(nlogba-ϵ) |
| Θ(nlogba \* lgbn) | f(n) = Θ(nlogba), |
| Θ(f(n)) | f(n) = Ω( nlog ba+ϵ) |

Exercício: **T(n) = 9T(n/3) + n**

**a=9; b=3; f(n) = n** => nlogba = nlog39 = n2; **f(n) = n = Θ(n2-ϵ), ϵ=1**

**Links Úteis**

1. **The Algorithms Design Manual (Second Edition)**

<http://www.algorist.com/algowiki/index.php/The_Algorithms_Design_Manual_(Second_Edition)>

1. **Blackstack**

<https://www.blackstackbrewing.com/>

1. **Análise de Algoritmos – Árvore de Recursão - Youtube**

<https://www.bing.com/videos/search?q=teorema+mestre+algoritmos+e+grafos&&view=detail&mid=3EC0D3CF1989BBBC73783EC0D3CF1989BBBC7378&&FORM=VRDGAR>