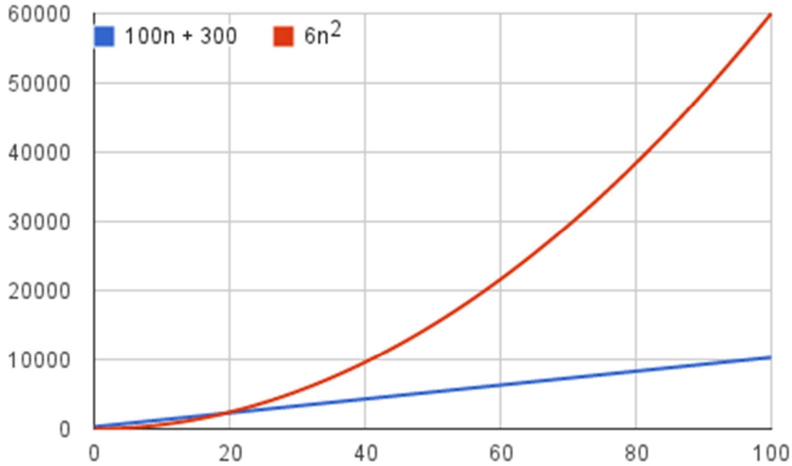
Análise Assintótica de Algoritmos

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**.

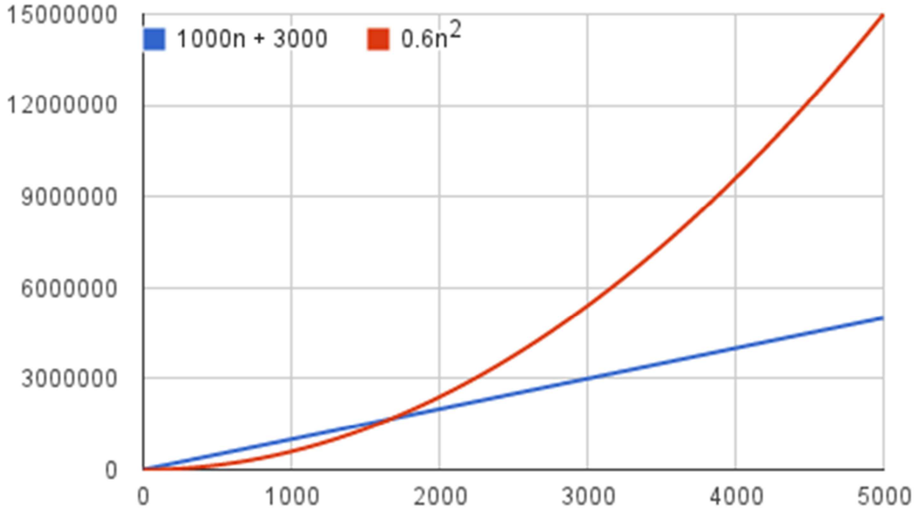
O tempo de execução de um algoritmo depende do quão demorado é para um computador executar as linhas de código do algoritmo, e isto, depende da velocidade deste computador, da linguagem de programação, e do compilador que traduziu o programa da linguagem de programação para o código que executa diretamente 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 **$6n^2 + 100n + 300$ instruções de máquina**. O termo **$6n^2$ 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 $6n^2$ e 100n + 300 para valores de n variando entre 0 e 100.



Podemos dizer que este **algoritmo cresce a uma taxa n^2**, 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 $an^2 + bn + c$, para alguns números a > 0, b e c, **sempre haverá um valor de n para o qual an^2 é maior que bn + c**, e essa diferença aumenta juntamente com n. Por exemplo, aqui está um gráfico mostrando os valores de **$0,6n^2$ e 1000n + 3000** de modo que **reduzimos o coeficiente de n^2 por um fator de 10 e aumentamos as outras duas constantes por um fator de 10**:



O valor de n para o $0,6n^2$ 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 $\Theta(n^0)$**.

Por que? Porque $n^o = 1$ e o tempo de execução do algoritmo está contido em um fator constante de 01. Na prática, não escrevemos Θ (n^0), ao invés disso, nós **escrevemos $\Theta(1)$**.

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 (n^a) cresce mais lentamente do que um tempo de execução $\Theta(n^b)$**. Por exemplo, o **tempo de execução $\Theta(n)$, que é $\Theta(n^1)$, cresce mais lentamente do que um tempo de execução $\Theta(n^2)$**. Por exemplo, o tempo de execução de Θ (n^2) cresce mais lentamente do que um tempo de execução Θ (n2*sqrt{n}), que é Θ $(n^{2,5})$.

Logaritmos crescem mais lentamente do que os polinômios. Ou seja, Θ(lgn) cresce mais lentamente do que $\Theta(n^a)$ 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 $\Theta(1)$**.

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.

- $\Theta(1)$
- Θ (lg n)
- Θ (n)
- Θ(nlgn)
- Θ (n^2)
- Θ(n^2lgn)
- Θ (n^3)
- Θ (2^n)

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

Análise Assintótica de Algoritmos T(n)

Exemplo de funções com consumo de tempo de algoritmos: **$n^2 + 3n – 3$; $(3n^2 +7n -8) / 2$**; da ordem: $c2n^2 +c1n + c0$.

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 n^2**. 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.

- n^6
- $n + 10000$
- $(3n^2 +7n -8) / 2$
- $83n^3 + 3n -3$
- $3n^5 + 8n -5$
- $\log n$
- $2^n + 3n$
- 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)**

$\mathbf{n^2}$ + 3n -3, (3$\mathbf{n^2}$ +7n -8) / 2 e c2$\mathbf{n^2}$ + c1n + c0, são O($\mathbf{n^2}$), n^2 + 3n -3 é O($\mathbf{n^2}$), isto é, $\mathbf{n^2}$+3n-3 não cresce mais rápido que $\mathbf{n^2}$;

(3$\mathbf{n^2}$ +7n -8) / 2 é O($\mathbf{n^2}$), isto é, (3$\mathbf{n^2}$+7n-8)/2 não cresce mais rápido que $\mathbf{n^2}$;

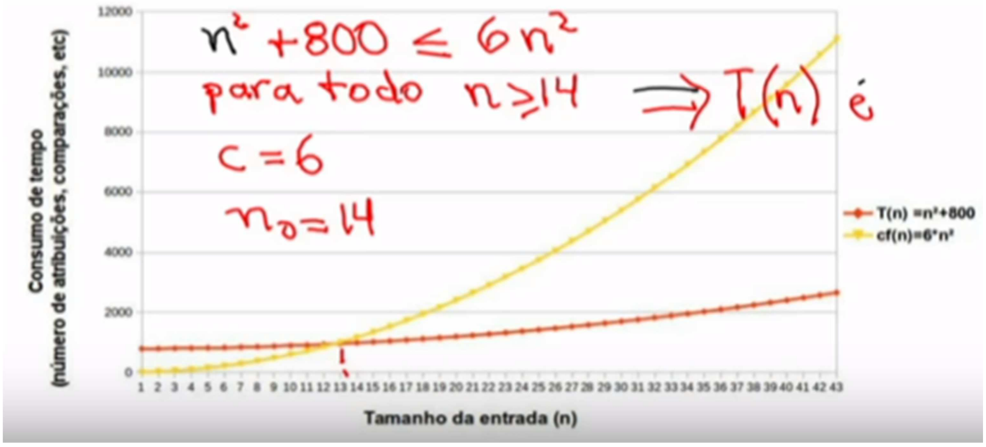
c2$\mathbf{n^2}$ + c1n + c0, é O($\mathbf{n^2}$), isto é, c2$\mathbf{n^2}$ + c1n + c0, não cresce mais rápido que $\mathbf{n^2}$.

Sejam **T(n) e f(n)** funções dos inteiros. Dizemos que **T(n) é O(f(n))** se existem constantes positivas **c e n_0**, tais que: **T(n) <= c.f(n)** para todo n >= n_0.

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: n^2 + 800 é O(n^2);

n^2 + 800 <= **$6n^2$** para todo **n >= 14** => **T(n) é O(f(n))**. **c= 6** e **n_0 = 14**.



Exemplo 2: Demonstrar que $n^2 + 800$ é $O(n^2)$; $n^2 + 800 <= c.f(n) => n^2 + 800 <= cn^2$

Prova: $n^2 + 800 <= n^2 + 800n^2 =>$ **$n^2 + 800 <= 801n^2$** para todo n >=1, logo, **$c = 801$ e $n_0 = 1$**.

Prova Alternativa:

Prova: $n^2 + 800 <= n^2 + n*n => n^2 + 800 <= 2n^2$ para todo n >= 800, logo, c = 2 e $n_0 = 800$.

Então, **$n^2 + 800$ é $O(n^2)$**, logo, $n^2 + 800 <= cf(n)$, pois, **$n^2 + 800 <= cn^2$**.

Exemplo 3: Demonstrar que $100n^2$ é $O(n^3)$;

$100n^2 <= n^3 => 100n^2 <= n * n^2$, logo, para todo n >= 100 => **$c = 1$ e $n_0 = 100$**.

Exemplo 4: Demonstrar que $10n^3-3n^2+27$ é $O(n^3)$;

$10n^3-3n^2+27 <= cf(n)$

$10n^3-3n^2+27 <= 10n^3$, para todo n >= 27 => **$c = 10$ e $n_0 = 3$**.

Exemplo 5: Demonstrar que n é $O(2^n)$

Será provado por indução que $n <= 2^n$ para todo n >= 1 => **$c = 1$ e $n_0 = 1$**.

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

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

<u>**Classes O**</u>:

1. O(1) = constante
2. O (lg n) = logarítmica
3. O (n) = linear
4. O (nlgn) = n log n
5. O (n^2) = quadrática
6. O (n^3) = cúbica
7. O (n^k) com k >= 1, polinomial
8. O (2^n) exponencial
9. O (a^n) 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(n^2)
5.          A[i+j] <- A[i]                         n O(n) = O(n^2)
6.          i <- i – 1                             n O(n) = O(n^2)
7.      fim enquanto
8.      A[i+1] <- chave                            O(n)
9. fim para
```

Somatório => $O(3n^2 + 4n) = O(n^2)$

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 $O(n^2)$ unidades de tempo**.

Portanto,

- $nO(n) = O(n^2)$
- $O(n) + O(n) + O(n) + O(n) = O(4n)$
- $O(n^2) + O(n^2) + O(n^2) = O(3n^2)$
- $O(3n^2) + O(4n) = O(3n^2+4n)$
- $O(3n^2+4n) = O(n^2)$

Sejam T(n) e f(n) funções dos inteiros. Dizemos que **T(n) é O(f(n)) se existem constantes positivas c e n_0**, tais que: **$T(n) <= cf(n)$** para todo **$n >= n_0$**.

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

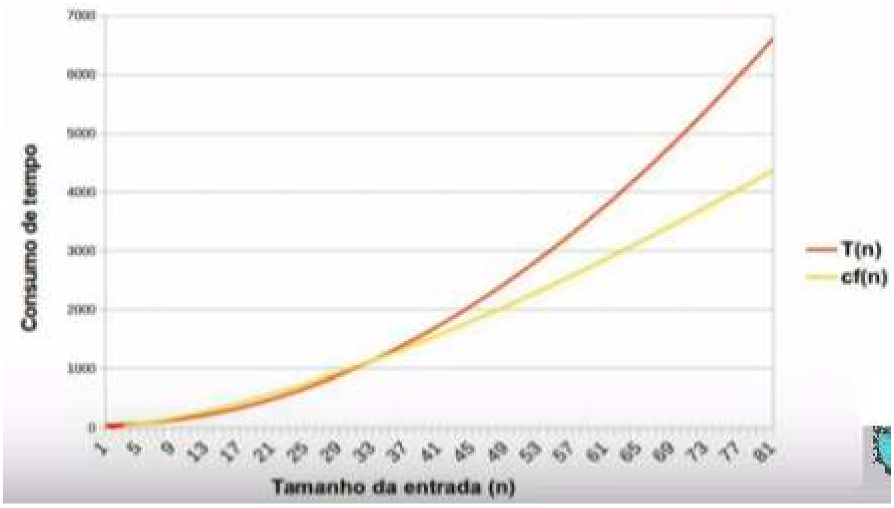
Prova: Existem constantes positivas c_1 e n_{01} tais que: $T(n) <= c_1n^2$ para todo $n >= n_{01}$ e existem constantes positivas c_2 e n_{02} tais que: $G(n) <= c_2n^2$ para todo $n >= n_{02}$.

$T(n) + G(n) <= c2n^2$.

$T(n) + G(n) <= c_1n^2 + c_2n^2 = 2n^2(c_1 + c_2)/2$ => **$c = (c_1 + c_2)/2$ e $n_0 = \max\{n_{01}, n_{02}\}$**

Notação Ω (Omega)

Dizemos que **T(n) é Ω(f(n))** se existem constantes positivas c e n_0, tais que: **T(n) >= cf(n)** para todo **n >=** $\mathbf{n_0}$.



Exemplo 1:

Demonstrar que **6n+5 é Ω(n)**

Prova: 6n+5 >= 6n; para todo **c = 6 e n >= 0,** $\mathbf{n_0}$**=0**.

Exemplo 2:

Demonstrar que $\mathbf{n^2 - 2}$ **é** $\mathbf{\Omega(n^2)}$

Prova: $n^2 - 2 >= n^2 / 2 + (n^2/2 - 2) >= n^2/2$ se $(n^2/2-2) >= 0$, isto é, para todo **n >=2 => c = ½ e** $\mathbf{n_0}$ **= 2**.

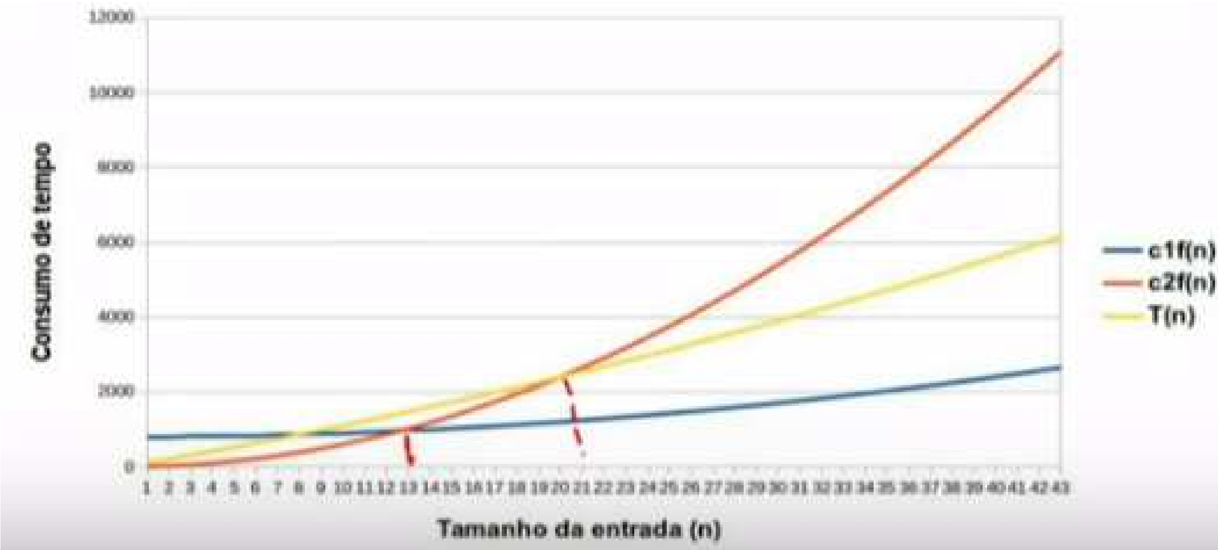
Exemplo 3:

Demonstrar que **nlogn é Ω(n)**

Prova: **logn >= 1** para todo **n >= 2**, portanto nlog n >= n => **c = 1 e** $\mathbf{n_0}$ **= 2**.

Notação Θ (Teta)

Dizemos que **T(n) é Θ(f(n))** se existem constantes positivas c1, c2 e n_0, tais que: **c1f(n) <= T(n) <= c2f(n)** para todo **$n >= n_0$**.



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 **n^2-2 é Θ(n^2)**

Prova:

$n^2 - 2$ é **O(n^2)**

$n^2 - 2 <= n^2$, para n >= 0, tais que **c2 = 1, n_{02} = 0**;

$n^2 - 2$ é **Ω(n^2)**

$n^2 - 2 >= n^2/2 + (n^2/2-2) >= n^2/2$, se $(n^2/2-2) >= 0$,

isto é, para todo n >= 2=> **c1 = 1/2, n_{01} = 2**;

Então,

$n^2/2 <= n^2-2 <= n^2$, **c1 = ½, c2 = 1 e n_0 = max{n_{01}, n_{02}} = 2**.

Classes Θ

$\Theta(1)$	constante
$\Theta(\lg n)$	logarítmica
$\Theta(n)$	linear
$\Theta(n\lg n)$	n log n
$\Theta(n^2)$	quadrática
$\Theta(n^3)$	cúbica
$\Theta(n^k)$ com k >= 1 é polinomial	
$\Theta(2^n)$	exponencial
$\Theta(a^n)$	exponencial

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

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

$\Theta(n)$ **linear**, n multiplicado por 10, tempo multiplicado por 10 algoritmos considerados muito rápidos;

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

$\Theta(n^2)$ **quadrático**, n multiplicado por 10, tempo multiplicado por 100, algumas vezes podem ser satisfatórios;

$\Theta(n^3)$ **cúbica**, => n multiplicado por 10, tempo multiplicado por 1000, algumas vezes podem ser satisfatórios, exemplos, são algoritmos que realizam multiplicação de matrizes;

$\Theta(n^k)$ **polinomial**, com k >= 1 exemplos: $\Theta(n)$, $\Theta(n^2)$, $\Theta(n^3)$, $\Theta(n^{50})$;

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

$\Theta(a^n)$ **exponencial**, com a > 1, exemplos: $\Theta(2^n)$, $\Theta(3^n)$, 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

$X^n * X^m = X^{n+m}$
$X^n / X^m = X^{n-m}$
$(X^n)^m = Xn^{n*m}$
$X^n + X^n = 2X^n$

Logaritmos

$\log_a(b*c) = \log_a b + \log_a c$
$\log_a(b/c) = \log_a b - \log_a c$
$\log_a b^m = m\log_a b$
$\log_a b = \log_c b + \log_c a$ (mudança de base)

Somatórias

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

Soma de Quadrados

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=1}^{n} i^3 = 1^3 + 2^3 + 3^3 + \cdots + n^3 = (\frac{n(n+1)}{2})^2$$

$$\sum_{i=m}^{n} x^i = x^m + x^{m+1} + x^{m+2} + \cdots + x^n = \frac{x(x^n - x^{m-1})}{x-1}$$

$$\sum_{i=0}^{n} x^i = x^m + x^{m+1} + x^{m+2} + \cdots + x^n = \frac{x(x^{n+1} - 1)}{x-1}$$

$$\sum_{i=0}^{n} a^i = \frac{(a^{n+1} - 1)}{a-1}$$

Série Aritmética – PA

$$\sum = n\frac{(1+n)}{2} = n\frac{a_1 + a_n}{2}$$

$$\sum a_1 + a_2 + a_3 + \cdots + a_{n=}n\frac{(a_n + a_1)}{2}$$

Série Geométrica - PG

Soma dos termos de uma PG.

$$\sum_{i=1}^{n} a_1 = \frac{a_1}{1-c}$$

$\sum n = a_1 + a_2 + a_3 + \cdots + a_{n=} a_1 \frac{(q^n - 1)}{q-1}$, 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) = $\Omega(2^{n/2})$**

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

$$f(n) = \begin{cases} 1, \text{ se } 1 <= n <= 2 \\ f(n-1) + f(n-2), \text{ se } n > 2 \end{cases}$$

Exemplo 3 – Algoritmo QSoft Merge

$$f(n) = \begin{cases} 1, \text{ se } n = 1 \\ 2g(n/2) + n, \text{ se } n > 1 \text{ (algoritmos menores)} \end{cases}$$

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, ..., 2^i.

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 $n = 2, 4, 6, 8, ..., 2^i$.

T(n)

$= 2*T(n/2) + 2^2 - 2$ it1

$= 2*[T(n/4) + 2] + 2 \Rightarrow 2^2 T(n/4) + 6 \Rightarrow 2^2 T(n/2^2) + 2^3 - 2$ it2

$= 2*[2*T(n/8) + 2] + 2 + 2] \Rightarrow 2^3 T(n/8) + 14 \Rightarrow 2^3 T(n/2^3) + 2^4 - 2$ it3

$= 2*[2*[2*T(n/16) + 2] + 2 + 2 + 2] \Rightarrow 2^4 T(n/16) + 30 \Rightarrow 2^4 T(n/2^4) + 2^5 - 2$ it4

e para a i-ésima iteração? = **$2^i T(n/2^i) + 2^{i+1} - 2$** it^i

Quando chegar no caso base? Será quando vai parar.

Quando $T(n/2^i) = T(1)$

$n/2^i = 1 \Rightarrow n = 2^i \Rightarrow i = \lg n$, substituindo na fórmula da it i, temos:

$T(n) = nT(1) + 2n - 2 = 3n - 2$, logo, **$T(n) = 3n - 2$, $T(n) = \Theta(n)$**

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.

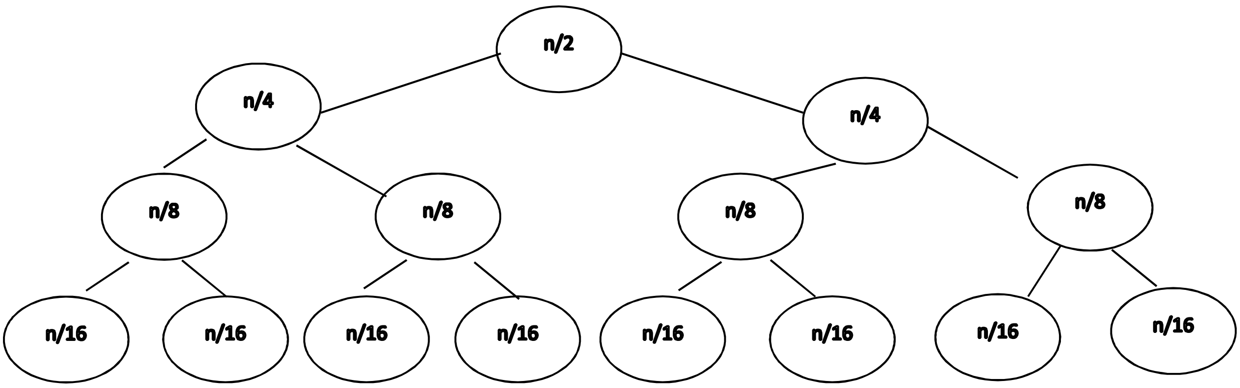
É útil para **estimar a solução de uma recorrência**. Representa em uma árvore o desenvolvimento da recorrência. **Permite visualizar melhor o que acontece quando a recorrência é iterada**. Cada **nó da árvore representa um subproblema**. Para calcular o consumo de tempo total, são somados todos os custos 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, ..., 2^i.



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

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

Nível	nós folhas	∑
0	1	2^1
1	2	2^2
2	2^2	2^3
3	2^3	2^4
...	...	...
i-1	2^{i-1}	2^i
i	2^i	

$T(n) = 2 + 2^2 + 2^3 + \ldots + 2^i + 2^i = 2^i + \sum_{k=1}^{i} 2^k = 2^i + (2^{i+1}-1)/(2-1)-1$

=> n + 2n -1 -1 = 3n -2

T(n) = 3n -2, logo o resultado encontrado pelo método de indução está correto.

- 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
n^2	Matriz
n^3	Algoritmos Grafos
n^2logn	Algoritmos Grafos
2^i (Exponencial)	
n! (Fatorial)	
n^n (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) = $\Theta(n^{\log_b a})$
2. Sef(n) = $\Theta(n^{\log_b a})$, então T(n) = $\Theta(n^{\log_b a} * \lg^b n)$
3. Se f(n) = $\Omega(n^{\log_b a+\epsilon})$, 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
$\Theta(n^{\log_b a})$	f(n) = $O(n^{\log_b a-\epsilon})$
$\Theta(n^{\log_b a} * \lg^b n)$	f(n) = $\Theta(n^{\log_b a})$,
Θ(f(n))	f(n) = $\Omega(n^{\log_b a+\epsilon})$

Exercício: **T(n) = 9T(n/3) + n**
a=9; b=3; f(n) = n => $n^{\log_b a} = n^{\log_3 9} = n^2$; **f(n) = n = $\Theta(n^{2-\epsilon})$, ϵ=1**

Links Úteis

1. **The Algorithms Design Manual (Second Edition)**
http://www.algorist.com/algowiki/index.php/The_Algorithms_Design_Manual_(Second_Edition)

2. **Blackstack**
https://www.blackstackbrewing.com/

3. **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