>> Eficiência de Algoritmos
- Não basta saber se o algoritmo funciona, precisamos saber se ele é o melhor possível
- Análises mais comuns:
    * Tempo <<Vamos focar nessa
    * Memória (Espaço) <<Iremos pra essa num caso de empate de eficiencia
    * Comunicação (Mensagens)

Vamos nos dedicar a analise de tempo.

Muitos fatores interferem no tempo de execução de um programa, de forma que e mais útil definirmos uma noção abstrata de tempo para analisar a eficiência de algoritmos quando o nosso objetivo e observar a proposta de resolução que eles oferecem. Vamos definir um passo em um algoritmo e usar essa noção
como unidade de tempo para analise.

Um passo na nossa linguagem, são:
- Cada instrução de atribuição.
- Cada instrução condicional.
- Cada instrução de retorno.

Algoritmo: Extremos(V)
Entrada: Vetor numérico V
Saída: Índices do maior e do menor elemento em V

```
se |V| = 0 entao:
|  retorne 0,0

maior ←1
menor ←1

para i ←2,3,...,|V| faça:
|  se V[menor] > V[i] entao:
|  |  menor<-i
|  se V[maior] < V[i] entao:
|  |  maior<-i

retorne maior, menor
```

A dimensão da entrada que interfere diretamente no numero de passos que um algoritmo executa damos o nome de tamanho de entrada.
Geralmente denotamos esse tamanho por "n", "m", ou outra letra, sendo "n" mais comum.
Mesmo para duas entradas de mesmo tamanho, ainda e possível que um mesmo algoritmo execute quantidades diferentes de passos.
Vamos então nos preocupar apenas com o máximo e o mínimo que esse algoritmo pode gerar, para um dado tamanho de entrada.
Vamos definir as funções Fp(n) e Fm(n) para relacionar o tamanho da entrada com o máximo e o mínimo respectivamente e denominá-las funções de complexidade de pior caso e de melhor caso.

Funções de calculo de melhor caso e de pior caso pro algoritmo "Extremos":

Pior caso - Fp(n) = 4+3(n-1)+2n = 5n+1

Algoritmo	Passos
se |V| = 0 entao: | retorne 0,0	+1
maior ←1	+1
menor ←1	+1
para i ←2,3,...,|V| faça:	+2n
| se V[menor] > V[i] entao:	+n
| | menor←i	+n <---|
| se V[maior] < V[i] entao:	+n |> ou, totalizando 3(n – 1)
| | maior←i	+n <---|
retorne maior, menor	+1

Melhor Caso - Fm(n) = 4+2(n-1)+2n = 4n+2

Algoritmo	Passos
se |V| = 0 entao: | retorne 0,0	+1
maior ←1	+1
menor ←1	+1
para i ←2,3,...,|V| faça:	+2n
| se V[menor] > V[i] entao:	+n <---|
| | menor←i	+n |> ou, totalizando 2(n – 1)
| se V[maior] < V[i] entao:	+n <---|
| | maior←i	+n
retorne maior, menor	+1

Entradas de pior caso:

São todas as entradas que, passadas para o algoritmo, o forçam a executar o máximo de passos.

Entradas de melhor caso:

São todas as entradas que, passadas para o algoritmo, o forçam ao executar o mínimo de passos.

Supondo outras função de complexidade:

$Gp(n) = 3.n^2+2n+1$
$Gp(5) = 3(25)+2(5)+1 = 86$

$Fp(n) = n^2+1$
$Fp(5) = 26$

Já sabemos qual algoritmo é mais eficiente, comparando as duas complexidades.
Vamos comparar algoritmos com relação ao seu comportamento quanto a variação no tamanho da entrada.
Para isso, vamos nos ater as comparações que levem em conta as ordens de grandezas das complexidades dos algor timos.
Para as definições a seguir, consideraremos funções naturais de domínio natural.

Definição:

Dadas as funções f e g, dizemos que g e limite superior assintótico para f quando existem constantes c pertencentes aos reais iguais ou maiores a 0 e n0 pertence aos naturais, tais que: f(n) <= c * g(n) para todo n>=n0

Dadas as funções f e h, dizemos que h e o limite inferior assintótico para f se existem constantes c pertencentes aos reais maiores ou iguais a 0 e n0 pertencente aos naturais tais que: 0 <= c * h(n) <= f(n) para todo n>=n0

Dadas duas funções f e k, dizemos que k e equivalente assintótico quando k é tanto limite superior como inferior assintótico para f.

Os conjuntos O(g), Ω(g) e Θ(g) são respectivamente os conjuntos de todas as funções para as quais a função g é limite superior, limite inferior e equivalente assintótico.

Limites Assintóticos:

- Quando f(n) pertence a O(g(n)), g é limite superior assintótico para f
- Quando f(n) pertence a Ω(g(n)), g e limite inferior assintótico para f
- Quando f(n) pertence a Θ(g(n)), g é equivalente assintótico para f

Caminho para provar O(g(n)):

Tendo: $f(n) = 2n^2-3n+100$ e $g(n) = n^2$

Vamos provar g(n) é um limite superior assintótico, demonstrando que existe uma constante que corresponderia a definição de limite superior assintótico.

$2n^2 – 3n + 100 <= c * n$, para todo n >= n0

Existem constantes 'c' que pertencem aos reais maiores ou iguais a 0 e n0 pertencente aos naturais, tais que:

$2n^2 – 3n + 100 <= c * n$, para todo n >= n0
$2n^2 – 3n + 100 <= 2n^2 – 3n + 100n^2$
$<= 2n^2 + 100n^2$
$<= 102n^2$ Isso prova, pois demonstra uma possível constante, que corresponde ao significado de O(g(n)) onde essa constante fica oculta na função.

Caminho para provar Ω(g(n))

Vamos provar que: $0 <= c * n^2 <= 2n^2 – 3n + 100$, para todo $n >= n0$

$2n^2-3n+100 >= 2n^2-3n$
$>= (2n-3)n$
$>= (2n-3)n$
$>= n^2$

Exemplo:

Tendo: $f(n) = n^3 / 10 - 7n^2 + 5n - 200$

- Provar que $g(n) = n^3$ é O(g(n))

$n^3/10 - 7n^2 + 5n - 200 <= n^3 + 5n^3$ para todo $n >= 0$
$<= 6n^3$

- Provar que $g(n) = n^3$ é Ω(g(n))

$n^3/10 - 7n^2 + 5n - 200 >= n^3/10 - 7n^2 - 200n^2$
$>= n^3/10 - 207n^2$
$>= (n/10 – 207)n^2$
$>= 4140n^2$

Caminho para provar f(n) pertence a Θ(g(n))

Usando a função dos últimos exemplos temos que provar que:
$0 <= c * g(n) <= f(n) <= C2g(n)$, para todo $n >= n0$
temos que: $0 <= 1/20 * g(n) <= f(n)$, para todo $n>=4140$
e que: $f(n) <= 6g(n)$, para todo $n>0$

Tendo que g(n) satisfaz tanto O quanto Ω, ele é Θ

Provando que um f(n) não pertence a O(g(n))
tendo $f(n) = n^2$ e $g(n) = 9n + 9$

Suponha, por absurdo, que existem c e n0 tais que
$n^2 <= c(9n+9)$, para todo $n>=n0$
$n^2 <= 9cn + 9c$
$n^2 <= 9cn + 9cn$
$n^2 <= 18cn$
$n <= 18c$ --absurdo, pois n é variável, ficando maior que 18c em algum momento

---

Outros modos de provar

f(n) = 3n _____ f(n) = O(g(n))
$g(n) = 2^2$ /

__| 1 | 2 | 3 | 4 | 5 | ...
3n| 3 | 6 | 9 | 12 | 15 | ...
2^2| 2 | 4 | 8 | 16 | 32 | ...
Vemos que $3n <= 2^n$ para todo $n >= 4$

Prova indutiva
Base: n=4
f(4) = 3 * 4 = 12 <= 16 = 2^4 = g(4)

Passo: f(n) <= g(n) → f(n+1) <= g(n+1)
f(n + 1) = 3(n + 1) = 3n + 3 <= 3n + 3n <= 2^n+2^n
<= 2^(n+1)
<= g(n+1) --provado

Limites Assintóticos, em suma:
-Se f(n) pertence a O(g(n)) então existe limite: lim(n→+∞) f(n)/g(n)
-Se f(n) pertence a Ω(g(n)) então existe limite: lim(n→+∞) g(n)/f(n)

Propriedades de Contagem de Passos de Algoritmo

Como vamos nos preocupar daqui pra frente com as ordens de grandeza e não com as funções em si, vamos utilizar a notação: f = O(g), f = Ω(g) ou f = Θ(g)

- Notação O

1) f=O(f) --reflexiva

2) f1 = O(g1) e f2=O(g2) => f1+f2 = O(g1+g2) --soma (obs.: max)
f1 <= c1*g1 -Para todo n >= n1
f2 <= c2*g2 -Para todo n >= n2
f1+f2 <= c1*g1 + c2*g2 -Para todo n >= max{n1,n2}
<= c*g1 + c*g2
= c(g1+g2)

3) f1 = O(g1) e f2 = O(g2) -> f1*f2 = O(g1*g2) --multiplicação (obs.: max)
f1 <= c*g1 -Para todo n >= n0
f2 <= c*g2 -n0 = max{n1,n2}
-c = max{c1,c2}
_____
f1*f2 <= c^2*g1*g2

4) Para todo k pertencente aos , f = O(g) -> k*f = O(g) --multi menor
Caso k < 0, k*f <= f <= c1*g -Para todo n >= n1

5) f = O(n^k), com k>=0, -> f=O($n^{(k+j)}$)
*) n^k <= n^k+j -k e j >= 0 - Para todo J >= 0
f <= c*n^k <= c*$n^{(k+j)}$ - Para todo n >= n0
*) Caso n=0. OK
*) Caso n=1. Ok
*) Caso n > 1
Por absurdo, suponha que existe n tal que n^k > $n^{(k+j)}$. Seja x esse valor para n.
Como x^k > 0, temos x^k > $x^{(k+j)}$ → $\log^{x^x^k}$ > $\log^{x^x^{k+j}}$ → k > k+j → 0 > j
Absurdo. Logo n^k <= $n^{(k+j)}$

6) f = O(g) e g = O(h) -> f=O(h) --transitividade
7) f1 = O(g) e f2 = O(g) -> f1+f2 = O(g) --dupla soma

8) $\log_a^n = O(\log_b^n)$ para todo a,b /= 1 e > 0
$\log_a^n = x \rightarrow a^x = n$ __\ $a^x = a^y$
$\log_b^n = y \rightarrow b^y = n$ /
$\log_b^n = \log_a^n / \log_a^b \rightarrow \log_a^n <= \log_a^b * \log_b^n$ -Para todo n>=1
$c = \log_a^b$ $\log_a^n = 0(\log_b^n)$, n0=1

9) k pertence aos Reais, k=O(1)
k < =c -Para todo n>=0
c = k + 1 -n0 = 0

10) f = O(g) <=> g = Ω(f)
0 <= f <= c*g -Para todo n >= n1
0 <= f*1/c <= g -Para todo n >= n1

Propriedades de Notação Θ:
Θ(1) → constantes
Θ(lg n) → logarítmicos
Θ(n) → lineares
Θ(n lgn) → linearítmicos
$\Theta(n^2)$ → quadráticos
$\Theta(n^e)$ → polinomial
$\Theta(c^n)$ → exponencial

Algoritmo: InsertionSort(V)
Entrada: Vetor V
Objetivo: V[1...|V|] ordenado não-decrescente.

```
para i ←2,3...|V| faça
|  j ← i – 1
|  valor ← V[i]
|  enquanto j >= 0 e V[j] > valor faça
|  |  V[j+1] ← V[j]
|  |  j ← j-1
|  V[j+1] ← V[valor]
```

Valor de i no para	Repetições do enquanto
2	1
3	2
.	.
.	.
.	.
n	n-1
=	
$((1+(n-1))(n-1))/2$ =	$\Theta(n^2)$