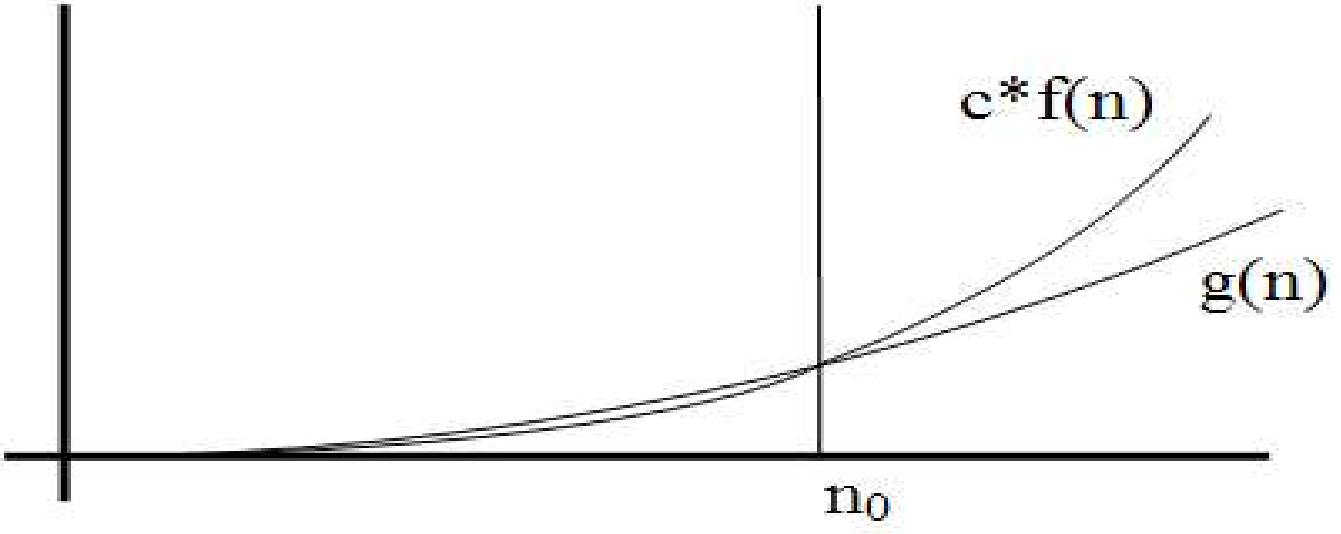
Cabe aqui ressaltar a importância dessa análise. Apressadamente poder-se-ia supor que, com a incrível melhoria em desempenho dos equipamentos recentes frente aos de algum tempo atrás, e mesmo com a expectativa de ainda melhores *performances* no futuro, seria desnecessária uma preocupação com a qualidades de uma solução em termos de eficiência. No entanto, o que se observa é um crescimento também acelerado na dificuldade dos problemas submetidos ao computador, além de um aumento significativo na quantidade de dados (tamanho de entradas). Ou seja: as máquinas melhoram mas os problemas *pioram*, e até numericamente pode-se verificar que uma solução de baixa qualidade torna-se economicamente *imune* ao aumento da capacidade de processamento das máquinas. Por fim, certamente há um limite para o aumento de desempenho das máquinas. Então, a escolha da melhor solução se tornará crítica.

A notação O

Diz-se que uma função **g(n)** é **O(f(n))**, notando-se **g = O(f(n))** se existir alguma constante c > 0 e um inteiro n_0, tal que **n > n_0** implica **g(n) <= c* f(n).**



A notação O

Diz-se também que **g(n)** tem *taxa de crescimento proporcional a f(n)*, é de *ordem máxima f(n)*, de *magnitude f(n)*, de *complexidade f(n)* ou simplesmente que é **O** *de f(n)*.

Isto é interpretado como uma constatação de que **f** expressa um limite superior para valores assintóticos de **g**. Ou que **O**(*f(n)*) tem como valor uma quantidade não conhecida explicitamente, sabendo-se que não excede c**f*(*n*), se n for suficientemente grande ($n>n_0$). Este n_0 é então o ponto a partir do qual *g(n)* é seguramente menor que (ou é ultrapassada por) *f(n)*.

UNIVASF

A notação O

Exemplo:

se $f(n)= n^2 + 1$, então $g(n)=\mathbf{O}(n^2)$;

também: $f(n)=n^3+5n^2-3$ é $\mathbf{O}(n^3)$;

$f(n)=517$ é $\mathbf{O}(1)$;

$f(n)=k2^n$ é $\mathbf{O}(2^n)$.

É bom ressaltar que $f(n)=n^2+1$ também é $\mathbf{O}(n^3)$ e $\mathbf{O}(n^4)$... pois estas expressões satisfazem a relação estabelecida.

Ao se tomar o tempo de execução T(n) de um algoritmo com uma função **g**, na implicação acima, pode-se naturalmente considerar sobre sua magnitude, ou sua complexidade **f**.

UNIVASF

A notação O

No trabalho com estruturas de dados as complexidades costumeiras, em ordem crescente, são as seguintes:

- **O**(1) ou constante;
- **O**(log n) ou logaritímica;
- **O**(n) ou linear;
- **O**(n log n) ou n log de n;
- **O**(n^2) ou quadrática;
- **O**(n^3) ou cúbica.

Embora não se apresente aqui um método formal para determinação da magnitude de uma função, um conjunto de regras práticas pode ajudar. As considerações anteriores sobre simplificações da expressão da taxa de crescimento do tempo de execução valem para a determinação de **O**(f(n)) de algoritmos, pois atendem à propriedade descrita.

UNIVASF

A notação O

Por exemplo:

a. regra da complexidade *polinomial*:

se P(n) é um polinômio de grau k, então P(n) = $\boldsymbol{O}(n^k)$**.**

Isto é certo, pois para valores grandes de *n,* os termos adicionais do polinômio podem ser desprezados, e se terá constantes *c* e n_0 tais que, para todo $n > n_0$, $P(n) <= c*n^k$. Além disso, é certo que:

b. $f(n) = \boldsymbol{O}(f(n))$;

UNIVASF

A notação O

c. regra da constante:

O(c*f(n)) = c* **O**(f(n)) = **O**(f(n));

O(f(n)) + **O**(f(n)) = **O**(f(n));

e. regra da soma de **tempos**:

se $T_1(n) = \mathbf{O}(f(n))$ e $T_2(n) = \mathbf{O}(g(n))$ então $T_1(n) + T_2(n) = \mathbf{O}(\max(f(n),g(n)))$

Isto significa que a complexidade de um algoritmo com dois trechos em sequência com tempos de execução diferentes é dada como a complexidade do trecho de maior complexidade.

UNIVASF

A notação O

f. regra do produto de **tempos**:

se $T_1(n) = \mathbf{O}(f(n))$ e $T_2(n) = \mathbf{O}(g(n))$ então

$T_1(n) * T_2(n) = \mathbf{O}(f(n) * g(n))$

Isto significa que a complexidade de um algoritmo com dois trechos aninhados, em que o segundo é repetidamente executado pelo primeiro, é dada como o produto da complexidade do trecho mais interno pela complexidade do trecho mais externo.

UNIVASF

Complexidade de algumas estruturas de controle

Regras rígidas sobre o cálculo da complexidade de qualquer algoritmo não existem, cada caso deve ser estudado em suas condições.

No entanto, as estruturas de controle clássicas da programação estruturada permitem uma estimativa típica de cada uma.

A partir disso, algoritmos construídos com combinações delas podem ter sua complexidade mais facilmente estabelecida.

UNIVASF

a. **comando simples –** tem um tempo de execução constante, **O**(c) = **O**(1).

b. **sequência –** tem um tempo igual à soma dos tempos de cada comando da sequência; se cada comando é **O**(1), assim, também será a sequência; senão, pela regra da soma, a sequência terá a complexidade do comando de maior complexidade.

c. **alternativa –** qualquer um dos ramos pode ter complexidade arbitrária; a complexidade resultante é a maior delas; isto vale para alternativa dupla (*if-else*) ou múltipla (*switch*).

UNIVASF

d. repetições

i. *repetição contada*: é aquela em que cada iteração (ou "volta") atualiza o controle mediante uma *adição* (geralmente, quando se usa uma estrutura do tipo *for*, que especifica incremento/decremento automático de uma variável inteira).

Se o número de iterações é independente do tamanho do problema, a complexidade de toda a repetição é a complexidade do corpo da mesma, pela regra da constante (ou pela regra da soma de tempos).

```
for (i=0; i<k ; i++)          // se k não é f(n) então
    trecho com O(g(n))        // o trecho é O(g(n))
```

```
for (i=0; i<10 ; i++)        // isto é O(1), logo toda
{                            // a repetição é O(1)
   x = x+v;
   printf ("%d", x);
}
```

Se o número de iterações é função de **n**, pela regra do produto teremos a complexidade da repetição como a complexidade do corpo multiplicada pela função que descreve o número de iterações. Isto é:

```
for (i=0; i<n ; i++)         // como o número de iterações é f(n)=n
                             // então o trecho é O(n*g(n))
```

Exemplo:

```
for (i=0; i<k*n ; i++)       // o trecho é O(f(n)*g(n)), no caso
    trecho com O(log n)      // O(k*n*log n), ou seja: O(n log n)
```

UNIVASF

Uma aplicação comum da regra do produto é a determinação da complexidade de repetições aninhadas.

Exemplo:

```
for (i=0; i<n ; i++)          // o trecho é O(f(n)*g(n)), no caso
    for (j=0; j<n ; j++)      // g(n)=n*1 (laço interno); logo,
        trecho com O(1)       // O(n*n), ou seja: O(n²)
```

Exemplo:

```
for (i=1; i<=n ; i++)         // o laço interno é executado 1+2+3
    for (j=1; j<=i ; j++)     // +...n-1 +n=n*(n+1)/2 vezes, logo,
                              // O(n*(n+1)/2), ou seja:
        trecho com O(1)       // O(0.5(n²+n)) ou seja O(n²)
```

UNIVASF

Exemplo:

```
for (i=1; i<=n ; i++)
    for (j=n; i<=j ; j--)
      trecho com O(1)
```

O laço interno é executado n+n-1 +n-2+...+2+1=n*(n+1)/2 vezes, ou seja: $O(n^2)$ como no caso anterior

Os dois últimos exemplos podem ser generalizados para quaisquer aninhamentos de repetições contadas em *k* níveis, desde que todos os índices dependam do tamanho do problema. Nesse caso, a complexidade da estrutura aninhada será da ordem de n^k.

Exemplo:

```
for (IndExt=1; IndExt<=n ; IndExt++)
   for (IndMed=IndExt; IndMed<=n ; IndMed++)
     for (IndInt=1; IndInt<=IndMed; IndInt++)
       trecho com O(1)
```

O laço mediano é executado n + n-1 + n-2 +…+ 2 +1= $(n^2+n)/2$ vezes; o laço mais interno será executado no máximo n vezes; logo, tem-se $O((n^2+n)*n)$, ou seja:$O(n^3)$

UNIVASF

ii. *repetição multiplicativa:* é aquela em que cada iteração atualiza o controle mediante uma *multiplicação* ou *divisão*.

Exemplo:

```
limite=1;                  // o número de iterações depende de n;
while (limite<=n)          // limite vai dobrando a cada iteração;
{                          // depois de k iterações, limite = 2^k e k = log2 limite;
   trecho com O(1)         // como o valor máximo de  limite é n, então o trecho
   limite = limite*2;      // é O(log2n) = O(log n)
}
```

OBS: Na verdade **O**(log n) independe da base do logaritmo, pois $\log_a n = \log_a b * \log_b n = c * \log_b n$.

Exemplo:

```
/* o número de iterações depende de n; limite vai-se
subdividindo a cada iteração; depois de k=log2n
iterações, encerra; então o trecho é O(log n)*/
int limite;
for (limite=n; limite!=0; limite /=2)
    trecho com O(1)
```

Os dois exemplos anteriores também podem ser generalizados, adotando-se um fator genérico de multiplicação *fator*. Nesse caso, o número de iterações será dado por $k = log_{fator}\ limite$ = **O**(log f(n)), se o limite é função de n.

Exemplo:

```
int limite=n;
while (limite!=0) {
    for (i=1; i<=n; i++)
        trecho com O(1)
    limite = limite/2;
}
```

```
/* o número de iterações depende
   de n; limite vai-se subdividindo
   a cada iteração; o laço interno
   é O(n), o externo O (log n);
   logo, o trecho é O (n log n)*/
```

e. Chamada de Procedimento

Pode ser resolvida considerando-se que o procedimento também tem um algoritmo com sua própria complexidade. Esta é usada como base para cálculo da complexidade do algoritmo invocador.

Por exemplo: se a invocação estiver num ramo de uma alternativa, sua complexidade será usada na determinação da máxima complexidade entre os dois ramos; se estiver no interior de um laço, será considerada no cálculo da complexidade da sequência repetida, etc.

UNIVASF

A questão se complica ao se tratar de uma chamada recursiva.

Embora não haja um método único para esta avaliação, em geral a complexidade de um algoritmo recursivo será função de componentes como: a *complexidade da base* e do *núcleo* da solução e a *profundidade da recursão.* Por este termo entende-se o *número de vezes que o procedimento é invocado recursivamente.* Este número, usualmente, depende do *tamanho do problema* e da *taxa de redução do tamanho do problema* a cada invocação; E é na sua determinação que reside a dificuldade da análise de algoritmos recursivos.

UNIVASF

Como exemplo, considere o algoritmo para o cálculo do fatorial.

```
int fatorial (int n)
{
    if (n==0)
        return 1;                    // Base
    else
        return n*fatorial(n-1);      //Núcleo
}
```

A redução do problema se faz de uma em uma unidade, a cada reinvocação do procedimento, a partir de *n,* até alcançar *n = 0.* Logo, a profundidade da recursão é igual a *n*. O núcleo da solução (que é repetido a cada reinvocação) tem complexidade **O**(1), pois se resume a uma multiplicação. A base tem complexidade **O**(1), pois envolve apenas uma atribuição simples. Nesse caso, conclui-se que o algoritmo tem um tempo $T(n) = n*1+1 =$ **O**(n).

f. Desvio Incondicional

A discussão anterior presumiu que os algoritmos sejam construídos com as estruturas da programação disciplinada. O uso indiscriminado de desvios incondicionais há de incrementar a dificuldade do cálculo da complexidade. No entanto, o **go to** não causará problemas em casos de uso restrito:

- quando for usado como desvio para a frente, associado a uma condição de parada, apenas para sair de um laço de repetição, diretamente para o comando subsequente ao corpo da repetição;

UNIVASF

nesse caso, o tempo do pior caso não é afetado, pois se assume que o pior caso é quando tal desvio nunca é executado e a repetição ocorre um número máximo de vezes;

- quando for um desvio para trás, estabelecendo uma repetição adequadamente estruturada (ou seja: desde que os laços estejam separados ou completamente aninhados); nesse caso, a análise pode ser feita sobre a estrutura de repetição resultante.

UNIVASF

Comparação Quantitativa

Pode-se ter uma ideia do que alguns autores chamam de “tirania da taxa de crescimento” observando-se o comportamento de diversos algoritmos de complexidade diferente, todos dedicados à solução do mesmo problema, sob as mesmas condições de processamento.

Nos valores a seguir, assume-se que uma operação elementar é executável em um décimo de microssegundo *(0,1 * 10^{-6}s).*

UNIVASF

É digna de nota a taxa de crescimento dos algoritmos de ordem exponencial. Em geral, seu desempenho torna-se de custo proibitivo, devendo ser usados apenas quando não se conheça solução de menor complexidade.

Complementarmente, pode-se considerar o efeito do *aumento da capacidade de processamento* sobre

FUNÇÃO DE COMPLEXIDADE	n (tamanho do problema)		
	20	40	60
n	0.0002 s	0.0004 s	0.0006 s
$n \log_2 n$	0.0009 s	0.0021 s	0.0035 s
n^2	0.0040 s	0.0160 s	0.0360 s
n^3	0.0800 s	0.6400 s	2.1600 s
2^n	10.0000 s	27 dias	3660 séculos
3^n	580 minutos	38550 séculos	$1.3*10^{14}$ séculos

o *tamanho do maior problema solucionável em um certo tempo*. A tabela a seguir apresenta resultados para os mesmos algoritmos, executados na máquina original e em máquinas 100 e 1000 vezes mais rápidas, tomando-se como básico o tamanho do problema solucionável em 1 hora de processamento.

FUNÇÃO DE COMPLEXIDADE	Tamanho da maior instância solucionável em 1 hora		
	máquina original	máquina 100 vezes mais rápida	máquina 1.000 vezes mais rápidas
n	N	100 N	1.000 N
$n \log_2 n$	N1	22.5 N1	140.2 N1
n^2	N2	10 N2	31.6 N2
n^3	N3	4.6 N3	10 N3
2^n	N4	N4 + 6	N4 +10
3^n	N5	N5 + 4	N5 + 6

Como já foi citado, algoritmos de alta complexidade têm um ganho pouco significativo em função da capacidade da máquina.

Os algoritmos exponenciais, principalmente, são quase imunes: no exemplo da tabela, o algoritmo com $O(2^n)$ tem um ganho de apenas *10 unidades* no tamanho do problema solucionável com uma máquina *1000 vezes* mais rápidas!

Na prática, são consideradas aceitáveis complexidades no máximo polinomiais na ordem de n^2.

UNIVASF

Exercício: O algoritmo a seguir calcula o valor do somatório de um determinado número de termos da série:

$$0, -\frac{1}{4!}, +\frac{1}{6!}, -\frac{2}{8!}, +\frac{3}{10!}, -\frac{5}{12!}, \ldots$$

Sendo que o número de termos a serem somados é fornecido pelo usuário.

$O(n^2)$

```
algoritmo "Exemplo"
var
   ind, i, fat, fib, aux, n: inteiro
   s: real
inicio
   s <- 0
   escreva ("Entre com o número de termos a
      serem somadas: ")
   leia (n)
   para ind de 1 ate n faca
      fat <- 1
      para i de 2 ate ind*2 faca
         fat <- fat * i
      fimpara
      i <- ind
      fib <- 0
      aux <- 1
      enquanto (i>1) faca
         aux <- fib + aux
         fib <- aux - fib
         i <- i - 1
      fimenquanto
      s <- s + -1^(ind+1) * fib / fat
   fimpara
   escreval ("O valor do somatório é: ", s)
fimalgoritmo
```