

Introdução à Análise de Algoritmos

Quanto tempo leva a execução de determinado algoritmo?

Quando temos dois algoritmos que fazem a mesma coisa, qual deles leva menos tempo?

A análise do algoritmo preocupa-se com as questões acima.

É sempre conveniente conhecer ou ter uma medida da eficiência de um algoritmo ao programá-lo.

O comportamento de alguns algoritmos

Raízes de equação do 2. grau

```
void raiz(double a, double b, double c, double *x1, double *x2) {  
    ...  
    *x1 = (-b+sqrt(b*b-4*a*c))/(2*a);  
    *x2 = (-b-sqrt(b*b-4*a*c))/(2*a);  
}
```

Se desconsiderarmos os casos particulares (delta negativo, a=0, etc.), esse algoritmo realiza sempre o mesmo número de operações.

Podemos afirmar então que o tempo que esse algoritmo leva é uma constante.

$t = k$

Máximo de uma seqüência de n elementos

```
float max(float a[], int N) {  
    int i; float m;  
    m=a[0];  
    for (i = 1; i < N; i++) if (m < a[i]) m = a[i];  
    return m;  
}
```

Esse algoritmo sempre repete um conjunto de operações N-1 vezes.

Podemos afirmar então que seu tempo é proporcional a N-1 mais uma constante.

$t = k_1 + k_2*(N-1)$

Conta a quantidade de nulos num vetor de N elementos

```
int nulos(float a[], int N) {  
    int i, c;  
    c = 0;  
    for (i = 0; i < N; i++) if (a[i] == 0) c++;  
    return c;  
}
```

Idem ao anterior. Repetindo N vezes. Portanto o seu tempo é da forma:

$$t = k_1 + k_2 \cdot N$$

Verifica se dois vetores de N elementos são iguais

```
/* devolve 1 se iguais e 0 se diferentes */
int compara(int a[], int b[], int N) {
    int i;
    for (i = 0; i < N; i++)
        if (a[i] != b[i]) return 0;
    return 1;
}
```

Neste caso o resultado depende dos dados, pois termina no primeiro elemento diferente encontrado. No pior caso (todos iguais ou o último diferente) também é proporcional a N.
 $t = k_1 + k_2 \cdot N$

Conta os algarismos significativos de um inteiro

```
int num_algar(int x) {
    int c=0;
    while (x != 0) {c++; x = x / 10;}
    return c;
}
```

outra forma:

```
int num_algar(int x) {
    int c;
    for (c = 0; x > 0; c++, x = x / 10;)
        return c;
}
```

O resultado c é o menor inteiro maior que $\log x$ (base 10).
 $10^{c-1} \leq x \leq 10^c$

O tempo é então proporcional a $\log x$.
 $t = k_1 + k_2 \cdot \log(x)$

Conta quantos bits significativos tem um inteiro

```
int num_bits(int x) {
    int c=0;
    while (x != 0) {c++; x = x / 2;}
    return c;
}
```

outra forma:

```
int num_bits(int x) {  
    int c;  
    for (c = 0; x > 0; c++, x = x / 2;)  
        return c;  
}
```

O resultado c é o menor inteiro maior que $\lg x$ (base 2).
 $2^{c-1} \leq x \leq 2^c$

O tempo é então proporcional a $\lg x$.
 $t = k_1 + k_2 \cdot \lg(x)$

Notação:

$\log x$ - (base 10)
 $\lg x$ - (base 2)
 $\ln x$ - (base e) – logaritmo natural

Observe que podemos dizer que os dois últimos exemplos acima são proporcionais ao logaritmo, sem mencionar a base, pois:

$\log N = \lg N / \lg 10$ ou $\log N = k \cdot \lg N$

Imprimir tabela de i / j ($1 \leq i \leq N$; $1 \leq j \leq M$)

```
void imp_tabela (int N, int M) {  
    int i, j;  
    for (i = 1; i <= N; i++) {  
        for (j = 1; j <= M; j++) printf("%8.3lf ", (float)i / (float)j);  
        printf("\n");  
    }  
}
```

O tempo é proporcional a $N \cdot M$. Um limitante superior é N^2 , supondo N o maior deles.
 $t = k_1 + k_2 \cdot N^2$.

Multiplicar matriz $A[N \times M]$ por vetor $X[M]$

Faça o algoritmo. São dois comandos **for** encaixados:

O tempo é proporcional a $N \cdot M$. Um limitante superior é N^2 , supondo N o maior deles.
 $t = k_1 + k_2 \cdot N^2$.

Idem imprimindo a tabela $i*j*k$ ($1 \leq i \leq N$; $1 \leq j \leq M$; $1 \leq k \leq P$)

Serão três comandos **for** encaixados. Neste caso será proporcional a $N*M*P$.

Podemos considerar $N*N*N$, supondo N o maior deles.

$$t = k_1 + k_2 * N^3.$$

Multiplicar matriz $A[N \times M]$ por matriz $X[M \times P]$

Faça o algoritmo. Serão três comandos **for** encaixados:

O tempo é proporcional a $N*M*P$. Um limitante superior é $N*N*N$, supondo N o maior deles.

$$t = k_1 + k_2 * N^3.$$

Proporcionalidade dos algoritmos anteriores:

Os algoritmos acima são proporcionais a:

1 – sempre executam as mesmas instruções uma só vez. Dizemos que o tempo de execução neste caso é uma constante.

N – dependem apenas de um parâmetro que é o número de vezes que um determinado laço é executado.

$\log N$ – Não importa a base, pois $\log N$, $\lg N$ ou $\ln N$ são proporcionais. Assim não vamos indicar a base. Vamos dizer apenas que o algoritmo leva um tempo logarítmico para ser executado. Note também que um algoritmo com essa característica é muito interessante. Quando N é 1.000 o tempo é proporcional a 3. Quando N fica 1.000 vezes maior (1.000.000) o tempo apenas dobra (proporcional a 6).

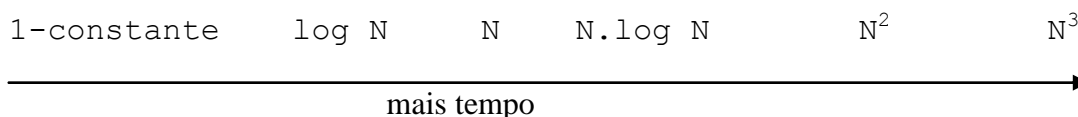
N^2 – em geral esses algoritmos têm um **for** encaixado em outro **for**. Quando N é 1.000, o tempo é proporcional a 1.000.000. Quando N dobra o tempo multiplica por 4.

N^3 - em geral possuem um **for** dentro dum um **for** dentro de um **for**. Quando N dobra, o tempo se multiplica por 8.

Outros algoritmos podem ainda serem proporcionais a $N \cdot \log N$, ou exponenciais, proporcionais a 2^N ou 10^N .

Um pouco de intuição

É intuitivo que para valores de N grandes, quanto maior a ordem do expoente, mais tempo leva o algoritmo.



Sobre os algoritmos

O tempo depende dos dados em cada execução do algoritmo.

Pode ser que para um conjunto de dados o algoritmo $A1$ seja mais rápido que o $A2$. Para outro conjunto o tempo pode se inverter.

Um algoritmo pode levar um tempo pequeno quando N é pequeno, mas demorar muito quando N é grande. Por exemplo, um algoritmo que tem um tempo proporcional a N^3 .

Às vezes não interessa muito se o algoritmo leva um tempo maior ou menor, pois a execução numa máquina é tão rápida que não faz diferença usarmos o algoritmo $A1$ ou $A2$.

O que realmente dá para se afirmar a respeito do tempo que um algoritmo vai demorar?

A notação $O(f(N))$ – Ordem de $f(N)$ ou notação Grande-O

Para expressar essa idéia de tempo proporcional a alguma função, foi inventada a notação $O(f(N))$ – Ordem de $f(N)$ ou ainda notação Grande-O (big-O notation).

Definição:

Dizemos que $g(N)$ é $O(f(N))$ se existirem constantes c_0 e N_0 tais que $g(N) < c_0 f(N)$ para todo $N > N_0$. Ou seja, a partir de um determinado N , a função $f(N)$ multiplicada por uma constante é sempre maior que $g(N)$. Veja o gráfico abaixo.

Outra forma é definir $O(f(N))$ como um conjunto:

$$O(f(N)) = \{ g(N) \text{ se existem constantes } c_0 \text{ e } N_0 \text{ tais que } g(N) < c_0 f(N) \text{ para todo } N > N_0 \}$$

Podemos dizer livremente que $g(N) = O(f(N))$, mas o mais correto é dizer: $g(N)$ é $O(f(N))$ ou $g(N) \in O(f(N))$.

Essa definição elimina os termos que contribuem em menor grau para o tempo total. Também é usada para classificar os algoritmos por um limitante superior.

Veja que c_0 e N_0 escondem coisas importantes sobre o funcionamento do algoritmo:

- Nada sabemos para $N < N_0$.
- c_0 pode esconder uma enorme ineficiência do algoritmo – por exemplo, é melhor N^2 nano-segundos que $\log N$ séculos.

Só interessa o termo de maior ordem.

$O(1)$ é o mesmo que $O(2)$ que é o mesmo que $O(K)$ – constante

$O(1+N+N^2)$ é $O(N^2)$

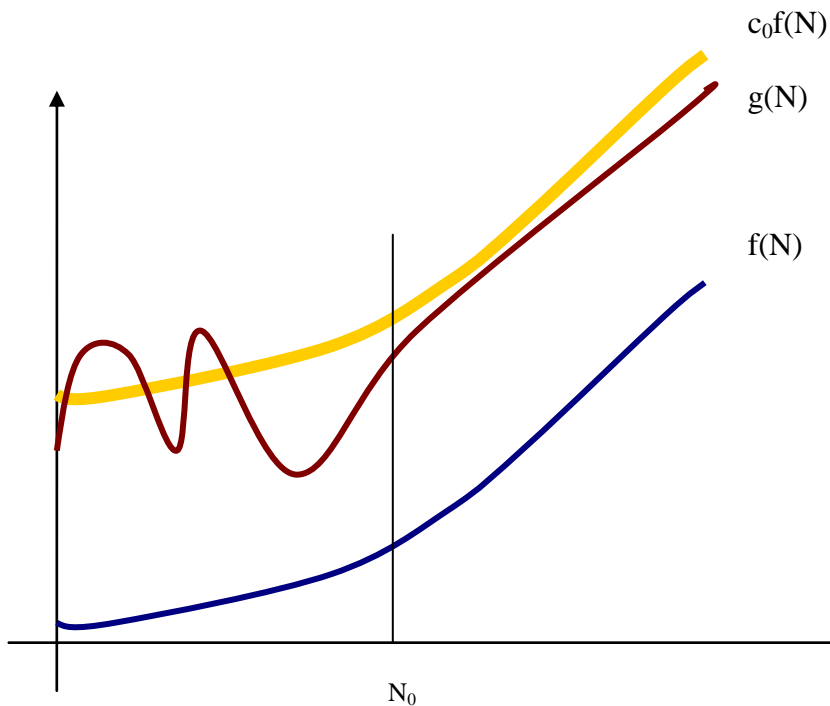
Observe que $1+N+N^2 < N^2 + N^2 + N^2 = 3 N^2$ para $N > N_0$

$O(N \cdot \log N + N^2)$ é $O(N^2)$

Observe que $N \cdot \log N + N^2 < N^2 + N^2 = 2 N^2$ para $N > N_0$

Propriedades:

- $O(f(N)) + O(g(N))$ é $O(\max\{f(N), g(N)\})$
- $O(f(N)) \cdot O(g(N))$ é $O(f(N) \cdot g(N))$
- $O(k \cdot f(N))$ é $O(f(N))$ desde que $k \neq 0$



Exemplo de análise de algoritmos

Vejamos um algoritmo simples e algumas características de sua análise.

Algoritmo de busca seqüencial:

```
/* procura x em a[0], a[1], ... a[N-1] */  
int busca(int a[], int x, int N) {  
    for (i = 0; i < N; i++) if(a[i] == x) return i;  
    return -1;  
}
```

Existem muitas variações deste algoritmo quando se programa em C, mas todas elas têm que percorrer o vetor.

Quantas comparações são necessárias até encontrar o elemento procurado ou concluir que ele não está na tabela?

Melhor caso: 1 - uma só comparação quando $x == a[0]$.

Pior caso: N – quando não encontra ou $x == a[N-1]$.

Caso médio: $(1+N)/2$ – média entre o pior e o melhor ?

Para considerarmos a média entre o melhor e o pior caso, estamos assumindo uma hipótese importante. A probabilidade de ser qualquer valor entre 1 e N é a mesma.

De uma maneira geral, a determinação do pior caso dá uma boa informação de como o algoritmo se comporta e oferece um limitante superior para o tempo que o algoritmo demandará.

O caso médio é o mais interessante de ser determinado, mas nem sempre é possível, pois muitas vezes depende de hipóteses adicionais sobre os dados.

Esse algoritmo é $O(N)$ (linear). Para determinar a ordem supõe-se o pior caso.

Outro exemplo

Determinar qual o último elemento do vetor igual a x.

A melhor solução é a busca seqüencial do fim para o começo do vetor.

```
/* procura x em a[N-1], a[N-2], ... a[0] */  
int busca(int a[], int x, int N) {  
    for (i = N - 1; i >= 0; i--) if(a[i] == x) return i;  
    return -1;  
}
```

Uma solução menos eficiente seria procurar a partir do início do vetor.

```
/* procura x em a[0], a[1], ... a[N-1] */  
int busca(int a[], int x, int N) {  
    int k = -1;  
    for (i = 0; i < N; i++) if (a[i] == x) k=i;  
    return k;  
}
```

Ambos são $O(N)$, embora a primeira solução seja preferível à segunda. O pior caso de ambas é o mesmo.

A notação $O(f(N))$ é a complexidade dos algoritmos

Como já vimos, a notação $O(f(N))$ ignora pontos importantes sobre o algoritmos:

Como ele funciona para N menores

Se vamos rodar num computador lento ou rápido

Mas do ponto de vista de complexidade, o que se pode afirmar e, portanto o que interessa sobre o algoritmo é o seu comportamento assintótico, isto é, qual a curva que melhor descreve o seu comportamento.

Veremos que existem algoritmos com vários comportamentos:

Constantes – $O(1)$

Lineares – $O(N)$

Quadráticos – $O(N^2)$

Polinomiais – $O(N^k)$

Exponenciais - $O(k^N)$

Logarítmicos – $O(\lg N)$ ou $O(N \cdot \lg N)$

Alguns exemplos

Vejamos os algoritmos acima:

Raízes de equação do 2. grau

$O(1)$

Máximo de uma seqüência de n elementos

$O(N)$

Conta a quantidade de nulos num vetor de N elementos

$O(N)$

Verifica se dois vetores de N elementos são iguais

$O(N)$

Conta os algarismos significativos de um inteiro

$O(\log N)$

Conta quantos bits significativos tem um inteiro

$O(\log N)$

Imprimir tabela de i/j ($1 \leq i \leq N$; $1 \leq j \leq M$)

$O(N \cdot M)$ ou $O(N^2)$ se $N > M$

Multiplicar matriz $A[N \times M]$ por vetor $X[M]$

$O(N \cdot M)$ ou $O(N^2)$ se $N > M$

Idem imprimindo a tabela $i*j*k$ ($1 \leq i \leq N$; $1 \leq j \leq M$; $1 \leq k \leq P$)

$O(N \cdot M \cdot P)$ ou $O(N^3)$ se $N > M, P$

Multiplicar matriz $A[N \times M]$ por matriz $X[M \times P]$

$O(N \cdot M \cdot P)$ ou $O(N^3)$ se $N > M, P$

Imprimir todos os números com N dígitos

$O(10^N)$