Análise de Algoritmos

Comportamento Assintótico

- $f(n) = O(1)$ (complexidade constante)
  O uso do algoritmo independe do tamanho de n. Neste caso, as instruções do programa são executadas um número fixo de vezes.

- $f(n) = O(\log n)$ (complexidade logaritmica)
  Ocorre tipicamente em algoritmos que resolvem um problema transformando-o em problemas menores. Nestes casos, o tempo de execução pode ser considerado como sendo menor que uma constante grande . Exemplos: $n = 1000$ -- $\log_2 n \sim 10$;
  $n = 10^6 - \log_2 n \sim 20$.

- $f(n) = O(n)$ (complexidade linear)
  Em geral, um pequeno trabalho é realizado sobre cada elemento de entrada. Esta é a melhor situação possível para um algoritmo que tem que processar n elementos de entrada ou produzir n elementos de saída. Cada vez que n dobra de tamanho n dobra.

Comportamento Assintótico

- $f(n) = O(n \log n)$
  Este tempo de execução ocorre tipicamente em algoritmos resolvidos com o método dividir&conquistar. Exemplos: n = 1 milhão – n log n ~20 milhões;
  n = 2 milhões, n log n ~ 42 milhões, pouco mais que o dobro.

- $f(n) = O(n^2)$ (complexidade quadrática)
  Algoritmos desta ordem de complexidade ocorrem quando os itens são processados aos pares, em um laço dentro de outro. São úteis para resolver problemas de tamanho pequeno – métodos diretos de ordenação.

Comportamento Assintótico

- $f(n) = O(n^3)$ (complexidade cúbica)
  Algoritmos desta ordem são úteis apenas para resolver pequenos problemas.

- $f(n) = O(2^n)$ (complexidade exponencial)
  Algoritmos desta ordem geralmente não são úteis sob o ponto de vista prático. Eles ocorrem na solução de problemas quando se usa a força bruta para resolvê-los.

Exponencial vs Polinomial

- Um algoritmo cuja função de complexidade é $O(c^n)$, $c>1$, é chamado de **algoritmo exponencial de tempo de execução**.
- Um algoritmo cuja função de complexidade é $O(p(n))$, onde $p(n)$ é um polinômio, é chamado de **algoritmo polinomial de tempo de execução**.
- Um problema é considerado **intratável** se ele é tão difícil que não existe um algoritmo polinomial para resolvê-lo, enquanto um problema é considerado **bem resolvido** quando existe um algoritmo polinomial para resolvê-lo.

Princípios para a Análise de Algoritmos

1. Atribuição, leitura e escrita: O(1)

Exceção linguagens que permitem atribuição direta em vetores de tamanho grande

2. Seqüência:

comando de maior tempo

3. Decisão:

avaliação da condição: O(1);

comandos dentro: *regra 2*.

4. Laço:

avaliação do término: O(1);

comandos dentro: *regra 2*.

Princípios para a Análise de Algoritmos

5. Programa com procedimentos não recursivos:

O tempo de execução de cada procedimento deve ser calculado separadamente, um a um.

Inicia-se calculando aqueles procedimento que não chamam outros.

A seguir, são avaliados as sub-rotinas que invocam os procedimentos anteriores, utilizando os tempos já calculados.

6. Programa com procedimentos recursivos:

Para cada procedimento é associada uma função de complexidade f(n) desconhecida, onde *n* mede o tamanho dos argumentos para o procedimento.

Operações com a Notação O

$f(n) = O(f(n))$

Ex.: $2n+5 = O(2n+5) = O(\ n\ )$

$c.\ O(f(n)) = O(f(n))$ se c = constante

Ex.: $3.O(n) = O(n)$

$O(f(n)) + O(f(n)) = O(f(n))$

Ex.: $O(\ lg(n)\ ) + O(\ lg(n)\ ) = O(\ lg(n)\ )$

$O(O(f(n))) = O(f(n))$

Ex.: $O(\ O(n^2)\ ) = O(n^2)$

$O(f(n)) + O(g(n)) = O(max(f(n),g(n)))$

Ex.: $O(n^2) + O(n^3) = O(\ max(n^2,n^3)\ ) = O(n^3)$

$O(f(n))O(g(n)) = O(f(n)g(n))$

Ex.: $O(n)\ .\ O(n^2) = O(\ n\ .\ n^2\ ) = O(n^3)$

$f(n)O(g(n)) = O(f(n)g(n))$

Ex.: $n\ .\ O(n^2) = O(\ n\ .\ n^2\ ) = O(n^3)$

Operações com a Notação O

- **Regra da Soma**

  Suponha 3 trechos de programas cujos tempos de execução são $O(n)$, $O(n^2)$ e $O(n \log n)$.

  O tempo dos 2 primeiros é $O(\max(n,n^2))$ que é $O(n^2)$ e dos 3 é: $O(\max(n^2, n \log n)$ que é $O(n^2)$.

- **Regra do Produto**

  Produto de $[\log n + k + O(1/n)]$ por

  $[n + O(\text{raiz}(n))] = n \log n + kn + O(\text{raiz}(n) \log n)$

Exemplo

/ busca seqüencial */*

```
int BSeq (int A[ ], int n, int x)
{
(1) int i = 0;
(2) while (i < n) {
(3)     if (A[ i ] == x)
(4)        return i;
(5)     i = i + 1;
        }
(6) return -1;
}
```

Esse algoritmo procura por um elemento *x* em um vetor *A* de tamanho *n*.

A procura consiste em verificar, posição por posição do vetor, comparando o elemento armazenado com o valor de *x* (chave) que desejamos localizar.

Assim, quando o elemento for encontrado, a condição dada na linha 3 será verdadeira, executando o comando *return* e, consequentemente, devolvendo o valor corrente de *i*, que é a posição do elemento no vetor.

Caso o valor de *x* não esteja presente no vetor, o laço terminará sua repetição assim que o contador *i* atingir o valor igual a *n*. O que implica que a próxima linha de comando (6) seria executada, retornando o valor -1 indicando que não foi encontrado o elemento no vetor.

Desta forma, pode se perceber que a principal operação é a comparação (3)

Exemplo

```
/* busca seqüencial */

int BSeq (int A[ ], int n, int x)
{
int i = 0;
while (i < n) {
    if (A[ i ] == x)
      return i;
    i = i + 1;
      }
return -1;
}
```

Para uma pesquisa qualquer, algumas situações podem ocorrer:

O valor que estamos procurando estar armazenado na 1ª posição

O valor que estamos procurando estar armazenado na 2ª posição

...

O valor que estamos procurando estar armazenado na kª posição

...

O valor que estamos procurando estar armazenado na última posição

O valor que estamos procurando não estar armazenado no vetor

Exemplo

```
/* busca seqüencial */

int BSeq (int A[ ], int n, int x)
{
(1) int i = 0;
(2) while (i < n) {
(3)     if (A[ i ] == x)
(4)         return i;
(5)     i = i + 1;
        }
(6) return -1;
}
```

Se *x* estiver:

- na 1ª posição: teremos 1 comparação
- na 2ª posição: teremos 2 comparações ...
- na kª posição : teremos *k* comparações
- na última posição: teremos realizado todas as comparações possíveis, ou seja, *n* comparações
- não estiver no vetor : neste caso também teremos *n* comparações

Se pensarmos no pior caso, sempre teremos *n* comparações

Porém, se pensarmos no caso médio, ou seja, que temos igual probabilidade de encontrarmos *x* em todas as posições do vetor, teremos:

$$1/n(1+2+3+4+\ldots+n) = (n+1)/2$$

E portanto, O(n).

Exemplo

```
{ ordena o vetor em ordem ascendente}
Procedure Ordena (var A: vetor);
Var i, j, min, x: integer;
Begin
(1)  For i:= 1 to n-1 do
       Begin
(2)      min := i;
(3)      for j:= i + 1 to n do
(4)          if A[j] < A[min]
(5)             then min := j;
         {troca A[min] e A[i]}
(6)      X := A[min];
(7)      A[mim] := A[i];
(8)      A[i] := X;
       End;
End;
```

O programa contém 2 laços, um dentro do outro.

Laço mais externo: (2) a (8)

Laço mais interno: (4) e (5)

Começamos pelo mais interno:

comando de atribuição e avaliação da condição levam tempo constante.

Como não sabemos se o corpo da decisão vai ou não ser executado, considera-se o pior caso: assumir que a linha (5) sempre será executada.

O tempo para incrementar o índice do laço e avaliar sua condição de termo é também O(1).

O tempo para executar uma vez o laço composto por (3), (4) e (5) é O(max(1,1,1)) = O(1).

→ Seguindo regra da soma para a notação O.

Como o número de iterações do laço é n-i então o tempo gasto é O(n-i . 1) = O(n – i).

→Seguindo a regra do produto para notação O.

Exemplo

```
{ ordena o vetor em ordem ascendente}
Procedure Ordena (var A: vetor);
Var i, j, min, x: integer;
Begin
(1)  For i:= 1 to n-1 do
     Begin
(2)     min := i;
(3)     for j:= i + 1 to n do
(4)          if A[j] < A[min]
(5)              then min := j;
        {troca A[min] e A[i]}
(6)     X := A[min];
(7)     A[mim] := A[i];
(8)     A[i] := X;
     End;
End;
```

O corpo do laço mais externo contém, além do laço interno, os comandos de atribuição nas linhas (2), (6), (7), (8).

O tempo das linhas (2) a (8) é:

$O(\max(1,(n-i),1,1,1)) = O(n-i)$

A linha (1) é executada n-1 vezes, o tempo total do programa é:

$$\sum_{1}^{N-1} (n - i) = n(n - 1)/2 = n^2/2 - n/2 = O(n^2)$$

Considerando o número de comparações como a mais relevante (4): $n^2/2 - n/2$ comparações

Considerando o número de trocas (6), (7), (8) a mais relevante: $3(n-1)$ trocas.