FIAP ON

CÓDIGOS DE ALTA PERFORMANCE

ANÁLISE DE COMPLEXIDADE (NOTAÇÃO BIG O)

PATRICIA MAGNA

6

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

6 ANÁLISE DE COMPLEXIDADE (NOTAÇÃO BIG O)

6.1 Motivos para Analisar a Eficiência de Algoritmos

Existe uma série de problemas reais que mesmo sendo processados computacionalmente são “críticos”, isto, ou o tempo de execução é bastante elevado ou então requerem uma quantidade enorme de memória. Normalmente, essa classe de problemas é amplamente estudada e, quando possível, são apresentadas formas diferentes de solução.

Assim, surgem alguns algoritmos diferentes cujo objetivo é resolver o mesmo problema. Quando nos deparamos com essa situação surge a seguinte pergunta: qual, entre esses vários algoritmos é o ideal para ser usado para resolver o meu específico problema?

Como exemplo dessa situação, vamos citar os diferentes métodos que podem ser usados para efetuar a busca de informações contidas em um conjunto grande de registros. Como já visto, podemos usar os métodos de busca sequencial simples, busca binária ou busca indexada. Como as operações de acesso a um determinado registro dentro de um grande arquivo são comumente utilizadas em sistemas de informação, é claro que essa operação deve ser feita da forma mais rápida possível. Como escolher o melhor método?

Para avaliarmos se um algoritmo é melhor do que outro, precisamos fazer uma comparação entre eles. Como realizar essa comparação? Quais parâmetros devem ser avaliados? Esse é o objetivo do que estudaremos agora.

6.2 Como Analisar a Eficiência de Algoritmos

Há dois aspectos de eficiência a serem considerados na medida de eficiência de programas: tempo requerido para o processamento e espaço de armazenamento em memória. Geralmente, a otimização de um aspecto é feita em detrimento do outro.

Podemos pensar em mensurar o aspecto tempo em unidades reais de tempo, ou seja, fazendo a execução do programa e medindo o tempo requerido. Porém, essa avaliação pode ter grande variação em função de:

- Qual compilador foi utilizado para gerar o programa executável, pois cada compilador tem escolhas de como implementar cada tipo de construção de linguagem de alto nível e isso interfere de forma direta no tempo de processamento;
- Qual hardware (máquina) está sendo executado, há uma grande diferença de desempenho em função de configurações como quantidade de memória RAM, geração do processador, *clock*, etc.
- O sistema operacional (SO) utilizado, pois ele também tem influência no tempo de execução, uma vez que é ele que gerencia o uso do processador pelos programas e também a ocupação e a troca de páginas (com dados e instruções do nosso programa) da memória RAM, sendo que cada SO utiliza algoritmos diferentes para realizar tais tarefas.

Assim, a eficiência em relação ao tempo não deve ser feita baseada no tempo de execução em uma determinada máquina. A forma mais usualmente utilizada é fazer a avaliação pelo número de operações significativas (também conhecidas por operações críticas) efetuadas em toda a execução do algoritmo.

Para melhor compreender o que se entende por operações significativas, vamos usar como exemplo nos algoritmos de busca de um elemento com determinado valor em um vetor, as operações críticas seriam as operações de comparações entre a chave (valor procurado) e cada elemento do vetor. Já em algoritmos de ordenação dos elementos de um vetor, as operações significativas seriam além das comparações as movimentações de elementos.

Normalmente, o que é interessante conhecer, supondo algoritmos de busca e ordenação de vetores, é a mudança na quantidade de tempo necessária para realizar o objetivo do algoritmo em função do tamanho do arquivo ou vetor. O tamanho do arquivo (ou vetor) sempre é referenciado em relação à quantidade **n** de registros que o compõem.

Devem ser analisados três tipos de casos: o pior, o melhor e o caso médio. O resultado dessa análise é, com frequência, uma equação dando o tempo médio ou o número de operações significativas, necessários para efetuar a ordenação ou busca em função do tamanho **n** do arquivo.

Para melhor entendimento considere o trecho de programa a seguir em que é feita a busca de uma chave em uma base de dados (em que a chave é chave primária, ou seja, não há valores de chaves repetidos).

```
int i = 0;
int n = basededados.length;
while (i < n && basededados[i] != chave)
          i++;
```

Código Fonte 6.1 – Código com trecho de programa para busca sequencial escrito em JAVA
Fonte: Elaborado pelo autor (2019)

Esse trecho de programa utiliza o método de busca sequencial da chave no vetor. Para esse algoritmo, a operação significativa é a comparação entre a chave e o elemento do vetor. Observe que nessa abordagem não são levadas em consideração as operações ditas secundárias, como por exemplo, a manipulação do índice do vetor. Com relação à quantidade de vezes que a operação de comparação será realizada por esse programa há três possíveis situações:

Melhor caso: 1, quando a chave for encontrada como primeiro elemento.

Pior caso: n, quando a chave for encontrada no último elemento.

Caso médio: (n+1)/2, usando distribuição de probabilidades em que todos os elementos têm probabilidade igual de ter o valor da chave procurada, cálculo apresentado por Ziviani (1999).

Um método para determinar as exigências de tempo de uma ordenação (classificação), em que as operações significativas são as comparações e as trocas de posição entre os elementos, é fazer uma análise matemática dos vários casos. Ocasionalmente esta análise torna-se complexa.

Em um exemplo apresentado em Tenembaum (1995, p.414) é feita a suposição de que tal análise matemática tenha sido aplicada sobre um determinado programa de ordenação e dela resulte a conclusão que o programa precisa de $0{,}01n^2 + 10n$ unidades de tempo (operações críticas) para executar. É interessante observar como cada parcela dessa soma influencia o resultado, o que é apresentado no quadro a seguir.

n	A=0,01 n^2	B= 10 n	A+B	(A+B) / n^2
10	1	100	101	1,01
50	25	500	525	0,21
100	100	1.000	1100	0,11
500	2.500	5.000	7500	0,03
1.000	10.000	10.000	20.000	0,02
5.000	250.000	50.000	300.000	0,01
10.000	1.000.000	100.000	1.100.000	0,01
50.000	25.000.000	500.000	25.500.000	0,01
100.000	100.000.000	1.000.000	101.000.000	0,01

Quadro 6.1 – Cálculo das parcelas e da função que descreve o número de operações críticas do exemplo
Fonte: Tenembaum (1995)

Para n pequeno, a parcela B domina a parcela A, pois a pequena diferença entre os valores de n não compensa a diferença entre fatores 0,01 e 10 que multiplicam n^2 e n, respectivamente. À medida que n aumenta, a diferença de n^2 e n aumenta com tanta rapidez que mais que compensa a diferença entre os fatores 0,01 e 10. Dessa forma, quando n é grande o termo n^2 domina a função tornando insignificante a contribuição da parcela B (10n) no resultado. Portanto, para n grande o tempo de ordenação do arquivo torna-se proporcional a n^2.

Evidentemente, para valores pequenos de n, a ordenação pode apresentar um comportamento completamente diferente, uma situação que pode ser levada em consideração ao analisar sua eficiência.

Contudo, devemos lembrar que a análise de eficiência é realmente necessária quando a execução do programa (algoritmo) é levada ao seu limite crítico. No nosso exemplo de ordenação de vetores, esse limite seria quando n torna-se muito grande.

No exemplo anterior, diz-se que de $0,01n^2 + 10n$ é “da ordem” da função n^2, pois com o aumento de n a ordem se torna proporcional a n^2. Utiliza-se a seguinte notação para representar essa proporcionalidade: $O(n^2)$. Diz-se da ordem de n^2.

Essa notação é muito utilizada para analisar com que proporção varia o tempo (ou número de operações críticas) efetuadas, e dessa forma, ter de alguma forma ideia da eficiência do algoritmo utilizado. Então, vamos estudar como utilizar essa notação.

6.3 Notação Big O - O ( )

Antes de iniciar a definição da notação big O, é necessário entendermos um termo utilizado por ela “comportamento assintótico”.

Definição: Uma função f(n) **domina assintoticamente** outra função g(n), se existirem inteiros positivos **a** e **b** tais que:

$|g(n)| \leq a * |f(n)|$, para todo $n \geq b$

Para entender melhor vamos a um exemplo.

Supondo que $g(n)= n^2 + 100n$ e $f(n)= n^2$, tem-se que $n^2 + 100n \leq 2\ n^2$, para a=2.

Analisando o gráfico apresentado na figura a seguir é possível observar que para b =100 tem-se satisfeita a condição que nos permite afirmar que f(n) domina g(n), ou seja, f(n) passa ser sempre maior do que g(n).

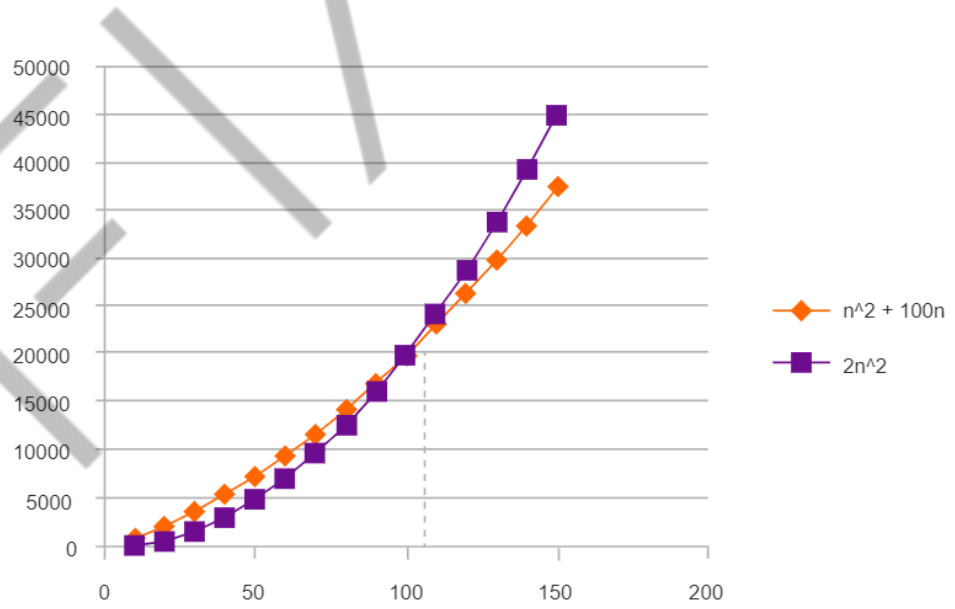

Figura 6.1 – Comparativo de entre as funções f(n) e g(n) do exemplo
Fonte: Elaborado pelo autor (2019)

Agora podemos passar para a definição da notação O(), conhecida com ***big O***.

Diz-se que f(n) é O(g(n)), lido como f(n) é da ordem de g(n), se f(n) **domina assintoticamente** a função g(n).

Assim voltando ao exemplo que $f(n)= n^2 + 100n$ e $g(n)= n^2$, para a=2 e b=100 tem-se satisfeita a condição para afirmar que f(n) é $O(2n^2)$, ou seja: f(n) é $O(n^2)$.

Se f(n) for O(g(n)) ocasionalmente ($n \geq b$) tornar-se-á permanentemente menor ou igual a algum múltiplo de g(n). Isso significa que f(n) está limitada por g(n) de cima para baixo, ou que f(n) é uma função menor que g(n). Outro modo, mais formal, é dizer que f(n) é assintoticamente limitada por g(n).

Pela propriedade transitiva:

- se f(n) é O(g(n)) e g(n) é O(h(n)) então, f(n) é O(h(n)).

Existem funções matemáticas que são referência na análise de eficiência de algoritmos, com n, n^2, entre outras.

Uma importante função de estudo de eficiência de algoritmo é a função logarítmica.

Lembrando que: se $x=\log_m n$ então $m^x = n$.

Exemplo: $x=\log_{10} 100 \Rightarrow 10^x = 100 \Rightarrow x=2$

Como cada função logarítmica é da ordem de qualquer outra função logarítmica, normalmente, omite-se a base. Simplesmente diz-se que é da ordem de uma função logarítmica, ou O(log(n)).

Segundo Ziviani (1999), algumas classes de problemas possuem as funções de complexidade (funções que descrevem o comportamento dos algoritmos), sendo elas: O(1), O(log n), O(n), O(n log n), $O(n^2)$, $O(n^3)$ e $O(2^n)$.

Para se entender a hierarquia das funções em termos de serem maiores que outras, vamos analisar o gráfico apresentado na Figura Comparativo de entre as funções log n e C. É possível verificar que a função constante 1 é assintoticamente limitada pelas funções log n, já que a partir de n=10 (foi usada função $\log_{10}$ n para gerar o gráfico) a função constante é sempre limitada de cima para baixo pela função log n.

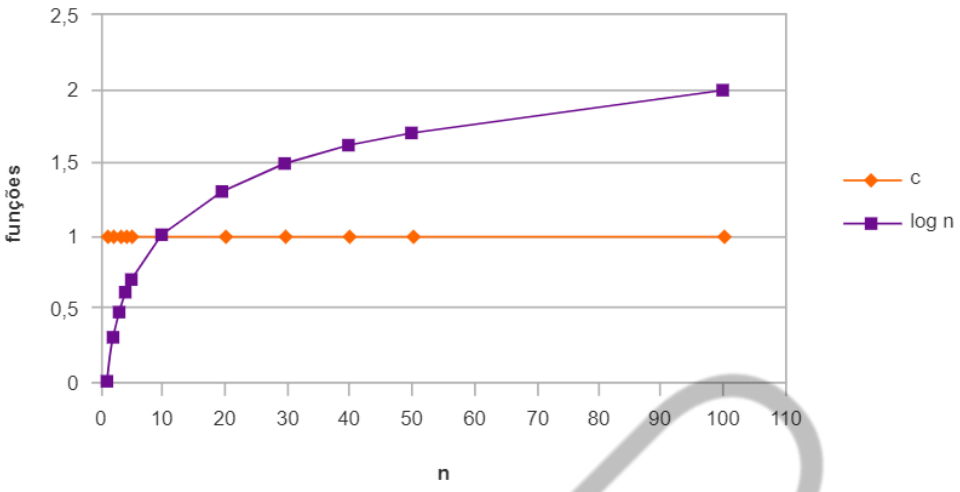

Figura 6.2 – Comparativo de entre as funções log n e C
Fonte: Elaborado pelo autor (2019)

Já pelo gráfico da Figura Comparativo de entre as funções log n, n e n log n é possível verificar que log n é assintoticamente limitada pelas funções n log n e n. E também que a função n é limitada pela função n log n.

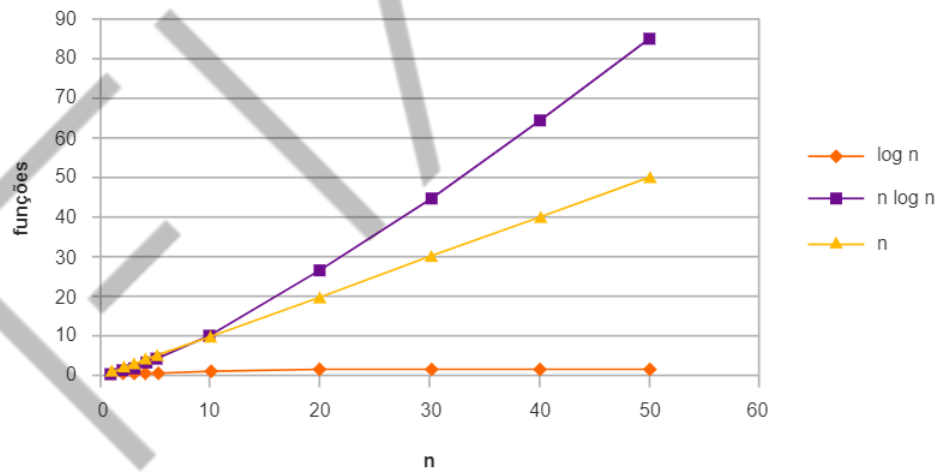

Figura 6.3 – Comparativo de entre as funções log n, n e n log n
Fonte: Elaborado pelo autor (2019)

Também analisando mais outras funções no gráfico da Figura Comparativo de entre as funções n log n, n^2 e n^2 log n , concluímos que: a função n log n é assintoticamente limitada pelas funções n^2 log n e n^2, já a função n^2 é assintoticamente limitada pela n^2 log n.

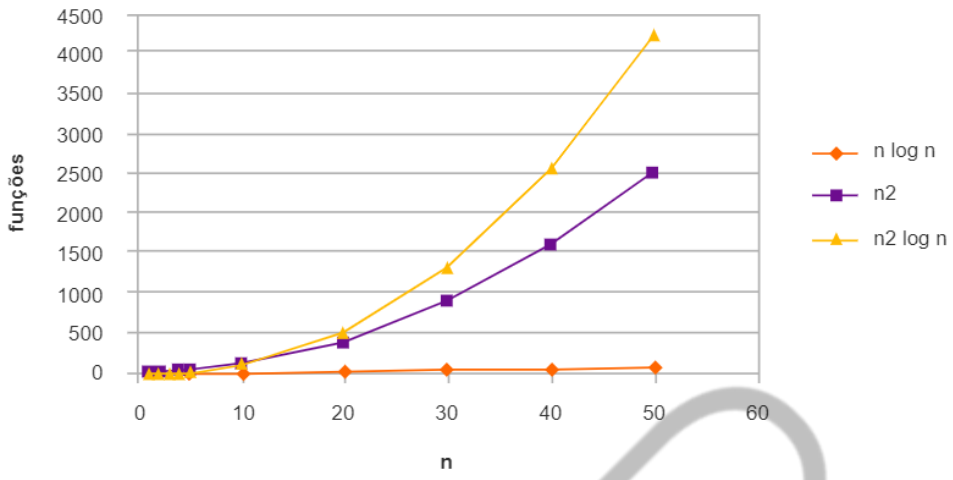

Figura 6.4 – Comparativo de entre as funções n log n, n^2 e n^2 log n
Fonte: Elaborado pelo autor (2019)

Também podem ser feitas mais comparações com o gráfico da Figura Comparativo de entre as funções n^2 log n e n^3. Nele é possível concluir que n^2 log n é assintoticamente limitada por n^k, para qualquer k maior que 2 (no gráfico foi usada a função n^3).

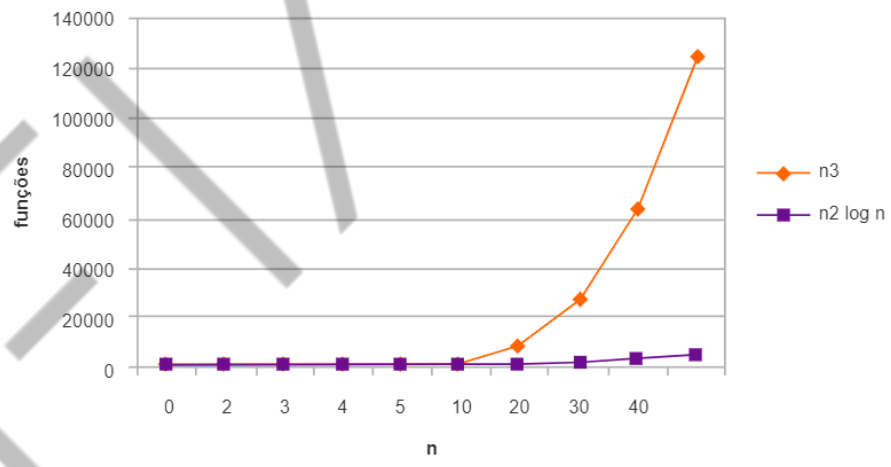

Figura 6.5 – Comparativo de entre as funções n^2 log n e n^3
Fonte: Elaborado pelo autor (2019)

Por meio dos gráficos apresentados nas figuras pode-se concluir que é estabelecida uma hierarquia em que cada função é mais baixa (ou assintoticamente limitada) que a função seguinte:

C, log n, n, n log n, n^k , n^k (log n) , n^{k+1} , d^n

As funções que são $O(n^k)$ são de ordem polinomial. Já as funções que são $O(d^n)$ para d>1, mas não $O(n^k)$ para qualquer k, são funções de ordem exponencial. As funções exponenciais são sempre muito maiores que as polinomiais.

Segundo Tenembaum (1995), os seguintes fatos estabelecem uma hierarquia de ordem de funções de complexidade de algoritmos:

C (constante) é O(1), pra todo C

C é O(log(n)) mas log n não é O(1)

C*$\log_k$ n é O(log n) para quaisquer C e k

C*$\log_k$ n é O(n), mas n não é O(log(n))

C* n^k é $O(n^k)$ para quaisquer C e k

C* n^k é $O(n^{k+j})$, mas n^{k+j} não é $O(n^k)$

C * n * $\log_k$ n é O(n log n) para quaisquer C e k

C * n^j * $\log_k$ n é O(n^j log n) para quaisquer C, j e k

C * n^j * $\log_k$ n é $O(n^{j+1})$, mas n^{j+1} não é $O(n^j \log n)$

C * n^j * $(\log_k n)^m$ é $O(n^j (\log n)^m)$, para quaisquer C, j, m e k

C * n^j * $(\log_k n)^m$ é $O(n^{j+1})$, mas n^{j+1} não é $O(n^j (\log n)^m)$

C * n^j * $(\log_k n)^m$ é $O(n^j (\log n)^{m+1})$, mas $n^j (\log n)^{m+1}$ não é $O(n^j (\log n)^m)$

C * n^j é $O(d^n)$, mas d^n não é $O(n^j)$ para quaisquer C,j, e d>1

6.3.1 CLASSES DE PROBLEMAS E FUNÇÕES DE COMPLEXIDADE

Agora que já conhecemos a hierarquia das funções que descrevem a complexidade de algoritmos, podemos analisar o significado em termos de comportamento dos algoritmos e suas eficiências.

- f(n) = O(1). O tempo de execução independe do tamanho n, são conhecidos como complexidade constante.
- f(n) = O(log n). Algoritmos da ordem de log n normalmente são aqueles em que o problema é resolvido transformando-os em problemas menores. Para citar um exemplo em que a complexidade é da ordem de log n, vamos analisar o código com o trecho de programa que realiza a busca de uma chave em um vetor usando o método de busca binária. Cada vez que a

chave não é encontrada, o vetor analisado é dividido ao meio. Sendo assim, o número máximo de comparações realizadas deve ser $\log_2 n$.

```
int i_baixo = 0;
int i_medio = 0;
int i_alto =  n-1;
int achou = 0;
int posicao = -1;
while( achou != 1 && i_baixo <= i_alto) {
   i_medio = (i_baixo + i_alto)/2;
   if (chaveproc== basededados[i_medio]) {
         posicao = i_medio;
         achou = 1;
         }
   else {
         if (chaveproc < basededados[i_medio])
               i_alto = i_medio - 1;
         else
               i_baixo = i_medio + 1;
         }
   }
```

Código Fonte 6.2 – Código com trecho de programa para busca binária escrito em JAVA
Fonte: Elaborado pelo autor (2019)

- f(n) = O(n). São chamados de algoritmos de ordem linear. Esses problemas são aqueles em que o número de operações críticas executadas é dado pelo tamanho da entrada n. Se n dobrar o tempo de execução do problema também dobra.
- f(n) = O(n log n). Assim como os algoritmos de O(log n) os algoritmos dessa classe têm como solução dividir o problema em problema menor até a solução. Esse tipo de complexidade é encontrado em algoritmos de ordenação.
- f(n) = $O(n^2)$. São chamados de algoritmos de ordem quadrática. Normalmente, esses problemas usam duas construções de repetição (loop) aninhadas.

  f(n) = $O(2^n)$. Os algoritmos dessa classe são chamados de complexidade exponencial. Pelo fato de funções exponenciais terem taxa de crescimento muito elevadas, essa classe de problema é considerada como “intratável”. Para exemplificar, supondo n = 10 o tempo de execução seria aproximadamente 1000, para n=100 o tempo seria 1.267.650.600.228.229.401.496.703.205.376.

Agora que você já sabe a importância de uma boa análise algorítmica, já dá para construir os melhores projetos de algoritmos, daqui para frente. Vamos em frente com alguns exercícios que vão ajudá-lo a compreender o assunto.

6.4 Exercícios Propostos

Analise os códigos em relação à eficiência, em seguida e responda:

a. Qual é a operação significativa ou crítica?

b. Qual o melhor caso?

c. Qual o pior caso?

d. Qual função da notação big O representa a função de complexidade?

1. Soma dos elementos de um vetor:

```
int i;
int soma = 0;
for(i=0; i<n; i++) {
   soma = soma + vetor[i];
  }
```

RESULTADO

Operação significativa: **`soma + vetor[i]`**

Melhor caso e pior caso: são efetuadas **n** somas.

Função de complexidade: O(n)

2. Procure um valor negativo entre os elementos de um vetor, parando assim que encontrar o primeiro:

```
int i=0;
boolean achou = false;
while(achou != true && i<n) {
  if (vetor[i]<0)
```

```
        achou = true;
        i++;
  }
```

RESULTADO

Operação significativa: **vetor[i]<0**

Melhor caso: 1, encontra o valor negativo no primeiro elemento.

Pior caso: são efetuadas **n** comparações (atinge fim do vetor).

Função de complexidade: O(n)

3. Altera o valor de cada elemento de um valor somando os elementos do vetor que estão em posições posteriores (posições com índices maiores):

```
int i, j;
double novo_valor;
for(i=0; i<n ; i++) {
    novo_valor = vetor[i];
    for (j=i+1; j<n; j++)
         novo_valor = vetor[j] + novo_valor;
    vetor[i] = novo_valor;
  }
```

RESULTADO

Operação significativa: **vetor[j] + novo_valor**

Melhor caso e pior caso:

1° elemento: (n-1) operações

2° elemento: (n-2) operações

3° elemento: (n-3) operações

...

(n-1)° elemento: 1 operação

Função de complexidade: $O(n^2)$

REFERÊNCIAS

DEITEL, P. J.; DEITEL, H. M., **Java**: como programar. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

TENEMBAUM, A. M. et al. **Estruturas de Dados usando C.** São Paulo: Makron Books Ltda, 1995.

ZIVIANI, N. **Projetos de Algoritmos**: com implmentações em Pascal e C. São Paulo: Pioneira, 1999.