Estrutura de Dados
Desempenho de Algoritmos

Profa. Ana Cristina dos Santos
email: ana.csantos@sp.senac.br

Conteúdo

- Comparação de algoritmos
- Complexidade de Tempo
- Busca linear e seu desempenho
- Notação O
- Busca binária
- Ordem de Magnitude
- Exercícios

Comparação de Algoritmos

- Um característica muito importante de qualquer **algoritmo** é o seu **tempo de execução**, com isso podemos comparar o tempo de execução de dois **algoritmos diferentes** que resolvem o **mesmo problema**.
- Isso nos leva à algumas questões:
  - Como comparar os algoritmos independente do computador usado para execução?
  - A linguagem de programação influencia ?
  - Qual a importância do sistema operacional na comparação ?

Complexidade de Tempo

- A complexidade de tempo de um algoritmo quantifica o tempo que um algoritmo gasta para processar e calcular a resposta de um problema em função do tamanho da entrada.
- Por exemplo:
- “Se meu algoritmo gasta 1 minuto para uma entrada de um vetor 1000 posições, quantos minutos ele gastará numa entrada de 2000 posições?“

Complexidade de Tempo

A **complexidade de tempo** de um algoritmo determina o **número de passos** (aproximados) que um algoritmo executa dado o tamanho de sua entrada, ou seja, uma expressão matemática que traduz o comportamento de tempo de um algoritmo.

Complexidade de Tempo

Se **n** é o número de elementos da entrada (tamanho da entrada), o **tempo de execução** pode ser representado por uma expressão matemática que denota o número de passos que o algoritmo executa em função de **n**.

Complexidade de Tempo

Considere que um passo são todas as instruções que um algoritmo executa a cada iteração do algoritmo.
Podemos dizer então que **número total de passos executados** pelo algoritmo é o **tempo de execução do algoritmo.**

Problema: Buscar um elemento no vetor

Considere o problema de determinar se um elemento está presente em uma lista de elementos, ou seja:

Dado um inteiro **x** e uma lista ou vetor v[0..n-1] com **n** elementos inteiros, o problema da busca consiste em encontrar **x** em v[], ou seja, encontrar um índice k tal que v[k] == x.

Problema: Buscar um elemento no vetor

Algoritmo da Busca Linear:
Realiza a busca comparando o valor do elemento a ser encontrado com os elementos do vetor, um a um, da esquerda para a direita, sequencialmente (=linearmente).

BuscaLinear(v,x)

Análise da Busca linear

Qual seria o número de passos da busca linear ? Os valores dentro do vetor e o valor do x influenciam no cálculo dos passos do algoritmo?

Qual seria a configuração de entrada para função BuscaLinear(v,x) que teríamos o **menor** número de passos ?

Qual seria a configuração de entrada para função BuscaLinear(v,x) que teríamos o **maior** número de passos ?

Análise da Busca Linear - Pior caso

Um algoritmo pode ter várias entradas possíveis, para calcular a complexidade de um algoritmo é considerada a configuração da entrada **mais desfavorável**.

A complexidade de tempo de **pior caso** corresponde ao número de passos que o algoritmo efetua no seu pior caso de execução, ou seja, para entrada mais desfavorável.

Assim o termo **complexidade** será empregado como o significado de complexidade de **pior caso**, pois ela fornece um **limite superior para o número de passos** que o algoritmo efetua.

Complexidade da Busca Linear

A Busca Linear tem o seu pior desempenho (**pior caso**), quando o valor procurado é igual ao do último elemento do vetor, ou quando o elemento não se encontra no vetor, pois o algoritmo realizará a comparação do valor procurado com todos os elementos do vetor.

Podemos dizer então que busca linear no **pior caso** executa aproximadamente **n passos** para verificar se um determinado elemento está presente no vetor.

Notação O()

A **notação O (lê-se Ó)** será utilizada para expressar a complexidade de um algoritmo no **pior caso**, sem entrar em detalhes sobre exatamente quantas operações constantes (**comparações, atribuições**) são executadas.

O que queremos é uma abordagem **assintótica** do algoritmo ignorando valores pequenos e constantes, concentrando se nos valores enormes de **n**, ou seja, quando $n \to \infty$.

Notação O()

Pela notação O podemos dizer que a Busca Linear tem complexidade O(n), ou seja é da ordem O(n).

Algoritmo da Busca Binária

A **Busca Binária** também resolve o problema de buscar de um elemento no vetor, a função que faz a Busca Binária recebe um número x e um vetor em **ordem crescente** v[0..n-1] com n elementos inteiros.

A Busca Binária sempre calcula o meio do vetor e compara o valor de x com o meio do vetor, de acordo com o resultado da comparação o algoritmo escolhe se vai para parte à esquerda do meio ou a parte à direita do meio no vetor. Ou seja, a cada iteração o vetor é reduzido o seu tamanho pela metade. O algoritmo repete até que só reste um elemento no vetor.

Complexidade da Busca Binária

Para encontrar a função de complexidade da Busca Binária é necessário identificar o **pior caso** do algoritmo.

O pior caso da busca binária é quando o elemento procurado não pertence a lista.

Agora, vamos realizar vários testes considerando o pior caso da Busca Binária, variando o tamanho do vetor (n) de 1 até 128.

Complexidade da Busca Binária

n	**Passos**
1	1
2	2
4	3
8	4
16	5
32	6
64	7
128	8

Note que os valores calculados para os passos são modificados somente para n=1, n=2, n=4, n=8, n=16,..., n=128.

A partir dessa análise poderíamos deduzir que se tivermos n =256 a Busca Binária executaria 9 passos, se n=1024 teríamos 11 passos.

Ou seja, para uma entrada de tamanho n a Busca Binária executa log2n+1 passos.

Podemos concluir então que a busca binária tem complexidade **O(logn)**, pois ignoramos constantes e valores pequenos, que é menor que O(n) , concordam ?

Ou seja, a busca binária é mais rápida que busca linear.

Ordem de Magnitude

- Classificação
  - $O(1)$ = constante
  - $O(n)$ = linear
  - $O(n^2)$ = quadrática
  - $O(n^3)$ = cúbica
  - $O(k^n)$ = exponencial
  - $O(\log n)$ = logarítmica
  - $O(n \log n)$ = n log n

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

Exercícios

1) Ao final da execução do trecho de algoritmo abaixo, qual o valor de S em função de n ? Dê uma fórmula fechada

(a)	(b)

S ← 0
para i de 1 até n faça
    para j de 1 até n faça
        S ← S + 1
	S ← 0
	i ← 1
	j ← 1
	enquanto i <= n faça
	    S ← S + 1
	    j ← j + 1
	    se j > n então
	        i ← i+1
	        j ← 1
	    fim-se
	fim-enquanto

Exercícios

(c)

```
S ← 0
para i de 1 até n faça
    para j de 1 até i faça
        S ← S + 1
```

(d)

```
S ← 0
para i de 1 até n faça
    para j de 1 até n faça
        para k de 1 até j faça
            S ← S + 1
```

(e)

```
S ← 0
enquanto n > 0 faça
        S ← S + 1
        n ← n / 2
```

Exercícios

2) Escreva uma função para **inverter** a ordem dos elementos de um vetor V[]. Você não pode usar outro vetor como área auxiliar. A função deve ter deve ter complexidade O(n), ou seja, o tamanho do vetor V[].

3) Escreva uma função que recebe um vetor A[] **e troca de posição seu maior e seu menor elementos**. A função deve ter deve ter complexidade O(n), ou seja, o tamanho do vetor V[].

4) Escreva o algoritmo que recebe um vetor A de tamanho n contendo inteiros e encontra o par de elementos distintos a e b do vetor que fazem com que a **diferença a-b seja a maior possível**. A função deve ter deve ter complexidade O(n), ou seja, o tamanho do vetor V[].

Exercícios

5) Dado um vetor de n números inteiros, faça uma função para determinar o **comprimento de um segmento crescente** de comprimento máximo. Exemplos:
Na sequência
{ 5, 10, 3, 2, 4, 7, 9, 8, 5} o comprimento do segmento crescente máximo é 4 {2, 4, 7, 9}.

Na sequência
{10, 8, 7, 5, 2} o comprimento de um segmento crescente máximo é 1.
A função deve ter deve ter complexidade O(n), ou seja, o tamanho do vetor.

Exercícios

6) Escreva uma função que receba dois vetores (A[] e B[]) já ordenados em ordem crescente e ambos possuem o mesmo tamanho. A sua função imprime a **intersecção** entre os dois vetores, ou seja, os elementos em comum entre os vetores A[] e B[]. Considere que os vetores **não contêm valores duplicados**. A função deve ter deve ter complexidade O(n), ou seja, o tamanho do vetor A[] e do vetor B[].

7) Repita o exercício anterior, agora deve ser impresso os elementos que estão em A[] mas não estão em B[]. A função deve ter deve ter complexidade O(n), ou seja, o tamanho dos vetores.

Exercícios

8) Escreva uma função que receba dois vetores (A[] e B[]), com n e m elementos, respectivamente. Os vetores estão ordenados em ordem crescente, a função aloca um vetor C[], exatamente com soma dos tamanhos de A e B, e **intercala** os elementos de A[] e B[] em C[], de forma que o vetor C[] fique em ordem crescente. A função deve ter deve ter complexidade O(n+m), ou seja, a soma dos tamanho dos vetores.

9) Escreva uma função que recebe um vetor como parâmetro, a sua função seleciona o primeiro elemento de um vetor e **rearranja** o vetor de forma que todos elementos menores ou iguais ao primeiro elemento fiquem a sua esquerda e os maiores a sua direita.
No vetor {5, 6, 2, 7, 9, 1, 8, 3, 7} após ser rearranjado teríamos {1, 3, 2, 5, 9, 7, 8, 6, 7}. A função deve rearranjar o vetor com a complexidade O(n).