Buscas em vetores

1. Introdução

Uma forma de armazenar dados em memória é em vetores. Se temos um conjunto de dados, por exemplo, nome, telefone e e-mail de alunos de uma classe, podemos armazenar as informações em três vetores diferentes, com o mesmo tamanho. O primeiro para armazenar os nomes, o segundo os telefones e o terceiro para os e-mail, sendo que as informações de determinado aluno estão no mesmo índice dos três vetores.

Na hora de utilizar as informações armazenadas, se queremos o telefone do aluno “João da Silva”, precisamos saber se esse nome consta do vetor de nomes. Se estiver, descobrimos a posição dele e na mesma posição do vetor de telefones estará o telefone dele.

Isso implica em fazermos um algoritmo de busca. A forma mais natural para essa busca é a busca sequencial, pela qual compara-se todos os elementos do vetor com o elemento buscado, até encontrá-lo.

A busca sequencial funciona qualquer que seja a ordem dos elementos no vetor. Caso o vetor esteja ordenado, é possível realizar uma busca mais otimizada, chamada de busca binária.

A seguir, veremos esses dois algoritmos.

2. Busca sequencial

Como o nome já diz, esse algoritmo busca sequencialmente, ou seja, compara o elemento buscado com todos os elementos do vetor, um a um, do início ao fim. O código normalmente retorna a posição do vetor em que o elemento é encontrado, ou um código especial caso ele não esteja no vetor. Esse código especial não deve ser possível de confundir com uma posição válida do vetor. Uma possibilidade é retornar -1 caso o elemento não exista no vetor.

Segue um exemplo de uma função de busca sequencial, para valores inteiros. Observe que o tamanho do vetor também é passado por parâmetro, além do vetor e do valor a ser buscado.

```
// Busca sequencial de um valor em um vetor
// vet é o vetor de inteiros
// tam é o tamanho do vetor
// valor é o número a ser buscado
// Retorna o índice do vetor onde está o valor, ou -1 se não encontrado
int buscaSeq(int *vet, int tam, int valor)
{
    int i;
    int pos = -1;

    for (i=0; i<tam; i++)
    {
        if (valor == vet[i])
           pos = i;
    }
    return (pos);
}
```

Note que essa versão do algoritmo de busca sequencial continua procurando até o final do vetor, mesmo depois de já ter encontrado o elemento. Ele pode ser otimizado, parando a execução quando encontrar o elemento buscado. Segue um exemplo dessa versão:

```
int buscaSeq(int *vet, int tam, int valor)
{
    int i=0;
    int pos = -1;
    while ((i<tam) && (pos==-1))
    {
        if (valor == vet[i])
           pos = i;
        i++;
    }
    return (pos);
}
```

3. Busca binária

A busca sequencial não é otimizada. Quando os dados estão armazenados aleatoriamente, a busca sequencial é a única forma efetiva. Entretanto, caso os valores estejam ordenados no vetor, pode-se usar uma pesquisa mais eficiente. Por exemplo, ao buscar uma palavra em um dicionário, as pessoas não fazem a busca sequencial. Ninguém inicia uma busca no dicionário pelo primeiro termo, buscando sequencialmente até chegar ao termo desejado. As pessoas abrem o dicionário em uma página próxima ao que estimam encontrar a palavra, e se a palavra está alfabeticamente mais para frente do que os termos encontrados na página aberta, a busca se repete para a frente, e em caso contrário, a busca continua na parte anterior do dicionário. E esse tipo de busca é muito mais rápido do que a busca sequencial.

Um raciocínio similar é utilizado pela chamada busca binária.

Para a busca binária funcionar, os dados precisam estar ordenados.

A primeira comparação é com o elemento do meio do vetor.

Caso o valor buscado seja maior do que o elemento, é realizada uma nova busca binária, considerando como início a posição seguinte à do elemento comparado, até o final do vetor.

Caso o valor buscado seja menor do que o elemento, é realizada uma nova busca binária, considerando como final a posição anterior à do elemento comparado, mantendo o mesmo início.

Caso o valor buscado seja igual ao elemento, a busca se encerra, retornando a posição do elemento.

Com a definição acima, o algoritmo pode ter a seguinte implementação recursiva:

```
// Busca sequencial de um valor inteiro em um vetor vet
// inicio é o índice de vet do inicio da busca
// fim é o índice de vet do final da busca
// Retorna o índice de vet onde valor foi encontrado, ou -1 se não encontrado
int buscaBin(int *vet, int inicio, int fim, int valor)
{
   int meio;
   int pos = -1;

   if (inicio > fim)   // Se o início passou do fim, não achou
      pos = -1;
   else {
      meio = (inicio + fim)/2;
      if (valor == vet[meio])
         pos = meio;   // Encontrou elemento, retorna posição
      else if (valor > vet[meio])
         pos = buscaBin(vet, meio+1, fim, valor);
      else
         pos = buscaBin(vet, inicio, meio-1, valor);
   }
   return (pos);
}
```

Observe que essa versão da busca binária lembra o algoritmo da busca em árvore binária.

O algoritmo de busca binária não necessita ser feito usando recursividade. A versão iterativa é relativamente simples, e mais leve em termos computacionais.
A seguir, um exemplo da implmentação da mesma busca, sem recursividade.

```
// Busca sequencial de um valor em um vetor
// vet é o vetor de inteiros
// inicio é o índice de vet do inicio da busca
// fim é o índice de vet do final da busca
// valor é o número a ser buscado
//  Retorna o índice do vetor onde o valor foi encontrado, ou -1 se não encontrado
int buscaBin(int *vet, int inicio, int fim, int valor)
{
   int meio;
   int pos = -1;
   int i, f;

   i=inicio;
   f=fim;

   while ((i <= f) && (pos == -1))
   {
      meio = (i + f)/2;
      if (valor == vet[meio])
         pos = meio;
      else if (valor > vet[meio])
         i = meio+1;
      else
         f = meio-1;
   }
   return (pos);
}
```

4. Análise de desempenho.

4.1. Busca Sequencial.

O algoritmo de busca sequencial é mais simples de execução. Entretanto, precisa comparar todos os elementos do vetor, para o caso de o elemento não ser encontrado, ou até encontrar o elemento.

Assim, o pior caso, para um vetor com `n` elementos, é `n` comparações. O caso médio, para `n` elementos, é `n/2`, para elementos encontrados, e `n` para elementos não encontrados.

Por isso, dizemos que o esforço computacional para buscar um elemento em um vetor com `n` posições é da ordem de `n`.

4.2. Busca Binária.

O algoritmo de busca binária é mais inteligente, precisando de menos comparações para encontrar um elemento no vetor.

Para um vetor com `1` elemento, é uma comparação.

Para um vetor com `2` elementos, no pior caso, `2` comparações

Para um vetor com `3` elementos, no pior caso, `2` comparações.

Para um vetor com `4` elementos, no pior caso, `3` comparações.

Para um vetor com `5` elementos, no pior caso, `3` comparações.

Para um vetor com 6 elementos, no pior caso, 3 comparações.

Para um vetor com 7 elementos, no pior caso, 3 comparações.

Para um vetor com 8 elementos, no pior caso, 4 comparações.

Generalizando, para um vetor com até $2^{(c-1)}$, são necessárias no máximo c comparações.

Dessa forma, pode-se dizer que o esforço computacional para uma busca binária em um vetor com n elementos é de ordem $\log_2 n$.

Comparando o esforço nos dois algoritmos, no pior caso, para alguns tamanhos de vetor.

N (tamanho do vetor)	Busca Sequencial	Busca Binária
1	1	1
3	3	2
7	7	3
15	15	4
31	31	5
63	63	6
127	127	7
255	255	8
511	511	9
1023	1023	10
1 milhão	1 milhão	20
1 bilhão	1 bilhão	30

Observe que para pequenos valores a diferença não é significativa. Mas para grandes volumes de dados, a diferença é muito grande.

Isso se deve ao fato de que o algoritmo de busca binária desconsidera metade dos dados a cada comparação.

Entretanto, a exigência de que os dados estejam ordenados não permite a aplicação a qualquer conjunto de dados.

Embora qualquer conjunto de dados possa ser ordenado, essa ordenação também implica em um custo computacional, que é maior do que o da busca sequencial. Assim, se o objetivo é fazer uma única busca no conjunto de dados que não está ordenado, vale a pena fazer a busca sequencial. Mas se o propósito é fazer muitas buscas sobre o mesmo conjunto de dados, pode valer a pena ordenar o conjunto de dados uma única vez, e apartir daí, realizar a consulta mais otimizada muitas vezes.