AULA 4

# BUSCA

Como temos visto ao longo de muitas das aulas anteriores, a busca de um elemento em um conjunto é uma operação básica em Computação. A maneira como esse conjunto é armazenado na memória do computador permite que algumas estratégias possam ser usadas para realizar a tarefa da busca. Na aula de hoje revemos os métodos de busca que vimos usando corriqueiramente até o momento como uma subtarefa realizada na solução de diversos problemas práticos. A busca será fixada, como antes, com os dados envolvidos como sendo números inteiros e o conjunto de números inteiros onde a busca se dará é armazenado em um vetor. Além de rever essa estratégia, chamada de busca seqüencial, vemos ainda uma estratégia nova e muito eficiente de busca em um vetor ordenado, chamada de busca binária. Esta aula está completamente baseada no livro de P. Feofiloff [2], capítulos 3 e 7.

## 4.1 Busca seqüencial

Vamos fixar o conjunto e o elemento onde a busca se dará como sendo constituídos de números inteiros, observando que o problema da busca não se modifica essencialmente se esse tipo de dados for alterado. Assim, dado um número inteiro $n \geqslant 0$, um vetor de números inteiros $v[0..n-1]$ e um número inteiro $x$, considere o problema de encontrar um índice $k$ tal que $v[k] = x$. O problema faz sentido com qualquer $n \geqslant 0$. Observe que se $n = 0$ então o vetor é vazio e portanto essa entrada do problema não tem solução.

Como em [2], adotamos a convenção de devolver $-1$ caso o elemento $x$ não pertença ao vetor $v$. A convenção é satisfatória já que $-1$ não pertence ao conjunto $\{0, \ldots, n-1\}$ de índices válidos do vetor $v$. Dessa forma, basta percorrer o vetor do fim para o começo, como mostra a função `busca_sequencial` a seguir.

```
/* Recebe um número inteiro n >= 0, um vetor v[0..n-1] com n nú-
   meros inteiros e um número inteiro x e devolve k no intervalo
   [0, n-1] tal que v[k] == x. Se tal k não existe, devolve -1. */
int busca_sequencial(int n, int v[MAX], int x)
{
   int k;

   for (k = n - 1; k >= 0 && v[k] != x; k--)
      ;

   return k;
}
```

Observe como a função é eficiente e elegante, funcionando corretamente mesmo quando o vetor está vazio, isto é, quando $n$ vale 0.

Um exemplo de uma chamada à função `busca_sequencial` é apresentado abaixo:

```
i = busca_sequencial(v, n, x);
if (i >= 0)
   printf("Encontrei %d!\n", v[i]);
```

A função `busca_sequencial` pode ser escrita em versão recursiva. A idéia do código é simples: se $n = 0$ então o vetor é vazio e portanto $x$ não está em $v[0..n-1]$; se $n > 0$ então $x$ está em $v[0..n-1]$ se e somente se $x = v[n-1]$ ou $x$ está no vetor $v[0..n-2]$. A versão recursiva é então mostrada abaixo:

```
/* Recebe um número inteiro n >= 0, um vetor de números in-
   teiros v[0..n-1] e um número x e devolve k tal que 0 <= k
   < n e v[k] == x. Se tal k não existe, devolve -1. */
int busca_sequencial_R(int n, int v[MAX], int x)
{
   if (n == 0)
      return -1;
   else
      if (x == v[n - 1])
         return n - 1;
      else
         return busca_sequencial_R(n - 1, v, x);
}
```

A correção da função `busca_sequencial` é mostrada de forma semelhante àquela do exercício 3.1. Seu tempo de execução é linear no tamanho da entrada, isto é, é proporcional a $n$, ou ainda, é $O(n)$, como mostrado na aula 2. A função `busca_sequencial_R` tem correção e análise de eficiência equivalentes.

Vale a pena consultar o capítulo 3, páginas 12 e 13, do livro de P. Feofiloff [2], para verificar uma série de "maus exemplos" para solução do problema da busca, dentre deselegantes, ineficientes e até incorretos.

## 4.2 Busca em um vetor ordenado

Como em [2], adotamos as seguintes definições. Dizemos que um vetor de números inteiros $v[0..n-1]$ é **crescente** se $v[0] \leqslant v[1] \leqslant \cdots \leqslant v[n-1]$ e **decrescente** se $v[0] \geqslant v[1] \geqslant \cdots \geqslant v[n-1]$. Dizemos ainda que o vetor é **ordenado** se é crescente ou decrescente. Nesta seção, vamos focar no problema da busca de um elemento $x$ em um vetor ordenado $v[0..n-1]$.

No caso de vetores ordenados, uma pergunta equivalente mas ligeiramente diferente é formulada: em qual posição do vetor $v$ o elemento $x$ deveria estar? Dessa forma, o problema da busca pode ser reformulado da seguinte maneira: dado um número inteiro $n \geqslant 0$, um vetor de números inteiros crescente $v[0..n-1]$ e um número inteiro $x$, encontrar um índice $k$ tal que

$$v[k-1] < x \leqslant v[k] \; . \tag{4.1}$$

Encontrar um índice $k$ como o da equação (4.1) significa então praticamente resolver o problema da busca, bastando comparar $x$ com $v[k]$.

Observe ainda que qualquer valor de $k$ no intervalo $[0, n]$ pode ser uma solução do problema da busca. Nos dois extremos do intervalo, a condição (4.1) deve ser interpretada adequadamente. Isto é, se $k = 0$, a condição se reduz apenas a $x \leqslant v[0]$, pois $v[-1]$ não faz sentido. Se $k = n$, a condição se reduz somente a $v[n-1] < x$, pois $v[n]$ não faz sentido. Tudo se passa como se o vetor $v$ tivesse um componente imaginário $v[-1]$ com valor $-\infty$ e um componente imaginário $v[n]$ com valor $+\infty$. Também, para simplificar um pouco o raciocínio, suporemos que $n \geqslant 1$.

Isso posto, observe então que uma busca seqüencial, recursiva ou não, como as funções `busca_sequencial` e `busca_sequencial_R` da seção 4.1, pode ser executada para resolver o problema. Vejamos então uma solução um pouco diferente na função `busca_ordenada`.

```
/* Recebe um número inteiro n > 0, um vetor de números in-
   teiros crescente v[0..n-1] e um número inteiro x e devol-
   ve um índice k em [0, n] tal que v[k-1] < x <= v[k] */
int busca_ordenada(int n, int v[MAX], int x)
{
   int k;

   for (k = 0; k < n && v[k] < x; k++)
      ;

   return k;
}
```

Uma chamada à função `buscaOrd` é mostrada a seguir:

```
i = busca_ordenada(n, v, x);
```

Se aplicamos a estratégia de busca seqüencial em um vetor ordenado, então certamente obtemos uma resposta correta, porém ineficiente do ponto de vista de seu consumo de tempo. Isso porque, no pior caso, a busca seqüencial realiza a comparação do elemento $x$ com cada um dos elementos do vetor $v$ de entrada. Ou seja, o problema da busca é resolvido em tempo proporcional ao número de elementos do vetor de entrada $v$, deixando de explorar sua propriedade especial de se encontrar ordenado. O tempo de execução de pior caso da função `busca_ordenada` permanece proporcional a $n$, o mesmo que das funções da seção 4.1.

Com uma busca binária, podemos fazer o mesmo trabalho de forma bem mais eficiente. A busca binária se baseia no método que usamos de modo automático para encontrar uma palavra no dicionário: abrimos o dicionário ao meio e comparamos a primeira palavra desta página com a palavra buscada. Se a primeira palavra é "menor" que a palavra buscada, jogamos fora a primeira metade do dicionário e repetimos a mesma estratégia considerando apenas a metade restante. Se, ao contrário, a primeira palavra é "maior" que a palavra buscada, jogamos fora a segunda metade do dicionário e repetimos o processo. A função `busca_binaria` abaixo implementa essa idéia.

```
/* Recebe um número inteiro n > 0, um vetor de números in-
   teiros crescente v[0..n-1] e um número inteiro x e devol-
   ve um índice k em [0, n] tal que v[k-1] < x <= v[k] */
int busca_binaria(int n, int v[MAX], int x)
{
   int esq, dir, meio;

   esq = -1;
   dir = n;
   while (esq < dir - 1) {
      meio = (esq + dir) / 2;
      if (v[meio] < x)
         esq = meio;
      else
         dir = meio;
   }

   return dir;
}
```

Um exemplo de chamada à função `busca_binaria` é mostrado abaixo:

```
k = busca_binaria(n, v, x);
```

Para provar a correção da função `busca_binaria`, basta verificar o seguinte invariante:

> no início de cada repetição `while`, imediatamente antes da comparação de `esq` com `dir - 1`, vale a relação `v[esq] < x <= v[dir]`.

Com esse invariante em mãos, podemos usar a estratégia que aprendemos na aula 3 para mostrar finalmente que essa função está de fato correta.

Quantas iterações a função `busca_binaria` executa? O total de iterações revela o valor aproximado que representa o consumo de tempo dessa função em um dado vetor de entrada. Observe que em cada iteração, o tamanho do vetor $v$ é dado por `dir - esq - 1`. No início da primeira iteração, o tamanho do vetor é $n$. No início da segunda iteração, o tamanho do vetor é aproximadamente $n/2$. No início da terceira, aproximadamente $n/4$. No início da $(k+1)$-ésima, aproximadamente $n/2^k$. Quando $k > \log_2 n$, temos $n/2^k < 1$ e a execução da função termina. Assim, o número de iterações é aproximadamente $\log_2 n$. O consumo de tempo da função é proporcional ao número de iterações e portanto proporcional a $\log_2 n$. Esse consumo de tempo cresce com $n$ muito mais lentamente que o consumo da busca seqüencial.

Uma solução recursiva para o problema da busca em um vetor ordenado é apresentada a seguir. Antes, é necessário reformular ligeiramente o problema. A função recursiva `busca_binaria_R` procura o elemento $x$ no vetor crescente `v[esq..dir]`, supondo que o valor $x$ está entre os extremos `v[esq]` e `v[dir]`.

```
/* Recebe dois números inteiros esq e dir, um vetor de números
   inteiros crescente v[esq..dir] e um número inteiro x tais
   que v[esq] < x <= v[dir] e devolve um índice k em
   [esq+1, dir] tal que v[k-1] < x <= v[k] */
int busca_binaria_R(int esq, int dir, int v[MAX], int x)
{
   int meio;

   if (esq == dir - 1)
      return dir;
   else {
      meio = (esq + dir) / 2;
      if (v[meio] < x)
         return busca_binaria_R(meio, dir, v, x);
      else
         return busca_binaria_R(esq, meio, v, x);
   }
}
```

Uma chamada da função `busca_binaria_R` pode ser realizada da seguinte forma:

```
k = busca_binaria_R(-1, n, v, x);
```

Quando a função `busca_binaria_R` é chamada com argumentos $(-1, n, v, x)$, ela chama a si mesma cerca de $\lfloor \log_2 n \rfloor$ vezes. Este número de chamadas é a profundidade da recursão e determina o tempo de execução da função.

## Exercícios

4.1 Tome uma decisão de projeto diferente daquela da seção 4.1: se $x$ não estiver em $v[0..n-1]$, a função deve devolver $n$. Escreva a versão correspondente da função `busca`. Para evitar o grande número de comparações de $k$ com $n$, coloque uma "sentinela" em $v[n]$.

```
/* Recebe um número inteiro n >= 0, um vetor de números intei-
   ros v[0..n-1] e um número inteiro x e devolve k no intervalo
   [0, n-1] tal que v[k] == x. Se tal k não existe, devolve n */
int busca_sequencial_sentinela(int n, int v[MAX+1], int x)
{
   int k;

   v[n] = x;
   for (k = 0; v[k] != x; k++)
      ;

   return k;
}
```

4.2 Considere o problema de determinar o valor de um elemento máximo de um vetor $v[0..n-1]$. Considere a função `maximo` abaixo.

```
int maximo(int n, int v[MAX])
{
   int i, x;

   x = v[0];
   for (i = 1; i < n; i++)
      if (x < v[i])
         x = v[i];

   return x;
}
```

(a) A função `maximo` acima resolve o problema?

(b) Faz sentido trocar `x = v[0]` por `x = 0`?

(c) Faz sentido trocar `x = v[0]` por `x = INT_MIN`[1]?

(d) Faz sentido trocar `x < v[i]` por `x <= v[i]`?

4.3 O autor da função abaixo afirma que ela decide se $x$ está no vetor $v[0..n-1]$. Critique seu código.

```
int buscaR2(int n, int v[MAX], int x)
{
   if (v[n-1] == x)
      return 1;
   else
      return buscaR2(n-1, v, x);
}
```

4.4 A operação de remoção consiste de retirar do vetor $v[0..n-1]$ o elemento que tem índice $k$ e fazer com que o vetor resultante tenha índices $0, 1, \ldots, n-2$. Essa operação só faz sentido se $0 \leqslant k < n$.

(a) Escreva uma função não-recursiva com a seguinte interface:

```
int remove(int n, int v[MAX], int k)
```

que remove o elemento de índice $k$ do vetor $v[0..n-1]$ e devolve o novo valor de $n$, supondo que $0 \leqslant k < n$.

(b) Escreva uma função recursiva para a remoção com a seguinte interface:

```
int removeR(int n, int v[MAX], int k)
```

[1] `INT_MIN` é o valor do menor número do tipo `int`.

4.5 A operação de inserção consiste em introduzir um novo elemento $y$ entre a posição de índice $k-1$ e a posição de índice $k$ no vetor $v[0..n-1]$, com $0 \leqslant k \leqslant n$.

(a) Escreva uma função não-recursiva com a seguinte interface:

```
int insere(int n, int v[MAX], int k, int y)
```

que insere o elemento $y$ entre as posições $k-1$ e $k$ do vetor $v[0..n-1]$ e devolve o novo valor de $n$, supondo que $0 \leqslant k \leqslant n$.

(b) Escreva uma função recursiva para a inserção com a seguinte interface:

```
int insereR(int n, int v[MAX], int k, int x)
```

4.6 Na busca binária, suponha que $v[i] = i$ para todo $i$.

(a) Execute a função `busca_binaria` com $n = 9$ e $x = 3$;

(b) Execute a função `busca_binaria` com $n = 14$ e $x = 7$;

(c) Execute a função `busca_binaria` com $n = 15$ e $x = 7$.

4.7 Execute a função `busca_binaria` com $n = 16$. Quais os possíveis valores de $m$ na primeira iteração? Quais os possíveis valores de $m$ na segunda iteração? Na terceira? Na quarta?

4.8 Confira a validade da seguinte afirmação: quando $n + 1$ é uma potência de 2, o valor da expressão `(esq + dir)` é divisível por 2 em todas as iterações da função `busca_binaria`, quaisquer que sejam $v$ e $x$.

4.9 Responda as seguintes perguntas sobre a função `busca_binaria`.

(a) Que acontece se a sentença `while (esq < dir - 1)` for substituída pela sentença `while (esq < dir)`?

(b) Que acontece se a sentença `if (v[meio] < x)` for substituída pela sentença `if (v[meio] <= x)`?

(c) Que acontece se `esq = meio` for substituído por `esq = meio + 1` ou então por `esq = meio - 1`?

(d) Que acontece se `dir = meio` for substituído por `dir = meio + 1` ou então por `dir = meio - 1`?

4.10 Se $t$ segundos são necessários para fazer uma busca binária em um vetor com $n$ elementos, quantos segundos serão necessários para fazer uma busca em $n^2$ elementos?

4.11 Escreva uma versão da busca binária para resolver o seguinte problema: dado um inteiro $x$ e um vetor decrescente $v[0..n-1]$, encontrar $k$ tal que $v[k-1] > x \geqslant v[k]$.

4.12 Suponha que cada elemento do vetor $v[0..n-1]$ é uma cadeia de caracteres (ou seja, temos uma matriz de caracteres). Suponha também que o vetor está em ordem lexicográfica. Escreva uma função eficiente, baseada na busca binária, que receba uma cadeia de caracteres $x$ e devolva um índice $k$ tal que $x$ é igual a $v[k]$. Se tal $k$ não existe, a função deve devolver $-1$.

4.13 Suponha que cada elemento do vetor $v[0..n-1]$ é um registro com dois campos: o nome do(a) estudante e o número do(a) estudante. Suponha que o vetor está em ordem crescente de números. Escreva uma função de busca binária que receba o número de um(a) estudante e devolva seu nome. Se o número não estiver no vetor, a função deve devolver a cadeia de caracteres vazia.

4.14 Escreva uma função que receba um vetor crescente $v[0..n-1]$ de números inteiros e devolva um índice $i$ entre 0 e $n-1$ tal que $v[i] = i$. Se tal $i$ não existe, a função deve devolver $-1$. A sua função não deve fazer mais que $\lfloor \log_2 n \rfloor$ comparações envolvendo os elementos de $v$.