MC102 – Algoritmos e Programação de Computadores

Instituto de Computação

UNICAMP

Primeiro Semestre de 2016

Roteiro

O problema da busca

- Vamos estudar alguns algoritmos para o seguinte problema:

Dada uma chave de busca e uma coleção de elementos, onde cada elemento possui um identificador único, desejamos encontrar o elemento da coleção que possui o mesmo identificador da chave de busca ou verificar que não existe nenhum elemento na coleção com a chave fornecida.

- Nos nossos exemplos, a coleção de elementos será representada por um vetor de inteiros.
  - O identificador do elemento será o próprio valor de cada elemento.
- Apesar de usarmos inteiros, os algoritmos que estudaremos servem para buscar elementos em qualquer coleção de elementos que possuam chaves que possam ser comparadas.

O problema da busca

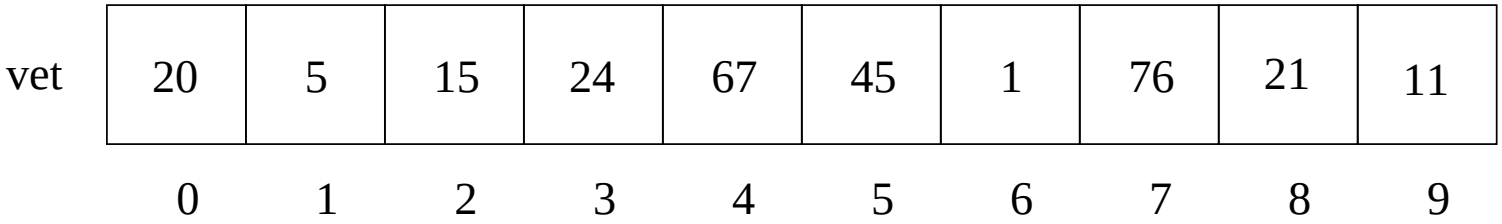
- O problema da busca é um dos mais básicos na área de Computação e possui diversas aplicações.
  - Buscar um aluno dado o seu RA.
  - Buscar um cliente dado o seu CPF.
  - Buscar uma pessoa dado o seu RG.
- Estudaremos algoritmos simples para realizar a busca assumindo que os dados estão em um vetor.
- Em disciplinas mais avançadas, estruturas de dados e algoritmos mais complexos serão estudados para armazenar e buscar elementos.

O problema da busca

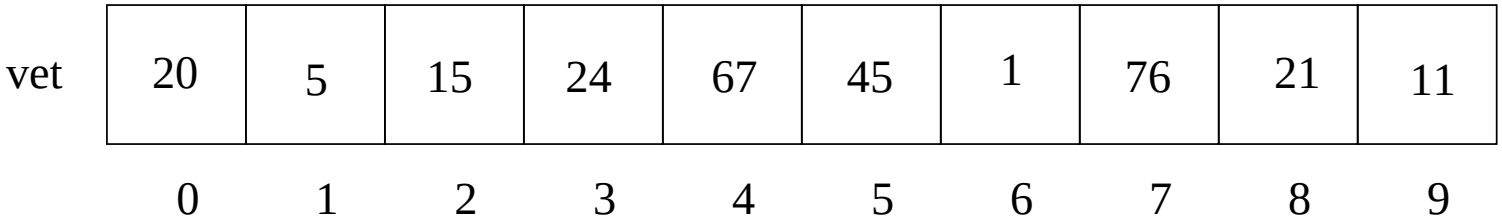
- Nos nossos exemplos vamos criar a função:
  - `int busca(int vet[], int n, int chave)`, que recebe um vetor com `n` inteiros e uma chave para busca.
  - A função deve retornar o índice do vetor que contém a chave ou -1 caso a chave não esteja no vetor.

O problema da busca

chave = 45 n = 10



chave = 100 n = 10



- No primeiro exemplo, a função deve retornar 5, enquanto no segundo exemplo, a função deve retornar -1.

Busca sequencial

- A busca sequencial é o algoritmo mais simples de busca:
  - Percorra o vetor comparando a chave com os valores dos elementos em cada um das posições.
  - Se a chave for igual a algum os elementos, retorne a posição correspondente no vetor.
  - Se o vetor todo foi percorrido e a chave não for encontrada, retorne o valor -1.

Busca sequencial

```
int buscaSequencial(int vet[], int n, int chave) {
  int i;

  for (i = 0; i < n; i++)
    if (vet[i] == chave)
      return i;

  return -1;
}
```

Busca sequencial

```
#include <stdio.h>

int buscaSequencial(int vet[], int n, int chave);

int main() {
  int pos, vet[] = {20, 5, 15, 24, 67, 45, 1, 76, 21, 11};

  pos = buscaSequencial(vet, 10, 45);
  if (pos != -1)
    printf("Posicao da chave 45 no vetor: %d\n", pos);
  else
    printf("A chave 45 nao se encontra no vetor.\n");

  pos = buscaSequencial(vet, 10, 100);
  if (pos != -1)
    printf("Posicao da chave 100 no vetor: %d\n", pos);
  else
    printf("A chave 100 nao se encontra no vetor.\n");

  return 0;
}
```

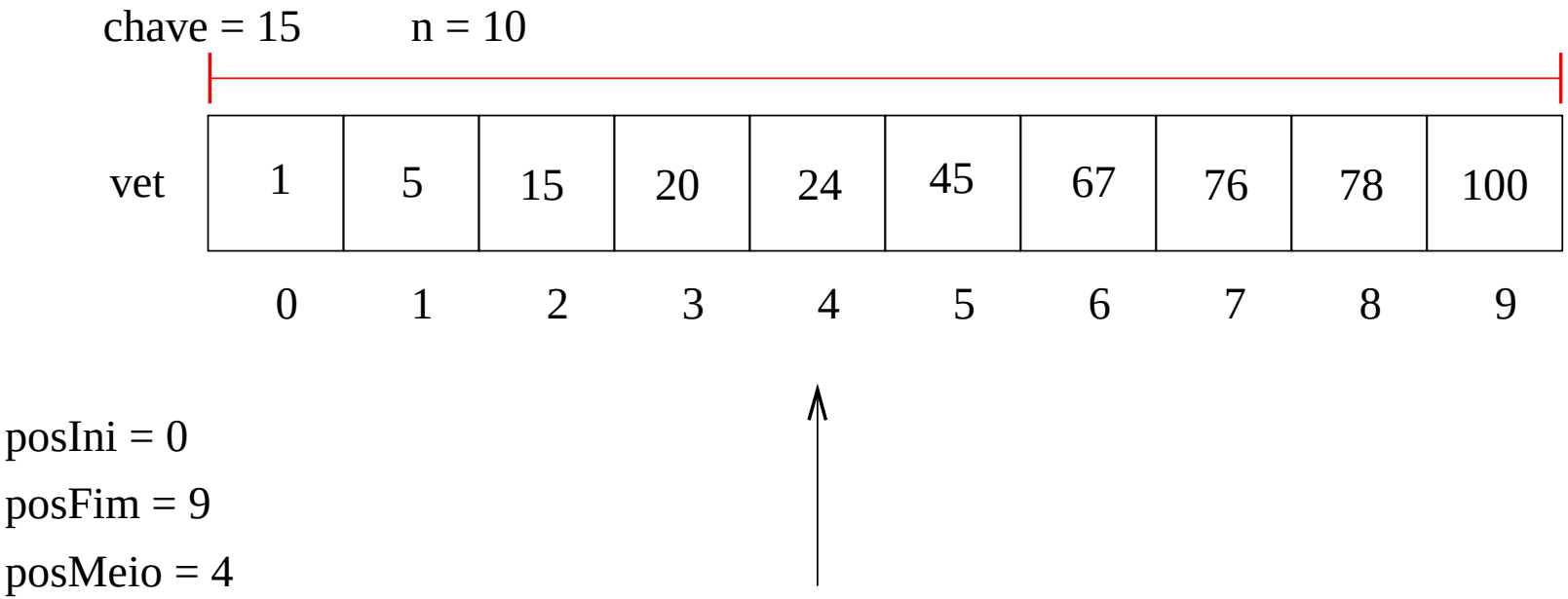
Busca sequencial

```
#include <stdio.h>

int buscaSequencial(int vet[], int n, int chave);

int main() {
  int pos, vet[] = {20, 5, 15, 24, 67, 45, 1, 76, 21, 11};

  pos = buscaSequencial(vet, 10, 45);
  if (pos != -1)
    printf("Posicao da chave 45 no vetor: %d\n", pos); /* pos = 5 */
  else
    printf("A chave 45 nao se encontra no vetor.\n");

  pos = buscaSequencial(vet, 10, 100);
  if (pos != -1)
    printf("Posicao da chave 100 no vetor: %d\n", pos);
  else
    printf("A chave 100 nao se encontra no vetor.\n"); /* pos = -1 */

  return 0;
}
```

Busca binária

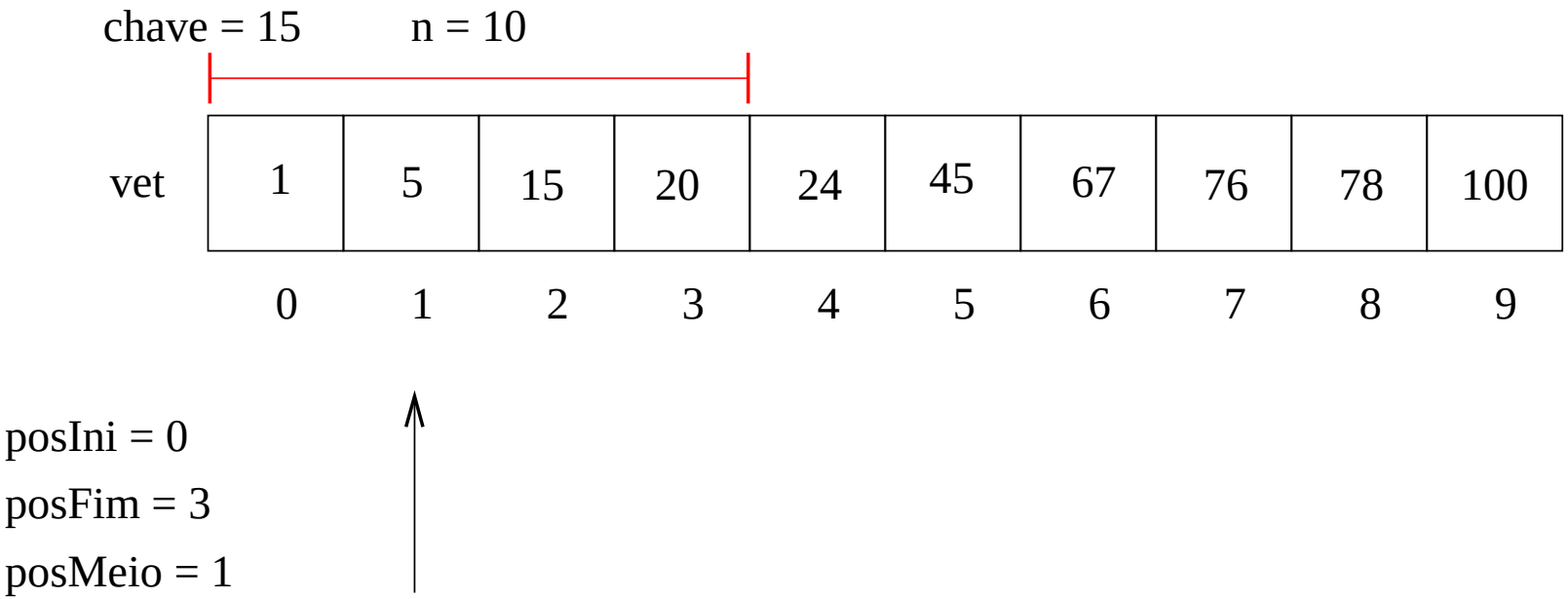
- A busca binária é um algoritmo mais eficiente, entretanto, requer que o vetor esteja ordenado pelos valores da chave de busca.
- A ideia do algoritmo é a seguinte (assuma que o vetor está ordenado pelos valores da chave de busca):
  - Verifique se a chave de busca é igual ao valor da posição do meio do vetor.
  - Caso seja igual, devolva esta posição.
  - Caso o valor desta posição seja maior, então repita o processo, mas considere que o vetor tem metade do tamanho, indo até a posição anterior a do meio.
  - Caso o valor desta posição seja menor, então repita o processo, mas considere que o vetor tem metade do tamanho e inicia na posição seguinte a do meio.

Busca binária



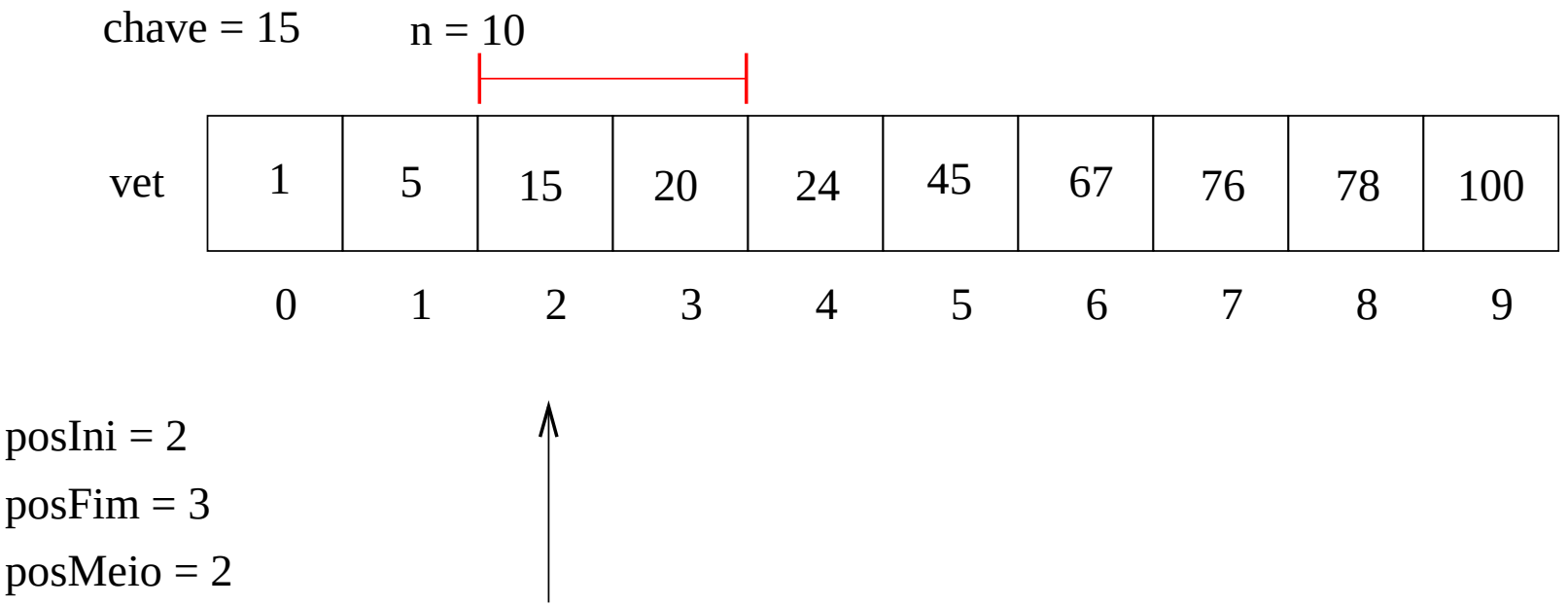
- Como `vet[posMeio] > chave`, devemos continuar a busca na primeira metade da região e, para isso, atualizamos a variável `posFim`.

Busca binária



- Como `vet[posMeio] < chave`, devemos continuar a busca na segunda metade da região e, para isso, atualizamos a variável `posIni`.

Busca binária



- Finalmente, encontramos a chave (`vet[posMeio] = chave`) e, sendo assim, devolvemos a sua posição no vetor (`posMeio`).

Busca binária

```
buscaBinaria(vet, n, chave)
  posIni = 1
  posFim = n

  enquanto posIni <= PosFim:
    posMeio = (posIni + posFim) div 2

    Se vet[posMeio] = chave então
      retorne posMeio
    Se vet[posMeio] > chave então
      posFim = posMeio - 1
    Se vet[posMeio] < chave então
      posIni = posMeio + 1

  retorne 0
```

Busca binária

```
int buscaBinaria(int vet[], int n, int chave) {
  int posIni = 0, posFim = n - 1, posMeio;

  while (posIni <= posFim) {
    posMeio = (posIni + posFim) / 2;

    if (vet[posMeio] == chave)
      return posMeio;
    else if (vet[posMeio] > chave)
      posFim = posMeio - 1;
    else
      posIni = posMeio + 1;
  }

  return -1;
}
```

Busca binária

```
#include <stdio.h>

int buscaBinaria(int vet[], int n, int chave);
int insertSort(int vet[], int n);

int main() {
  int vet[] = {20, 5, 24, 15, 100, 67, 45, 78, 1, 76};
  int pos, i;

  /* A busca binaria deve utilizada num vetor ordenado */
  insertionSort(vet, 10);

  pos = buscaBinaria(vet, 10, 15);
  if (pos != -1)
    printf("Posicao da chave 15 no vetor: %d\n", pos);
  else
    printf("A chave 15 nao se encontra no vetor.\n");

  retutn 0;
}
```

Busca binária

```
#include <stdio.h>

int buscaBinaria(int vet[], int n, int chave);
int insertSort(int vet[], int n);

int main() {
  int vet[] = {20, 5, 24, 15, 100, 67, 45, 78, 1, 76};
  int pos, i;

  /* A busca binaria deve utilizada num vetor ordenado */
  insertionSort(vet, 10);

  pos = buscaBinaria(vet, 10, 15);
  if (pos != -1)
    printf("Posicao da chave 15 no vetor: %d\n", pos); /* pos = 2 */
  else
    printf("A chave 15 nao se encontra no vetor.\n");

  retutn 0;
}
```

Eficiência dos algoritmos

- Podemos medir a eficiência de um algoritmo analisando a quantidade de recursos (tempo, memória, largura de rede, energia, etc) que o algoritmo usa para resolver o problema para o qual foi proposto.
- A forma mais simples é medir a eficiência em relação ao tempo de processamento. Para isso, analisamos quantas instruções um algoritmo usa para resolver o problema.
- Podemos fazer uma análise simplificada dos algoritmos de busca analisando o número de acessos ao vetor.

Eficiência da busca sequencial

- Na melhor das hipóteses, a chave de busca estará na posição 0. Portanto, teremos um único acesso em `vet[0]`.
- Na pior das hipóteses, a chave é o último elemento ou não pertence ao vetor e, portanto, faremos acesso a todas as n posições do vetor.
- É possível mostrar que, se as chaves possuirem a mesma probabilidade de serem requisitadas, o número médio de acessos nas buscas cujas chaves encontram-se no vetor será igual a:

$$\frac{n+1}{2}$$

Eficiência da busca binária

- Na melhor das hipóteses, a chave de busca estará na posição do meio do vetor. Portanto, teremos um único acesso.
- Na pior das hipóteses, teremos $(\log_2 n)$ acessos.
  - Para observar isso, note que, a cada verificação de uma posição do vetor, o tamanho do vetor considerado é dividido pela metade.
  - No pior caso, a busca é repetida até que o vetor considerado tenha tamanho 1.
  - Assim, o número de acessos x pode ser encontrado resolvendo-se a equação:

$$\frac{n}{2^x} = 1$$

    cuja solução é $x = \log_2 n$.
- É possível mostrar que, se as chaves possuirem a mesma probabilidade de serem requisitadas, o número médio de acessos nas buscas cujas chaves encontram-se no vetor será igual a:

$$(\log_2 n) - 1$$

Eficiência dos algoritmos

- Para se ter uma ideia da diferença de eficiência dos dois algoritmos, considere um vetor com um milhão de itens (10^6 itens).
- Com a busca sequencial, para buscar um elemento qualquer do vetor necessitamos, em média, de:

$$(10^6 + 1)/2 \approx 500000 \text{ acessos.}$$

- Com a busca binária, para buscar um elemento qualquer do vetor necessitamos, em média, de:

$$(\log_2 10^6) - 1 \approx 19 \text{ acessos.}$$

Eficiência dos algoritmos

- Uma ressalva importante deve ser feita: para utilizar a busca binária, o vetor precisa estar ordenado.
- Se você tiver um cadastro onde vários itens são removidos e inseridos com frequência e a busca deve ser feita de forma intercalada com essas operações, então a busca binária pode não ser a melhor opção, já que você precisará manter o vetor ordenado.
- Caso o número de buscas seja muito maior que as demais operações de atualização do cadastro, então a busca binária pode ser uma boa opção.

Exercícios

- Refaça as funções de busca sequencial e busca binária assumindo que o vetor possui chaves que podem ocorrer múltiplas vezes no vetor. Neste caso, você deve retornar, em um outro vetor, todas as posições onde a chave foi encontrada. Protótipo:
  `int busca(int vet[], int n, int chave, int posicoes[]);`
  - Sua função deve retornar o número de ocorrências da chave no vetor e, para cada uma destas ocorrências, indicar no vetor `posicoes[]`, as posições de `vet` que possuem valor igual à `chave`.
  - Na chamada desta função, o vetor `posicoes` deve ter espaço suficiente (`n`) para guardar todas as possíveis ocorrências da `chave` no vetor.
- Refaça o exercício acima, desta vez criando dinamicamente o vetor com as posições que possuem valores iguais à `chave`. Você deve alocar a quantidade exata de memória necessária para armazenar as posições encontradas.

Exercícios

- Mostre como implementar uma variação da busca binária que retorne um inteiro `k` entre `0` e `n`, tal que, ou `vet[k] = chave`, ou a chave não se encontra no vetor, mas poderia ser inserida entre as posições `(k-1)` e `k` de forma a manter o vetor ordenado. Note que, se `k = 0`, então a chave deveria ser inserida antes da primeira posição do vetor, assim como, se `k = n`, a chave deveria ser inserida após a última posição do vetor.
- Use a função desenvolvida acima para, dado um vetor ordenado de `n` números inteiros e distintos e dois outros inteiros `X` e `Y`, retornar o número de chaves do vetor que são maiores ou iguais a `X` e menores ou iguais a `Y`.