Programação Estruturada

Ordenação

Professores Emílio Francesquini e Carla Negri Lintzmayer

2018.Q3

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

O problema da ordenação

Ordenação

Problema

Dado uma coleção de elementos com uma relação de ordem entre si, devemos gerar uma saída com os elementos ordenados.

Nos nossos exemplos usaremos um vetor de inteiros para representar tal coleção.

- É claro que quaisquer inteiros possuem uma relação de ordem entre si.

Apesar de usarmos inteiros, os algoritmos servem para ordenar qualquer coleção de elementos que possam ser comparados.

Ordenação

- O problema de ordenação é um dos mais básicos em computação.
  - Mas muito provavelmente é um dos problemas com o maior número de aplicações diretas ou indiretas (como parte da solução para um problema maior).
- Criar *rankings*
- Definir preferências em atendimentos por prioridade
- Criar listas
- Otimizar sistemas de busca
- Manter estruturas de bancos de dados
- Etc.

Ordenação

Existem **muitos** algoritmos de ordenação:

- Selection Sort
- Merge Sort
- Insertion Sort
- Quicksort
- Heapsort
- Bubble Sort
- Shell Sort
- Bogosort
- ...
- `https://nicholasandre.com.br/sorting/`

Selection Sort

Selection Sort

- Seja `vet` um vetor contendo números inteiros.
- Devemos deixar `vet` em ordem crescente.
- A ideia do algoritmo é a seguinte:
  - Ache o menor elemento a partir da posição 0. Troque então este elemento com o elemento da posição 0.
  - Ache o menor elemento a partir da posição 1. Troque então este elemento com o elemento da posição 1.
  - Ache o menor elemento a partir da posição 2. Troque então este elemento com o elemento da posição 2.
  - E assim sucessivamente...

Selection Sort

Exemplo: (5,3,2,1,90,6).

Iteração 0. Acha o menor: (5,3,2,1,90,6).
Faz troca: (1,3,2,5,90,6).
Iteração 1. Acha o menor: (1,3,2,5,90,6).
Faz troca: (1,2,3,5,90,6).
Iteração 2. Acha o menor: (1,2,3,5,90,6).
Faz troca: (1,2,3,5,90,6).
Iteração 3. Acha o menor: (1,2,3,5,90,6).
Faz troca: (1,2,3,5,90,6).
Iteração 4. Acha o menor: (1,2,3,5,90,6).
Faz troca: (1,2,3,5,6,90).

Selection Sort

- Como achar o menor elemento a partir de uma posição inicial?
- Vamos achar **o índice** do menor elemento em um vetor, a partir de uma posição inicial **ini**:

```
1  int min = ini, j;
2  for (j = ini+1; j < tam; j++) {
3      if (vet[min] > vet[j])
4          min = j;
5  }
```

Selection Sort

Criamos então uma função que retorna o índice do elemento mínimo de um vetor, a partir de uma posição **ini** passada por parâmetro.

```
1  int indiceMenor(int vet[], int tam, int ini) {
2      int min = ini, j;
3      for (j = ini+1; j < tam; j++) {
4          if (vet[min] > vet[j])
5              min = j;
6      }
7      return min;
8  }
```

Selection Sort

- Dada a função anterior para achar o índice do menor elemento, como implementar o algoritmo de ordenação?
- Ache o menor elemento a partir da posição 0, e troque com o elemento da posição 0.
- Ache o menor elemento a partir da posição 1, e troque com o elemento da posição 1.
- Ache o menor elemento a partir da posição 2, e troque com o elemento da posição 2.
- E assim sucessivamente...

Selection Sort

```
1  void selectionSort(int vet[], int tam) {
2      int i, min, aux;
3
4      for (i = 0; i < tam; i++) {
5          /* Acha posicao do menor elemento a partir
           ↪  de i */
6          min = indiceMenor(vet, tam, i);
7          /* Troca o menor elemento com o da posição
           ↪  i */
8          aux = vet[i];
9          vet[i] = vet[min];
10         vet[min] = aux;
11     }
12 }
```

Selection Sort

Com as funções anteriores podemos executar o exemplo:

```c
int main() {
    int vetor[10] = {14,7,8,34,56,4,0,9,-8,100};
    int i;

    printf("Vetor Antes: (%d", vetor[0]);
    for (i = 1; i < 10; i++)
        printf(", %d", vetor[i]);
    printf(")\n");

    selectionSort(vetor, 10);

    printf("Vetor Depois: (%d", vetor[0]);
    for (i = 1; i < 10; i++)
        printf(", %d", vetor[i]);
    printf(")\n");

    return 0;
}
```

Selection Sort

- A função para achar o índice do menor elemento não é estritamente necessária.
- Podemos refazer a função `selectionSort` como segue:

```
1  void selectionSort(int vet[], int tam) {
2      int i, j, min, aux;
3      for (i = 0; i < tam; i++) {
4          min = i;
5          for (j = i+1; j < tam; j++) {
6              if (vet[min] > vet[j])
7                  min = j;
8          }
9          aux = vet[i];
10         vet[i] = vet[min];
11         vet[min] = aux;
12     }
13 }
```

Selection Sort

Mas podemos também criar uma função troca:

```
1  void troca(int *a, int *b) {
2      int aux = *a;
3      *a = *b;
4      *b = aux;
5  }
```

E assim `selectionSort` pode ser reescrita:

```
1  void selectionSort(int vet[], int tam) {
2      int i, min;
3      for (i = 0; i < tam; i++) {
4          min = indiceMenor(vet, tam, i);
5          troca(&vet[i], &vet[min]);
6      }
7  }
```

Merge Sort

Merge Sort

- Seja `vet` um vetor contendo `tam` números inteiros.
- Devemos deixar `vet` em ordem crescente.
- E se soubéssemos ordenar vetores de tamanho menor?

Merge Sort

- Suponha que temos dois vetores `A` e `B` de tamanho `tam`/2 cada que já estão em ordem crescente.
- Como podemos colocá-los em um único vetor `vet` de tamanho `tam` totalmente ordenado?

$$(3, 5, 7, 10, 25, 36) \qquad (1, 2, 6, 13, 17, 28)$$

- Qual elemento vai ficar em `vet[0]`?

Merge Sort

- Em `vet[0]` só podemos ter `A[0]` ou `B[0]`, o que for menor dentre esses dois.
- Se `vet[0] = A[0]`, então `vet[1]` só pode ter `A[1]` ou `B[0]`, o que for menor dentre esses dois.
- Mas se `vet[0] = B[0]`, então `vet[1]` só pode ter `A[0]` ou `B[1]`, o que for menor dentre esses dois.
- Note que uma vez que um elemento `A[i]` é copiado para `vet`, esse elemento não deve mais ser considerado (comparado com um elemento de `B`).
- Da mesma forma, uma vez que um elemento `B[j]` é copiado para `vet`, esse elemento não deve mais ser considerado (comparado com um elemento de `A`).

Merge Sort

Precisamos manter:

- Um índice `i` para percorrer o vetor `A`.
- Um índice `j` para percorrer o vetor `B`.
- Um índice `k` para percorrer o vetor `vet`.

A cada iteração, precisamos colocar um elemento em `vet[k]`:

- Se `A[i] < B[j]`, então `vet[k] = A[i]` e `i` deve ser incrementado, para que `A[i]` não seja considerado novamente. Nesse caso, `j` não deve ser incrementado.
- Se `B[j] < A[i]`, então `vet[k] = B[j]` e `j` deve ser incrementado para que `B[j]` não seja considerado novamente. Nesse caso, `i` não deve ser incrementado.

Merge Sort

- O que garantimos com as ações anteriores?
- Na hora de colocar um elemento na posição `k` de `vet`, sabemos que $vet[0..k-1]$ está preenchido com os menores elementos de `A` e `B` e que está ordenado.
- `A[i]` e `B[j]` são menores do que todos os elementos em $A[i+1..tam/2]$ e $B[j+1..tam/2]$.
- Além disso, `A[i]` e `B[j]` são maiores do que todos os elementos em $vet[0..k-1]$, então o menor dos dois é de fato o elemento que deve ir para `vet[k]`.

Merge Sort

Exemplo: $A = (3, 5, 7, 10)$, $B = (1, 2, 6, 13)$ e $vet = ()$.

Iteração $k = 0$. $A = (\underline{3}, 5, 7, 10)$ $B = (\underline{1}, 2, 6, 13)$
$vet = (1)$ (preenche posição k)
Iteração $k = 1$. $A = (\underline{3}, 5, 7, 10)$ $B = (1, \underline{2}, 6, 13)$
$vet = (1, 2)$
Iteração $k = 2$. $A = (\underline{3}, 5, 7, 10)$ $B = (1, 2, \underline{6}, 13)$
$vet = (1, 2, 3)$
Iteração $k = 3$. $A = (3, \underline{5}, 7, 10)$ $B = (1, 2, \underline{6}, 13)$
$vet = (1, 2, 3, 5)$
Iteração $k = 4$. $A = (3, 5, \underline{7}, 10)$ $B = (1, 2, \underline{6}, 13)$
$vet = (1, 2, 3, 5, 6)$
Iteração $k = 5$. $A = (3, 5, \underline{7}, 10)$ $B = (1, 2, 6, \underline{13})$
$vet = (1, 2, 3, 5, 6, 7)$
Iteração $k = 6$. $A = (3, 5, 7, \underline{10})$ $B = (1, 2, 6, \underline{13})$
$vet = (1, 2, 3, 5, 6, 7)$
Com outras duas iterações, $vet = (1, 2, 3, 5, 6, 7, 10, 13)$

Merge Sort

Cuidados extras:

- Na verdade, se temos um total de `tam` elementos, `A` tem tamanho $\lfloor$`tam`$/2\rfloor$ e `B` tem tamanho $\lceil$`tam`$/2\rceil$.
- Devemos ainda tomar cuidado para que `i` e `j` sejam posições válidas em `A` e `B`.
  - Se $A = (1, 2, 3, 4, 5)$ e $B = (6, 7, 8, 9, 10)$, `A` será totalmente copiado para `vet` antes mesmo que `B[0]` seja considerado.

Merge Sort

```
1  void intercala(int A[], int B[], int vet[], int tam) {
2      int i, j, k, tamA, tamB;
3      i = j = k = 0;
4      tamA = tamB = tam/2;
5      if (tam%2 == 1)
6          tamB = tam/2 + 1;
7      while (i < tamA && j < tamB) {
8          if (A[i] < B[j]) { /* copie A[i] se ele for o
           ↪  menor */
9              vet[k] = A[i];
10             i++;
11         } else { /* copie B[j] se ele for o menor */
12             vet[k] = B[j];
13             j++;
14         }
15         k++;
16     }
17     /* aqui sabemos que OU i == tamA OU j == tamB */
18     ...
```

Merge Sort

```
1        ...
2        /* continuamos a copiar elementos de A, se existirem
         ↪  */
3        while (i < tamA) {
4            vet[k] = A[i];
5            i++;
6            k++;
7        }
8
9        /* continuamos a copiar elementos de B, se existirem
         ↪  */
10       while (j < tamB) {
11           vet[k] = B[j];
12           j++;
13           k++;
14       }
15   }
```

Outra forma de implementar a função `intercala`:

```
1  /* o vetor vet está ordenado da posição ini até meio
2     e da posição meio+1 até fim */
3  void intercala(int vet[], int ini, int meio, int fim) {
4      int i, j, k, tamA, tamB, *A, *B;
5      tamA = meio - ini + 1;
6      tamB = fim - meio;
7
8      /* alocando espaço suficiente para A e B */
9      A = malloc(tamA * sizeof(int));
10     B = malloc(tamB * sizeof(int));
11
12     /* copiando os elementos de vet para A e B: */
13     for (i = 0; i < tamA; i++)
14         A[i] = vet[ini + i];
15     for (i = 0; i < tamB; i++)
16         B[i] = vet[meio+1 + i];
17
18     ...
```

Merge Sort

```
1       ...
2       i = j = 0;
3       k = ini;
4       /* o resto se mantém igual */
5       while (i < tamA && j < tamB) {
6           if (A[i] < B[j]) { /* copie A[i] se ele for o menor */
7               vet[k] = A[i];    i++;
8           } else { /* copie B[j] se ele for o menor */
9               vet[k] = B[j];    j++;
10          }
11          k++;
12      }
13      while (i < tamA) {
14          vet[k] = A[i];    i++;    k++;
15      }
16      while (j < tamB) {
17          vet[k] = B[j];    j++;    k++;
18      }
19      /* libere os vetores alocados dinamicamente! */
20      free(A); free(B);
21  }
```

Merge Sort

- Com a função `intercala`, sabemos como ordenar um vetor de tamanho `tam` qualquer.
- Basta que esse vetor seja dividido em dois vetores de tamanho (quase) `tam`/2, e que esses vetores menores sejam ordenados!
- Mas então o problema continua o mesmo: ordenar um vetor.
- Acontece que agora o problema tem tamanho menor.
- Podemos então usar recursão para resolvê-lo!

Merge Sort

- A ideia do algoritmo Merge Sort é:
  - Divida o vetor inicial em dois vetores de tamanho (quase) metade do tamanho inicial.
  - Ordene cada um desses dois novos vetores de forma recursiva.
  - Utilize o algoritmo `intercala` para juntar os dois vetores e ter o vetor inicial ordenado.
- É claro que se o vetor inicial tem tamanho pequeno o suficiente, ele pode ser ordenado diretamente.
  - Um vetor de tamanho 1, em particular, já está ordenado!

Merge Sort

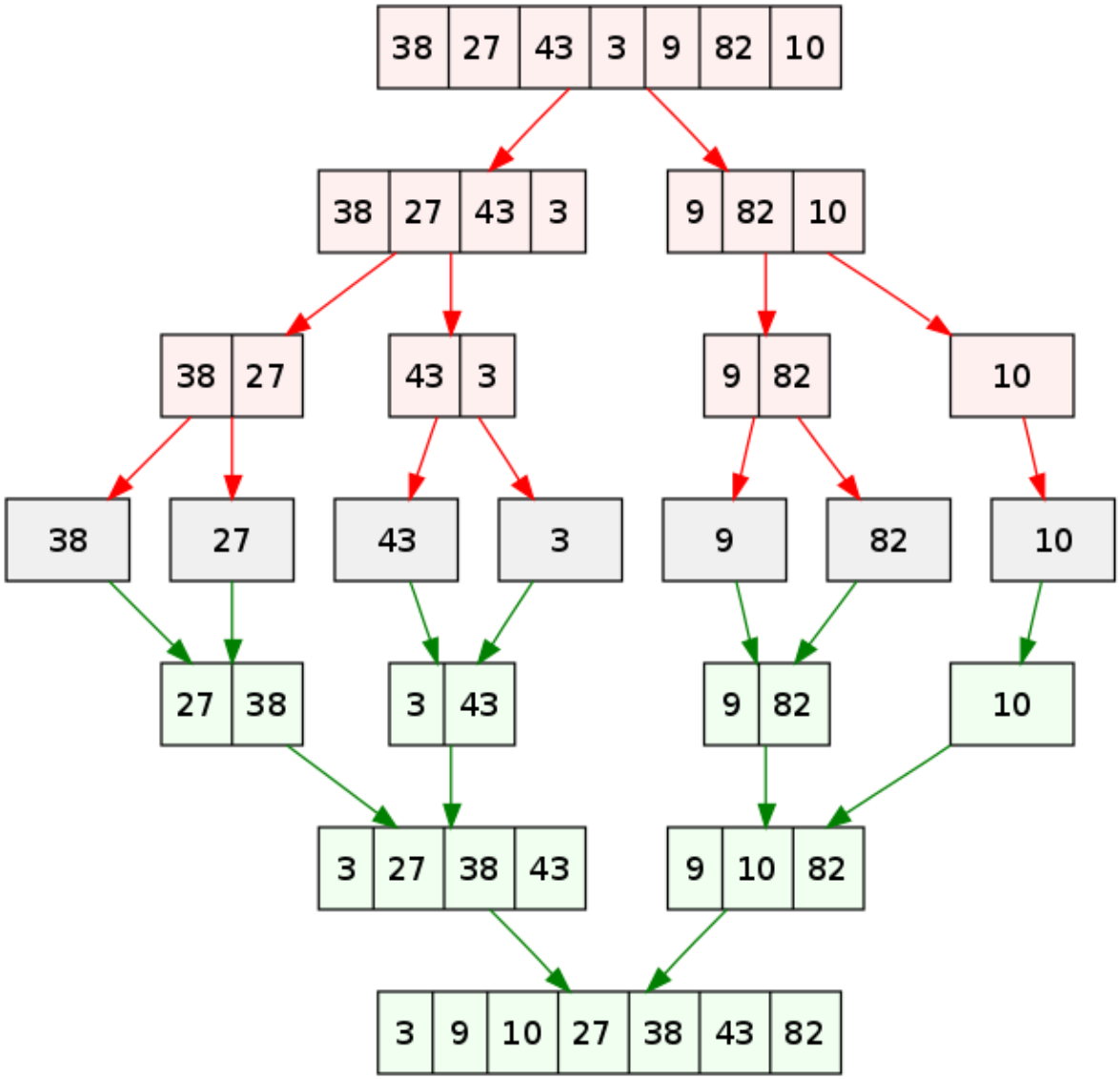


Figura 1: Fonte:

Merge Sort

```
1  void mergeSort(int vet[], int ini, int fim) {
2      if (fim <= ini)
3          return;
4
5      int meio = (ini + fim)/2;
6      mergeSort(vet, ini, meio);
7      mergeSort(vet, meio+1, fim);
8      intercala(vet, ini, meio, fim);
9  }
```

Merge Sort

Com as funções anteriores podemos executar o exemplo:

```c
int main() {
    int vetor[10] = {14,7,8,34,56,4,0,9,-8,100};
    int i;

    printf("Vetor Antes: (%d", vetor[0]);
    for (i = 1; i < 10; i++)
        printf(", %d", vetor[i]);
    printf(")\n");

    mergeSort(vetor, 0, 9);

    printf("Vetor Depois: (%d", vetor[0]);
    for (i = 1; i < 10; i++)
        printf(", %d", vetor[i]);
    printf(")\n");

    return 0;
}
```

Exercícios

Exercício 1

Altere os algoritmos vistos nesta aula para que estes ordenem um vetor de inteiros em ordem decrescente ao invés de ordem crescente.

Exercício 2

No algoritmo `selectionSort`, o laço principal é executado de 0 até `tam-2` e não `tam-1`. Por quê?

Informações extras: outros algoritmos famosos de ordenação

Insertion Sort

- Seja `vet` um vetor contendo números inteiros, que devemos deixar ordenado.
- A ideia do algoritmo é a seguinte:
  - A cada passo, uma porção de 0 até $i - 1$ do vetor já está ordenada.
  - Devemos inserir o item da posição i na posição correta para deixar o vetor ordenado até a posição i.
  - No passo seguinte poderemos então consider que o vetor está ordenado até i.

Insertion Sort

Exemplo: (5,3,2,1,90,6).

O valor sublinhado representa onde está o índice *i*:

$(5, \underline{3}, 2, 1, 90, 6)$: vetor ordenado de $0 - 0$.

$(3, 5, \underline{2}, 1, 90, 6)$: vetor ordenado de $0 - 1$.

$(2, 3, 5, \underline{1}, 90, 6)$: vetor ordenado de $0 - 2$.

$(1, 2, 3, 5, \underline{90}, 6)$: vetor ordenado de $0 - 3$.

$(1, 2, 3, 5, 90, \underline{6})$: vetor ordenado de $0 - 4$.

$(1, 2, 3, 5, 6, 90)$: vetor ordenado de $0 - 5$.

Insertion Sort

- Vamos supor que o vetor está ordenado de 0 até $i - 1$.
- Vamos inserir o elemento da posição i no lugar correto.

```
1  j = i;
2  while (j > 0) {
3      /* trocar v[i] com elementos anteriores até
       ↪  achar sua posicao correta relativa a esses
       ↪  elementos */
4      if (vet[j-1] > vet[j]) {
5          troca(&vet[j], &vet[j-1]);
6          j--;
7      } else {
8          break;
9      }
10 }
```

Insertion Sort

Código completo:

```c
void insertionSort(int vet[], int tam) {
    int i, j;

    for (i = 1; i < tam; i++) {
        /* Colocar elemento v[i] na pos. correta */
        j = i;
        while (j > 0) {
            /* trocar v[i] com elementos anteriores até achar
            ↪  sua posicao correta relativa a esses elementos
            ↪  */
            if (vet[j-1] > vet[j]) {
                troca(&vet[j], &vet[j-1]);
                j--;
            } else {
                break;
            }
        }
    }
}
```

Insertion Sort

- Vamos apresentar uma forma alternativa de colocar `v[i]` na posição correta.
- Vamos supor que o vetor está ordenado de 0 até $i - 1$.
- Vamos inserir o elemento da posição i no lugar correto.

```
1  aux = vet[i];  /* precisamos inserir aux na posição correta */
2  j = i - 1;     /* vamos analisar elementos das posições
   ↪  anteriores, a começar por i-1 */
3
4  while (j >= 0 && vet[j] > aux) {
5      /* enquanto vet[j] > aux, "empurra" vet[j] para a direita */
6      vet[j+1] = vet[j];
7      j--;
8  }
9
10 /* Nesse ponto, OU j = -1 OU vet[j] <= aux
11    De qualquer forma, j+1 é a posição correta para vet[i] (aux)
   ↪  */
12 vet[j+1] = aux;
```

Insertion Sort

Exemplo $(1, 3, 5, 10, 20, 2^*, 4)$ com $i = 5$.

$(1, 3, 5, 10, \underline{20}, 2, 4)$: `aux` = 2; `j` = 4;

$(1, 3, 5, \underline{10}, 20, 20, 4)$: `aux` = 2; `j` = 3;

$(1, 3, \underline{5}, 10, 10, 20, 4)$: `aux` = 2; `j` = 2;

$(1, \underline{3}, 5, 5, 10, 20, 4)$: `aux` = 2; `j` = 1;

$(\underline{1}, 3, 3, 5, 10, 20, 4)$: `aux` = 2; `j` = 0;

Aqui temos que `vet[j] < aux`, logo, fazemos `vet[j+1] = aux`

$(1, 2, 3, 5, 10, 20, 4)$: `aux` = 2; `j` = 0;

Insertion Sort

Código completo.

```c
void insertionSort(int vet[], int tam) {
    int i, j, aux;

    for (i = 1; i < tam; i++) {
        /* Assuma vetor ordenado de 0 até i-1 */
        aux = vet[i];
        j = i-1;

        while (j >= 0 && vet[j] > aux) {
            /* Coloca elementos v[j] > v[i] para a direita */
            vet[j+1] = vet[j];
            j--;
        }

        vet[j+1] = aux; /* coloca v[i] na pos. correta */
    }
}
```

BubbleSort

- Seja `vet` um vetor contendo números inteiros.
- Devemos deixar `vet` em ordem crescente.
- O algoritmo começa fazendo o seguinte:
  - Compare `vet[0]` com `vet[1]` e troque-os se `vet[0] > vet[1]`.
  - Compare `vet[1]` com `vet[2]` e troque-os se `vet[1] > vet[2]`.
  - ...
  - Compare `vet[tam-2]` com `vet[tam-1]` e troque-os se `vet[tam-2] > vet[tam-1]`.
- Após uma iteração repetindo estes passos o que podemos garantir?
  - Que o maior elemento estará na posição correta!

BubbleSort

- Após uma iteração de trocas, o maior elemento estará na última posição.
- Após outra iteração de trocas, o segundo maior elemento estará na posição correta.
- E assim sucessivamente.
- Quantas iterações repetindo estas trocas precisamos para deixar o vetor ordenado?

BubbleSort

Exemplo: (5,3,2,1,90,6).

Valores sublinhados estão sendo comparados:

$(\underline{5}, \underline{3}, 2, 1, 90, 6)$

$(3, \underline{5}, \underline{2}, 1, 90, 6)$

$(3, 2, \underline{5}, \underline{1}, 90, 6)$

$(3, 2, 1, \underline{5}, \underline{90}, 6)$

$(3, 2, 1, 5, \underline{90}, \underline{6})$

$(3, 2, 1, 5, 6, 90)$

- Isto termina a primeira iteração de trocas.
- Temos que repetir todo o processo mais 4 vezes!
- Mas note que não precisamos mais avaliar a última posição.

BubbleSort

- O código abaixo realiza as trocas de uma iteração.
- São comparados e trocados os elementos das posições: 0 e 1, 1 e 2, . . ., $i-1$ e i.
- Assumimos que de $(i+1)$ até $(tam-1)$ o vetor já está com os maiores elementos ordenados.

```
1  for (j = 0; j < i; j++)
2      if (vet[j] > vet[j+1])
3          troca(&vet[j], &vet[j+1]);
```

BubbleSort

```
1  void bubbleSort(int vet[], int tam) {
2      int i, j, aux;
3
4      for (i = tam-1; i > 0; i--) {
5          for (j = 0; j < i; j++) /* Troca até pos. i
           ↪  */
6              if (vet[j] > vet[j+1])
7                  troca(&vet[j], &vet[j+1]);
8      }
9  }
```

BubbleSort

- Note que as trocas na primeira iteração ocorrem até a última posição.
- Na segunda iteração ocorrem até a penúltima posição.
- E assim sucessivamente.
- Por quê?

Informações extras: o problema da busca

O problema da busca

Temos uma coleção de elementos, onde cada elemento possui um identificador/chave único, e recebemos uma chave para busca. Devemos encontrar o elemento da coleção que possui a mesma chave ou identificar que não existe nenhum elemento com a chave dada.

Nos nossos exemplos usaremos um vetor de inteiros como a coleção.

O valor da chave será o próprio valor de cada número.

Apesar de usarmos inteiros, os algoritmos servem para buscar elementos em qualquer coleção de elementos que possuam chaves que possam ser comparadas, como registros com algum campo de identificação único (RA, ou RG, ou CPF, etc.).

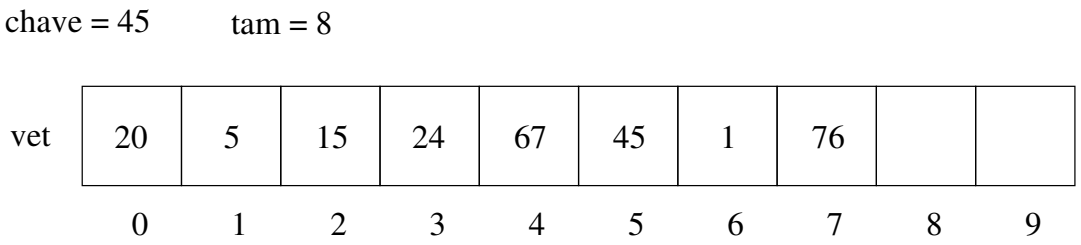
O Problema da busca

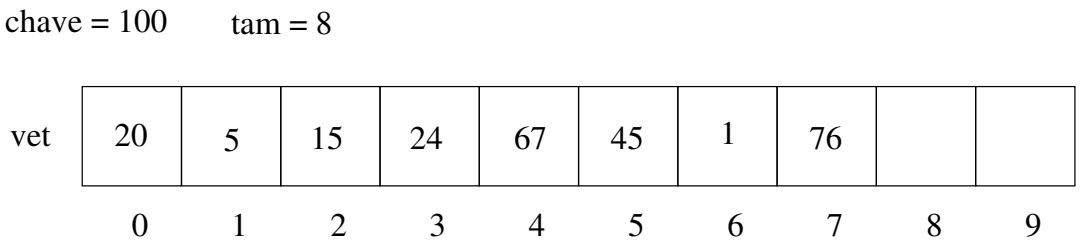
- O problema da busca é um dos mais básicos em computação e também possui diversas aplicações.
  - Suponha que temos um cadastro com registros de motoristas.
  - Um vetor de registros é usado para armazenar as informações dos motoristas. Podemos usar como chave o número da carteira de motorista, ou o RG, ou o CPF.
- Veremos algoritmos simples para realizar a busca assumindo que dados estão em um vetor.
- Em disciplinas mais avançadas são estudados outros algoritmos e estruturas (que não um vetor) para armazenar e buscar elementos.

O Problema da busca

- Nos nossos exemplos vamos criar a função `int busca(int vet[], int tam, int chave)` que recebe um vetor com um determinado tamanho, 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





No exemplo mais acima, a função deve retornar 5, enquanto no exemplo mais abaixo a função deve retornar -1.

Busca sequencial

- A busca sequencial é o algoritmo mais simples de busca:
  - Percorra todo o vetor comparando a chave com o valor de cada posição.
  - Se for igual para alguma posição, então devolva esta posição.
  - Se o vetor todo foi percorrido então devolva -1.

Busca sequencial

```
1  int buscaSequencial(int vet[], int tam, int chave)
   ↪  {
2      int i;
3      for (i = 0; i < tam; i++) {
4          if (vet[i] == chave)
5              return i;
6      }
7      return -1;
8  }
```

Busca sequencial

```c
#include <stdio.h>
int buscaSequencial(int vet[], int tam, int chave);
int main() {
    int pos, vet[] = {20, 5, 15, 24, 67, 45, 1, 76};

    pos = buscaSequencial(vet, 8, 45);
    if (pos != -1)
        printf("A posicao da chave 45 no vetor é: %d\n",
        ↪  pos);
    else
        printf("A chave 45 não está no vetor!\n");

    pos = buscaSequencial(vet, 8, 100);
    if (pos != -1)
        printf("A posicao da chave 100 no vetor é:
        ↪  %d\n", pos);
    else
        printf("A chave 100 não está no vetor!\n");

```

Busca binária

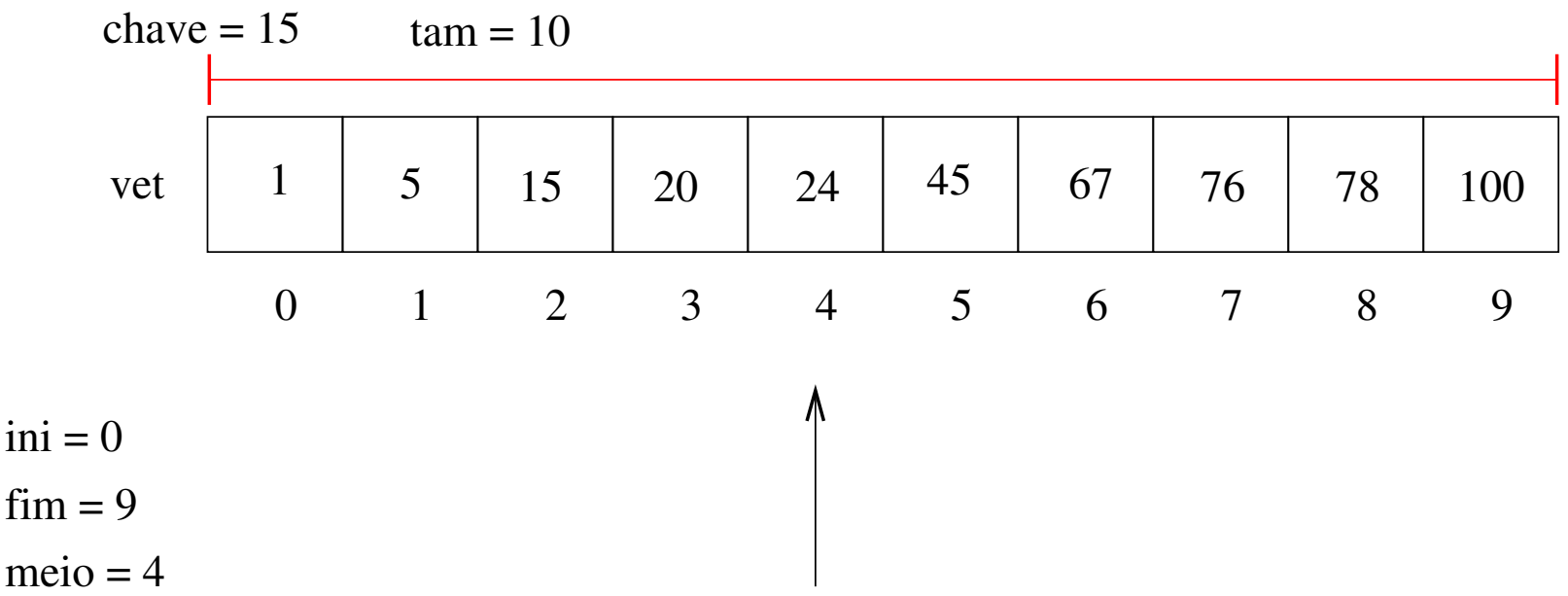
- A busca binária é um algoritmo um pouco mais sofisticado.
- É mais eficiente (tempo), mas requer que o vetor esteja ordenado pelos valores da chave de busca.
- A ideia do algoritmo é a seguinte (assuma que o vetor está ordenado):
  - Verifique se a chave de busca é igual ao valor da posição do meio 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

Pseudocódigo:

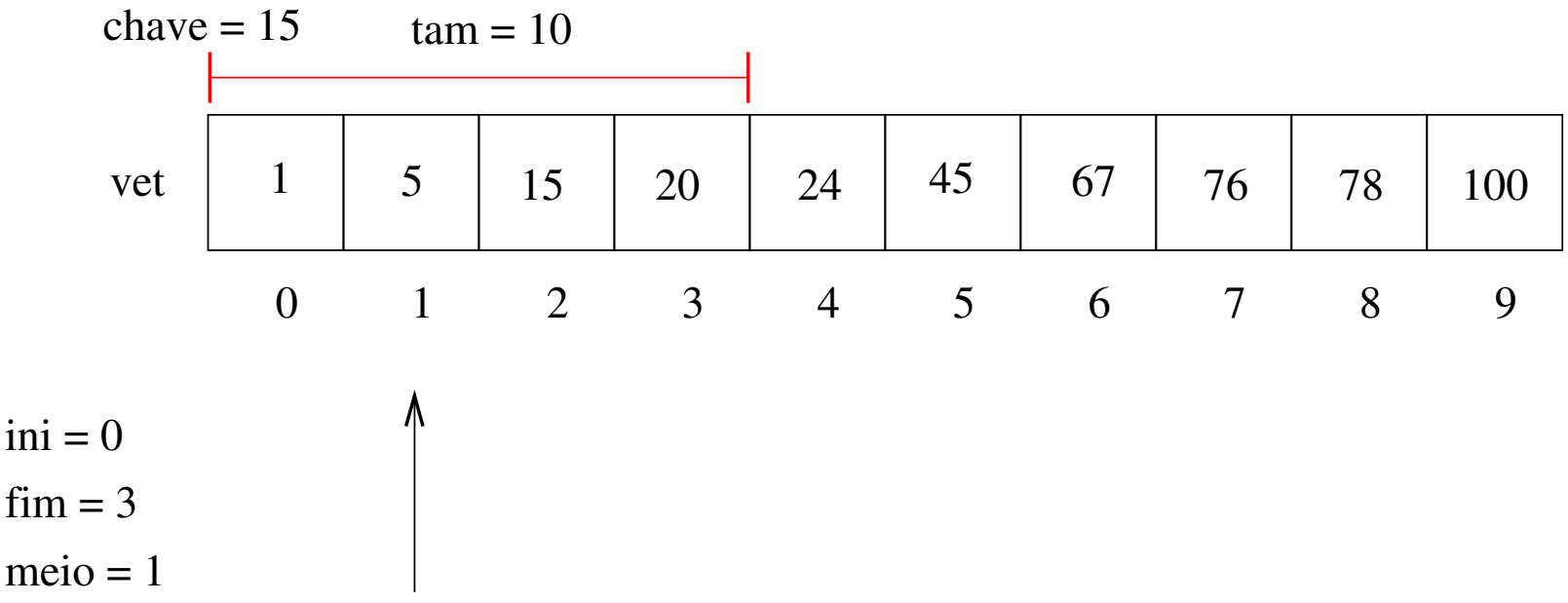
```
1  /* vetor começa em ini e termina em fim */
2  ini = 0
3  fim = tam-1
4
5  Repita enquanto tamanho do vetor considerado for >= 1:
6      meio = (ini + fim)/2
7      Se vet[meio] == chave Então
8          devolva meio
9      Se vet[meio] > chave Então
10         fim = meio - 1
11     Se vet[meio] < chave Então
12         ini = meio + 1
```

Busca binária



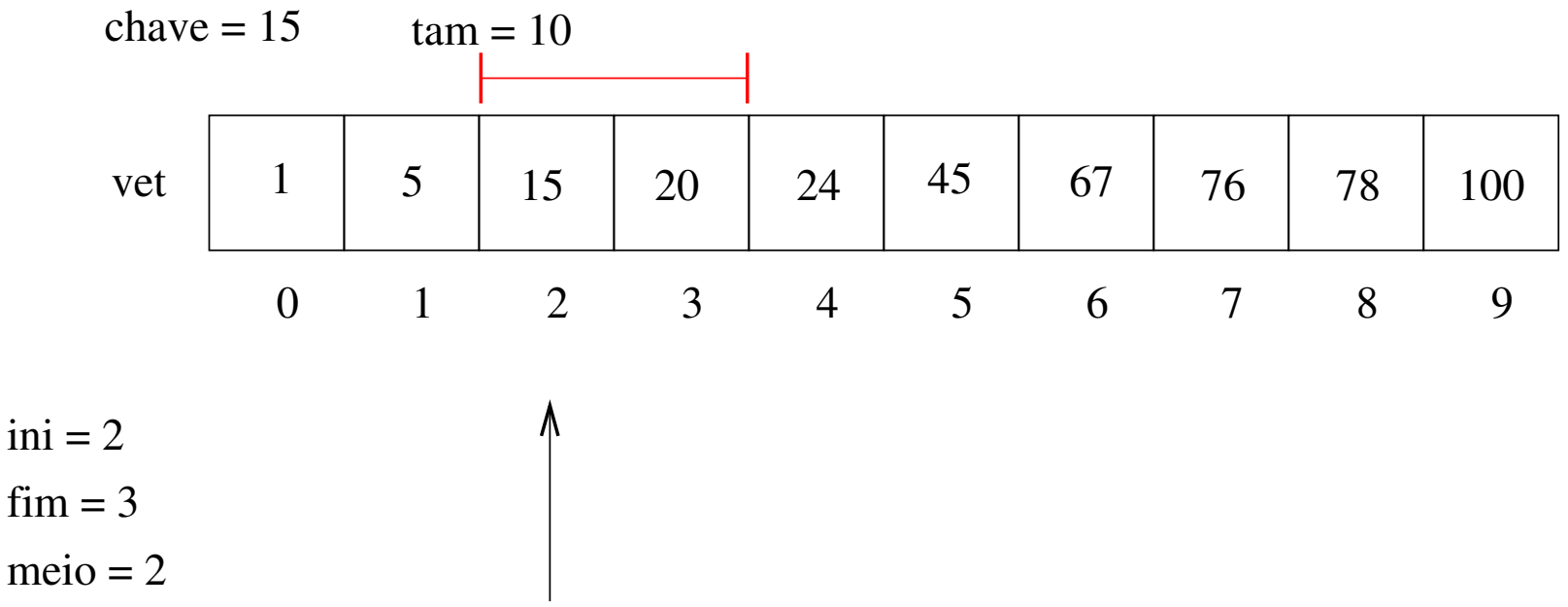
Como o valor da posição do meio é maior que a chave, atualizamos `fim` do vetor considerado.

Busca binária



Como o valor da posição do meio é menor que a chave, atualizamos `ini` do vetor considerado.

Busca binária



Finalmente encontramos a chave e podemos devolver sua posição 2.

Busca binária

Código completo:

```
1  int buscaBinaria(int vet[], int tam, int chave) {
2      int ini = 0, fim = tam-1, meio;
3
4      while (ini <= fim) { /* enquanto o vetor tiver
       ↪  pelo menos 1 elemento */
5          meio = (ini+fim)/2;
6          if (vet[meio] == chave)
7              return meio;
8          else if (vet[meio] > chave)
9              fim = meio - 1;
10         else
11             ini = meio + 1;
12     }
13     return -1;
14 }
```

Busca binária

Exemplo de uso:

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

    /* antes de usar a busca devemos ordenar o vetor */
    selectionSort(vet, 10);
    printf("Vetor Ordenado: (%d", vet[0]);
    for (i = 1; i < 10; i++)
        printf(", %d", vet[i]);
    printf(")\n");

    pos = buscaBinaria(vet, 10, 15);
    if (pos != -1)
        printf("A posicao da chave 15 no vetor é: %d\n", pos);
    else
        printf("A chave 15 não está no vetor!\n");

    return 0;
}
```

Exercício

Refaça as funções de busca sequencial e busca binária assumindo que o vetor possui chaves que podem aparecer repetidas.

Neste caso, você deve retornar em um outro vetor todas as posições onde a chave foi encontrada.

Protótipo: `int busca(int vet[], int tam, int chave, int *posicoes)`

Você deve devolver em `posicoes[]` as posições de `vet` que possuem a `chave`, e o retorno da função é o número de ocorrências da chave.

OBS: O vetor `posições` deve ser alocado dentro da função busca e ter espaço suficiente para guardar todas as possíveis ocorrências.

Informações extras: questões sobre eficiência

Eficiência dos algoritmos

Podemos medir a eficiência de qualquer algoritmo analisando a quantidade de recursos (tempo, memória, banda de rede, 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.

Para isso, analisamos quantas instruções um algoritmo usa para resolver o problema.

Podemos fazer uma análise simplificada dos algoritmos de busca analisando a quantidade de vezes que os algoritmos **acessam** uma posição do vetor.

Eficiência dos algoritmos

No caso da busca sequencial existem três possibilidades:

- 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 acessaremos todas as `tam` posições do vetor.
- É possível mostrar que se uma chave qualquer pode ser requisitada com a mesma probabilidade, então o número de acessos será `(tam+1)/2` em média.

Eficiência dos algoritmos

No caso da busca binária temos também três possibilidades:

- Na melhor das hipóteses a chave de busca estará na posição do meio. Portanto teremos um único acesso.
- Na pior das hipóteses, teremos $(\log_2 \texttt{tam})$ acessos.
  - Para ver isso note que a cada verificação de uma posição do vetor, o tamanho do vetor considerado é dividido pela metade. No pior caso repetimos a busca até o vetor considerado ter tamanho 1. Se você pensar um pouco, o número de acessos x pode ser encontrado resolvendo-se a equação: $\frac{\texttt{tam}}{2^x} = 1$, cuja solução é $x = (\log_2 \texttt{tam})$.
- É possível mostrar que se uma chave qualquer pode ser requisitada com a mesma probabilidade, então o número de acessos será $(\log_2 \texttt{tam}) - 1$ em média.

Eficiência dos algoritmos

Para se ter uma ideia da diferença de eficiência dos dois algoritmos, considere que temos um cadastro com 10^6 (um milhão) de itens.

- Com a busca sequencial, a procura de um item qualquer gastará na média $(10^6 + 1)/2 \approx 500000$ acessos.
- Com a busca binária teremos $(\log_2 10^6) - 1 \approx 20$ acessos.

Eficiência dos algoritmos

Mas uma ressalva 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 intercalada com estas operações, então a busca binária pode não ser a melhor opção, já que você precisará ficar mantendo o vetor ordenado.
- Caso o número de buscas feitas seja muito maior, quando comparado com outras operações, então a busca binária é uma boa opção.