Linguagem C: Ordenação

Luis Martí
Instituto de Computação
Universidade Federal Fluminense
lmarti@ic.uff.br - http://lmarti.com

Tópicos Principais

- Introdução
- Algoritmos de ordenação
  - Ordenação por Seleção
  - Ordenação por Inserção (*insertion sort*)
  - Ordenação Bolha (*bubble sort*)
  - Ordenação por Intercalação (*merge sort*)
  - Ordenação Rápida (*quick sort*)
- Ordenação de vetores complexos
  - Algoritmos genéricos
  - Função de comparação
  - Função *qsort()*

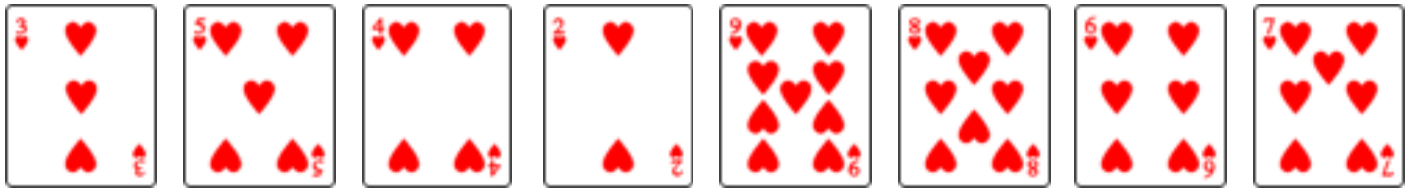
Introdução

- Ordenação de vetores:
  - entrada: vetor com os elementos a serem ordenados
  - saída: mesmo vetor com elementos na ordem especificada
  - ordenação:
    - pode ser aplicada a qualquer dado com ordem bem definida
    - vetores com dados complexos (structs)
      - chave da ordenação escolhida entre os campos
      - elemento do vetor contém apenas um ponteiro para os dados
      - troca da ordem entre dois elementos = troca de ponteiros

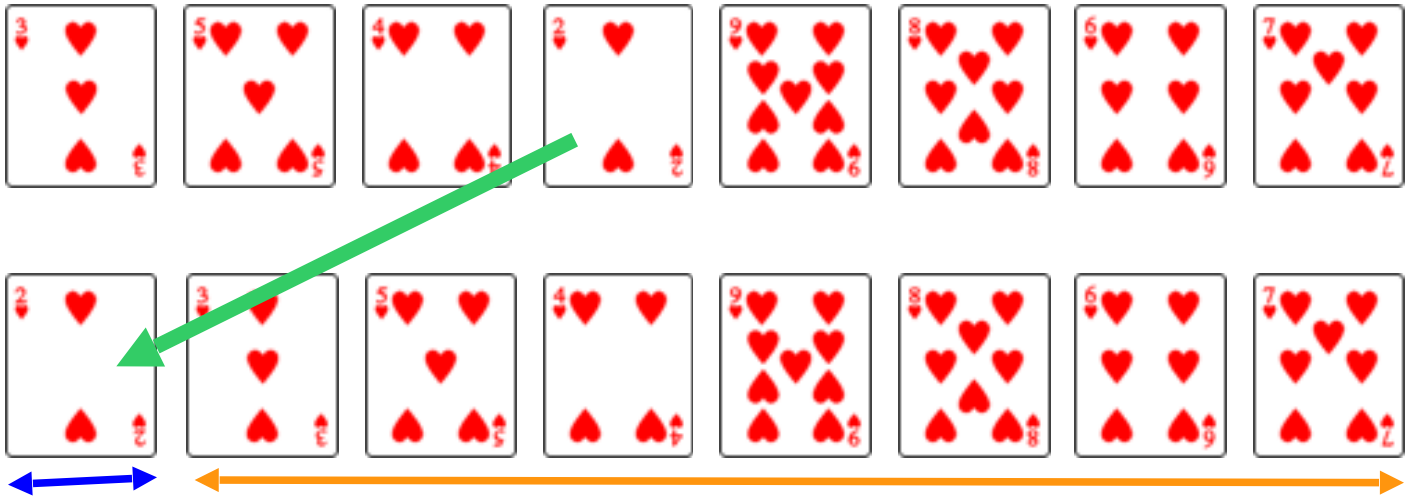
Ordenação por Seleção

- Ordenação por Seleção:
  - processo básico:
    - passar o menor valor do vetor para a primeira posição, depois o de segundo menor valor para a segunda posição, e assim é feito sucessivamente com os (n-1) elementos restantes, até os últimos dois elementos.

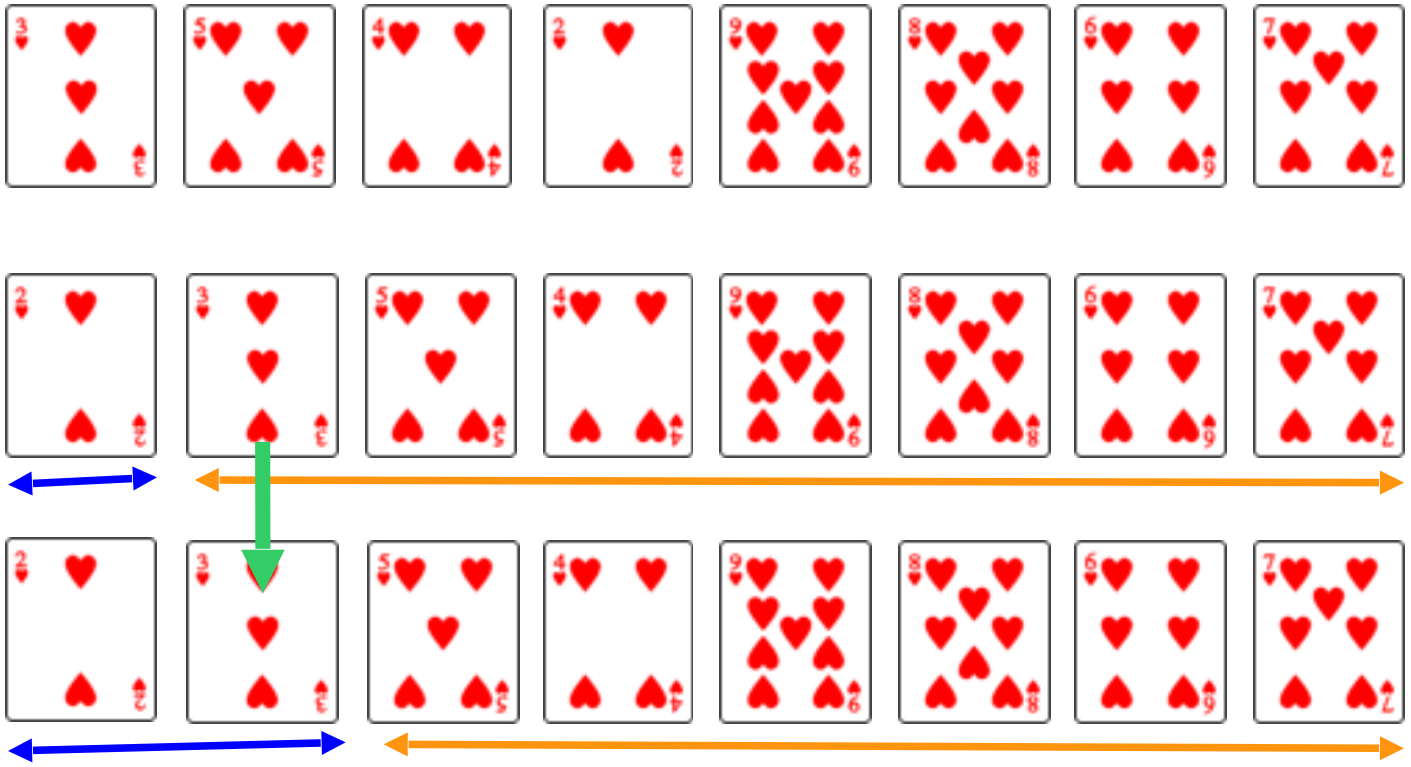
Ordenação por Seleção



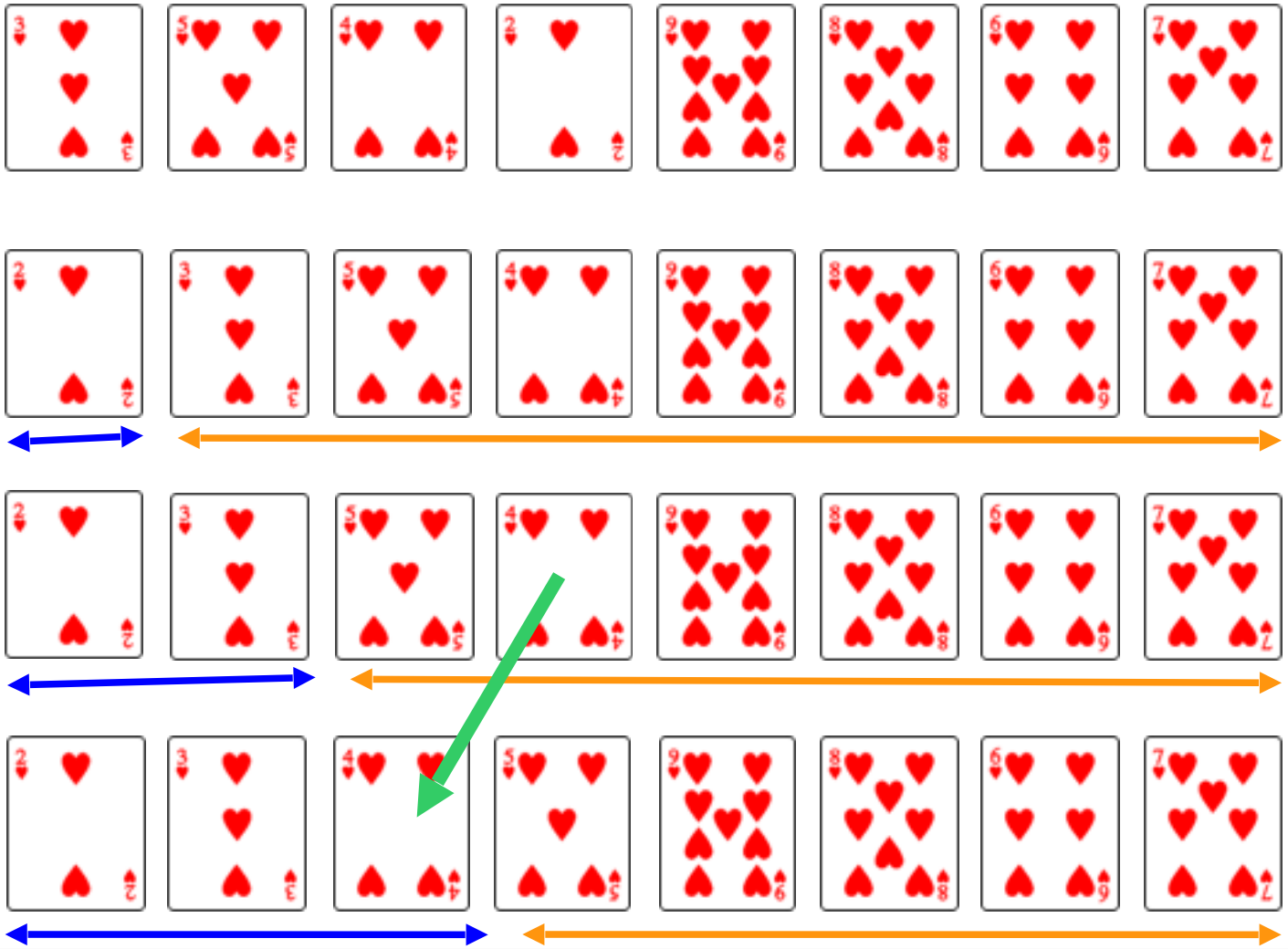
Ordenação por Seleção



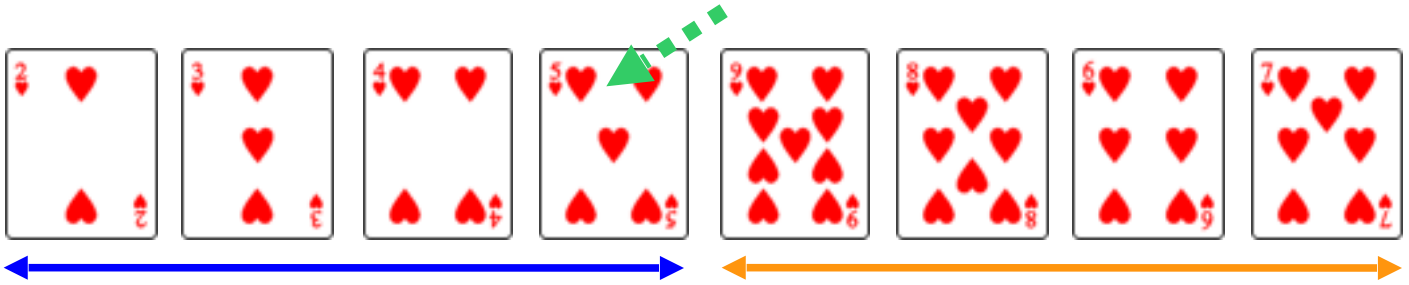
Ordenação por Seleção



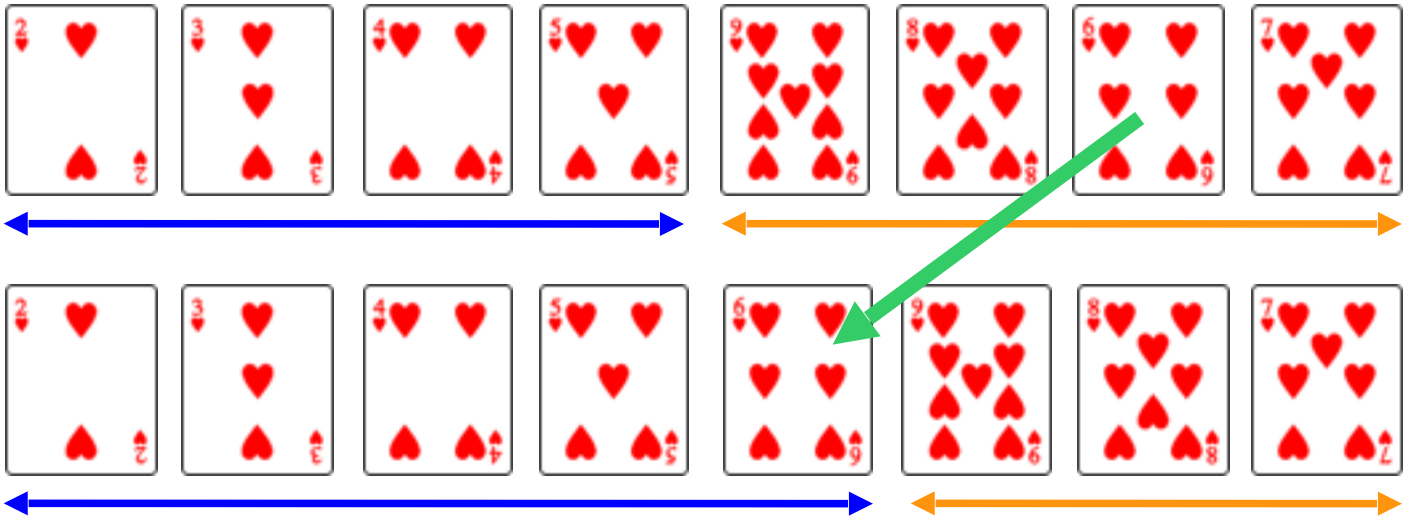
Ordenação por Seleção



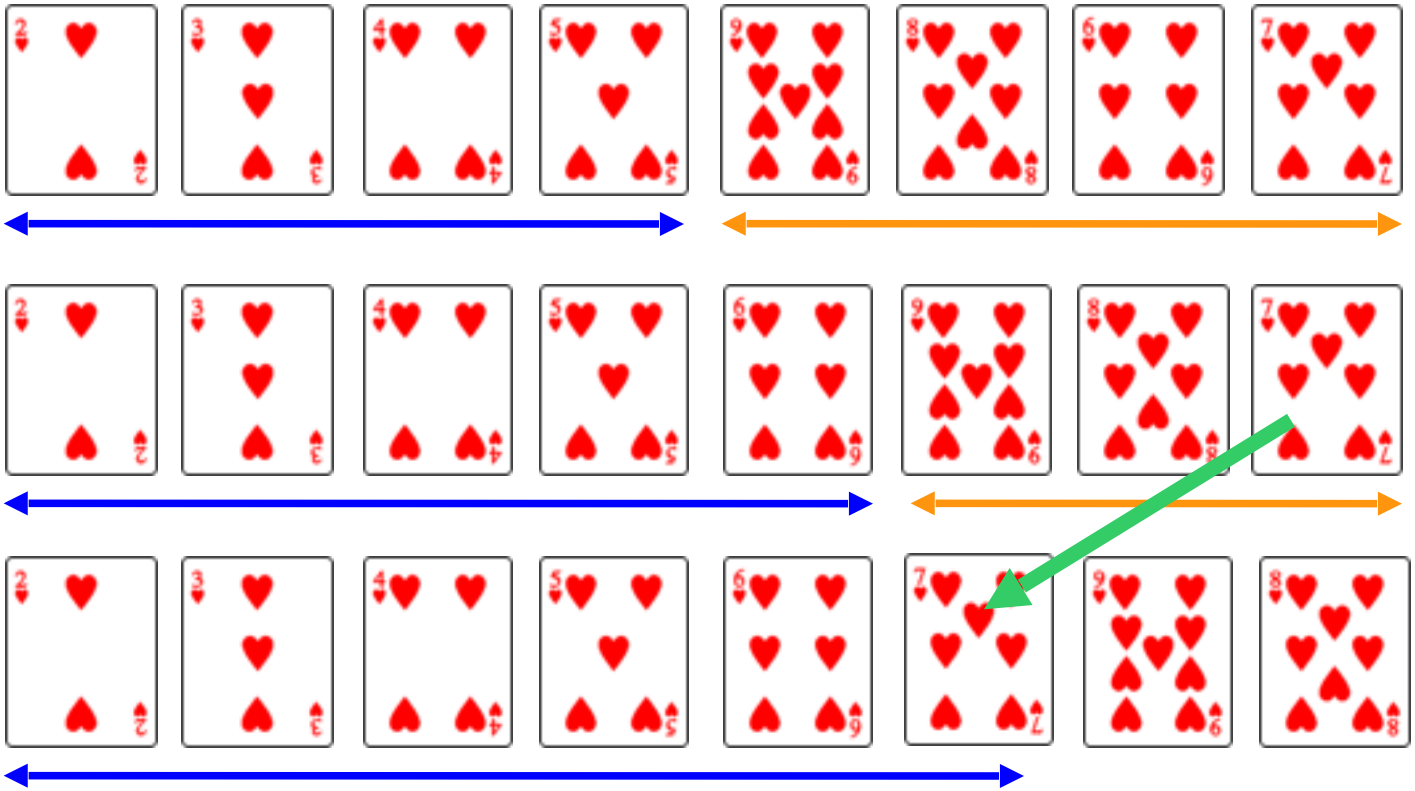
Ordenação por Seleção



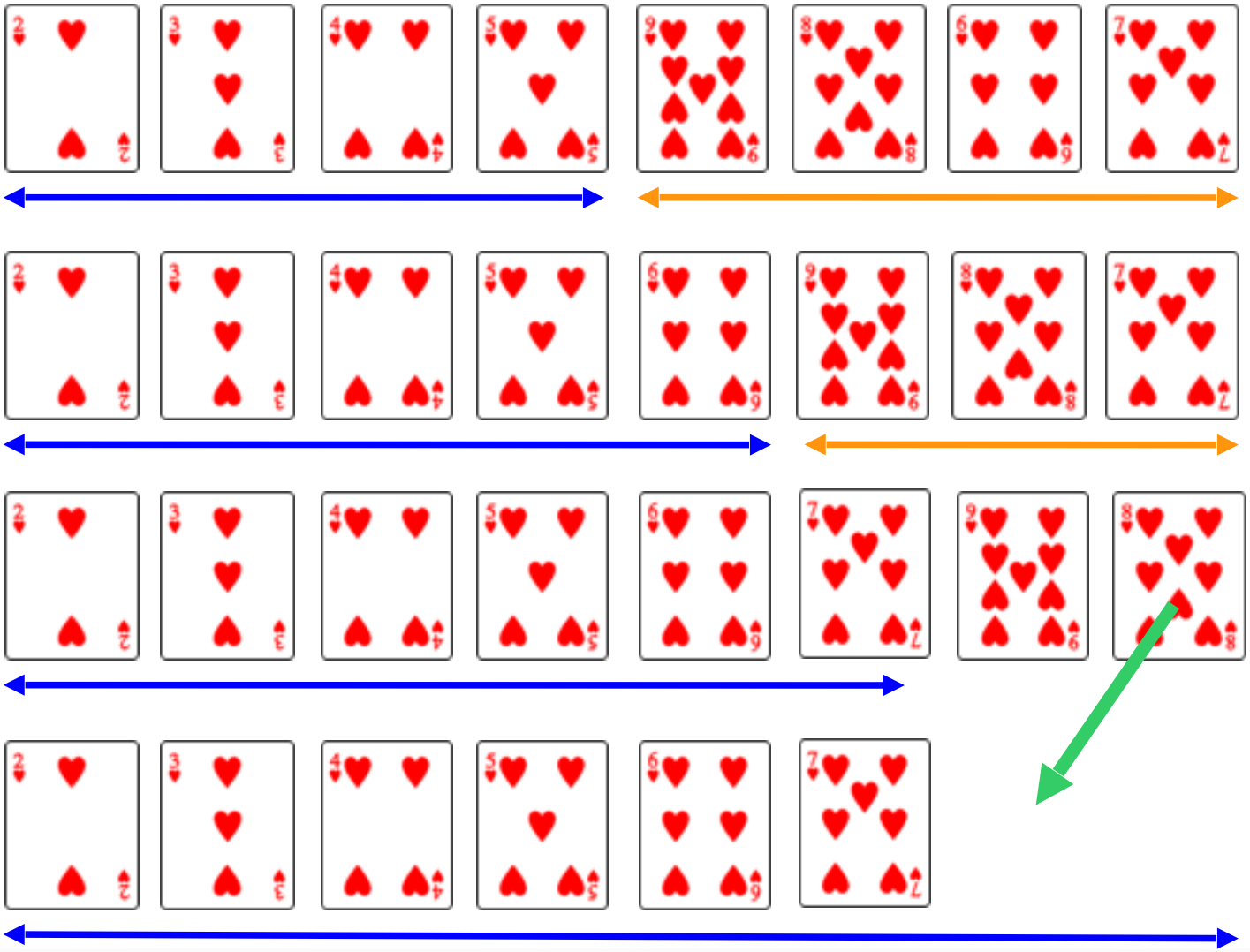
Ordenação por Seleção



Ordenação por Seleção



Ordenação por Seleção



Ordenação por Seleção

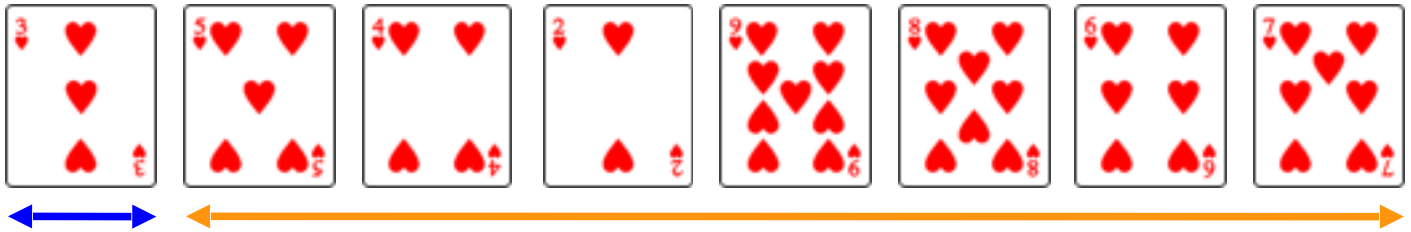
- Implementação:

```
void selecao(int *v, int n) {
  int i, j, min, temp;
  for (i = 0; i < (n-1); i++) {
    min = i;
    for (j = (i+1); j < n; j++) {
      if(v[j] < v[min]) {
        min = j;
      }
    }
    if (i != min) {
      temp = v[i];
      v[i] = v[min];
      v[min] = temp;
    }
  }
}
```

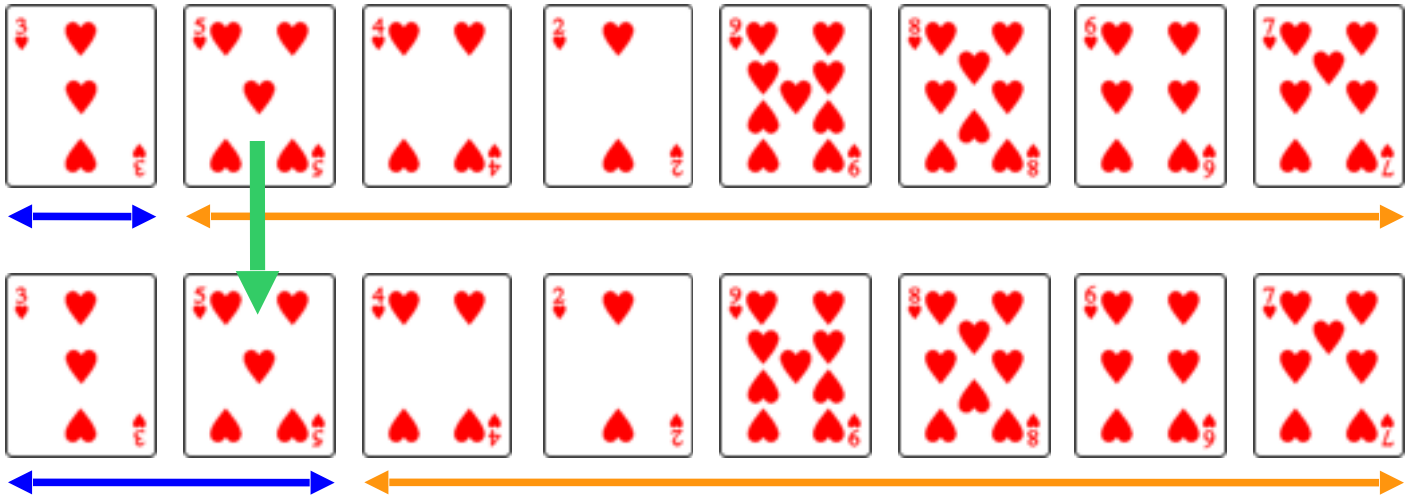
Ordenação por Inserção

- Ordenação por Inserção:
  - processo básico:
    - percorrer o vetor da esquerda para a direita e à medida em que avançar tomar o primeiro elemento à direita e inseri-lo na posição correta no subvetor ordenado (entre os elementos mais à esquerda).

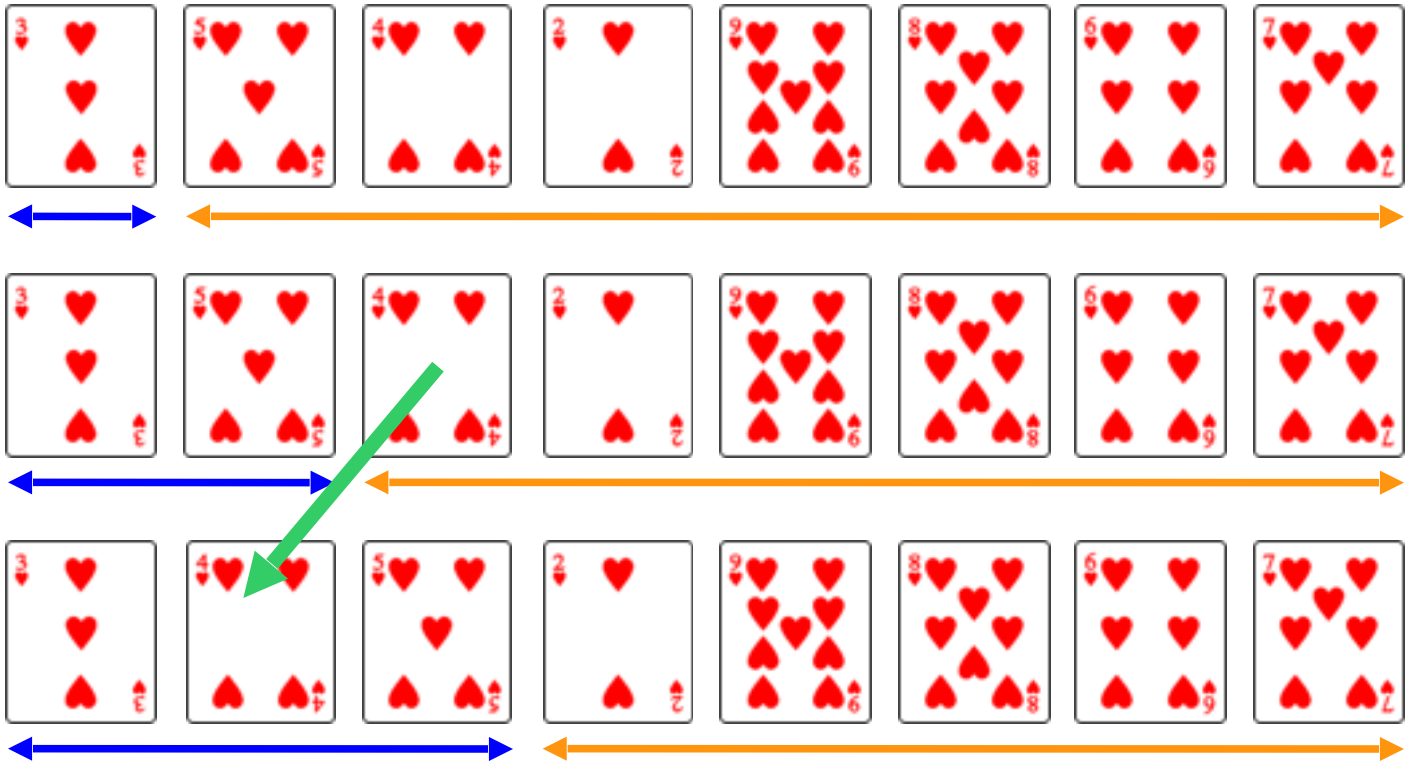
Ordenação por Inserção



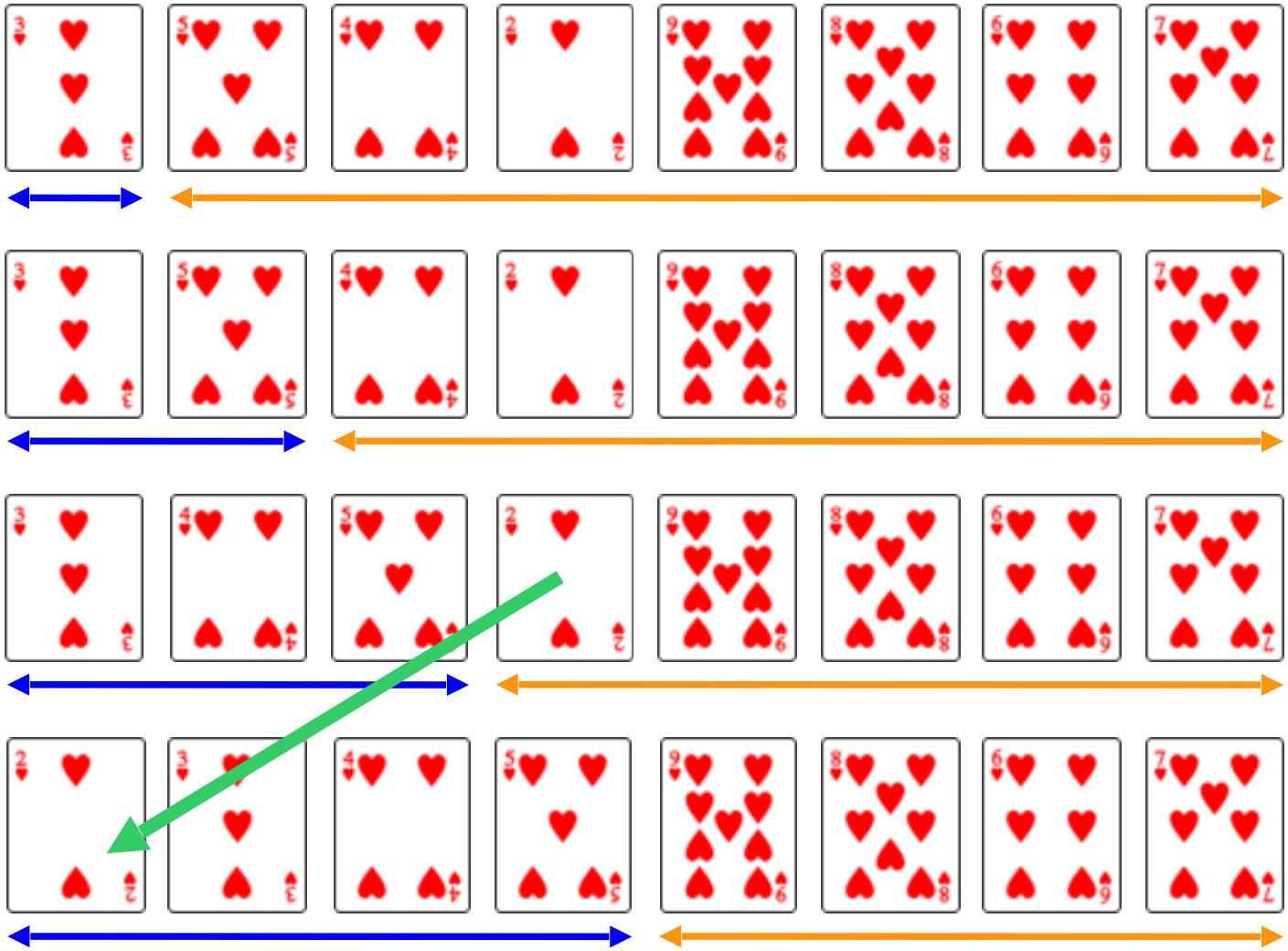
Ordenação por Inserção



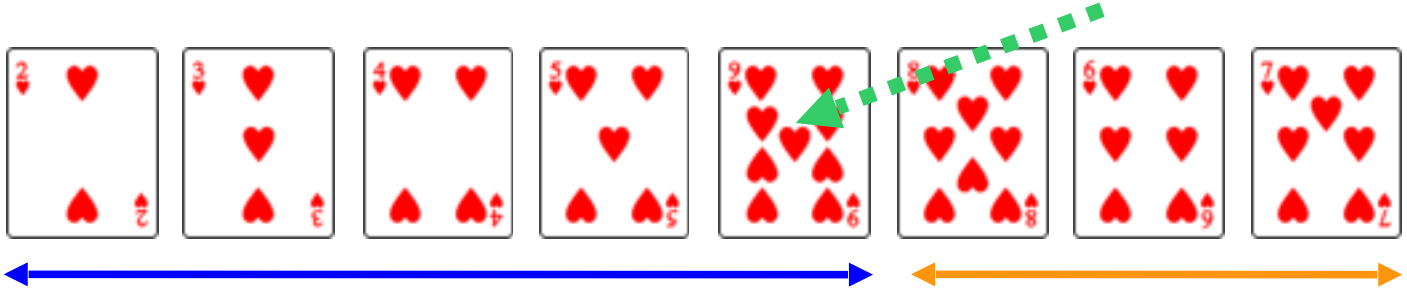
Ordenação por Inserção



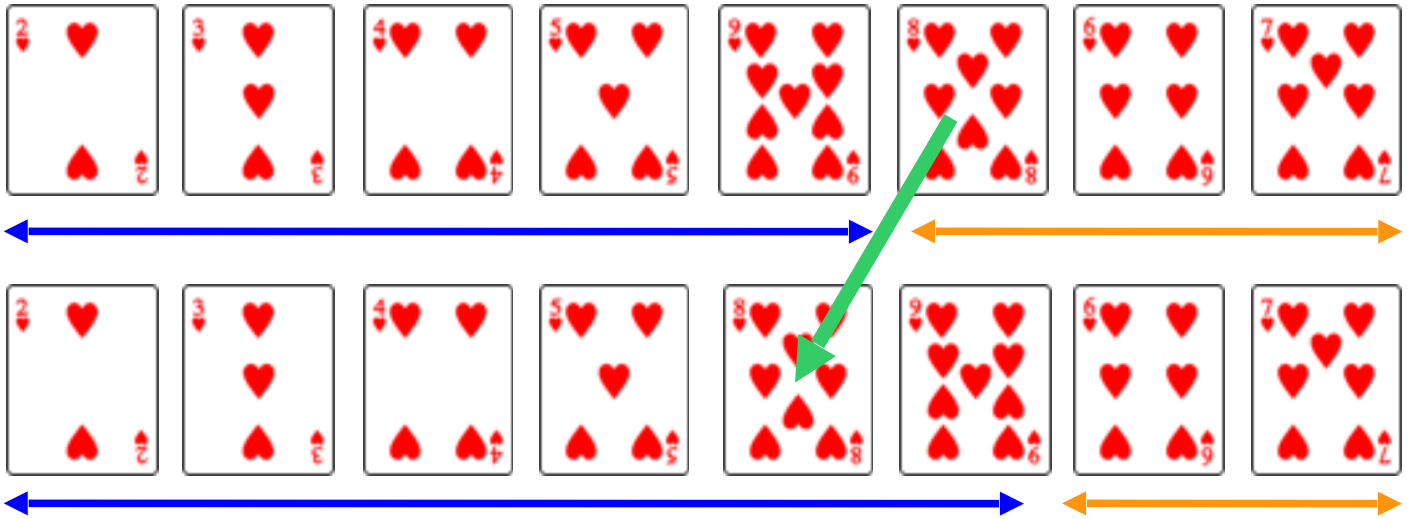
Ordenação por Inserção



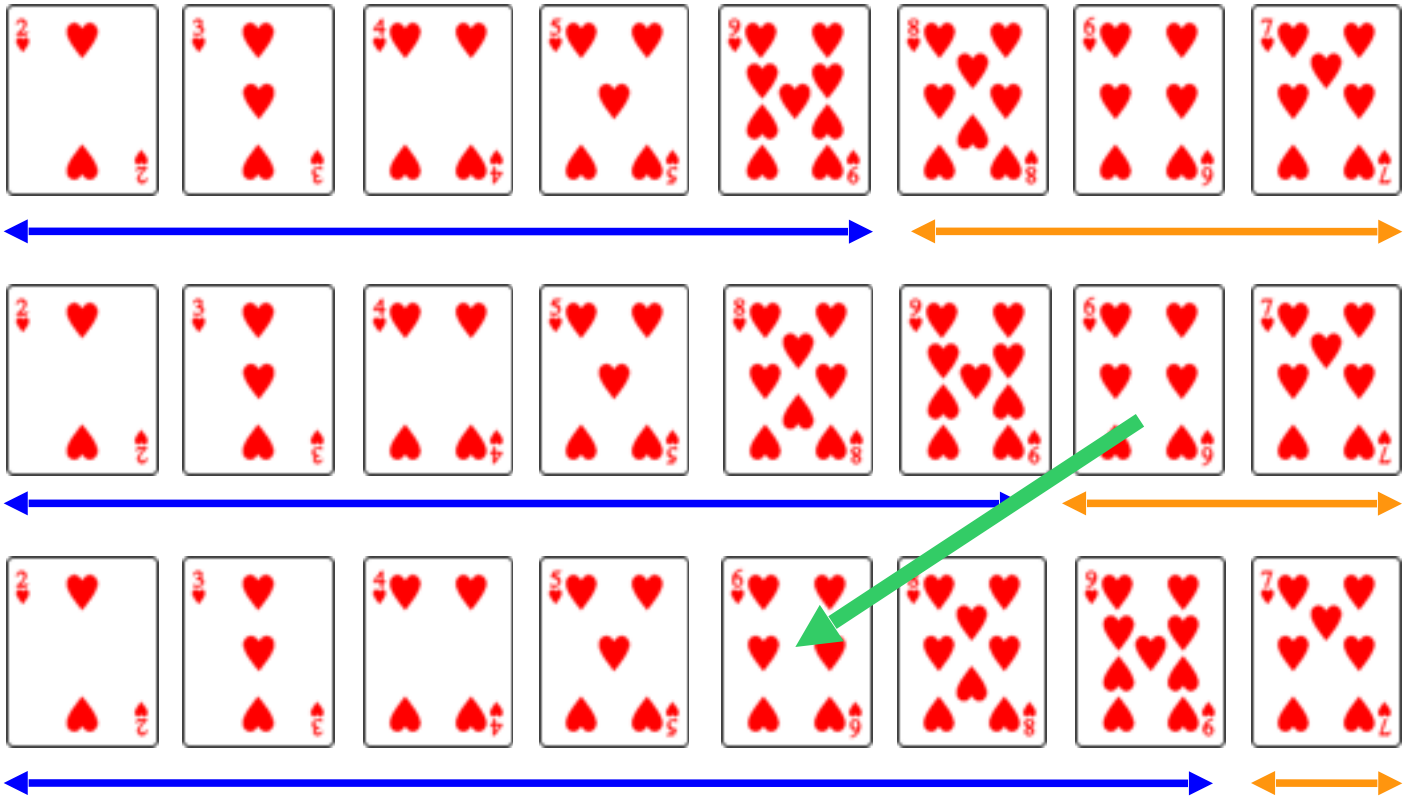
Ordenação por Inserção



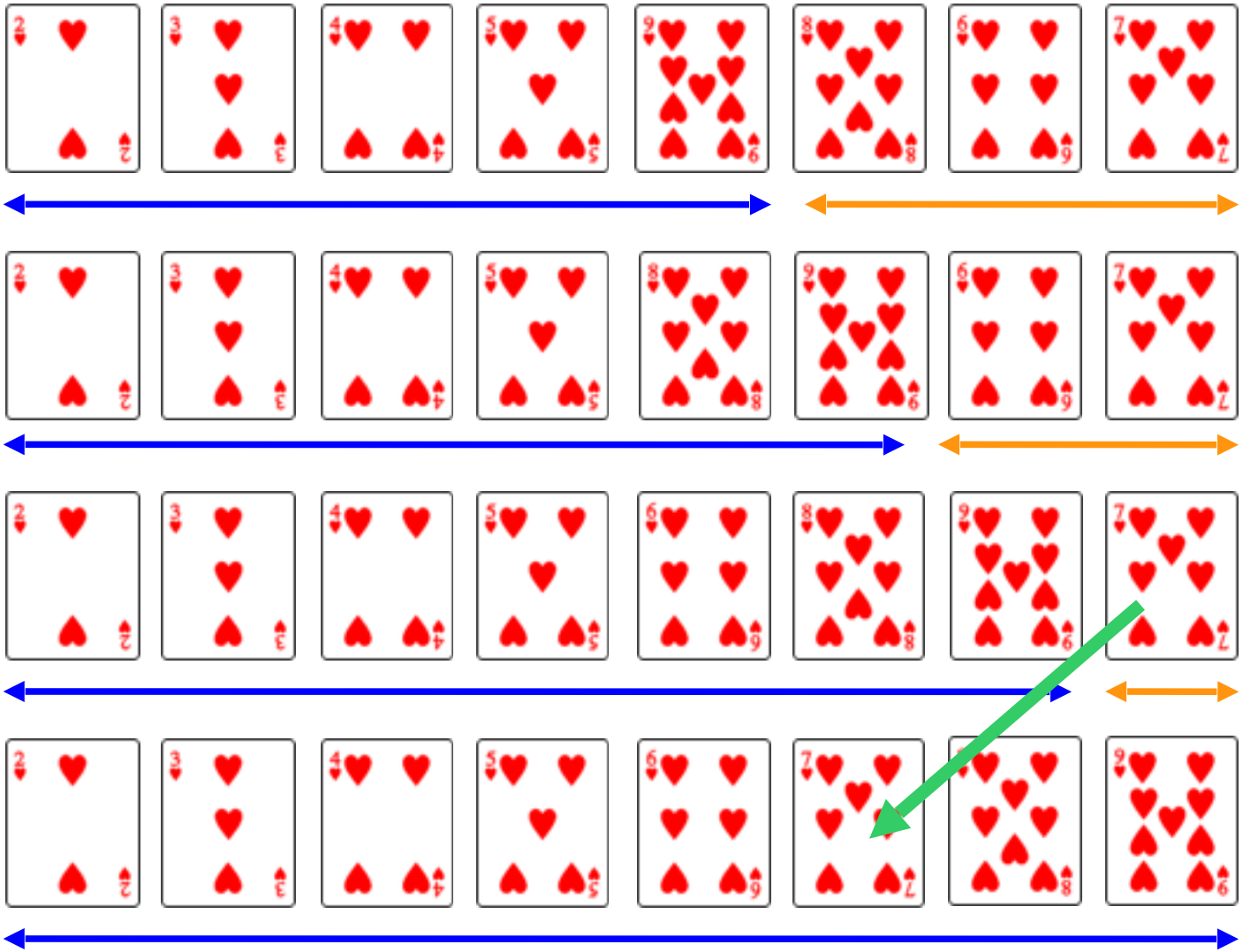
Ordenação por Inserção



Ordenação por Inserção



Ordenação por Inserção



Ordenação por Inserção

- Implementação:

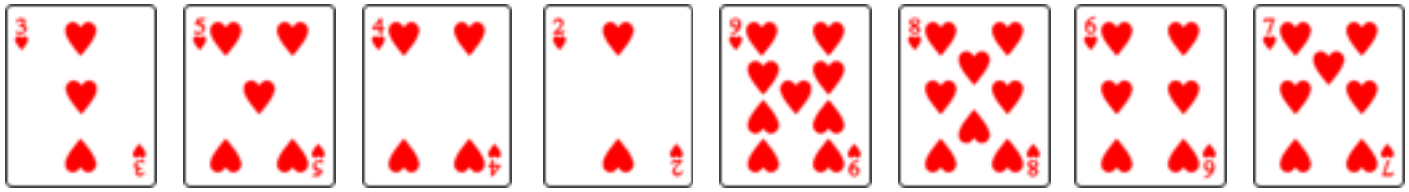
```
void insercao(int *v, int n)
{
    int i, j, chave;

    for(j=1; j<n; j++)
    {
            chave = v[j];
            i = j-1;
            while(i >= 0 && v[i]>chave)
            {
                v[i+1] = v[i];
                i--;
            }
            v[i+1] = chave;
        }
}
```

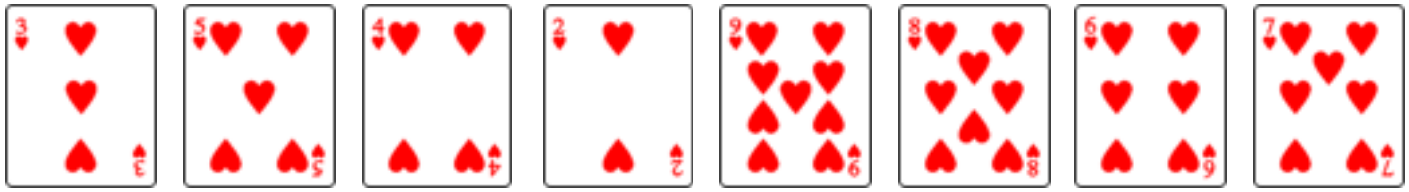
Ordenação Bolha

- Ordenação bolha:
  - processo básico:
    - quando dois elementos estão fora de ordem, troque-os de posição até que o i-ésimo elemento de maior valor do vetor seja levado para as posições finais do vetor
  - continue o processo até que todo o vetor esteja ordenado

Ordenação Bolha

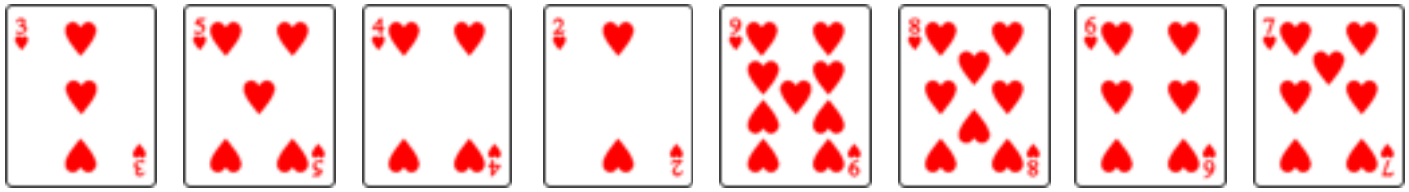


Ordenação Bolha



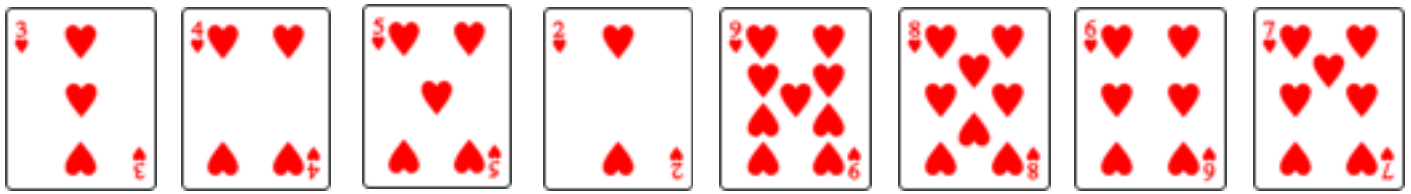
Não troca!

Ordenação Bolha



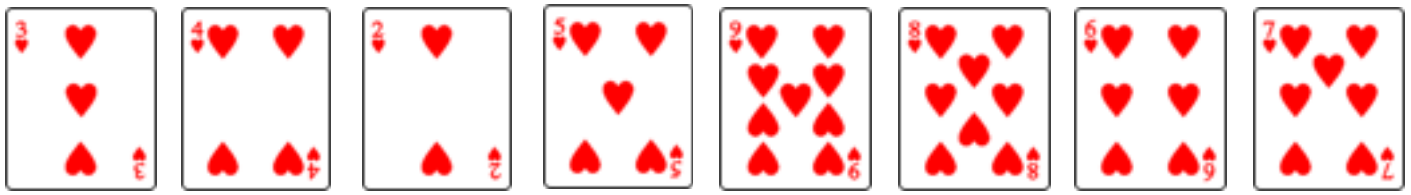
Troca!!!

Ordenação Bolha



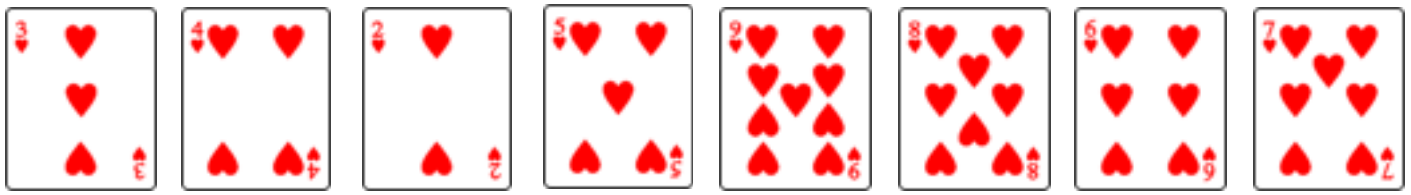
Troca!!!

Ordenação Bolha



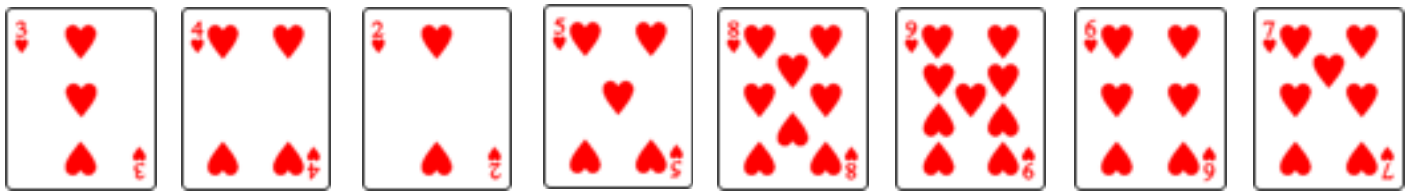
Não troca!!!

Ordenação Bolha



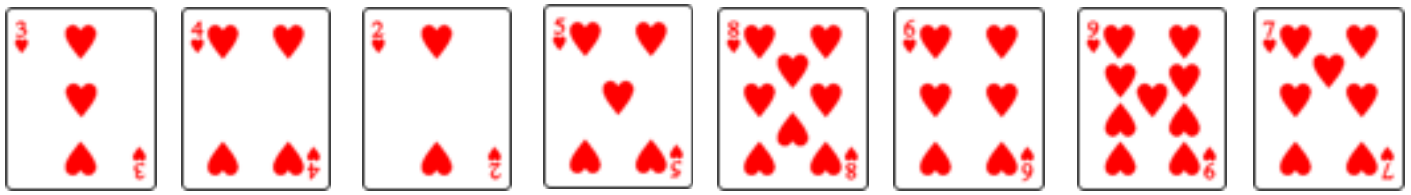
Troca!!!

Ordenação Bolha



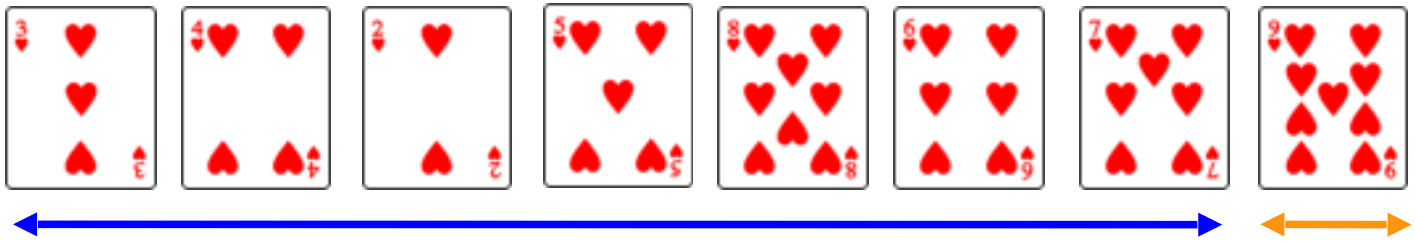
Troca!!!

Ordenação Bolha



Troca!!!

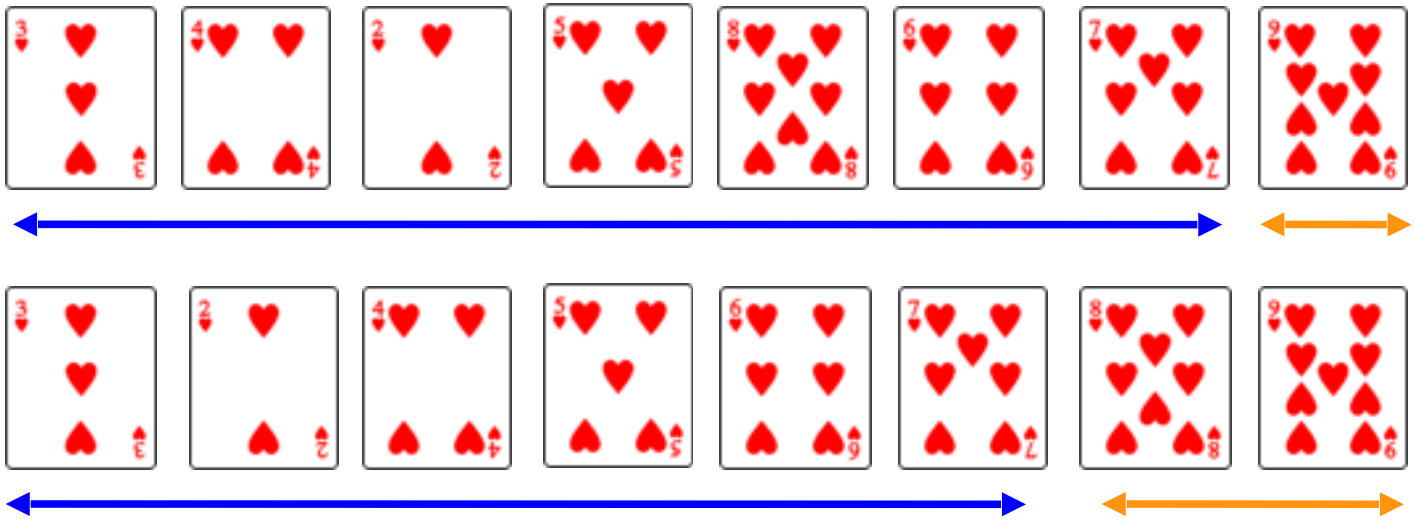
Ordenação Bolha



Fim da 1ª passada: o maior elemento já está na sua posição correta!

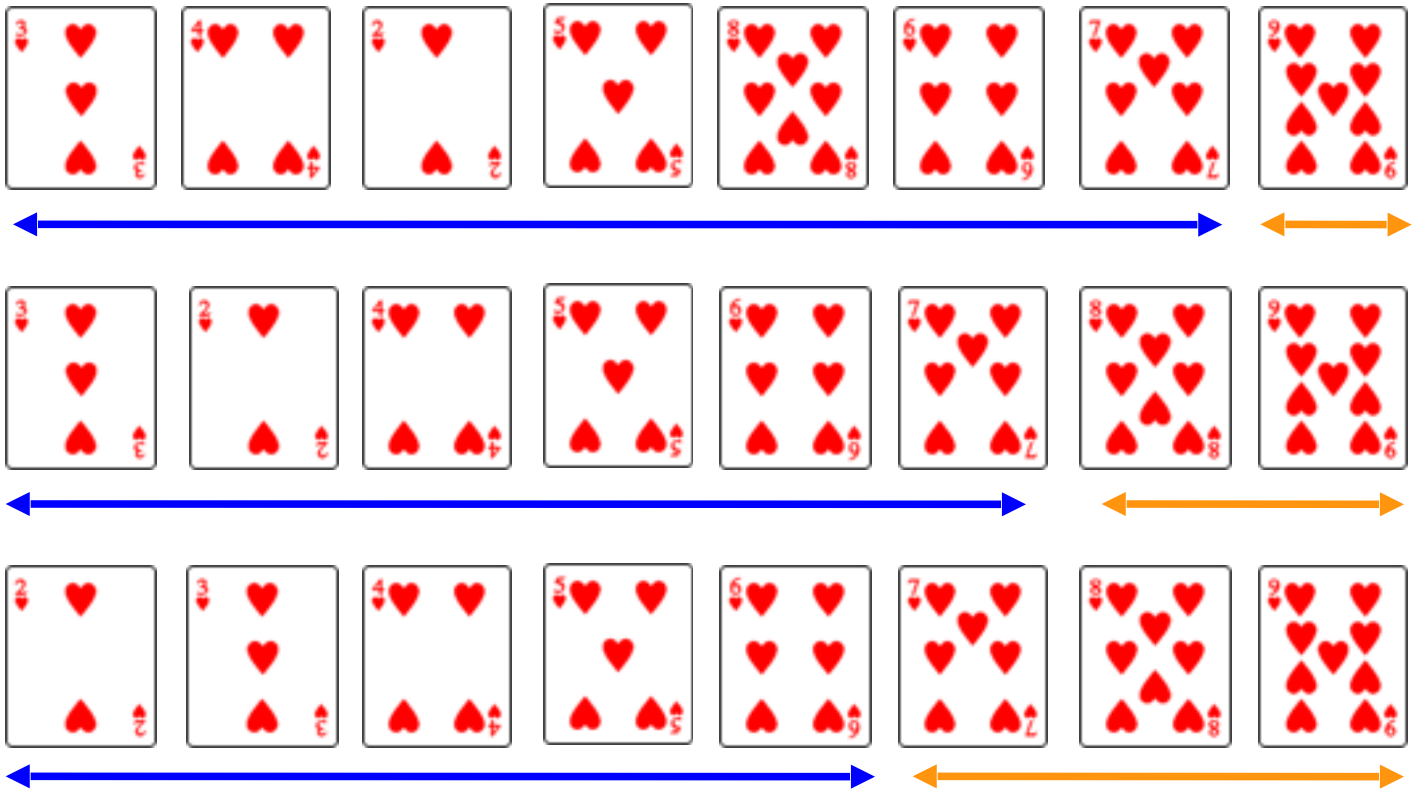
Serão necessárias no máximo *n-1* passadas

Ordenação Bolha



Fim da 2ª passada!

Ordenação Bolha



Fim da 3ª passada: o vetor já está ordenado!!!

Ordenação Bolha

Outro Exemplo:

```
25 48 37 12 57 86 33 92
```

```
25 48 37 12 57 86 33 92    25x48
25 48 37 12 57 86 33 92    48x37 troca
25 37 48 12 57 86 33 92    48x12 troca
25 37 12 48 57 86 33 92    48x57
25 37 12 48 57 86 33 92    57x86
25 37 12 48 57 86 33 92    86x33 troca
25 37 12 48 57 33 86 92    86x92
25 37 12 48 57 33 86 92    final da primeira passada
```

o maior elemento, 92, já está na sua posição final

Ordenação Bolha

```
25 37 12 48 57 33 86 92    25x37
25 37 12 48 57 33 86 92    37x12 troca
25 12 37 48 57 33 86 92    37x48
25 12 37 48 57 33 86 92    48x57
25 12 37 48 57 33 86 92    57x33 troca
25 12 37 48 33 57 86 92    57x86
25 12 37 48 33 57 86 92    final da segunda passada
```

o segundo maior elemento, 86, já está na sua posição final

Ordenação Bolha

```
25 12 37 48 33 57 86 92    25x12 troca
12 25 37 48 33 57 86 92    25x37
12 25 37 48 33 57 86 92    37x48
12 25 37 48 33 57 86 92    48x33 troca
12 25 37 33 48 57 86 92    48x57
12 25 37 33 48 57 86 92    final da terceira passada
```

Idem para `57`.

```
12 25 37 33 48 57 86 92    12x25
12 25 37 33 48 57 86 92    25x37
12 25 37 33 48 57 86 92    37x33 troca
12 25 33 37 48 57 86 92    37x48
12 25 33 37 48 57 86 92    final da quarta passada
```

Idem para `48`.

```
12 25 33 37 48 57 86 92    12x25
12 25 33 37 48 57 86 92    25x33
12 25 33 37 48 57 86 92    33x37
12 25 33 37 48 57 86 92    final da quinta passada
```

Idem para `37`.

```
12 25 33 37 48 57 86 92    12x25
12 25 33 37 48 57 86 92    25x33
12 25 33 37 48 57 86 92    final da sexta passada
```

Idem para `33`.

```
12 25 33 37 48 57 86 92    12x25
12 25 33 37 48 57 86 92    final da sétima passada
```

Idem para `25` e, conseqüentemente, `12`.

```
12 25 33 37 48 57 86 92    final da ordenação
```

Ordenação Bolha

- Implementação Iterativa(I):

```
/* Ordenação bolha */
void bolha (int n, int* v)
{
  int fim,i, temp;
  for (fim=n-1; fim>0; fim--)
    for (i=0; i<fim; i++)
       if (v[i]>v[i+1]) {
          temp = v[i]; /* troca */
          v[i] = v[i+1];
          v[i+1] = temp;
       }
}
```

Ordenação Bolha

- Implementação Iterativa (II):

```
/* Ordenação bolha (2a. versão) */
void bolha (int n, int* v)
{  int i, fim, temp, troca;
   for (fim=n-1; fim>0; fim--) {
        troca = 0;
        for (i=0; i<fim; i++)
            if (v[i]>v[i+1]) {
                temp = v[i]; /* troca */
                v[i] = v[i+1];
                v[i+1] = temp;
                troca = 1;
        }
        if (troca == 0) return; /* não houve troca */
   }
}
```

pára quando há

uma passagem inteira

sem trocas

Ordenação Bolha

- Esforço computacional:
  - esforço computacional ≅ número de comparações
    ≅ número máximo de trocas
    - primeira passada: n-1 comparações
    - segunda passada: n-2 comparações
    - terceira passada: n-3 comparações
    - ...
  - tempo total gasto pelo algoritmo:
    - T proporcional a: $(n-1) + (n-2) + \ldots + 2 + 1 = n(n-1+1) / 2 = \mathbf{n^2 / 2}$
    - algoritmo de ordem quadrática: ***O(n²)***

Ordenação Bolha

- **Implementação recursiva:**

```
/* Ordenação bolha recursiva */
void bolha_rec (int n, int* v)
{
   int i;
   int troca = 0;
   for (i=0; i<n-1; i++)
        if (v[i]>v[i+1]) {
           int temp = v[i];   /* troca */
           v[i] = v[i+1];
           v[i+1] = temp;
           troca = 1;
       }
    if (troca != 0)&&(n>1) /* houve troca e n>1 */
      bolha_rec(n-1,v);
}
```

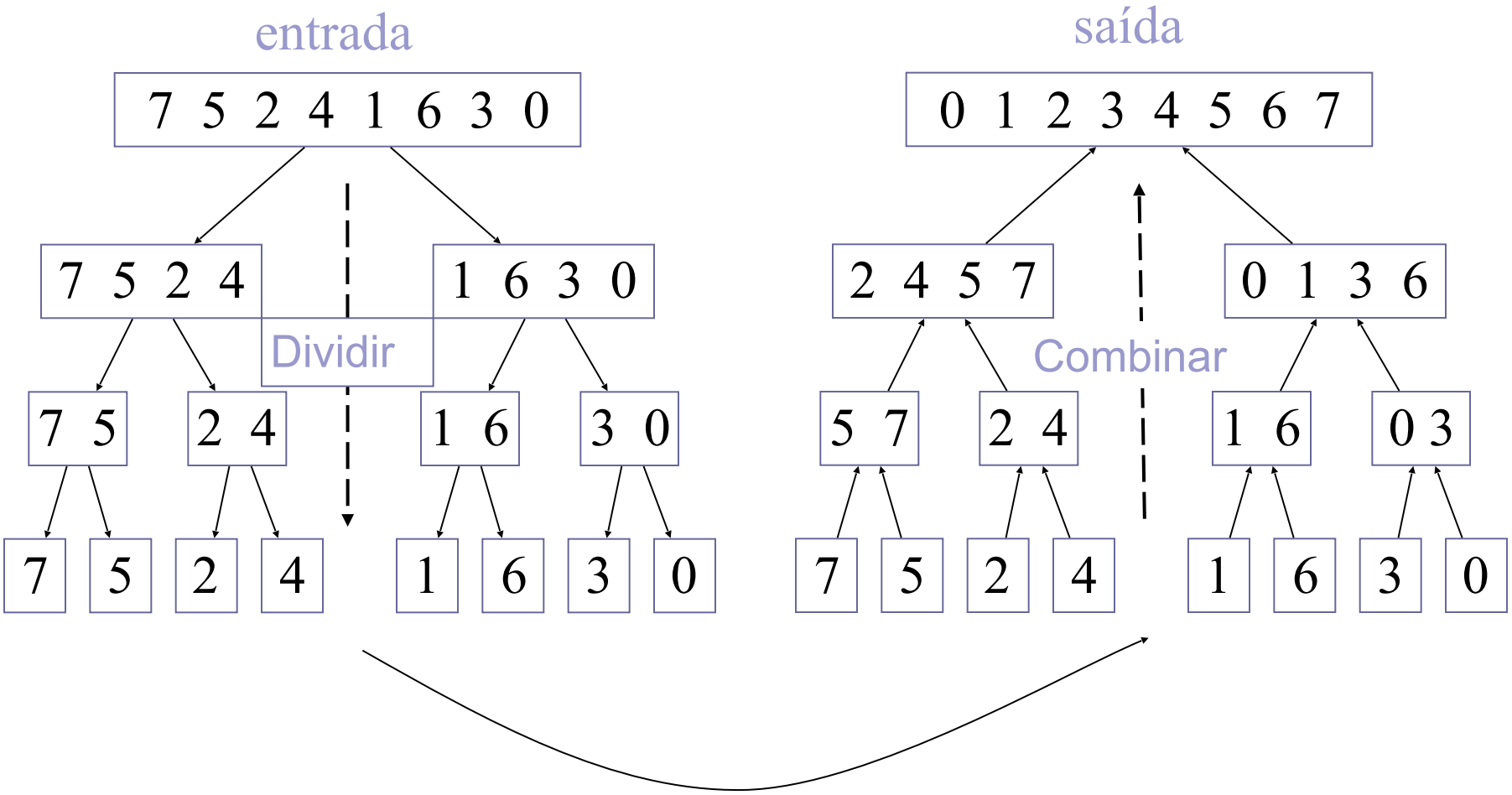
Ordenação por Intercalação

- Se baseia na criação de uma sequência ordenada a partir de duas outras sequências já ordenadas (intercalação)
  - Se a lista é de tamanho 0 ou 1 então ela já está ordenada; Se não:
  - Divida a lista aproximadamente na metade
  - Ordene cada sub-lista recursivamente (chamando o próprio mergesort)
  - Efetue o merge (intercala) nas duas sub-listas ordenadas

Ordenação por Intercalação

- Seja uma lista *A* de *n* elementos:
  - **Dividir** *A* em 2 sub-listas de tamanho ≈ *n*/2
  - **Conquistar**: ordenar cada sub-lista chamando *MergeSort* recursivamente
  - **Combinar** as sub-listas ordenadas formando uma única lista ordenada (*Merge*)
- caso base: lista com um elemento

Ordenação por Intercalação



Ordenação por Intercalação

```
void mergeSort(int *v, int n) {
   int mid;

   if (n > 1) {
      mid = n/2;
      mergeSort(v, mid);
      mergeSort(v + mid, n - mid);
      merge(v, n);
   }
}
```

O(1)

Ordenação por Intercalação

```
void merge(int *v, int n) {
  int mid, i, j, k, *t;
  t = (int*) malloc(n* sizeof(int));
  mid = n / 2;
  i = 0; j = mid; k = 0;
  while (i < mid && j < n) {
    if (v[i] < v[j]) {
      t[k] = v[i];
      ++i;
    }
    else {
      t[k] = v[j];
      ++j;
    }
    ++k;
  }
  if (i == mid)
    while (j < n) {
      t[k] = v[j];
      ++j; ++k;
    }
  else
    while (i < mid) {
      t[k] = v[i];
      ++i; ++k;
    }
  for(i = 0; i < n; i++){
    v[i] = t[i];
  }
  free(t);
}
```

O(n)

Ordenação Rápida

- Ordenação rápida (“*quick sort*”):
  - escolha um elemento arbitrário x, o *pivô*
  - rearrume o vetor de tal forma que x fique na posição correta v[i]
    - x deve ocupar a posição i do vetor sse
      todos os elementos v[0], … v[i−1] são menores que x e
      todos os elementos v[i+1], …, v[n−1] são maiores que x
  - chame recursivamente o algoritmo para ordenar os (sub-)vetores v[0], … v[i−1] e v[i+1], …, v[n−1]
  - continue até que os vetores que devem ser ordenados tenham 0 ou 1 elemento

Ordenação Rápida

- Esforço computacional:
  - melhor caso:
    - pivô representa o valor mediano do conjunto dos elementos do vetor
    - após mover o pivô para sua posição, restarão dois sub-vetores para serem ordenados, ambos com o número de elementos reduzido à metade, em relação ao vetor original
    - algoritmo é *O(n log(n))*
  - pior caso:
    - pivô é o maior elemento e algoritmo recai em ordenação bolha
  - caso médio:
    - algoritmo é *O(n log(n))*

Ordenação Rápida

- Rearrumação do vetor para o pivô de x=v[0]:
  - do início para o final, compare x com v[1], v[2], …
    até encontrar v[a]>x
  - do final para o início, compare x com v[n−1], v[n−2], …
    até encontrar v[b]<=x
  - troque v[a] e v[b]
  - continue para o final a partir de v[a+1] e
    para o início a partir de v[b−1]
  - termine quando os pontos de busca se encontram (b<a)
  - a posição correta de x=v[0] é a posição b e v[0] e v[b] são trocados

25 48 37 12 57 86 33 92
25 48 37 12 57 86 33 92
25 48 37 12 57 86 33 92 v[1]>25, a=1
25 48 37 12 57 86 33 92
25 48 37 12 57 86 33 92
25 48 37 12 57 86 33 92
25 48 37 12 57 86 33 92
25 48 37 12 57 86 33 92 v[3]<25, b=3
25 12 37 48 57 86 33 92 troca v[1] com v[3]
25 12 37 48 57 86 33 92 a=b=2

25 12 37 48 57 86 33 92 a=2, pois todos v[2]>25
25 12 37 48 57 86 33 92 b=1, pois v[1]<25 (índices cruzaram)
12 25 37 48 57 86 33 92

12 25 37 48 57 86 33 92
12 25 37 48 57 86 33 92, a=1, no vetor que começa em 37
12 25 37 48 57 86 33 92, b=4, no vetor que começa em 37
12 25 37 33 57 86 48 92, faz a troca, a=2
12 25 37 33 57 86 48 92, b=1 (b<a)
12 25 33 37 57 86 48 92, troca v[0] por v[b]

Ordenação Rápida

- vetor inteiro de v[0] a v[7]

  (0-7) 25 48 37 12 57 86 33 92
- determine a posição correta de x=v[0]=25
  - de a=1 para o fim: 48>25 (a=1)
  - de b=7 para o início: 25<92, 25<33, 25<86, 25<57 e 12<=25 (b=3)

  (0-7) 25 48 37 12 57 86 33 92

  a↑ *b*↑
- troque v[a]=48 e v[b]=12, incrementando a e decrementando b
- nova configuração do vetor:

  (0-7) 25 12 37 48 57 86 33 92

  a,b↑

Ordenação Rápida

- configuração atual do vetor:

  (0-7) 25 12 37 48 57 86 33 92

  a,b↑

- determine a posição correta de x=v[0]=25
  - de a=2 para o final: 37>25 (a=2)
  - de b=2 para o início: 37>25 e 12<=25 (b=1)
- os índices a e b se cruzaram, com b<a

  (0-7) 25 12 37 48 57 86 33 92

  b↑ *a*↑
  - todos os elementos de 37 (inclusive) para o final são maiores que 25 e
    todos os elementos de 12 (inclusive) para o início são menores que 25 – com exceção de 25
- troque o pivô v[0]=25 com v[b]=12, o último dos valores menores que 25 encontrado
- nova configuração do vetor, com o pivô 25 na posição correta:

  (0-7) 12 25 37 48 57 86 33 92

Ordenação Rápida

- dois vetores menores para ordenar:
  - valores menores que 25:

(0-0) 12

    - vetor já está ordenado pois possui apenas um elemento

  - valores maiores que 25:

(2-7) 37 48 57 86 33 92

    - vetor pode ser ordenado de forma semelhante, com 37 como pivô

```
void rapida (int n, int* v){
   if (n > 1) {
      int x = v[0];
      int a = 1;
      int b = n-1;
      do {
         while (a < n && v[a] <= x) a++;  /* teste a<n */
         while (v[b] > x) b--;            /* nao testa */
         if (a < b) {                  /* faz troca */
            int temp = v[a];
            v[a] = v[b];
            v[b] = temp;
            a++; b--;
         }
      } while (a <= b);
      /* troca pivô */
      v[0] = v[b];
      v[b] = x;

      /* ordena sub-vetores restantes */
      rapida(b,v);
      rapida(n-a,&v[a]);
   }
}
```

Ordenação de vetores complexos

- Exemplo:
  - Ordenação de vetor de ponteiros para o tipo Aluno definido a seguir:

```
/* Dados do aluno */
struct aluno {
    int mat;
    char nome[81];
    double p1, p2, p3;
    char email[41];
};
typedef struct aluno Aluno;
```

Qual o critério de ordenação?

```
/* função de ordenacao do vetor de ponteiros */
void ordena (Aluno **v, int n);
```

Ordenação de vetores complexos

```
/* Usando algoritmo bolha */
void ordena (Aluno **v, int n)
{  int i, j, troca;
   Aluno *temp;
   for (i=n-1; i>0; i--) {
       troca = 0;
       for (j=0; j<i; j++)
            if (compara(v[j],v[j+1])) {
                temp = v[j]; /* troca */
                v[j] = v[j+1];
                v[j+1] = temp;
                troca = 1;
          }
      if (troca == 0)    /* não houve troca */
          return;
   }
}
```

Ordenação de vetores complexos

- Algoritmo genérico (I):
  - independente dos dados armazenados no vetor
  - usa uma função auxiliar para comparar elementos

```
/* Função auxiliar para comparar matrículas */
static int compara (Aluno *a, Aluno *b)
{
   if (a->mat > b->mat)
      return 1;
   else
      return 0;
}
```

Ordenação de vetores complexos

- Algoritmo genérico (I):
  - Considerando ordenação em ordem decrescente de média

```
/* Função auxiliar para comparar médias */
static int compara (Aluno *a, Aluno *b)
{
   if ((a->p1+a->p2+a->p3)/3 < (b->p1+b->p2+b->p3)/3)
      return 1;
   else
      return 0;
}
```

Ordenação de vetores complexos

- Algoritmo genérico (II):
  - função de ordenação e assinatura da função de comparação independentes do tipo do elemento
  - função de ordenação: void ordena (int n, void* v, int tam);

v - ponteiro de qualquer tipo (definido como void*)

tam - tamanho de cada elemento em bytes (para percorrer o vetor)

  - função de comparação: int compara (void* a, void* b);

a e b - dois ponteiros genéricos um para cada elemento que se deseja comparar

Ordenação de vetores complexos

```
/* Ordenação genérica */
void ordena (int n, void* v, int tam, int(*cmp)(void*,void*))
{
   int i, fim, troca;
   void *p1, *p2;
   for (fim=n-1; fim>0; fim--) {
       troca = 0;
       for (i=0; i<fim; i++) {
            p1 = acessa(v,i,tam);
            p2 = acessa(v,i+1,tam);
            if (cmp(p1,p2)) {
                trocar(p1,p2,tam);
                troca = 1;
            }
       }
       if (troca == 0)    /* nao houve troca */
           return;
   }
}
```

Ordenação de vetores complexos

- Quick sort genérico da biblioteca padrão:
  - disponibilizado via a biblioteca *stdlib.h*
  - independe do tipo de dado armazenado no vetor
  - implementação segue os princípios discutidos na implementação do algoritmo de ordenação bolha genérico

Ordenação de vetores complexos

- Protótipo do quick sort da biblioteca padrão:

```
void qsort (void *v, int n, int tam,
                    int (*cmp)(const void*, const void*));
```

v: ponteiro para o primeiro elemento do vetor (é do tipo genérico para acomodar qualquer tipo de elemento do vetor

n: número de elementos do vetor

tam: tamanho, em bytes, de cada elemento do vetor

cmp: ponteiro para a função de comparação

const: modificador de tipo para garantir que a função não modificará os valores dos elementos (devem ser tratados como constantes)

Ordenação de vetores complexos

- Função de comparação:

```
int nome_func (const void*, const void*);
```

  - definida pelo cliente do quick sort
  - recebe dois ponteiros genéricos (do tipo void*)
    - apontam para os dois elementos a comparar
    - modificador de tipo const garante que a função não modificará os valores dos elementos (devem ser tratados como constantes)
  - deve retornar –1, 0, ou 1, se o primeiro elemento for anterior, indiferente, ou posterior ao segundo elemento, respectivamente, de acordo com o critério de ordenação adotado

Ordenação de vetores complexos

- Exemplo 1:
  - ordenação de valores reais

Ordenação de vetores complexos

- Função de comparação para float:
  - os dois ponteiros genéricos passados para a função de comparação representam ponteiros para float

```
/* função de comparação de reais */
static int comp_reais (const void* p1, const void* p2)
{
   /* converte para ponteiros de float */
   float *f1 = (float*)p1;
   float *f2 = (float*)p2;
   /* dados os ponteiros de float, faz a comparação */
   if (*f1 < *f2) return -1;
   else if (*f1 > *f2) return 1;
   else return 0;
}
```

```c
/* Ilustra uso do algoritmo qsort  para vetor de float */
#include <stdio.h>
#include <stdlib.h>

/* função de comparação de reais */
static int comp_reais (const void* p1, const void* p2)
{…}

/* ordenação de um vetor de float */
int main (void)
{
   int i;
   float v[8] = {25.6,48.3,37.7,12.1,57.4,86.6,33.3,92.8};
   qsort(v,8,sizeof(float),comp_reais);
   printf("Vetor ordenado: ");
   for (i=0; i<8; i++)
      printf("%g ",v[i]);
   printf("\n”);
   return 0;
}
```

Ordenação de vetores complexos

- Exemplo 2:
  - vetor de ponteiros para *Aluno*
  - chave de ordenação dada pelo nome do aluno

```
/* estrutura representando um aluno*/
struct aluno {
   int mat;
   char nome[81];     /* chave de ordenação */
   float p1, p2, p3;
   char email[41];
};
typedef struct aluno Aluno;

Aluno* vet[100];    /* vetor de ponteiros para Aluno */
```

Ordenação de vetores complexos

- Função de comparação 2
  - os dois ponteiros genéricos passados para a função de comparação representam ponteiros de ponteiros para Aluno
  - função deve tratar uma indireção a mais

```
/* Função de comparação: elemento é do tipo Aluno* */
static int comp_alunos (const void* p1, const void* p2)
{
   /* converte p/ ponteiros de ponteiros de Aluno */
   Aluno **a1 = (Aluno**)p1;
   Aluno **a2 = (Aluno**)p2;

   /* faz a comparação */
   return strcmp((*a1)->nome,(*a2)->nome);
}
```

Ordenação de vetores complexos

- Exemplo 3:
  - vetor de ponteiros para *Aluno*
  - chave de ordenação dada pelo ordem decrescente da média do aluno, com desempate pela ordem crescente matrícula do aluno.

```
/* estrutura representando um aluno*/
struct aluno {
   int mat;
   char nome[81];    /* chave de ordenação */
   float p1, p2, p3;
   char email[41];
};
typedef struct aluno Aluno;

Aluno* vet[100];    /* vetor de ponteiros para Aluno */
```

Ordenação de vetores complexos

- Função de comparação 3

```
static int comp_alunos (const void* p1, const void* p2)
{
   /* converte p/ ponteiros de ponteiros de Aluno */
   Aluno **a1 = (Aluno**)p1;
   Aluno **a2 = (Aluno**)p2;
   float m1 = ((*a1)->p1 + (*a1)->p2 + (*a1)->p3)/3
   float m2 = ((*a2)->p1 + (*a2)->p2 + (*a2)->p3)/3

   /* faz a comparação da média */
   if(m1<m2) return 1;
   if(m1>m2) return -1;

   /* desempate com a matrícula*/
   if((*a1)->mat > (*a2)->mat) return 1;
   if((*a1)->mat < (*a2)->mat) return -1;

   return 0;
}
```

```
#include <string.h>
 ...
int main( )
{
  Aluno a[5]={ {123, "Pedro", 7.0, 5.6, 8.9,"pedro@puc"},
               {124, "Joao",  3.0, 4.0, 6.2,"joao@puc"},
               {125, "Maria", 4.5, 6.7, 7.2,"maria@puc"},
               {126, "Jose",  8.9, 9.2, 9.9,"jose@puc"},
               {127, "Felipe",6.5, 8.3, 7.9,"felipe@puc"}};

   vet[0]=&a[0]; vet[1]=&a[1]; vet[2]=&a[2];
   vet[3]=&a[3]; vet[4]=&a[4];

    show_vet(5,vet);

    qsort(vet,5,sizeof(Aluno*),comp_alunos);

    show_vet(5,vet);
    return 1;
}
```

Resumo

- Bubble sort
  - quando dois elementos estão fora de ordem, troque-os de posição até que o i-ésimo elemento de maior valor do vetor seja levado para as posições finais do vetor
  - continue o processo até que todo o vetor esteja ordenado
- Quick sort
  - coloque um elemento arbitrário x, o *pivô,* em sua posição k
  - chame recursivamente o algoritmo para ordenar os (sub-)vetores v[0], … v[k−1] e v[k+1], …, v[n−1]
  - continue até que os vetores que devem ser ordenados tenham 0 ou 1 elemento
- Quick sort genérico da biblioteca padrão:
  - disponibilizado via *stdlib.h*, com protótipo

```
void qsort (void *v, int n, int tam, int (*cmp)(const void*, const void*))
```

Referências

Waldemar Celes, Renato Cerqueira, José Lucas Rangel, *Introdução a Estruturas de Dados*, Editora Campus (2004)

Capítulo 16 – Ordenação

Material adaptado por Luis Martí a partir dos slides de José Viterbo Filho que forem elaborados por Marco Antonio Casanova e Marcelo Gattas para o curso de Estrutura de Dados para Engenharia da PUC-Rio, com base no livro *Introdução a Estrutura de Dados*, de Waldemar Celes, Renato Cerqueira e José Lucas Rangel, Editora Campus (2004).