BUSCA E ORDENAÇÃO EM ARRAYS

Prof. André Backes | @progdescomplicada

BUSCA

Definição

- Ato de procurar por um elemento em um conjunto de dados
  - Recuperação de dados armazenados em um repositório ou base de dados
- A operação de busca visa responder se um determinado valor está ou não presente em um conjunto de elementos
  - Por exemplo, em um array

Definição

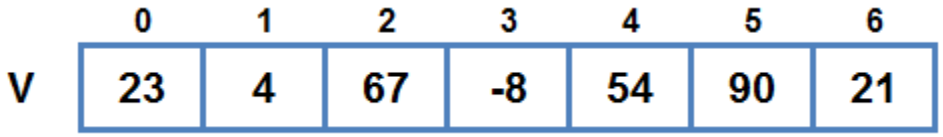
- Baseada em uma chave
  - A chave de busca é o **campo** do item utilizado para comparação
    - Valor armazenado em um array de inteiros
    - Campo de uma struct
    - etc
  - É por meio dela que sabemos se dado elemento é o que buscamos
    - No caso do item estar presente no conjunto de elementos, seus dados são retornados para o usuário

Definição

- Existem vários tipos de busca
- Sua utilização depende de como são estes dados
  - Os dados estão estruturados ?
    - Array, lista, árvore, etc.?
    - Existe também a busca em dados não estruturados
  - Os dados estão ordenados?
  - Existem valores duplicados?

Definição

- Tipos de busca abordados
  - Dados armazenados em um array
  - Dados ordenados ou não
- Métodos
  - Busca Sequencial ou Linear
  - Busca Sequencial Ordenada
  - Busca Binária



Busca Sequencial ou Linear

- Estratégia de busca mais simples que existe
  - Basicamente, esse algoritmo percorre o array que contém os dados desde a sua primeira posição até a última
    - Assume que os dados não estão ordenados, por isso a necessidade de percorrer o array do seu início até o seu fim

Busca Sequencial ou Linear

- Funcionamento
  - Para cada posição do array, o algoritmo compara se a posição atual do array é igual ao valor buscado.
    - Se os valores forem iguais, a busca termina
    - caso contrário, a busca continua com a próxima posição do array

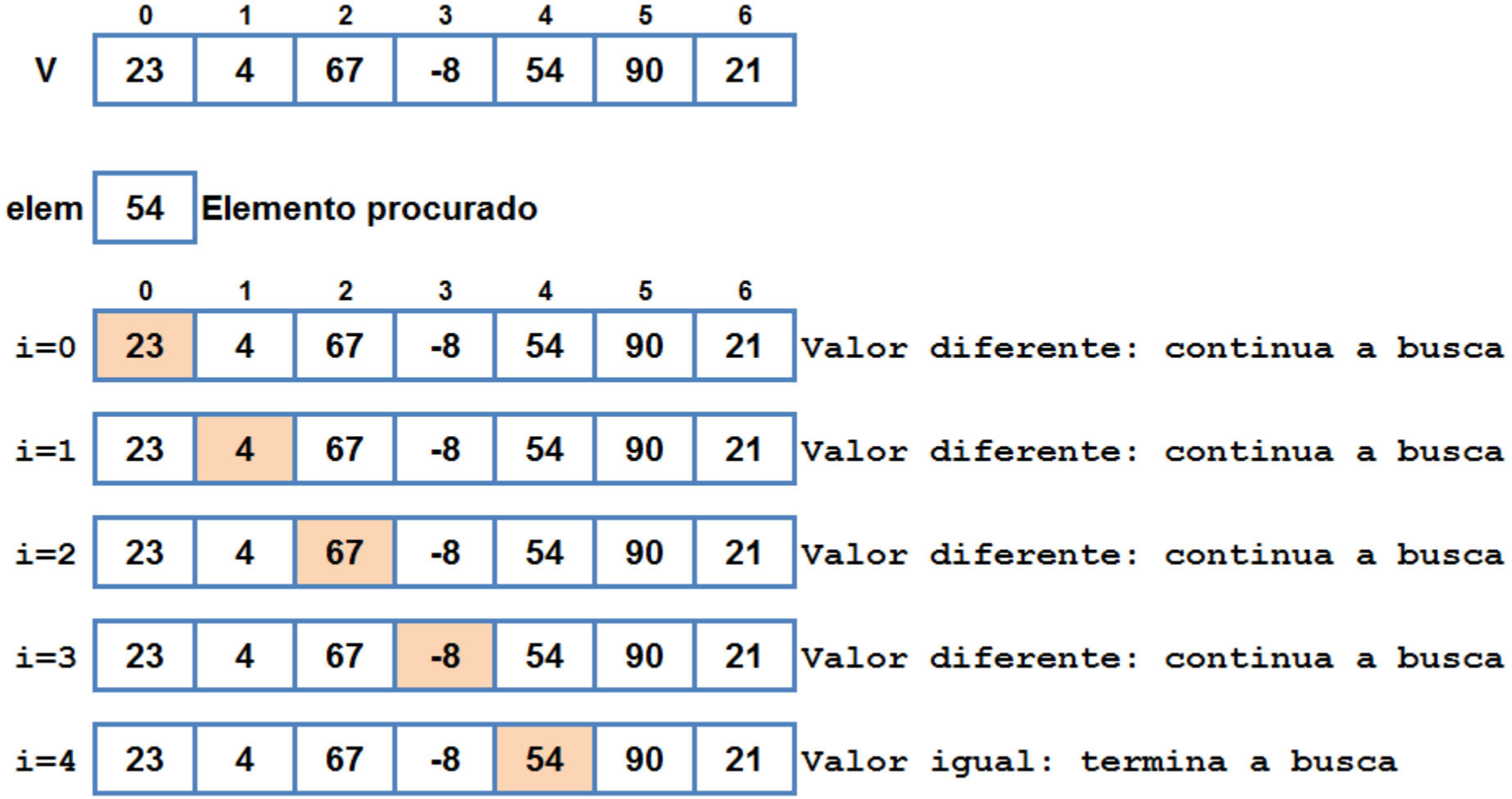
Busca Sequencial ou Linear

- Algoritmo

```
13
14  int buscaLinear(int *V, int N, int elem){
15      int i;
16      for(i = 0; i<N; i++){
17          if(elem == V[i])
18              return i;//elemento encontrado
19      }
20      return -1;//elemento não encontrado
21  }
```

Busca Sequencial ou Linear

- Exemplo



Busca Sequencial ou Linear

- Complexidade
  - Considerando um array com **N** elementos
    - ***O(1)***, melhor caso: o elemento é o primeiro do array
    - ***O(N)***, pior caso: o elemento é o último do array ou não existe
    - ***O(N/2)***, caso médio

Busca Sequencial Ordenada

- Procurar por um determinado valor em um array desordenado é uma tarefa bastante cara

- Como melhorar isso?
  - Organizando o array segundo alguma ordem, isto é, devemos ordenar o array
  - Isso facilita a tarefa de busca

Busca Sequencial Ordenada

- Funcionamento
  - Assume que os dados estão ordenados
  - Se o elemento procurado for **menor** do que o valor em uma determinada posição do array, temos a certeza de que ele não estará no restante do array
    - Isso evita a necessidade de percorrer o array do seu início até o seu fim

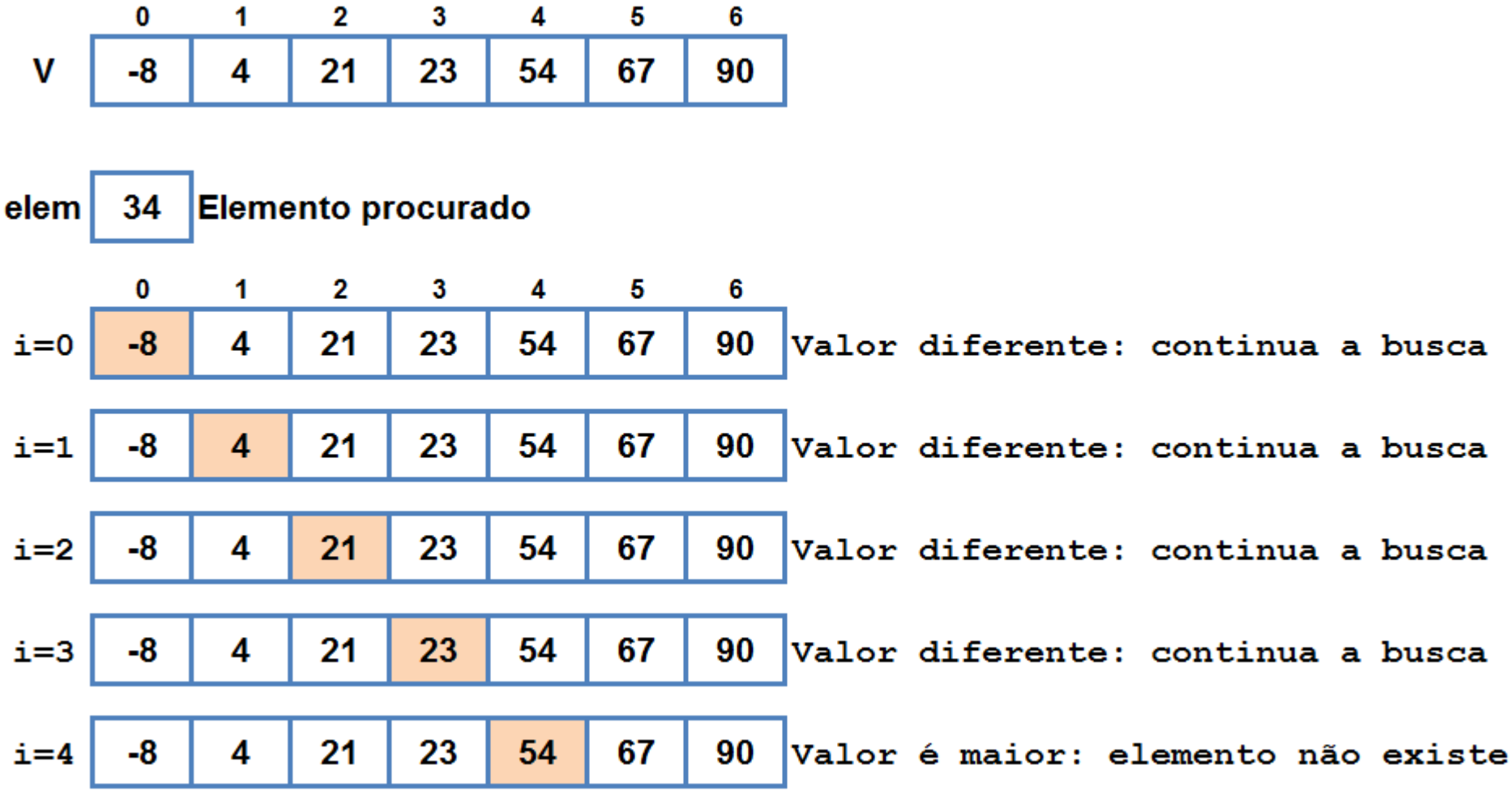
Busca Sequencial Ordenada

- Algoritmo

```
14
15  int buscaOrdenada(int *V, int N, int elem){
16      int i;
17      for(i = 0; i<N; i++){
18          if(elem == V[i])
19              return i;//elemento encontrado
20          else
21              if(elem < V[i])
22                  return -1;//para a busca
23      }
24      return -1;//elemento não encontrado
25  }
```

Busca Sequencial Ordenada

- Exemplo



Busca Sequencial Ordenada

- Desvantagens
  - Ordenar um array também tem um custo
    - Esse custo é superior ao custo da busca sequencial no seu pior caso
  - Se for para fazer a busca de um único elemento, não compensa ordenar o array
    - Porém, se mais de um elemento for recuperado do array, o esforço de ordenar o array pode compensar

Busca Binária

- Fazer a busca em um array ordenado representa um ganho de tempo
  - Podemos terminar a busca mais cedo se o elemento procurado for menor que o valor da posição atual do array

Busca Binária

- A Busca Sequencial Ordenada é uma estratégia de busca extremamente simples
  - Ela percorre todo o array linearmente
  - Não utiliza adequadamente a ordenação dos dados
- Uma estratégia de busca mais sofisticada é a Busca Binária
  - Muito mais eficiente do que a Busca Sequencial Ordenada

Busca Binária

- Funcionamento
  - É uma estratégia baseada na idéia de *dividir para conquistar*
  - A cada passo, esse algoritmo analisa o valor do meio do array
    - Caso esse valor seja igual ao elemento procurado, a busca termina
    - Caso contrário, a busca continua na metade do array que condiz com o valor procurado

Busca Binária

- Algoritmo

```
27
28  int buscaBinaria(int *V, int N, int elem){
29      int i, inicio, meio, final;
30      inicio = 0;
31      final = N-1;
32      while(inicio <= final){
33          meio = (inicio + final)/2;
34          if(elem < V[meio])
35              final = meio-1;//busca na metade da esquerda
36          else
37              if(elem > V[meio])
38                  inicio = meio+1;//busca na metade da direita
39              else
40                  return meio;
41      }
42      return -1;//elemento não encontrado
43  }
```

Busca Binária

- Exemplo

Busca Binária

- Complexidade
  - Considerando um array com **N** elementos, o tempo de execução é:
    - ***O(1)***, melhor caso: o elemento procurado está no meio do array;
    - ***O(log$_2$ N)***, pior caso: o elemento não existe;
    - ***O(log$_2$ N)***, caso médio.

Busca Binária

- Complexidade
  - Para se ter uma idéia da sua vantagem, em um array contendo **N = 1000** elementos, no pior caso
    - A Busca Sequencial irá executar **1000 comparações**
    - A Busca Binária irá executar apenas **10 comparações**

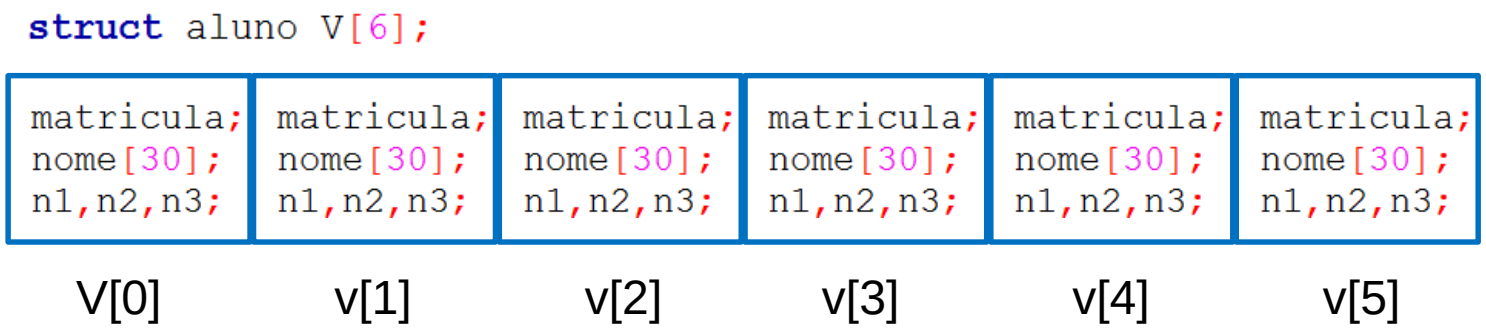
Busca em array de struct

- A busca em um array de inteiros é uma tarefa simples
  - Na prática, trabalhamos com dados um pouco mais complexos, como estruturas
  - Mais dados para manipular

```
11  struct aluno{
12      int matricula;
13      char nome[30];
14      float n1,n2,n3;
15  };
```

Busca em array de struct

- Como fazer a busca quando o que temos é um array de struct?



Busca em array de struct

- **Relembrando**
- A busca é baseada em uma chave
  - A chave de busca é o **campo** do item utilizado para comparação
    - Valor armazenado em um array de inteiros
    - **Campo de uma struct**
    - etc
  - É por meio dela que sabemos se dado elemento é o que buscamos
    - No caso do item estar presente no conjunto de elementos, seus dados são retornados para o usuário

Busca em array de struct

- Ou seja, devemos modificar o algoritmo para que a comparação das chaves seja feita utilizando um determinado campo da **struct**
- Exemplo
  - Vamos modificar a **busca linear**
    - Essa modificação vale para os outros tipos de busca

```
13
14  int buscaLinear(int *V, int N, int elem){
15      int i;
16      for(i = 0; i<N; i++){
17          if(elem == V[i])
18              return i;//elemento encontrado
19      }
20      return -1;//elemento não encontrado
21  }
```

Busca em array de struct

- Duas novas buscas
  - Busca por **matricula**
  - Busca por **nome**

```
28  int buscaLinearMatricula(struct aluno *V, int N, int elem){
29      int i;
30      for(i = 0; i<N; i++){
31          if(elem == V[i].matricula)
32              return i;//elemento encontrado
33      }
34      return -1;//elemento não encontrado
35  }
36
37
38  int buscaLinearNome(struct aluno *V, int N, char* elem){
39      int i;
40      for(i = 0; i<N; i++){
41          if(strcmp(elem,V[i].nome)==0)
42              return i;//elemento encontrado
43      }
44      return -1;//elemento não encontrado
45  }
```

ORDENAÇÃO

Conceitos básicos

- Ordenação
  - Ato de colocar um conjunto de dados em uma determinada ordem predefinida
  - Fora de ordem
    - 5, 2, 1, 3, 4
  - Ordenado
    - 1, 2, 3, 4, 5 **OU** 5, 4, 3, 2, 1
- Algoritmo de ordenação
  - Coloca um conjunto de elementos em uma certa ordem

Conceitos básicos

- A ordenação permite que o acesso aos dados seja feito de forma mais eficiente
  - É parte de muitos métodos computacionais
    - Algoritmos de busca, intercalação/fusão, utilizam ordenação como parte do processo
    - Aplicações em geometria computacional, bancos de dados, entre outras necessitam de listas ordenadas para funcionar

Conceitos básicos

- A ordenação é baseada em uma chave
  - A chave de ordenação é o **campo** do item utilizado para comparação
    - Valor armazenado em um array de inteiros
    - Campo nome de uma struct
    - etc
  - É por meio dela que sabemos se um determinado elemento está a frente ou não de outros no conjunto

Conceitos básicos

- Podemos usar qualquer tipo de chave
  - Deve existir uma regra de ordenação bem-definida
- Alguns tipos de ordenação
  - numérica
    - 1, 2, 3, 4, 5
  - lexicográfica (ordem alfabética)
    - André, Carlos, Eduardo, Ricardo

Conceitos básicos

- Independente do tipo, a ordenação pode ser

- Crescente
  - 1, 2, 3, 4, 5
  - André, Carlos, Eduardo, Ricardo

- Decrescente
  - 5, 4, 3, 2, 1
  - Ricardo, Eduardo, Carlos, André

Conceitos básicos

- Os algoritmos de ordenação podem ser classificados como de

- Ordenação interna
  - O conjunto de dados a ser ordenado cabe todo na memória principal (RAM)
  - Qualquer elemento pode ser imediatamente acessado

Conceitos básicos

- Os algoritmos de ordenação podem ser classificados como de

- Ordenação externa
  - O conjunto de dados a ser ordenado não cabe na memória principal
  - Os dados estão armazenados em memória secundário
    - Por exemplo, um arquivo
  - Os elementos são acessados sequencialmente ou em grandes blocos

Conceitos básicos

- Além disso, a ordenação pode ser estável ou não
  - Um algoritmo de ordenação é considerado **estável** se a ordem dos elementos com chaves iguais não muda durante a ordenação
  - O algoritmo preserva a **ordem relativa** original dos valores

Conceitos básicos

- Dados não ordenados
  - **5a**, 2, **5b**, 3, 4, 1
  - **5a** e **5b** são o mesmo número

- Dados ordenados
  - 1, 2, 3, 4, 5a, 5b: ordenação **estável**
  - 1, 2, 3, 4, 5b, 5a: ordenação **não-estável**

Métodos de ordenação

- Os métodos de ordenação estudados podem ser divididos em

- Básicos
  - Fácil implementação
  - Auxiliam o entendimento de algoritmos complexos

- Sofisticados
  - Em geral, melhor desempenho

Algoritmo Bubble Sort

- Também conhecido como ordenação por bolha
  - É um dos algoritmos de ordenação mais conhecidos que existem
  - Remete a idéia de bolhas flutuando em um tanque de água em direção ao topo até encontrarem o seu próprio nível (ordenação crescente)

Algoritmo Bubble Sort

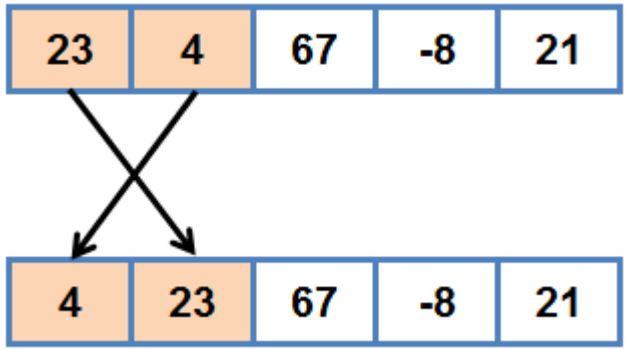
- Funcionamento
  - Compara pares de valores adjacentes e os troca de lugar se estiverem na ordem errada
    - Trabalha de forma a movimentar, uma posição por vez, o maior valor existente na porção não ordenada de um array para a sua respectiva posição no array ordenado
  - Esse processo se repete até que mais nenhuma troca seja necessária
    - Elementos já ordenados

Algoritmo Bubble Sort

- Algoritmo

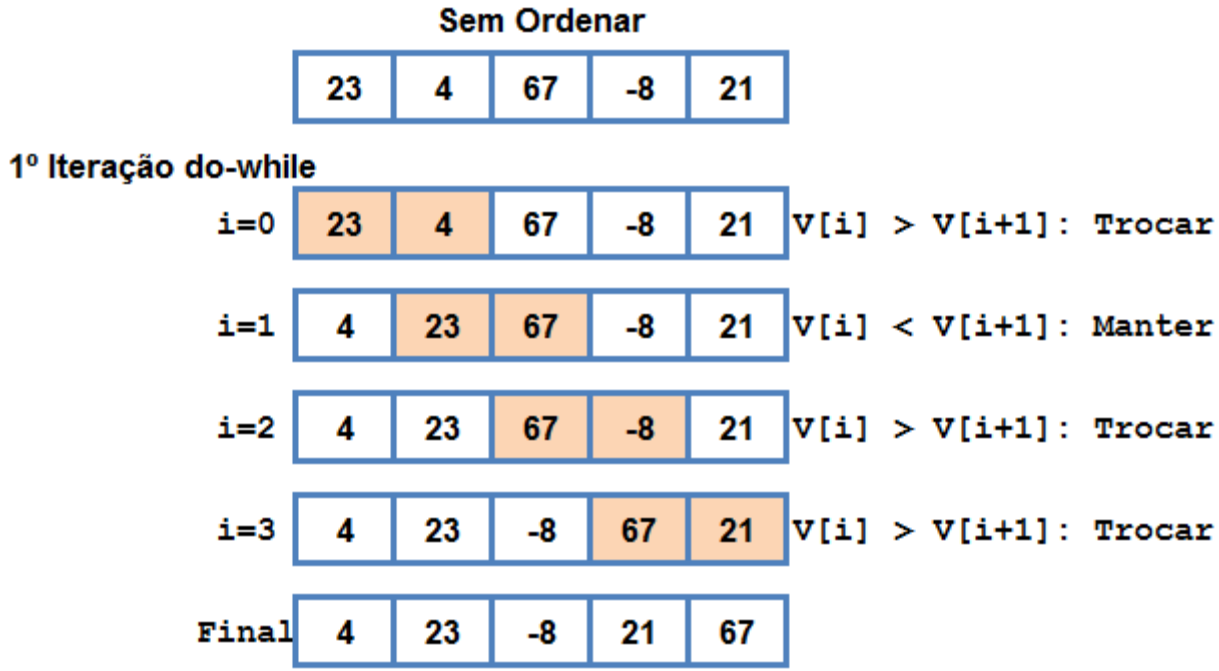
```
41
42  void bubbleSort(int *V  , int N){
43      int i, continua, aux, fim = N;
44      do{
45          continua = 0;
46          for(i = 0; i < fim-1; i++){
47              if (V[i] > V[i+1]){
48                  aux = V[i];
49                  V[i] = V[i+1];
50                  V[i+1] = aux;
51                  continua = i;
52              }
53          }
54          fim--;
55      }while(continua != 0);
56  }
```

Troca dois valores consecutivos no vetor



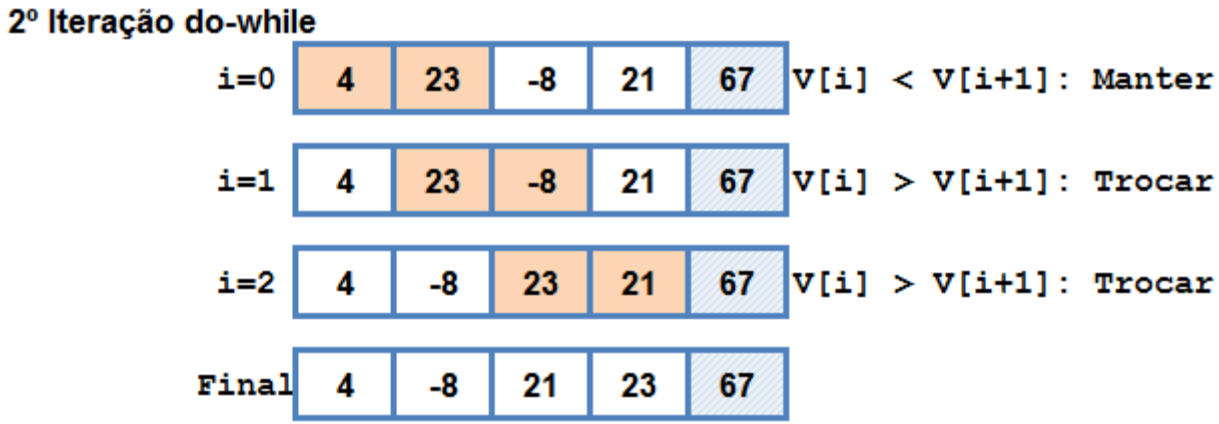
Algoritmo Bubble Sort | Passo a passo

- 1º iteração do-while: encontra o maior valor e o movimenta até a última posição



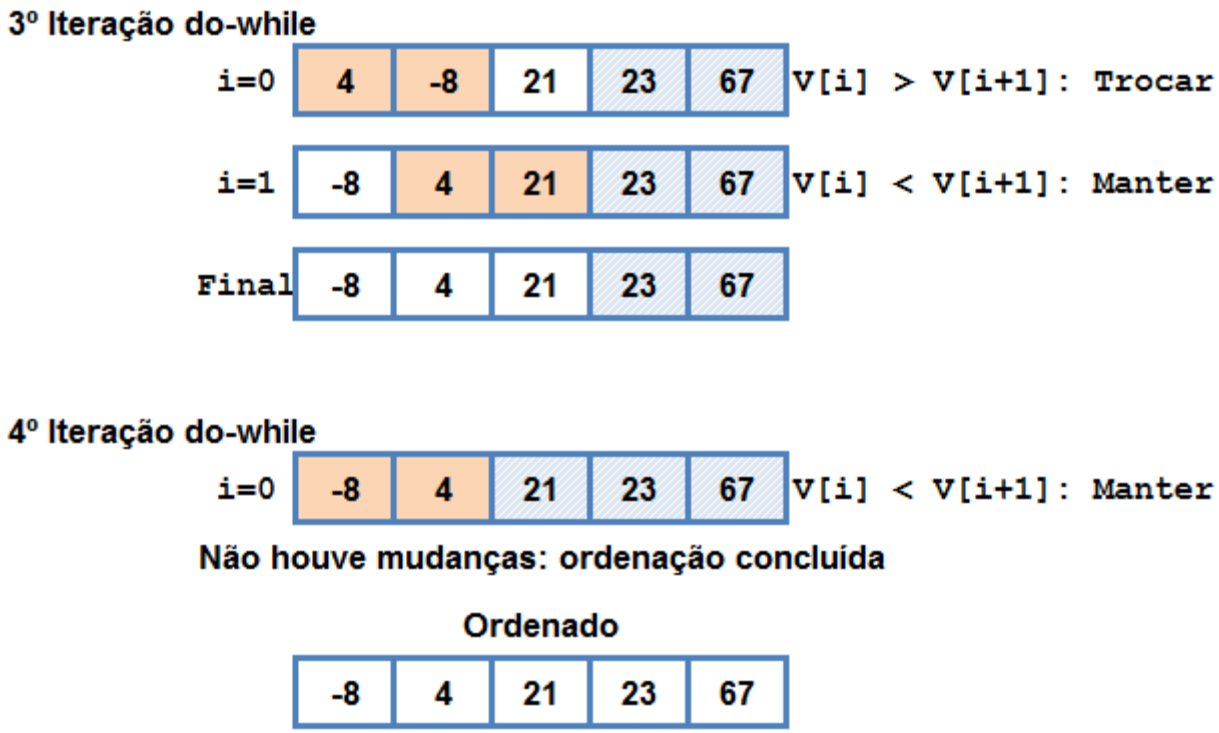
Algoritmo Bubble Sort | Passo a passo

- 2º iteração do-while: encontra o segundo maior valor e o movimenta até a penúltima posição



Algoritmo Bubble Sort | Passo a passo

- Processo continua até todo o array estar ordenado



Algoritmo Bubble Sort

- Vantagens
  - Simples e de fácil entendimento e implementação
  - Está entre os métodos de ordenação mais difundidos existentes
- Desvantagens
  - Não é um algoritmo eficiente
    - Sua eficiência diminui drasticamente a medida que o número de elementos no array aumenta
    - É estudado apenas para fins de desenvolvimento de raciocínio

Algoritmo Bubble Sort

- Complexidade
  - Considerando um array com **N** elementos, o tempo de execução é:
    - ***O(N)***, melhor caso: os elementos já estão ordenados.
    - ***O(N^2)***, pior caso: os elementos estão ordenados na ordem inversa.
    - ***O(N^2)***, caso médio.

Algoritmo Selection Sort

- Também conhecido como ordenação por seleção
  - É outro algoritmo de ordenação bastante simples
  - A cada passo ele **seleciona** o melhor elemento para ocupar aquela posição do array
    - Maior ou menor, dependendo do tipo de ordenação
    - Na prática, possui um desempenho quase sempre superior quando comparado com o bubble sort

Algoritmo Selection Sort

- Funcionamento
  - A cada passo, procura o menor valor do array e o coloca na primeira posição do array
    - Divide o array em duas partes: a parte ordenada, a esquerda do elemento analisado, e a parte que ainda não foi ordenada, a direita do elemento.
  - Descarta-se a primeira posição do array e repete-se o processo para a segunda posição
  - Isso é feito para todas as posições do array

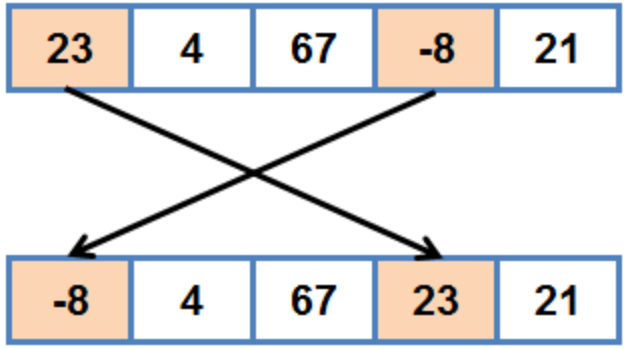
Algoritmo Selection Sort

- Algoritmo

```
74
75  void selectionSort(int *V, int N){
76      int i, j, menor, troca;
77      for(i = 0; i < N-1; i++){
78          menor = i;
79          for(j = i+1; j < N; j++){
80              if(V[j] < V[menor])
81                  menor = j;
82          }
83          if(i != menor){
84              troca = V[i];
85              V[i] = V[menor];
86              V[menor] = troca;
87          }
88      }
89  }
```

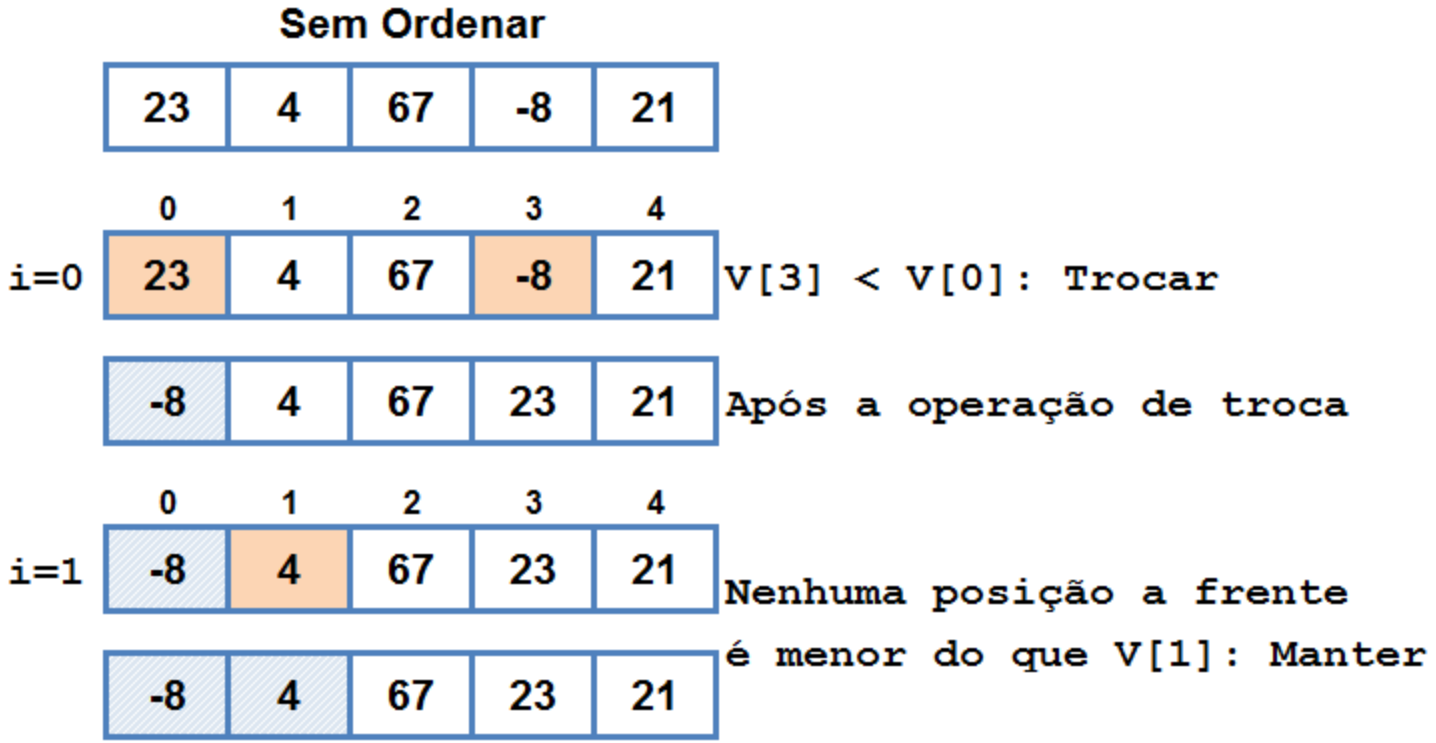
Procura o menor elemento em relação a “i”

Troca os valores da posição atual com a “menor”



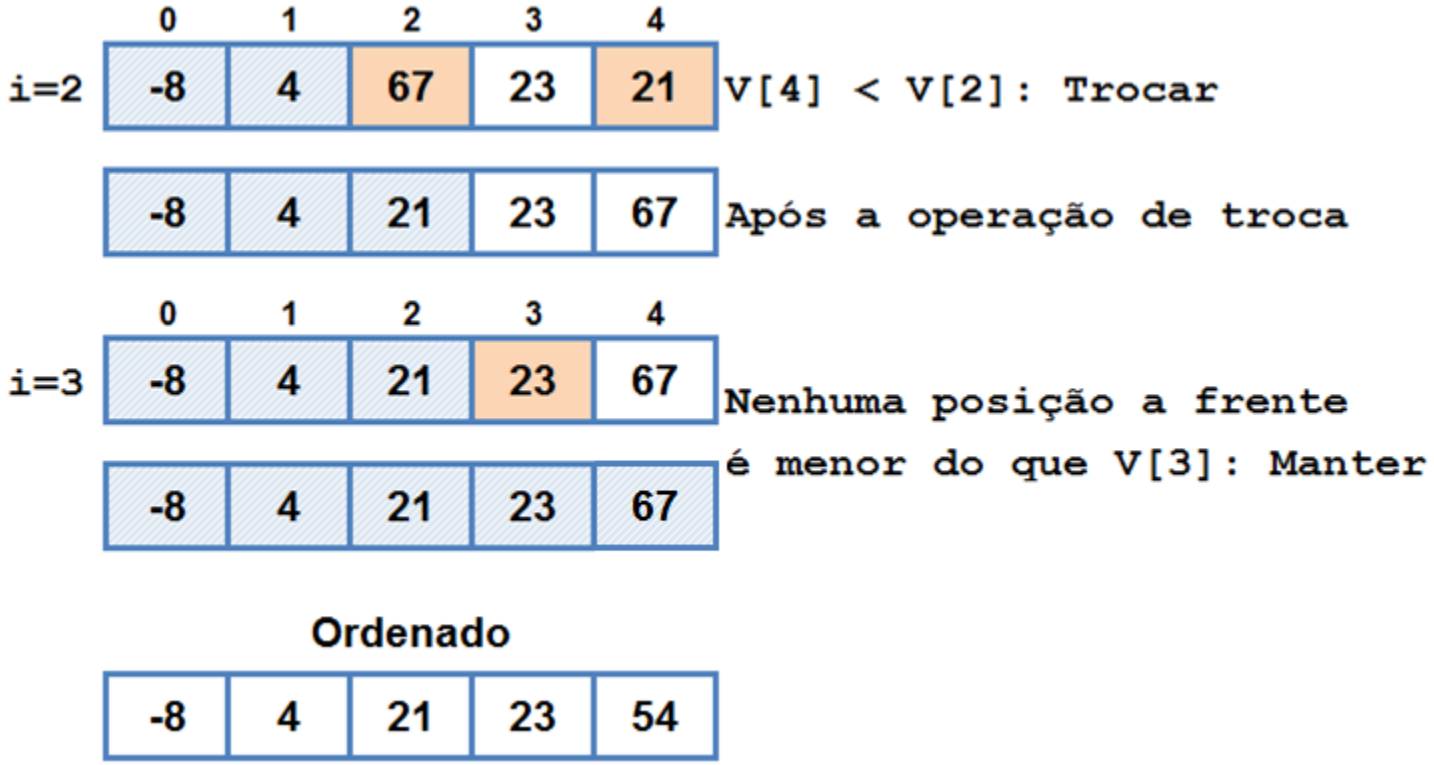
Algoritmo Selection Sort | Passo a passo

- Para cada posição **i**, procura no restante do array o menor valor para ocupá-la



Algoritmo Selection Sort | Passo a passo

- Para cada posição **i**, procura no restante do array o menor valor para ocupá-la



Algoritmo Selection Sort

- Vantagem
  - Simples de ser implementado
  - Não requer memória adicional

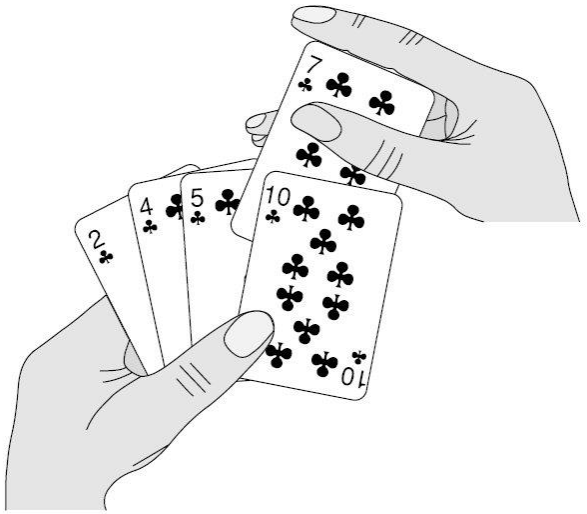
- Desvantagens
  - Sua eficiência diminui drasticamente a medida que o número de elementos no array aumenta
    - Não é recomendado para aplicações que que envolvam grandes quantidade de dados ou que precisem de velocidade

Algoritmo Selection Sort

- Complexidade
  - Considerando um array com **N** elementos, o tempo de execução é sempre de ordem $O(N^2)$
    - A eficiência do selection sort não depende da ordem inicial dos elementos
  - Melhor do que o bubble sort
    - Apesar de possuírem a mesma complexidade no caso médio, na prática o selection sort quase sempre supera o desempenho do bubble sort pois envolve um número menor de comparações

Algoritmo Insertion Sort

- Também conhecido como ordenação por inserção
  - Similar a ordenação de cartas de baralho com as mãos
    - Pegue uma carta de cada vez e a insira em seu devido lugar, sempre deixando as cartas da mão em ordem



Algoritmo Insertion Sort

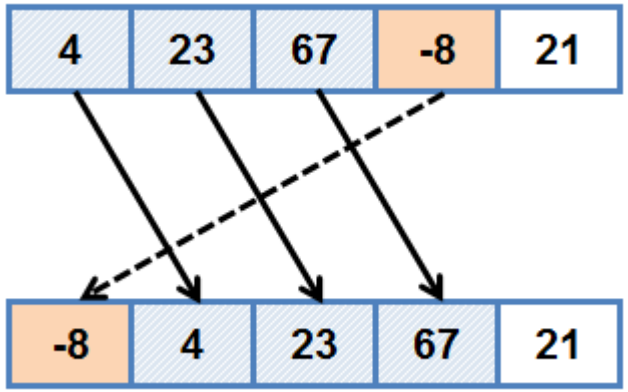
- Funcionamento
  - O algoritmo percorre o array e para cada posição **X** verifica se o seu valor está na posição correta
    - Isso é feito andando para o começo do array a partir da posição **X** e movimentando uma posição para frente os valores que são maiores do que o valor da posição **X**
    - Desse modo, teremos uma posição livre para inserir o valor da posição **X** em seu devido lugar

Algoritmo Insertion Sort

- Algoritmo

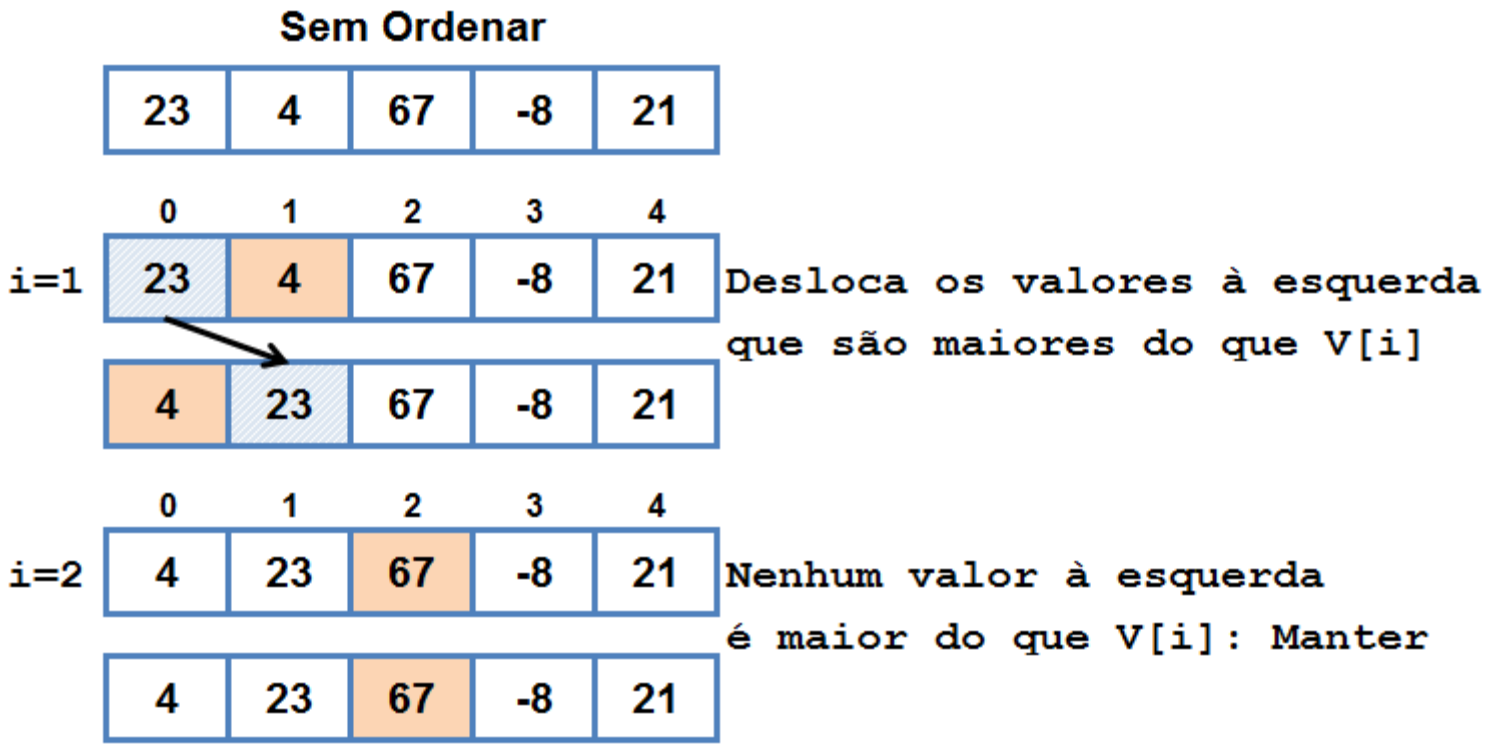
```
61
62  void insertionSort(int *V, int N){
63      int i, j, aux;
64      for(i = 1; i < N; i++){
65          aux = V[i];
66          for(j = i; (j > 0) && (aux < V[j - 1]); j--)
67              V[j] = V[j - 1];
68          V[j] = aux;
69      }
70  }
```

Move as cartas maiores para frente e insere na posição vaga



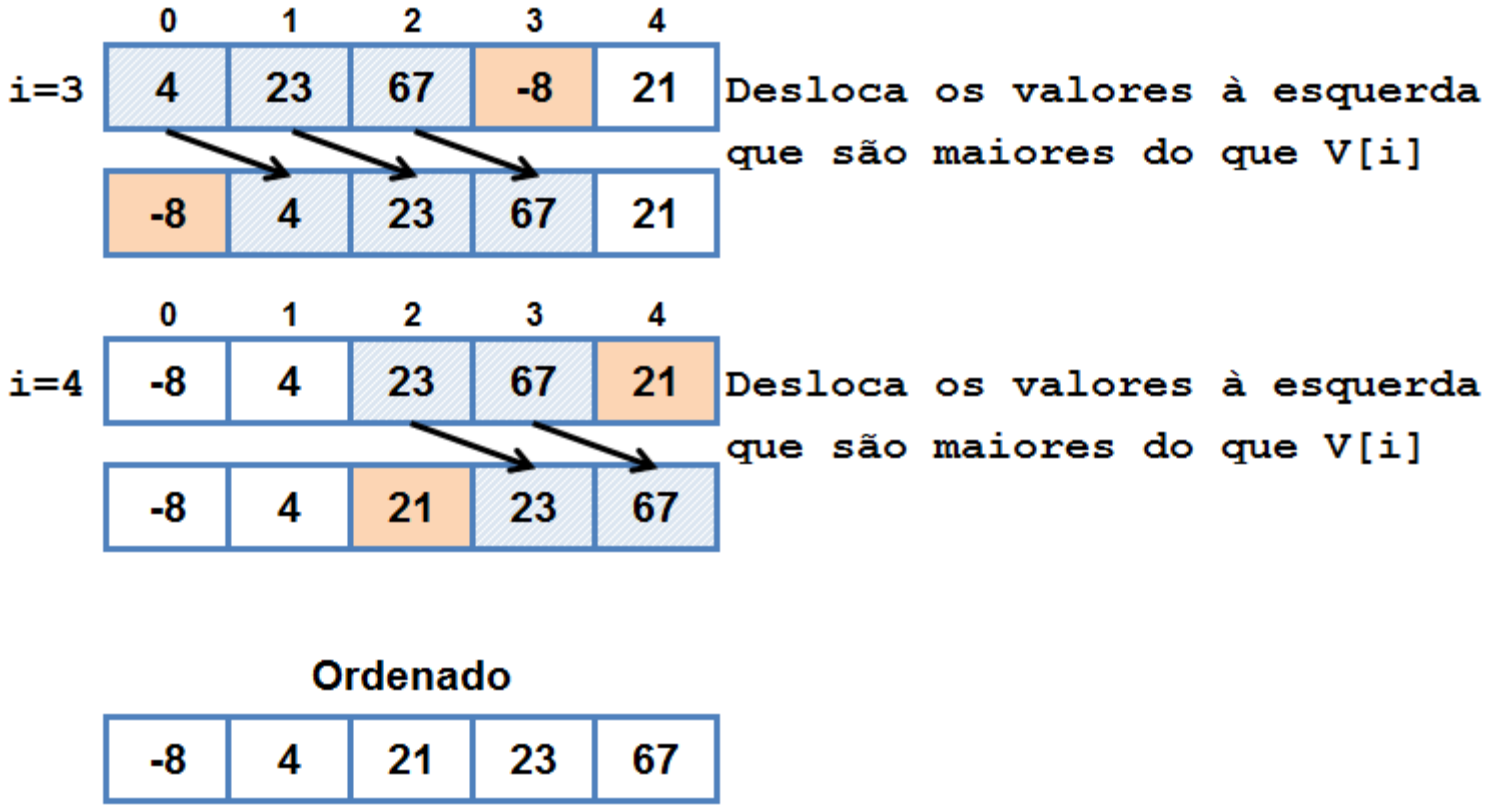
Algoritmo Insertion Sort | Passo a passo

- Para cada posição **i**, movimenta os valores maiores uma posição para frente no array



Algoritmo Insertion Sort | Passo a passo

- Para cada posição **i**, movimenta os valores maiores uma posição para frente no array



Algoritmo Insertion Sort

- Vantagens
  - Fácil implementação
  - Na prática, é mais eficiente que a maioria dos algoritmos de ordem quadrática
    - Como o selection sort e o bubble sort.
  - Um dos mais rápidos algoritmos de ordenação para conjuntos pequenos de dados
    - Superando inclusive o quick sort

Algoritmo Insertion Sort

- Vantagens
  - Estável: não altera a ordem dos dados iguais
  - Online
    - Pode ordenar elementos a medida que os recebe (tempo real)
    - Não precisa ter todo o conjunto de dados para colocá-los em ordem

Algoritmo Insertion Sort

- Complexidade
  - Considerando um array com **N** elementos, o tempo de execução é:
    - ***O(N)***, melhor caso: os elementos já estão ordenados.
    - ***O(N^2)***, pior caso: os elementos estão ordenados na ordem inversa.
    - ***O(N^2)***, caso médio.

Algoritmo Merge Sort

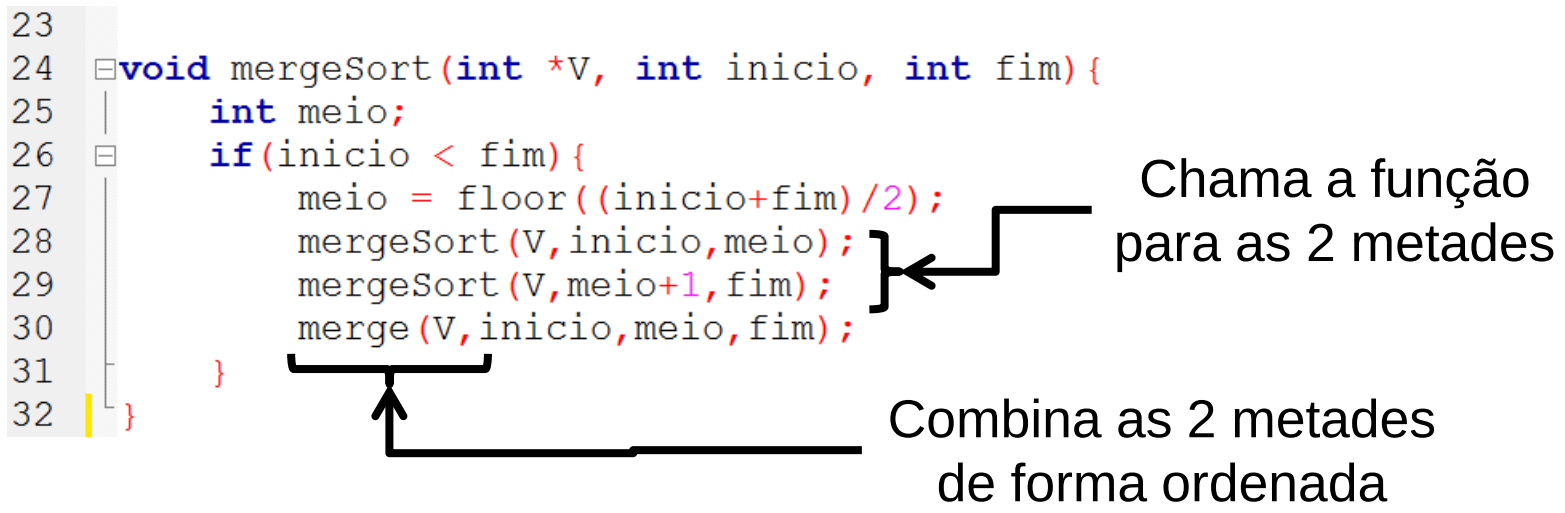
- Também conhecido como ordenação por intercalação
  - Algoritmo recursivo que usa a idéia de *dividir para conquistar* para ordenar os dados
    - Parte do princípio de que é mais fácil ordenar um conjunto com poucos dados do que um com muitos
  - O algoritmo divide os dados em conjuntos cada vez menores para depois ordená-los e combina-los por meio de intercalação (merge)

Algoritmo Merge Sort

- Funcionamento
  - Divide, recursivamente, o array em duas partes
    - Continua até cada parte ter apenas um elemento
  - Em seguida, combina dois array de forma a obter um array maior e ordenado
    - A combinação é feita intercalando os elementos de acordo com o sentido da ordenação (crescente ou decrescente)
  - Este processo se repete até que exista apenas um array

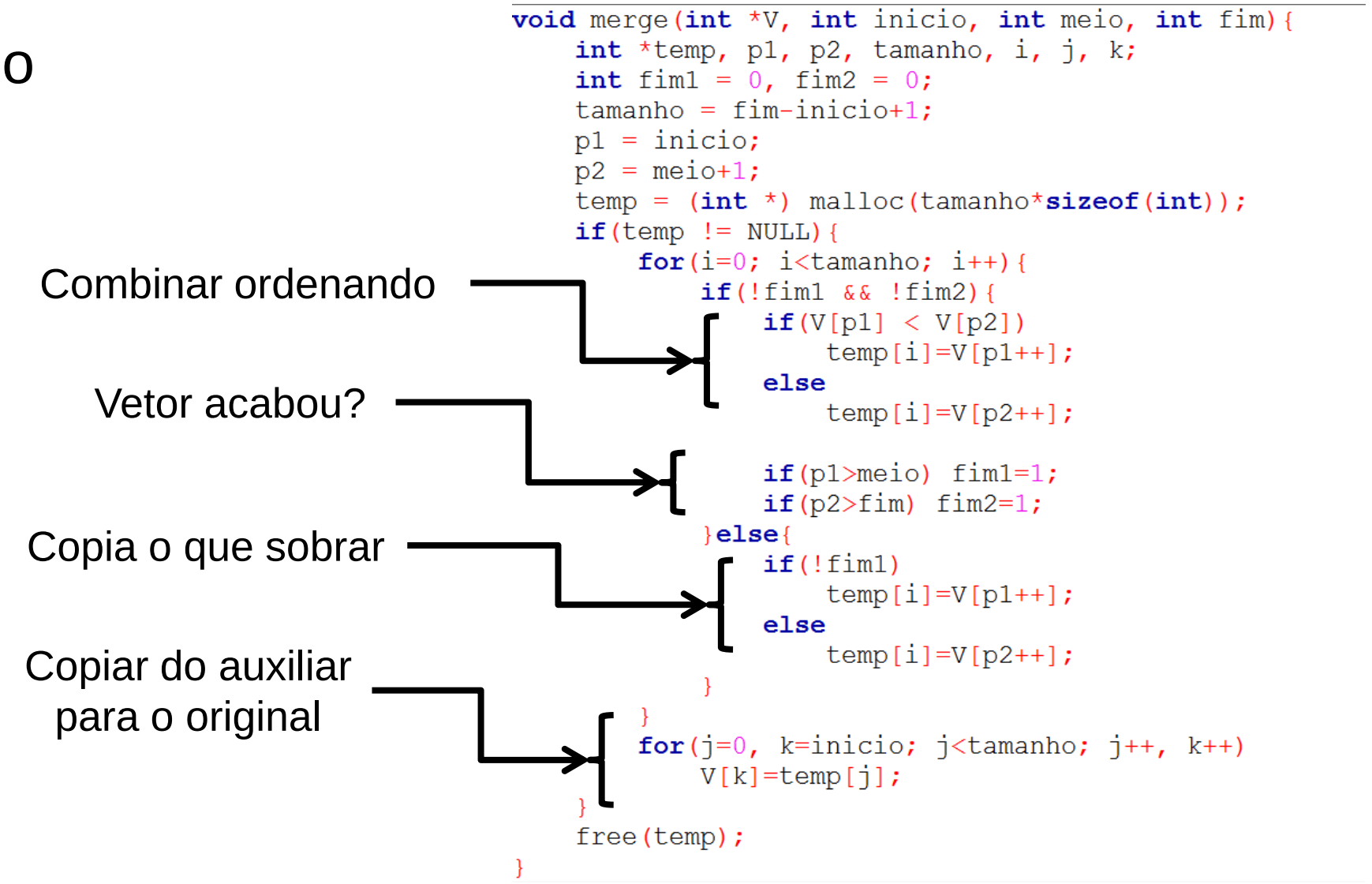
Algoritmo Merge Sort

- Algoritmo usa 2 funções
  - mergeSort : divide os dados em arrays cada vez menores
  - merge: intercala os dados de forma ordenada em um array maior



Algoritmo Merge Sort

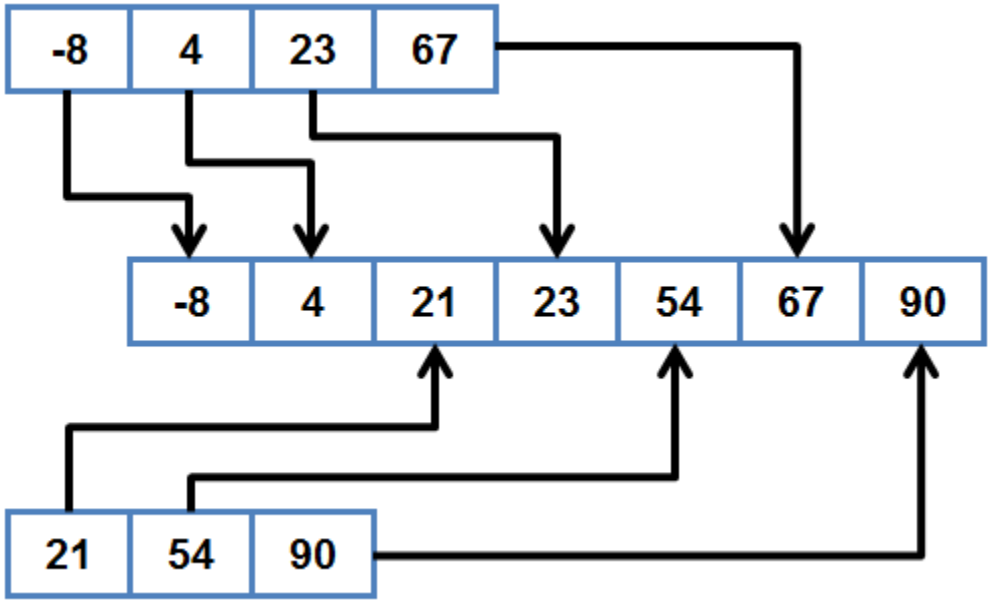
- Algoritmo




```
void merge(int *V, int inicio, int meio, int fim){
    int *temp, p1, p2, tamanho, i, j, k;
    int fim1 = 0, fim2 = 0;
    tamanho = fim-inicio+1;
    p1 = inicio;
    p2 = meio+1;
    temp = (int *) malloc(tamanho*sizeof(int));
    if(temp != NULL){
        for(i=0; i<tamanho; i++){
            if(!fim1 && !fim2){
                if(V[p1] < V[p2])
                    temp[i]=V[p1++];
                else
                    temp[i]=V[p2++];

                if(p1>meio) fim1=1;
                if(p2>fim) fim2=1;
            }else{
                if(!fim1)
                    temp[i]=V[p1++];
                else
                    temp[i]=V[p2++];
            }
        }
        for(j=0, k=inicio; j<tamanho; j++, k++)
            V[k]=temp[j];
    }
    free(temp);
}
```

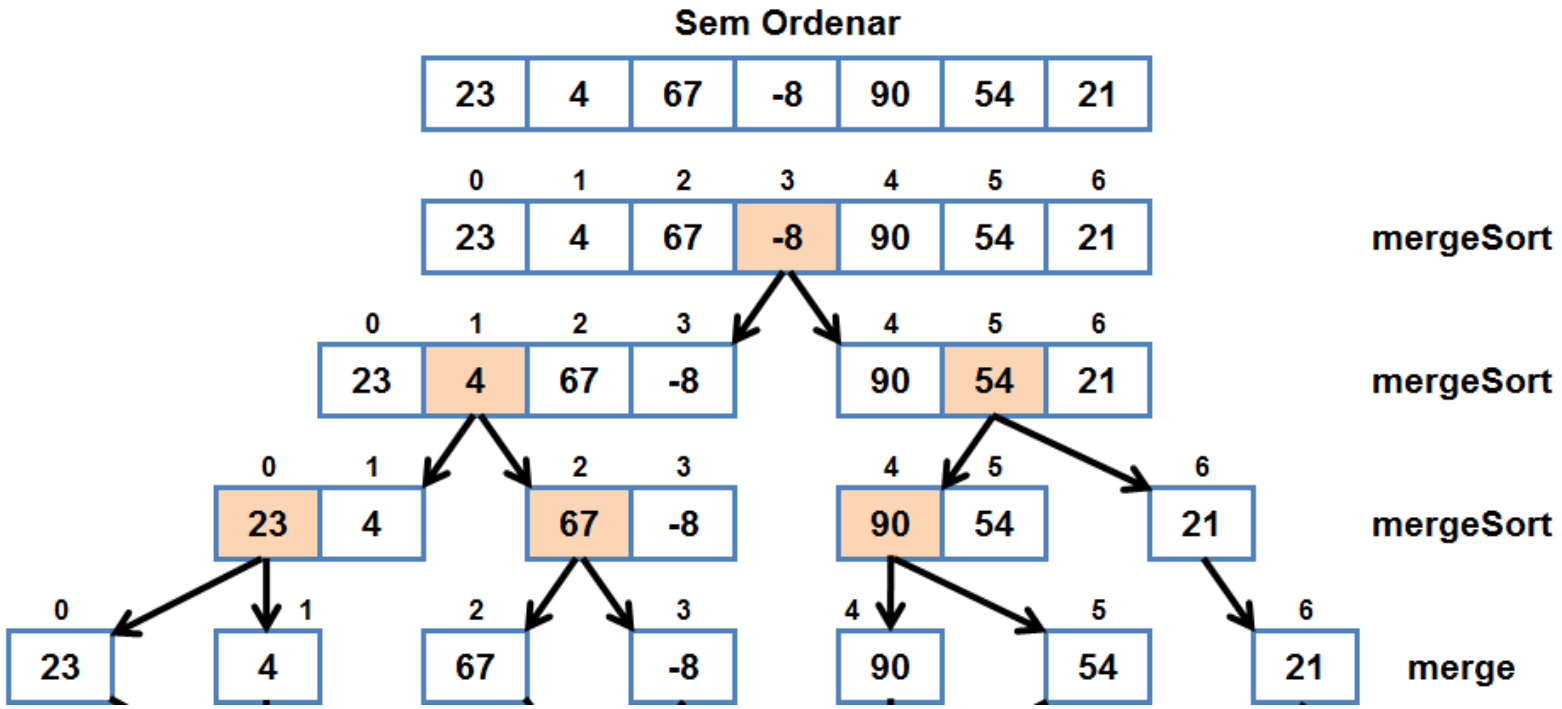
Algoritmo Merge Sort | Passo a passo

- Função merge
  - Intercala os dados de forma ordenada em um array maior
  - Utiliza um array auxiliar



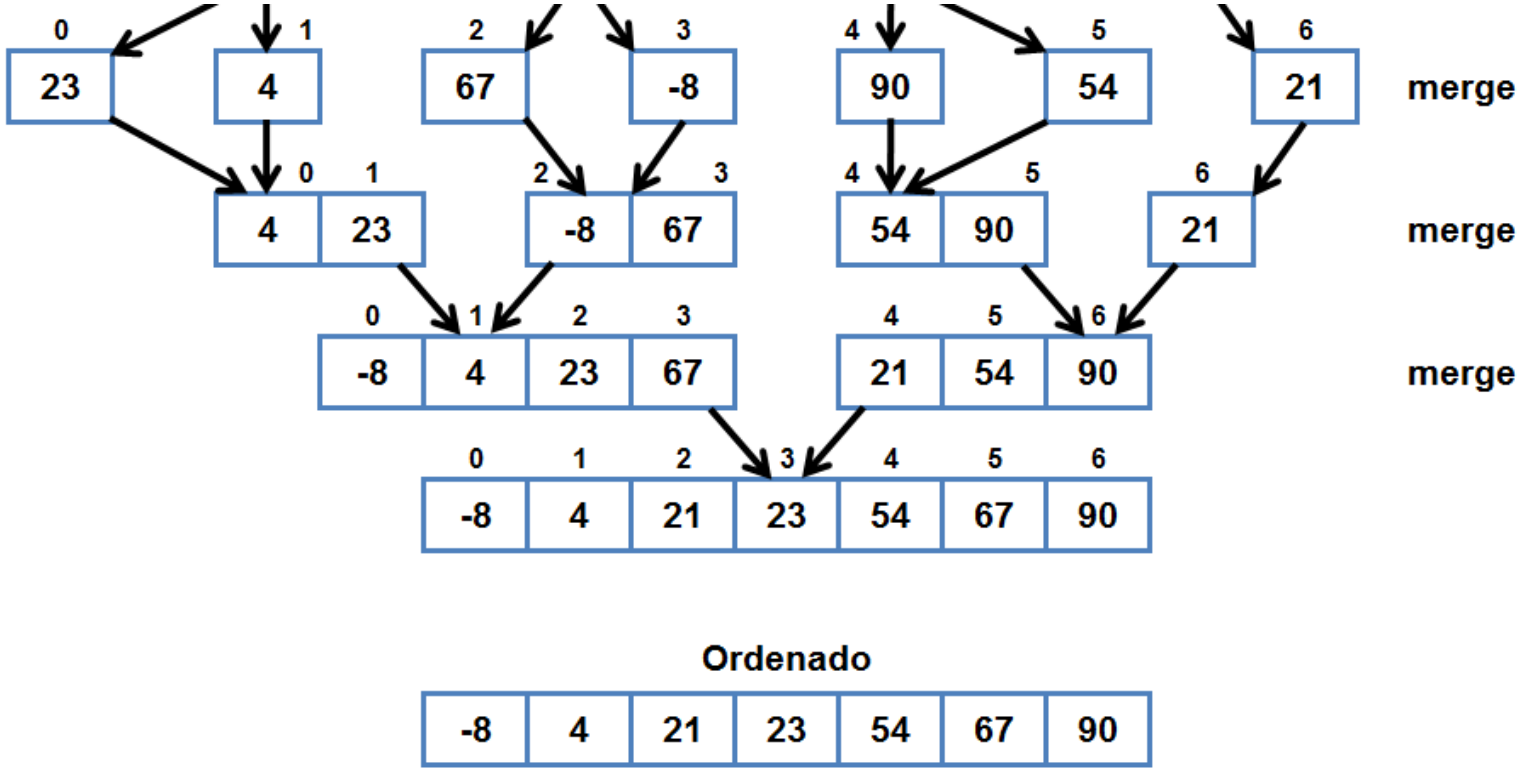
Algoritmo Merge Sort | Passo a passo

- Divide o array até ter **N** arrays de 1 elemento cada



Algoritmo Merge Sort | Passo a passo

- Intercala os arrays até obter um único array de **N** elementos



Algoritmo Merge Sort

- Complexidade
  - Considerando um array com **N** elementos, o tempo de execução é de ordem ***O(N log N)*** em todos os casos
  - Sua eficiência não depende da ordem inicial dos elementos
    - No pior caso, realiza cerca de 39% menos comparações do que o quick sort no seu caso médio
    - Já no seu melhor caso, o merge sort realiza cerca de metade do número de iterações do seu pior caso

Algoritmo Merge Sort

- Vantagens
  - Estável: não altera a ordem dos dados iguais

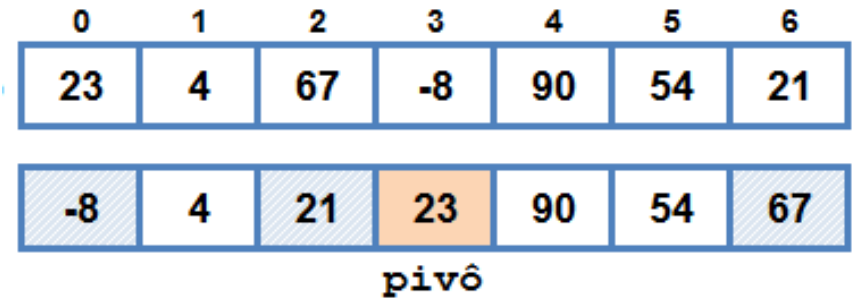
- Desvantagens
  - Possui um gasto extra de espaço de memória em relação aos demais métodos de ordenação
    - Ele cria uma cópia do array para cada chamada recursiva
    - Em outra abordagem, é possível utilizar um único array auxiliar ao longo de toda a sua execução

Algoritmo Quick Sort

- Também conhecido como ordenação por partição
  - É outro algoritmo recursivo que usa a idéia de *dividir para conquistar* para ordenar os dados
  - Se baseia no problema da separação
    - Em inglês, *partition subproblem*

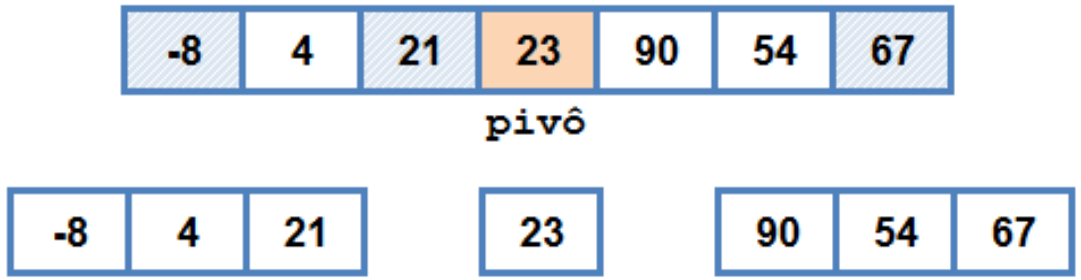
Algoritmo Quick Sort

- Problema da separação
  - Em inglês, *partition subproblem*
  - Consiste em rearranjar o array usando um valor como **pivô**
    - Valores menores do que o **pivô** ficam a esquerda
    - Valores maiores do que o **pivô** ficam a direita



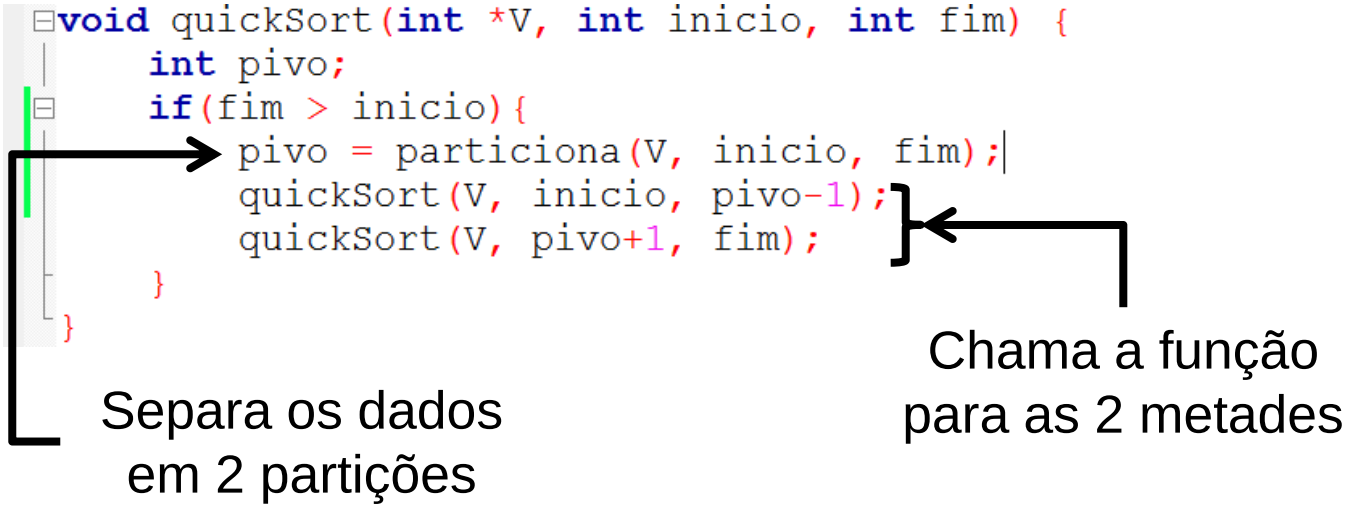
Algoritmo Quick Sort

- Funcionamento
  - Um elemento é escolhido como pivô
  - Valores menores do que o pivô são colocados antes dele e os maiores, depois
    - Supondo o pivô na posição **X**, esse processo cria duas partições:
    - **[0,...,X-1]** e **[X+1,...,N-1]**.
  - Aplicar recursivamente a cada partição
    - Até que cada partição contenha um único elemento



Algoritmo Quick Sort

- Algoritmo usa 2 funções
  - quickSort : divide os dados em arrays cada vez menores
  - particiona: calcula o pivô e rearranja os dados



Algoritmo Quick Sort

- Algoritmo

```
19  int particiona(int *V, int inicio, int final ){
20      int esq, dir, pivo, aux;
21      esq = inicio;
22      dir = final;
23      pivo = V[inicio];
24      while(esq < dir){
25          while(esq <= final && V[esq] <= pivo)
26              esq++;
27
28          while(dir >= 0 && V[dir] > pivo)
29              dir--;
30
31          if(esq < dir){
32              aux = V[esq];
33              V[esq] = V[dir];
34              V[dir] = aux;
35          }
36      }
37      V[inicio] = V[dir];
38      V[dir] = pivo;
39      return dir;
40  }
```

Avança posição da esquerda

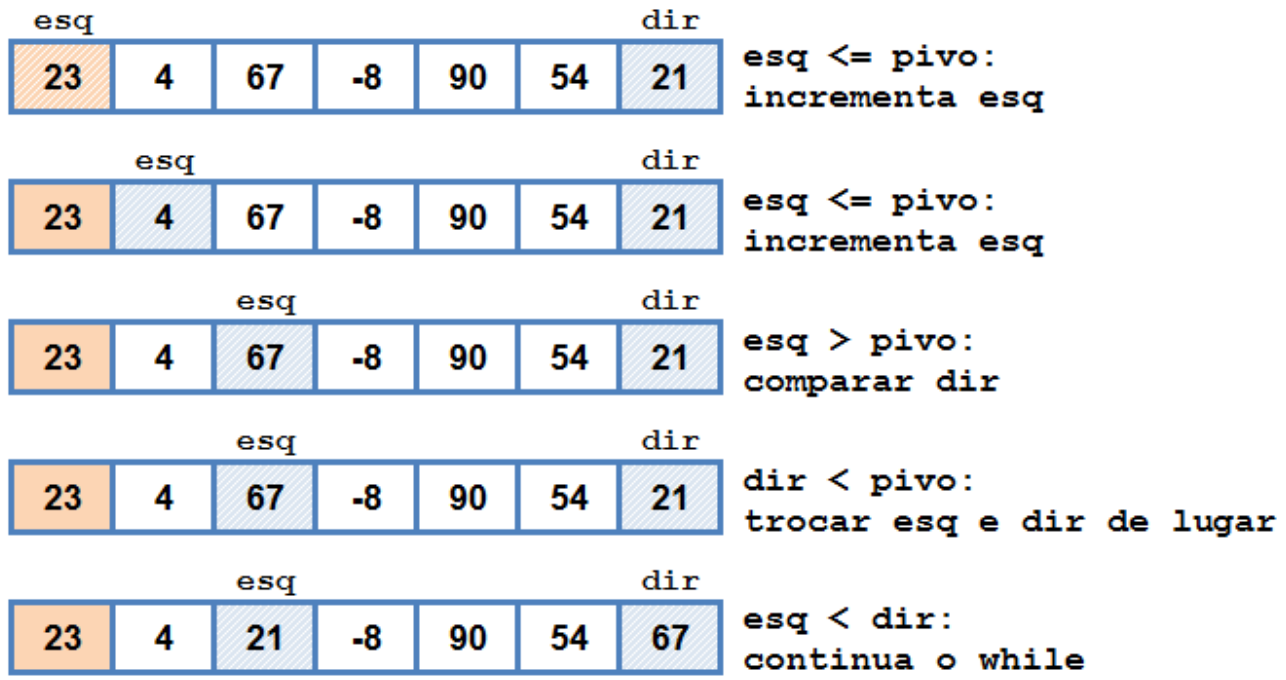
Recua posição da direita

Trocar esq e dir

Algoritmo Quick Sort | Passo a passo

- Função particiona

particiona(V,0,6)



Primeira chamada

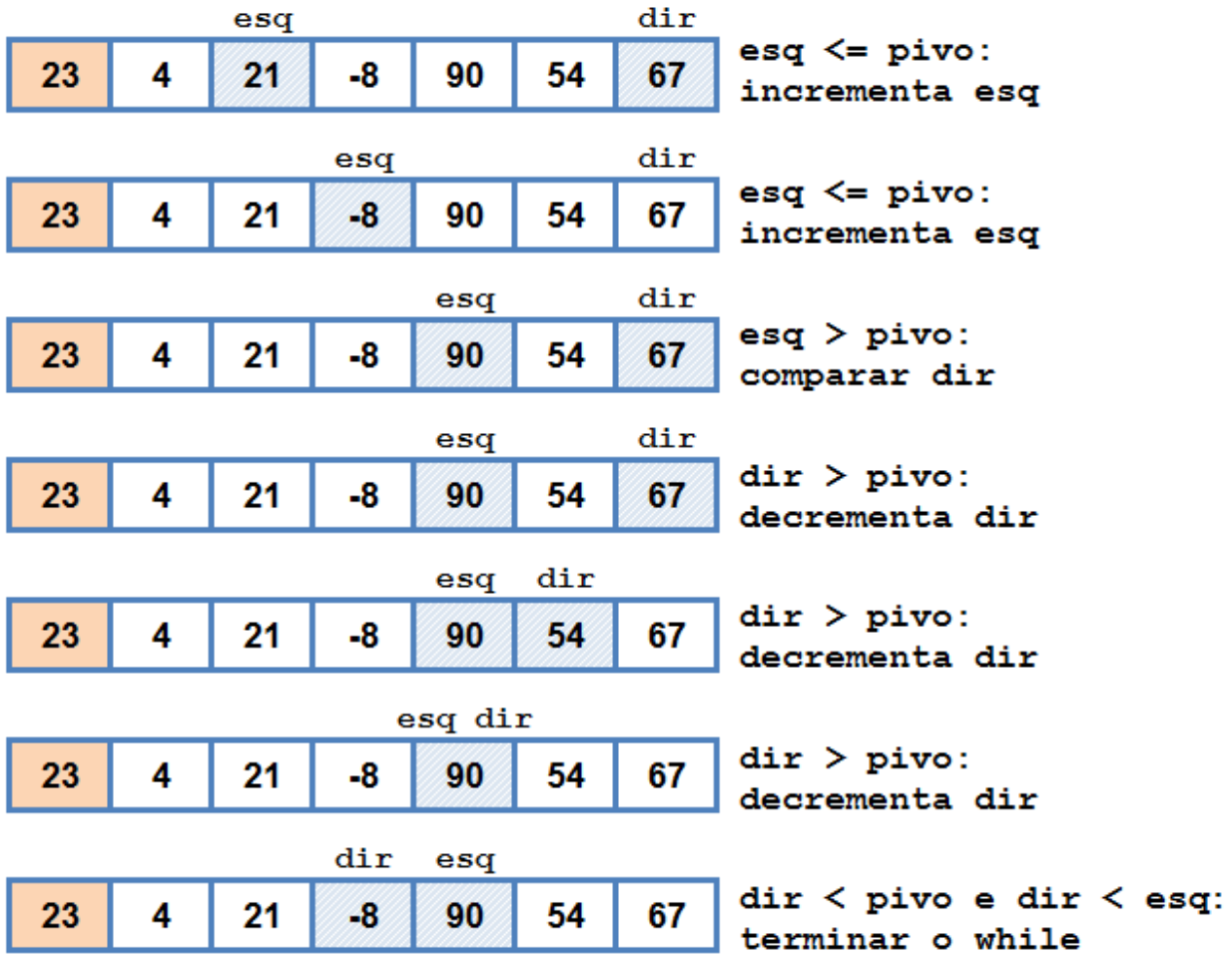
`while(esq < dir)`

Algoritmo Quick Sort | Passo a passo

- Função particiona

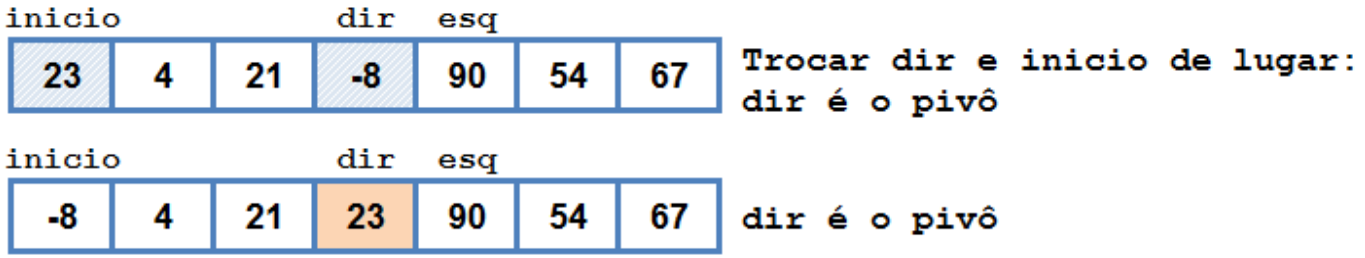
Segunda chamada

`while(esq < dir)`

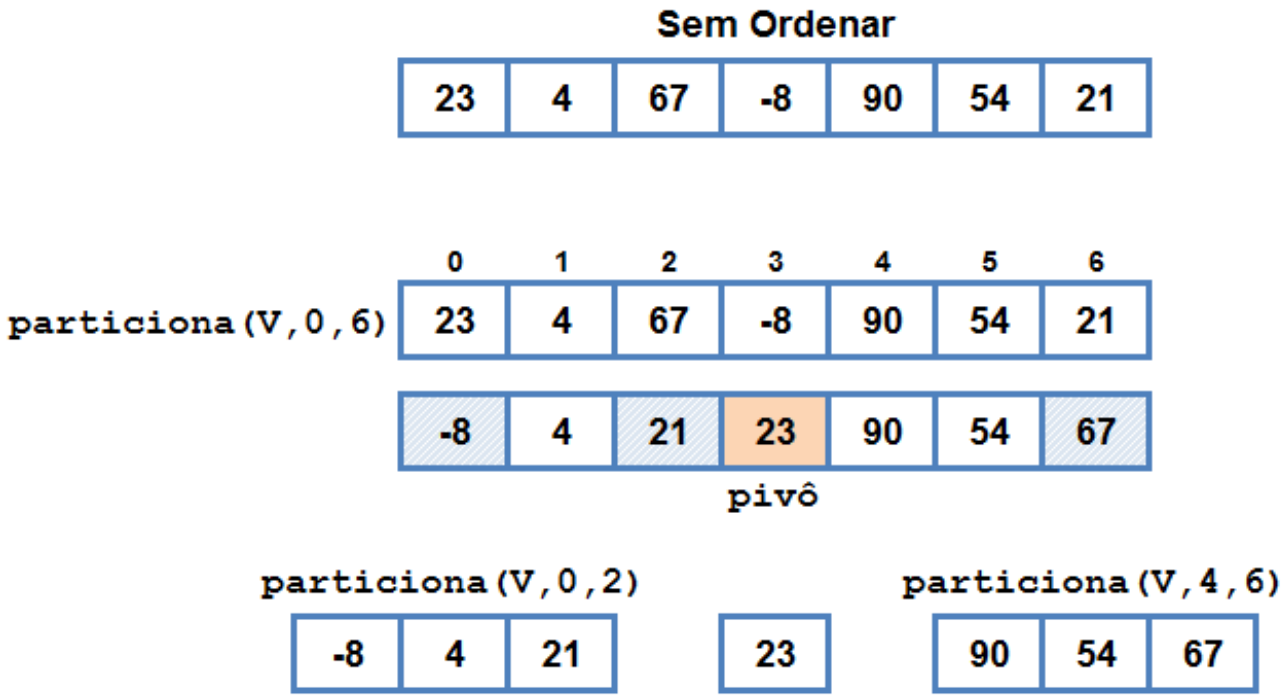


Algoritmo Quick Sort | Passo a passo

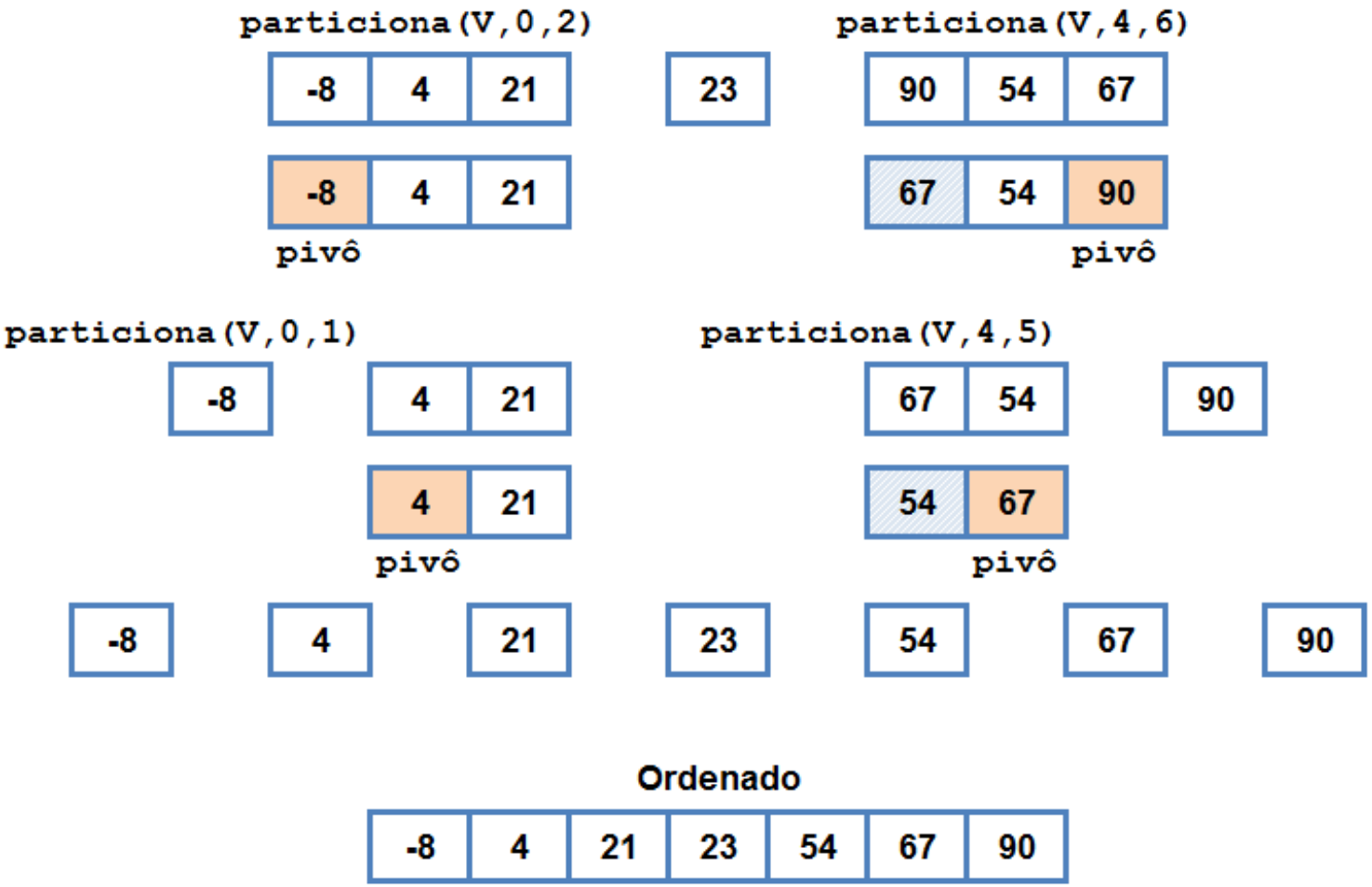
- Função particiona



Algoritmo Quick Sort | Passo a passo



Algoritmo Quick Sort | Passo a passo



Algoritmo Quick Sort

- Complexidade
  - Considerando um array com **N** elementos, o tempo de execução é:
    - ***O(N log N)***, melhor caso e caso médio;
    - ***O(N^2)***, pior caso.
  - Em geral, é algoritmo muito rápido. Porém, é um algoritmo lento em alguns casos especiais
    - Por exemplo, quando o particionamento não é balanceado

Algoritmo Quick Sort

- Desvantagens
  - Não é um algoritmo estável
  - **Como escolher o pivô?**
    - Existem várias abordagens diferentes
    - No pior caso o pivô divide o array de **N** em dois: uma partição com **N-1** elementos e outra com **0** elementos
    - **Particionamento não é balanceado**
    - Quando isso acontece a cada nível da recursão, temos o tempo de execução de $\boldsymbol{O(N^2)}$

Algoritmo Quick Sort

- Desvantagens
  - No caso de um particionamento não balanceado, o insertion sort acaba sendo mais eficiente que o quick sort
    - O pior caso do quick sort ocorre quando o array já está ordenado, uma situação onde a complexidade é ***O(N)*** no insertion sort
- Vantagem
  - Apesar de seu pior caso ser quadrático, costuma ser a melhor opção prática para ordenação de grandes conjuntos de dados

Algoritmo Counting Sort

- Também conhecido como ordenação por contagem
  - Algoritmo de ordenação para valores inteiros
  - Esse valores devem estar dentro de um determinado intervalo
  - A cada passo ele conta o número de ocorrências de um determinado valor no array

Algoritmo Counting Sort

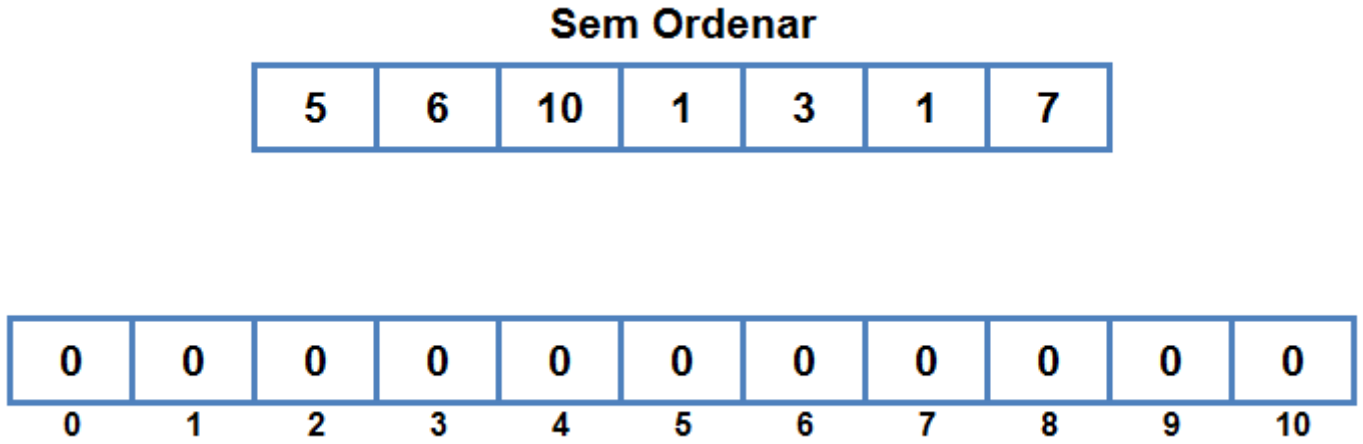
- Funcionamento
  - Usa um array auxiliar de tamanho igual ao maior valor a ser ordenado, **K**
  - O array auxiliar é usado para contar quantas vezes cada valor ocorre
  - Valor a ser ordenado é tratado como índice.
  - Percorre o array auxiliar verificando quais valores existem e os coloca no array ordenado

Algoritmo Counting Sort

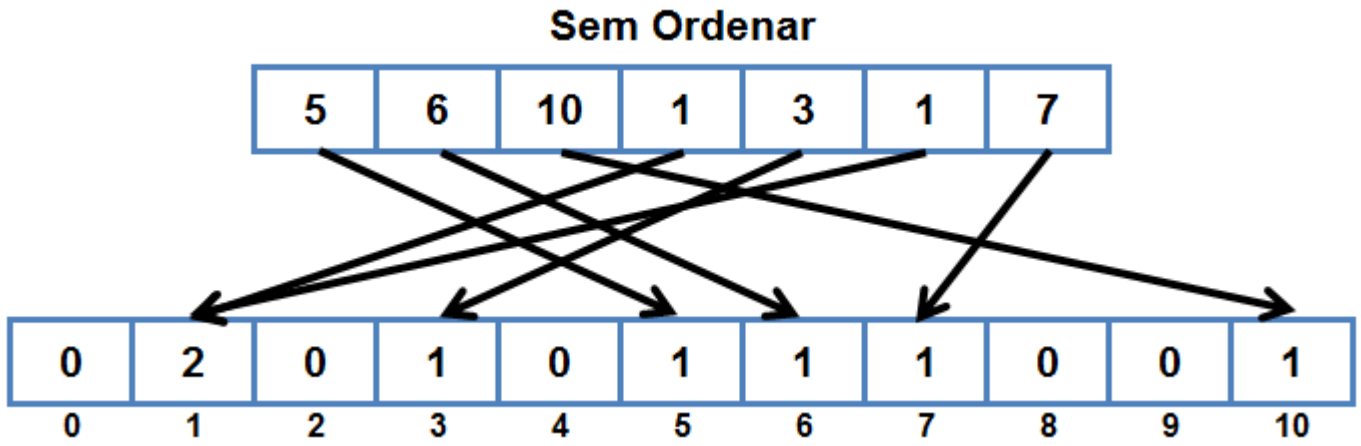
- Algoritmo

```
#define K 100
void countingSort(int *V, int N){
    int i, j, k;
    int baldes [K];
    for(i = 0; i < K; i++)
        baldes[i] = 0;
    for(i = 0; i < N; i++)
        baldes[V[i]]++;

    for(i = 0, j = 0; j < K; j++)
        for(k = baldes[j]; k > 0; k--)
            V[i++] = j;
}
```

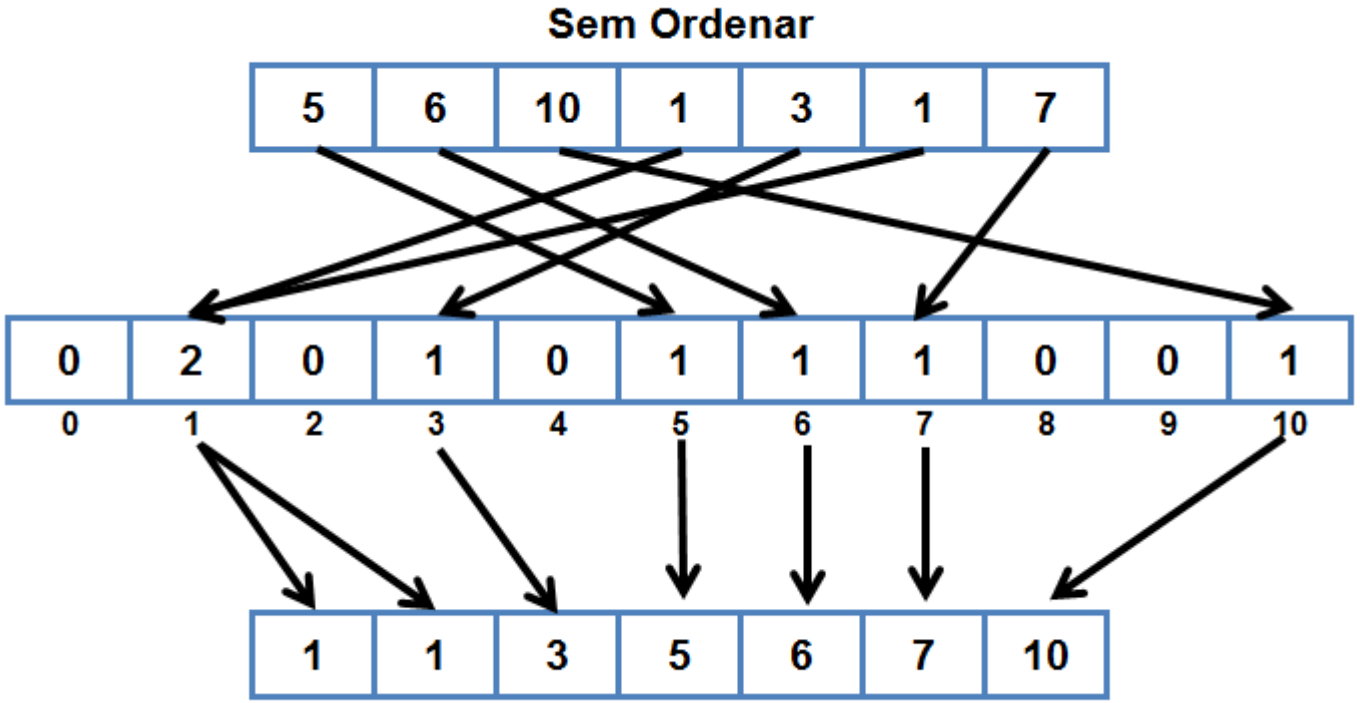
Algoritmo Counting Sort | Passo a passo



Algoritmo Counting Sort | Passo a passo



Algoritmo Counting Sort | Passo a passo



Algoritmo Counting Sort

- Complexidade
  - Complexidade linear
  - Considerando um array com **N** elementos e o maior valor sendo **K**, o tempo de execução é sempre de ordem ***O(N+K)***
  - **K** é o tamanho do array auxiliar

Algoritmo Counting Sort

- Vantagem
  - Estável: não altera a ordem dos dados iguais
  - Processamento simples

- Desvantagens
  - Não recomendado para grandes conjuntos de dados (**K** muito grande)
  - Ordena valores inteiros positivos (pode ser modificado para outros valores)

Algoritmo Bucket Sort

- Também conhecido como ordenação usando baldes
  - Algoritmo de ordenação para valores inteiros
  - Usa um conjunto de **K** baldes para separar os dados
  - A ordenação dos valores é feita por balde

Algoritmo Bucket Sort

- Funcionamento
  - Distribui os valores a serem ordenados em um conjunto de baldes.
    - Cada balde é um array auxiliar
    - Cada balde guarda uma faixa de valores
  - Ordena os valores de cada balde.
    - Isso é feito usando outro algoritmo de ordenação ou ele mesmo
  - Percorre os baldes e coloca os valores de cada balde de volta no array ordenado

Algoritmo Bucket Sort

- Algoritmo

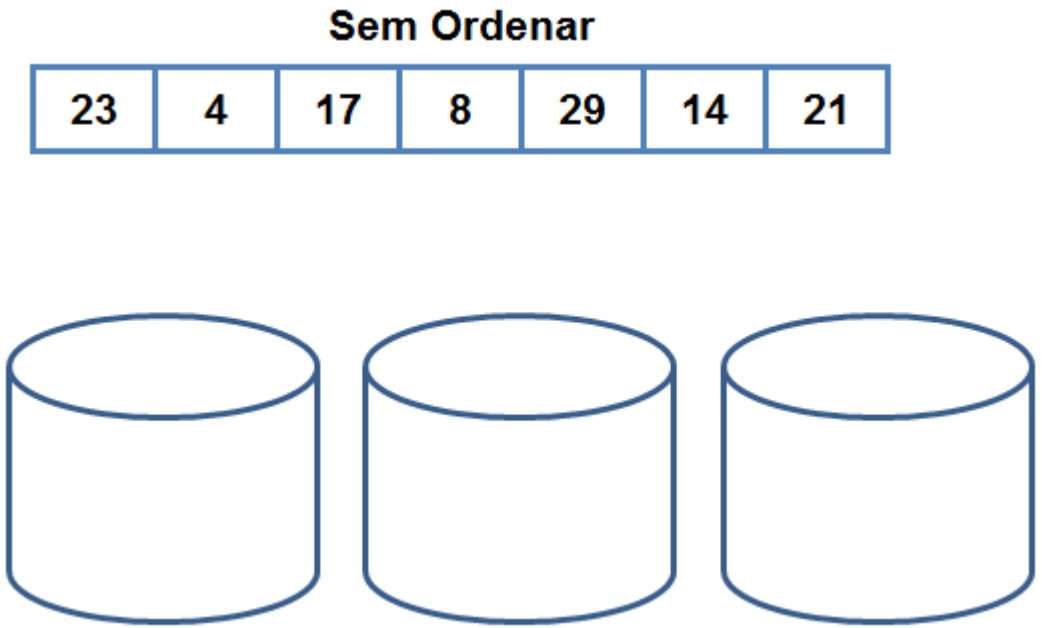
```
#define TAM 5 // tamanho do balde
struct balde{
    int qtd;
    int valores[TAM];
};

void bucketSort(int *V, int N){
    int i, j, maior, menor, nroBaldes, pos;
    struct balde *bd;
    // Acha maior e menor valor
    maior = menor = V [0];
    for(i = 1; i < N; i++) {
        if(V[i] > maior) maior = V[i];
        if(V[i] < menor) menor = V[i];
    }
    // Inicializa baldes
    nroBaldes = (maior - menor) / TAM + 1;
    bd = (struct balde *) malloc(nroBaldes * sizeof(struct balde));
    for(i = 0; i < nroBaldes; i++)
        bd[i].qtd = 0;
```

Algoritmo Bucket Sort

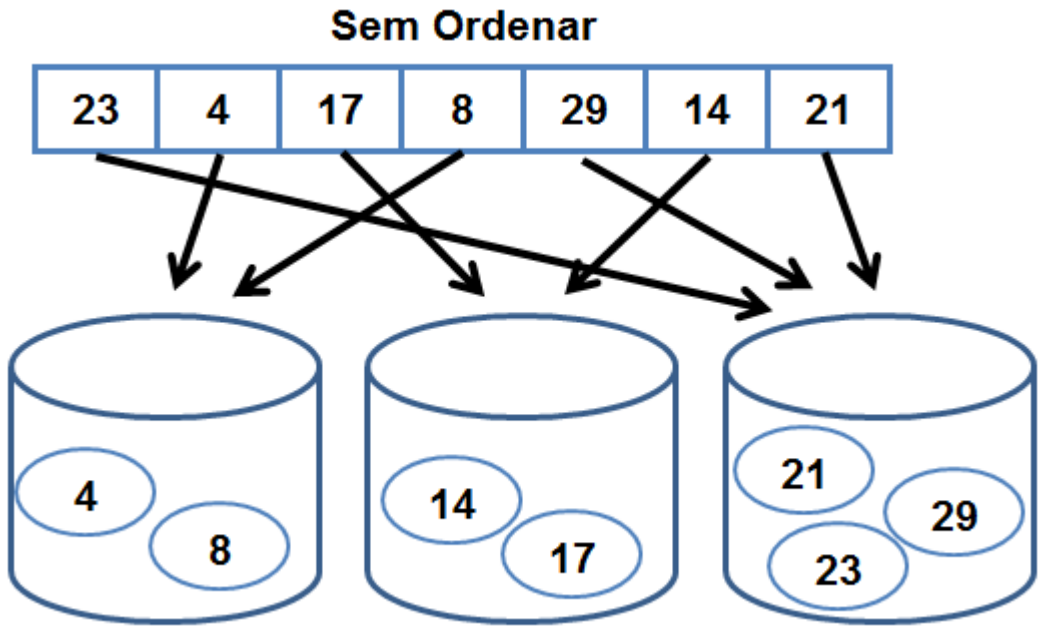
- Algoritmo

```
        // Distribui os valores nos baldes
        for(i = 0; i < N; i++){
            pos = (V[i] - menor)/ TAM;
            bd[pos].valores[bd[pos].qtd] = V[i];
            bd[pos].qtd++;
        }
        // Ordena baldes e coloca no array
        pos = 0;
        for(i = 0; i < nroBaldes; i++){
            insertionSort(bd[i].valores,bd[i].qtd);
            for (j = 0; j < bd[i].qtd; j++){
                V[pos] = bd[i].valores[j];
                pos++;
            }
        }
        free(bd);
}
```

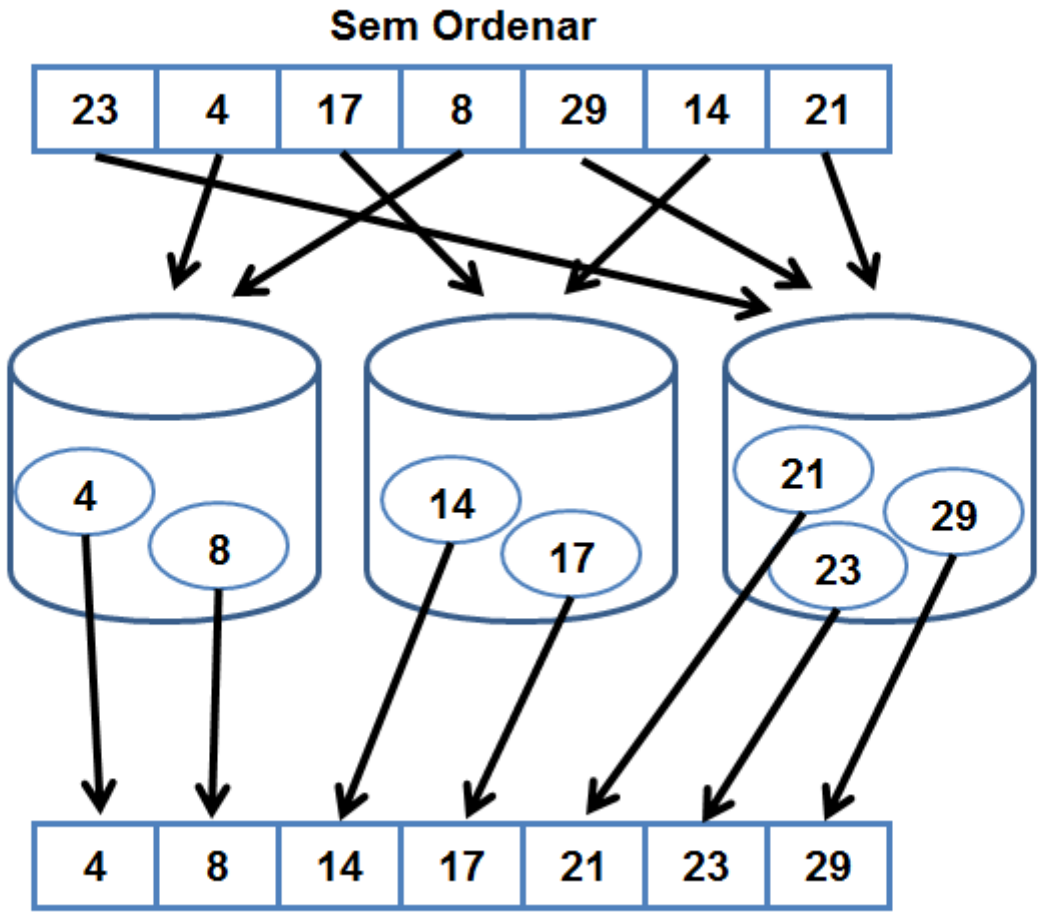
Algoritmo Bucket Sort | Passo a passo



Algoritmo Bucket Sort | Passo a passo



Algoritmo Bucket Sort | Passo a passo



Algoritmo Bucket Sort

- Vantagem
  - Estável: não altera a ordem dos dados iguais
    - Exceto se usar um algoritmo não estável nos baldes
  - Processamento simples
  - Parecido com o Counting Sort
    - Mas com baldes mais sofisticados
- Desvantagens
  - Dados devem estar uniformemente distribuídos
  - Não recomendado para grandes conjuntos de dados
  - Ordena valores inteiros positivos (pode ser modificado para outros valores)

Algoritmo Bucket Sort

- Complexidade
  - Considerando um array com **N** elementos e **K** baldes, o tempo de execução é
  - ***O(N+K)***, melhor caso: dados estão uniformemente distribuídos
  - ***O(N^2)***, pior caso: todos os elementos são colocados no mesmo balde

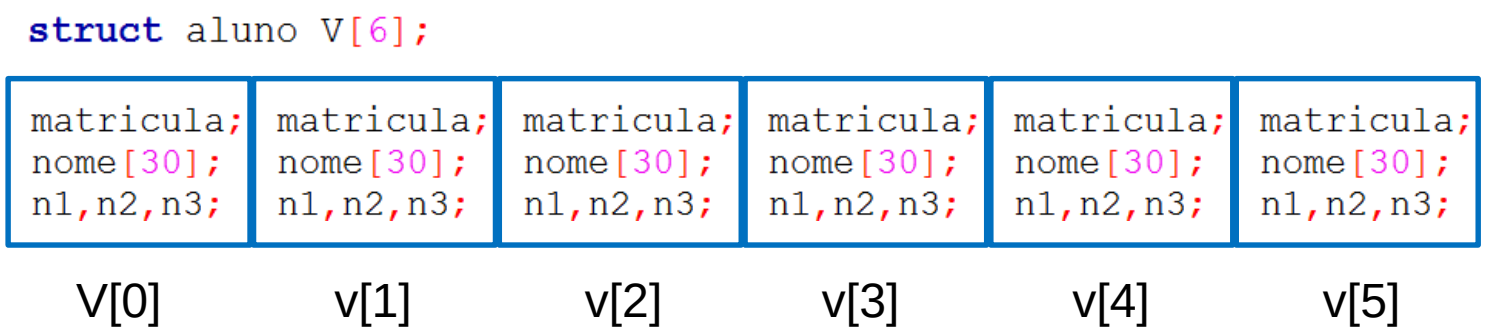
Ordenação de array de struct

- A ordenação de um array de inteiros é uma tarefa simples
  - Na prática, trabalhamos com dados um pouco mais complexos, como estruturas
  - Mais dados para manipular

```
11  struct aluno{
12      int matricula;
13      char nome[30];
14      float n1,n2,n3;
15  };
```

Ordenação de array de struct

- Como fazer a ordenação quando o que temos é um array de struct?



Ordenação de array de struct

- **Relembrando**
- A ordenação é baseada em uma chave
  - A chave de ordenação é o **campo** do item utilizado para comparação
    - Valor armazenado em um array de inteiros
    - **Campo de uma struct**
    - etc
  - É por meio dela que sabemos se um determinado elemento está a frente ou não de outros no conjunto

Ordenação de array de struct

- Ou seja, devemos modificar o algoritmo para que a comparação das chaves seja feita utilizando um determinado campo da **struct**
- Exemplo
  - Vamos modificar o **insertion sort**
    - Essa modificação vale para os outros métodos

```
62  void insertionSort(int *V, int N){
63      int i, j, aux;
64      for(i = 1; i < N; i++){
65          aux = V[i];
66          for(j = i; (j > 0) && (aux < V[j - 1]); j--)
67              V[j] = V[j - 1];
68          V[j] = aux;
69      }
70  }
```

Ordenação de array de struct

- Duas novas formas de ordenação
  - Por **matricula**

```
void insertionSortMatricula(struct aluno *V, int N){
    int i, j;
    struct aluno aux;
    for(i = 1; i < N; i++){
        aux = V[i];
        for(j=i;(j>0) && (aux.matricula<V[j-1].matricula);j--)
            V[j] = V[j - 1];
        V[j] = aux;
    }
}
```

Ordenação de array de struct

- Duas novas formas de ordenação
  - Por **nome**

```
/*saída strcmp(str1,str2)
     == 0: str1 é igual a str2
      > 0: str1 vem depois de str2
      < 0: str1 vem antes de str2
*/
void insertionSortNome(struct aluno *V, int N){
     int i, j;
     struct aluno aux;
     for(i = 1; i < N; i++){
          aux = V[i];
          for(j=i;(j>0) && (strcmp(aux.nome,V[j-1].nome)<0);j--)
               V[j] = V[j-1];
          V[j] = aux;
     }
}
```

Material Complementar | Vídeo Aulas

- Aula 47: Ordenação de Vetores:
  - youtu.be/vPHHV6iAU2E
- Aula 48: Ordenação: BubbleSort:
  - youtu.be/qU8N_bmebQ4
- Aula 49: Ordenação: InsertionSort:
  - youtu.be/79buQYoWszA
- Aula 50: Ordenação: SelectionSort:
  - youtu.be/zjcGGqskf5s
- Aula 51: Ordenação: MergeSort:
  - youtu.be/RZbg5oT5Fgw
- Aula 52: Ordenação: QuickSort:
  - youtu.be/spywQ2ix_Co

Material Complementar | Vídeo Aulas

- Aula 53: Ordenação: HeapSort:
  - youtu.be/zSYOMJ1E52A
- Aula 54: Ordenação em Vetor de Struct:
  - youtu.be/LFs-sIQesVw
- Aula 55: Ordenação – Usando a função qsort():
  - youtu.be/HtvfgqO0IM4
- Aula 123 - Ordenação: CountingSort:
  - youtu.be/En8daEdcpJU
- Aula 124 - Ordenação: BucketSort:
  - youtu.be/4J89y2Pv_qM

Material Complementar | Vídeo Aulas

- Aula 140 - Ordenação: ShellSort:
  - youtu.be/aiafbYZB7S4
- Aula 143 - Ordenação estável: exemplo de uso
  - youtu.be/aEE_hGEV92Q
- Bubble Sort: sorting color lines
  - youtu.be/qNs61uMKBlg
- Selection Sort: sorting color lines
  - youtu.be/e85-oWQQFqk
- Insertion Sort: sorting color lines
  - youtu.be/n5g-knVw2lo

Material Complementar | Vídeo Aulas

- Merge Sort: sorting color lines
  - youtu.be/lzXeeGAdXWc
- Quick Sort: sorting color lines
  - youtu.be/0bwUNoTXVmQ
- Heap Sort: sorting color lines
  - youtu.be/fuUr3y8jcP4
- Sorting colors with 6 algorithms
  - youtu.be/IMZqil4TLU0
- Sorting Mona Lisa with 6 algorithms
  - youtu.be/1oZcfj39F6o

Material Complementar | Vídeo Aulas

- Aula 45: Busca em Vetores:
  - youtu.be/ptvnLzqcJuA
- Aula 46: Busca em Vetor de struct:
  - youtu.be/zxwCSxbntKA

Material Complementar | GitHub

- https://github.com/arbackes