ALGORITMOS DE ORDENAÇÃO

Prof. André Backes

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

3

- 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

4

- 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

5

- 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)
    - Ana, André, Bianca, Ricardo

Conceitos básicos

6

- Independente do tipo, a ordenação pode ser
  - Crescente
    - 1, 2, 3, 4, 5
    - Ana, André, Bianca, Ricardo
  - Decrescente
    - 5, 4, 3, 2, 1
    - Ricardo, Bianca, André, Ana

Conceitos básicos

7

- 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

8

- 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

9

- 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

10

- Exemplo
  - 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

11

- 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

12

- 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

13

- 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

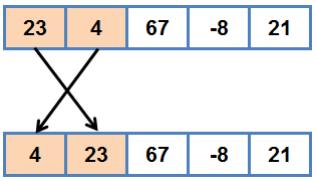
Algoritmo Bubble Sort

14

- 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  }
```

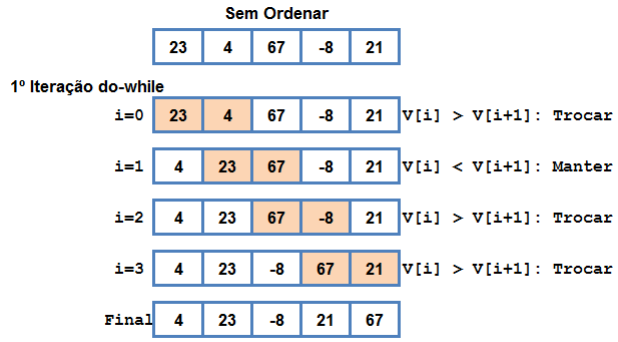
Troca dois valores consecutivos no vetor



Algoritmo Bubble Sort

15

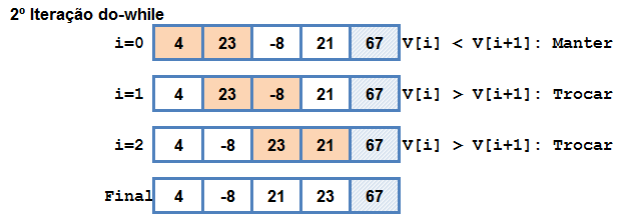
- Passo a passo
  - 1º iteração do-while: encontra o maior valor e o movimenta até a última posição



Algoritmo Bubble Sort

16

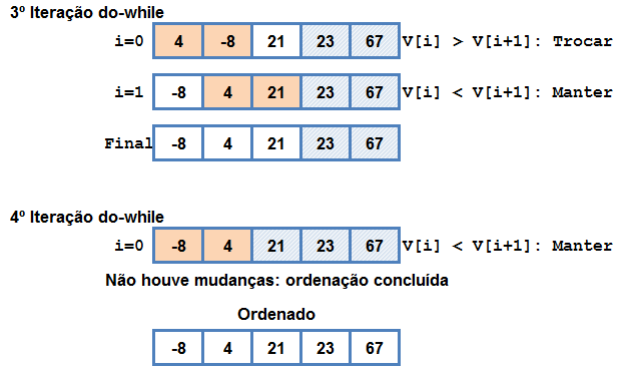
- Passo a passo
  - 2º iteração do-while: encontra o segundo maior valor e o movimenta até a penúltima posição



Algoritmo Bubble Sort

17

- Passo a passo
  - Processo continua até todo o array estar ordenado



Algoritmo Bubble Sort

18

- 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

19

- 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

20

- 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

21

- 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

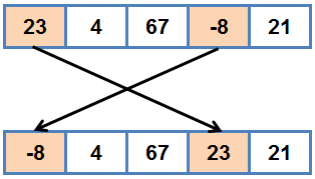
22

- 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"

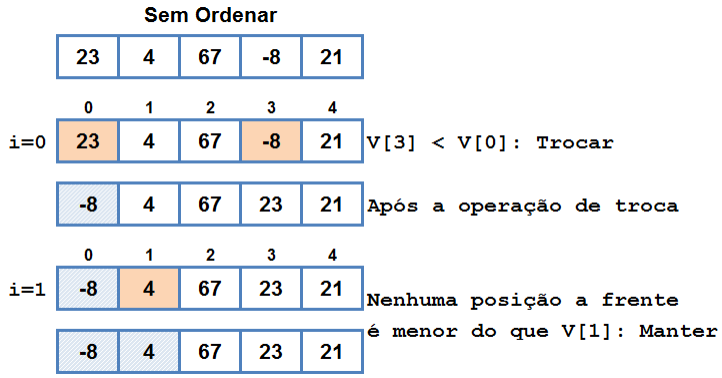
Troca os valores da posição atual com a "menor"



Algoritmo Selection Sort

23

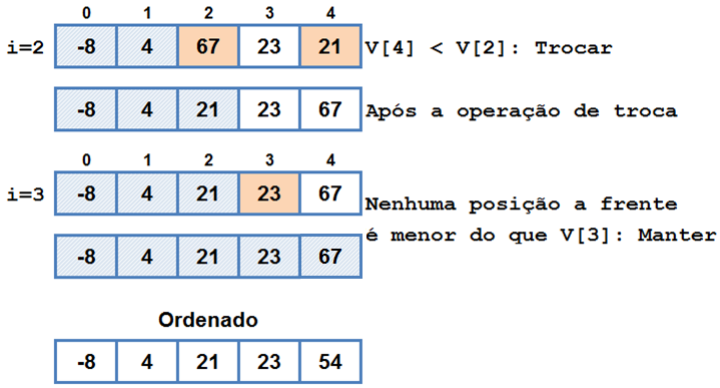
- Passo a passo
  - Para cada posição **i**, procura no restante do array o menor valor para ocupá-la



Algoritmo Selection Sort

24

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



Algoritmo Selection Sort

25

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

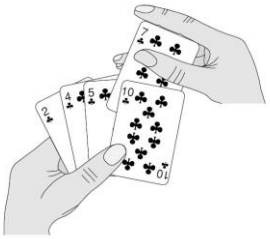
26

- 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

27

- 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

28

- 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

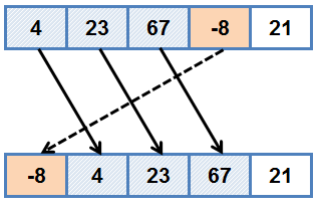
Algoritmo Insertion Sort

29

- 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  }
```

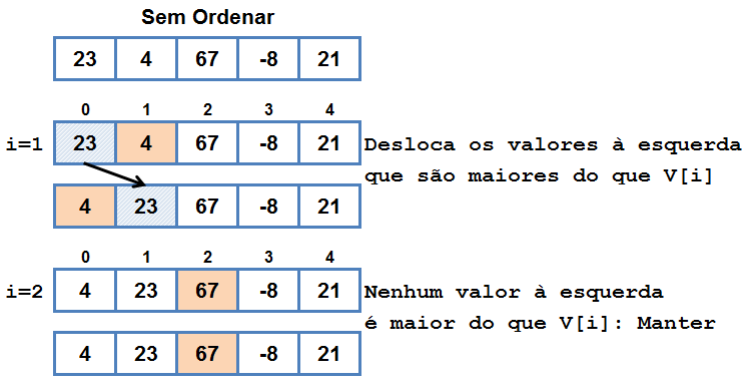
Move as cartas maiores para frente e insere na posição vaga



Algoritmo Insertion Sort

30

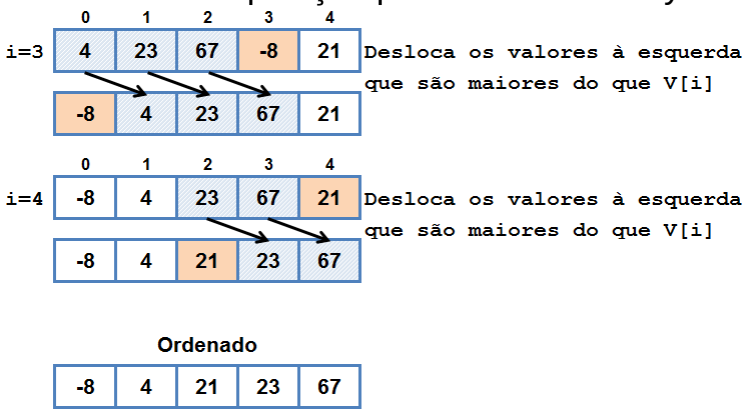
- Passo a passo
  - Para cada posição **i**, movimenta os valores maiores uma posição para frente no array



Algoritmo Insertion Sort

31

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



Algoritmo Insertion Sort

32

- 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

33

- 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

34

- 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

35

- 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

36

- 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

37

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

```
23
24  void mergeSort(int *V, int inicio, int fim){
25      int meio;
26      if(inicio < fim){
27          meio = floor((inicio+fim)/2);
28          mergeSort(V,inicio,meio);
29          mergeSort(V,meio+1,fim);
30          merge(V,inicio,meio,fim);
31      }
32  }
```

Chama a função para as 2 metades

Combina as 2 metades de forma ordenada

Algoritmo Merge Sort

38

- 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);
}
```

Combinar ordenando

Vetor acabou?

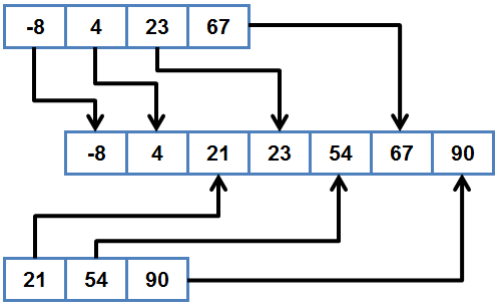
Copia o que sobrar

Copiar do auxiliar para o original

Algoritmo Merge Sort

39

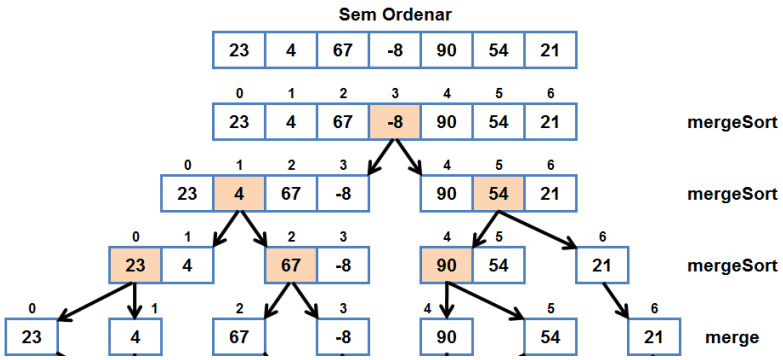
- Passo a passo: função merge
  - Intercala os dados de forma ordenada em um array maior
  - Utiliza um array auxiliar



Algoritmo Merge Sort

40

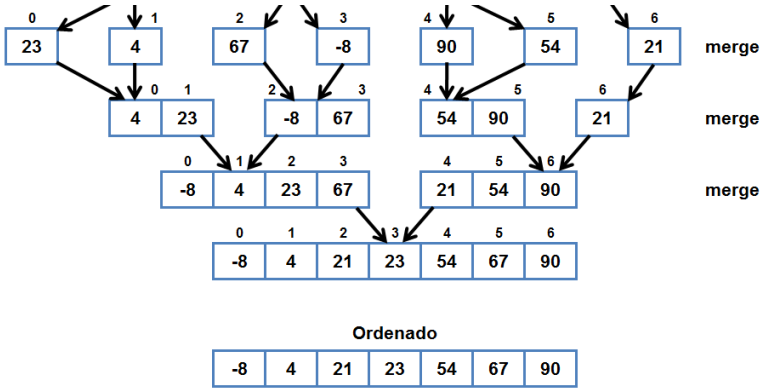
- Passo a passo
  - Divide o array até ter **N** arrays de 1 elemento cada



Algoritmo Merge Sort

41

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



Algoritmo Merge Sort

42

- 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

43

- 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

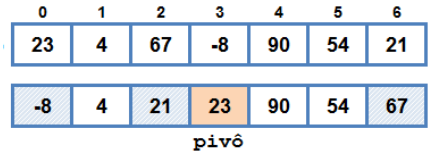
44

- 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

45

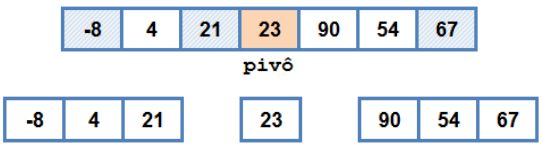
- Problema da separação
  - Em inglês, *partition subproblem*
  - Consiste em rearranjar o array usando um valor como **pivô**
    - Valores menores do que o **pivô** ficam a esquerda
    - Valores maiores do que o **pivô** ficam a direita



Algoritmo Quick Sort

46

- 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

47

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

```
void quickSort(int *V, int inicio, int fim) {
    int pivo;
    if(fim > inicio){
        pivo = particiona(V, inicio, fim);
        quickSort(V, inicio, pivo-1);
        quickSort(V, pivo+1, fim);
    }
}
```

Separa os dados em 2 partições

Chama a função para as 2 metades

Algoritmo Quick Sort

48

- 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

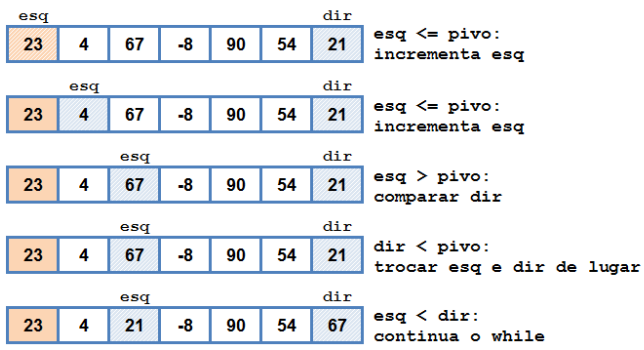
Trocar esq e dir

Algoritmo Quick Sort

49

- Passo a passo: função particiona

particiona(V,0,6)



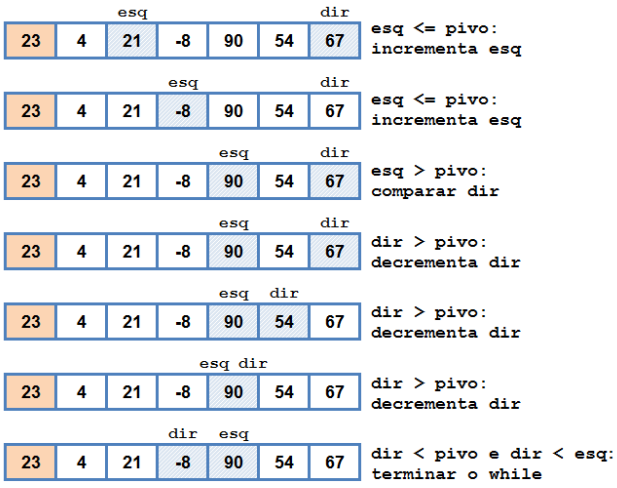
Primeira chamada

`while(esq < dir)`

Algoritmo Quick Sort

50

- Passo a passo: função particiona



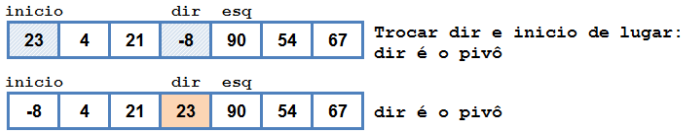
Segunda chamada

`while(esq < dir)`

Algoritmo Quick Sort

51

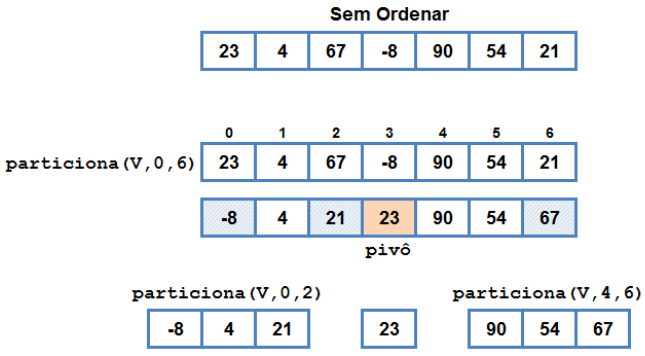
- Passo a passo: função particiona



Algoritmo Quick Sort

52

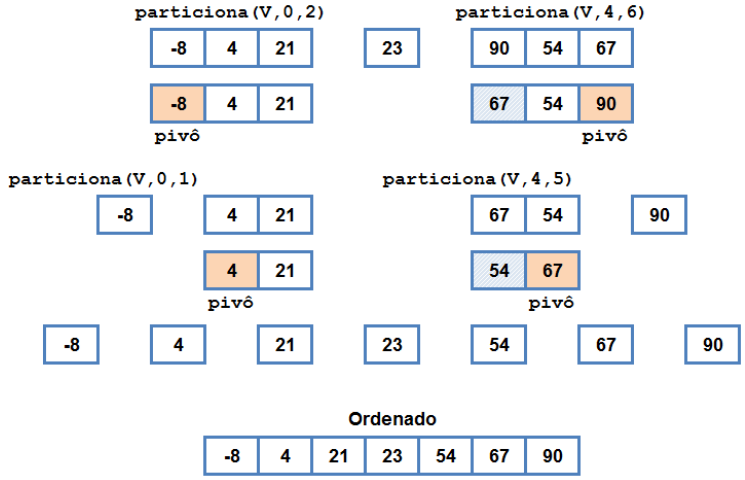
- Passo a passo



Algoritmo Quick Sort

53

- Passo a passo



Algoritmo Quick Sort

54

- 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

55

- 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 ***O(N^2)***

Algoritmo Quick Sort

56

- 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

57

- 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

58

- 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

59

- 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

60

- Passo a passo

Sem Ordenar

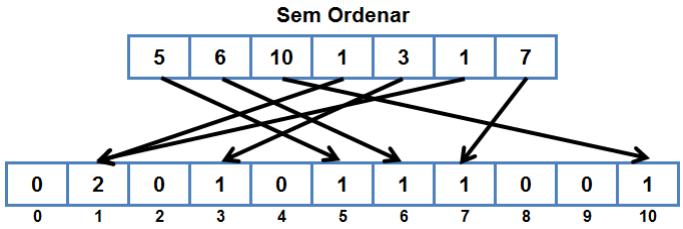
5	6	10	1	3	1	7

0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10

Algoritmo Counting Sort

61

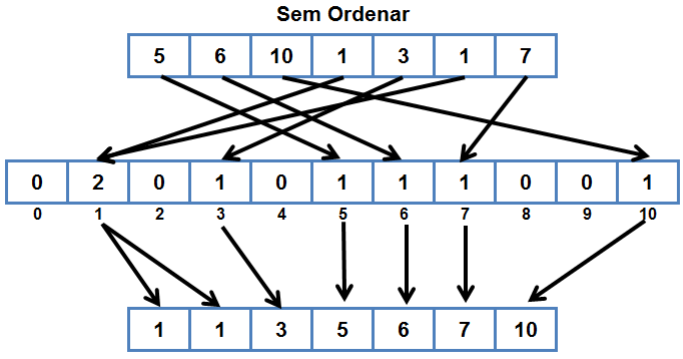
- Passo a passo



Algoritmo Counting Sort

62

- Passo a passo



Algoritmo Counting Sort

63

- 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

64

- 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

65

- 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

66

- 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

67

- 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

68

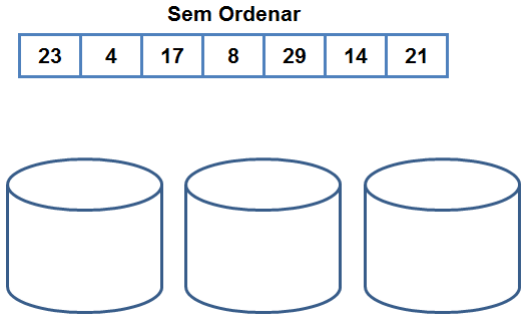
- Algoritmo

```
    // 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

69

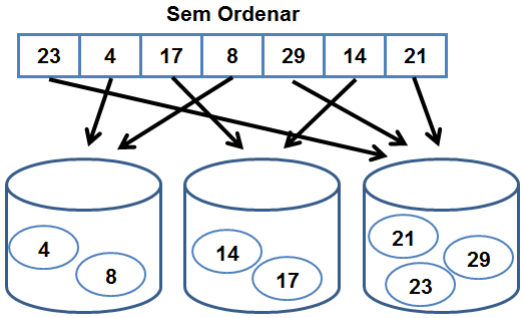
- Passo a passo



Algoritmo Bucket Sort

70

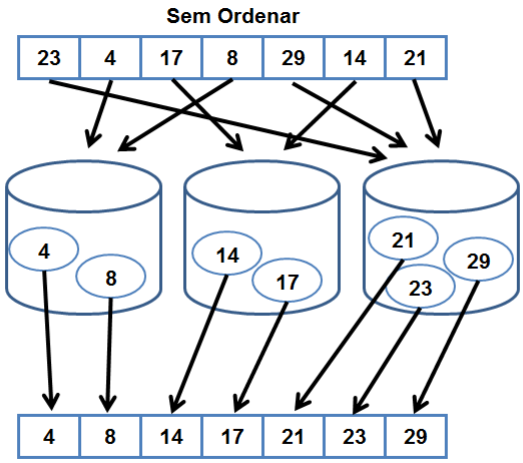
- Passo a passo



Algoritmo Bucket Sort

71

- Passo a passo



Algoritmo Bucket Sort

72

- 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

73

- 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

74

- 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

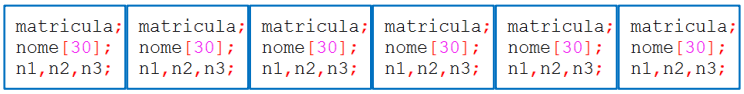
```
11  struct aluno{
12      int matricula;
13      char nome[30];
14      float n1,n2,n3;
15  };
```

Ordenação de array de struct

75

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

```
struct aluno V[6];
```



V[0] v[1] v[2] v[3] v[4] v[5]

Ordenação de array de struct

76

- **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

77

- 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

78

- 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

79

- 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

80

- 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

81

- 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