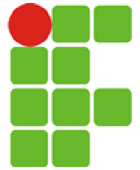
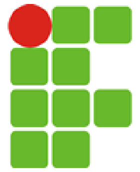
Estruturas de Dados



Ordenação

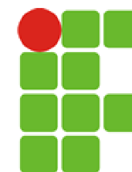
Prof. MSc. Raphael Gomes
raphael@ifg.edu.br



O que veremos hoje...

- *Conceitos sobre ordenação de dados*
- *Ordenação por Seleção*
- *Ordenação por Inserção*
- *Ordenação por Troca e Partição*

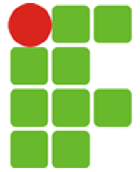
Por que ordenar?



- *O que é ordenação?*
  - Ordenação é o processo de organização de um conjunto de informações semelhantes em ordem crescente ou decrescente.
- *O conceito de um conjunto de elementos ordenado tem considerável impacto sobre nossa vida cotidiana!*

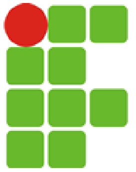
+ 200.000 palavras

Classificação dos Métodos de Ordenação



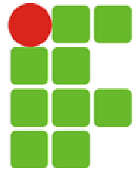
- *Para organizar as informações usamos algoritmos de ordenação.*
- *Os algoritmos atuam em cima dos **registros** de um arquivo ou estrutura.*
- *A comparação é feita através de uma determinada **chave** escolhida.*

Classificação – Quanto à Estabilidade



- ***Métodos Estáveis:*** *a ordem relativa dos itens com chaves iguais mantém-se inalterada durante o processo*
- ***Métodos Instáveis:*** *a ordem relativa dos itens com chaves iguais é alterada durante o processo de ordenação*
  - Ex.: Se uma lista dos funcionários ordenada pelo campo "Nome" é reordenada pelo campo "Salário", um método estável produz uma lista em que os funcionários com o mesmo salário aparecem em ordem alfabética.
- *Alguns dos métodos de ordenação mais eficientes são instáveis*

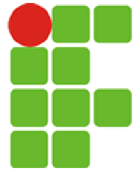
Classificação – Quanto ao Conjunto de Registros



- ***Ordenação Interna:*** *o conjunto de registros cabe todo em na memória principal.*
- ***Ordenação Externa:*** *o conjunto de registros não cabe completamente em memória principal, e deve ser armazenado em disco ou fita.*

  - ✓ Principal diferença: na ordenação interna, o registro pode ser acessado diretamente, enquanto na ordenação externa, o registros são acessados sequencialmente ou em blocos

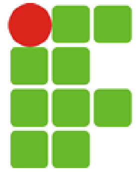
Classificação – Quanto à complexidade



- *Para calcular a complexidade de um algoritmo de ordenação interna são necessários dois itens:*
  - ✓ O número de comparações entre as chaves.
  - ✓ O número de movimentações de itens.

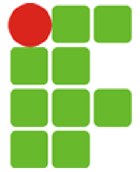
- *Portanto, a complexidade do algoritmo sofre influência dos números de chaves contida nos registros.*

Classificação – Quanto à complexidade



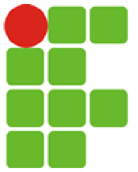
- ***Métodos Simples:*** *mais recomendados para conjuntos pequenos de dados. Usam mais comparações, mas produzem códigos menores e mais simples.*
  - Ordenação por Seleção Direta (SelectionSort)
  - Ordenação por Inserção Direta (InsertionSort)
  - Ordenação por Seleção e Troca – Bolha (BubbleSort)

Classificação – Quanto à complexidade



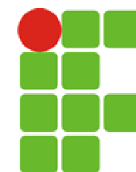
- ***Métodos Eficientes ou Sofisticados:*** *adequados para conjuntos maiores de dados. Usam menos comparações, porém produzem códigos mais complexos e com muitos detalhes.*
  - Ordenação por Intercalação (MergeSort)
  - Ordenação por Seleção em Árvore (HeapSort)
  - Ordenação por Troca e Partição (QuickSort)

Ordenação por Seleção (Selection)



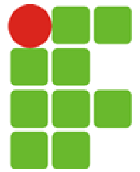
- *A ordenação por seleção consiste em trocar o menor elemento (ou maior) de uma lista com o elemento posicionado no início da lista, depois o segundo menor elemento para a segunda posição e assim sucessivamente com os (n – 1) elementos restantes, até os últimos dois elementos.*
  - Seleciona sempre o elemento chave

Seleção - o que ocorre em cada passo



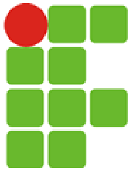
inicial	13	7	5	1	4
passo 1	**1**	7	5	**13**	4
passo 2	1	**4**	5	13	**7**
passo 3	1	4	5	13	7
passo 4	1	4	5	**7**	**13**

Seleção - Implementação



```
typedef int T;

void selection(T piItem[], int iQtdElementos) {
      int i, j, iMinimo;
      T aux;

      /* Para cada elemento do vetor ... */
      for (i = 0; i < iQtdElementos - 1; i++) {
            iMinimo = i;

            /*... verifica os demais para encontrar o menor */
            for (j = i + 1; j < iQtdElementos; j++) {
                  if (piItem[j] < piItem[iMinimo]) {
                        iMinimo = j;
                  }
            }

            /* Faz a troca entre o elemento considerado e o menor
            encontrado */
            aux = piItem[i];
            piItem[i] = piItem[iMinimo];
            piItem[iMinimo] = aux;
      }
}
```

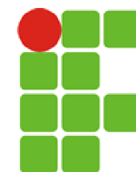
Ordenação por Inserção



- *O algoritmo de inserção funciona da mesma maneira com que muitas pessoas ordenam cartas em um jogo de baralho como o pôquer. Uma das características deste algoritmo é o menor número de trocas e comparações se a lista estiver ordenada (parcialmente).*

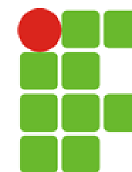
- *O número de comparações é n^2 no pior caso e no melhor caso 2(n - 1) comparações, no caso médio o número de comparações é $n^2/4$*

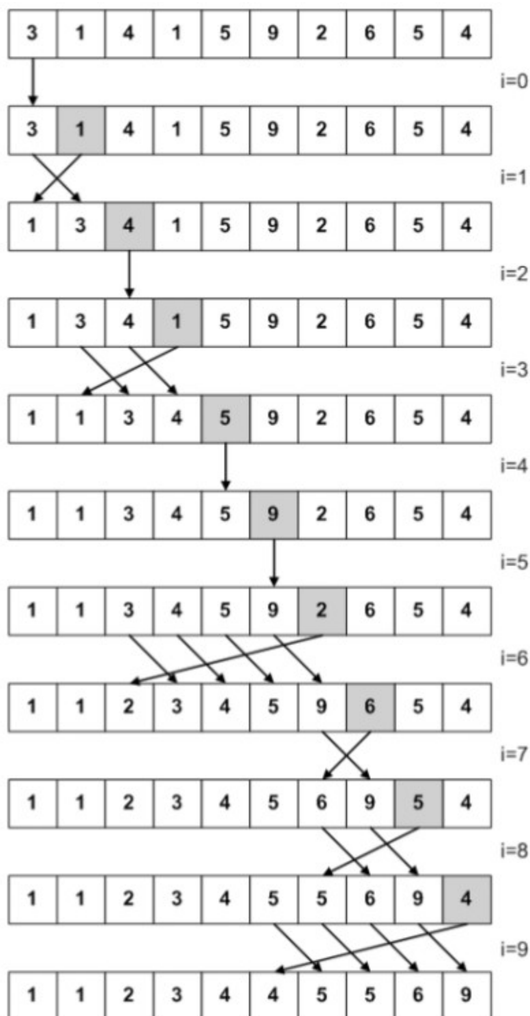
Inserção - o que ocorre em cada passo



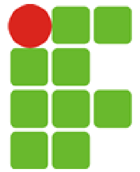
inicial	13	7	5	1	4
passo 1	**7**	**13**	5	1	4
passo 2	**5**	7	13	1	4
passo 3	**1**	5	7	13	4
passo 4	1	**4**	5	7	13

Inserção - o que ocorre em cada passo



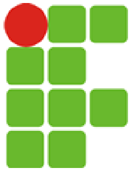


Inserção – Implementação



```
void insertion(T piItem[], int iQtdElementos) {
     int i, j;
     T aux;

     /* Para cada elemento do vetor a partir do segundo... */
     for (i = 1; i < iQtdElementos; i++) {
          j = i;

          /*... insere o elemento na posição correta */
          while (piItem[j] < piItem[j-1] && j > 0) {
               /* Faz a troca entre o elemento considerado e
               seu vizinho */
               aux = piItem[j];
               piItem[j] = piItem[j-1];
               piItem[j-1] = aux;
               j--;
          }
     }
}
```

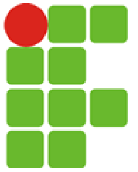
Troca e Partição (QuickSort)



- *O Quicksort é o algoritmo mais rápido para ordenação interna conhecido para uma grande quantidade de situações, sendo por isso o mais utilizado entre todos os algoritmos de ordenação.*

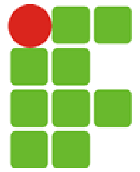
- *Princípio*
  - Dividir para Conquistar
  - Ordenar independentemente os problemas menores
  - Combinar os resultados para produzir a solução do problema maior

QuickSort – Partição

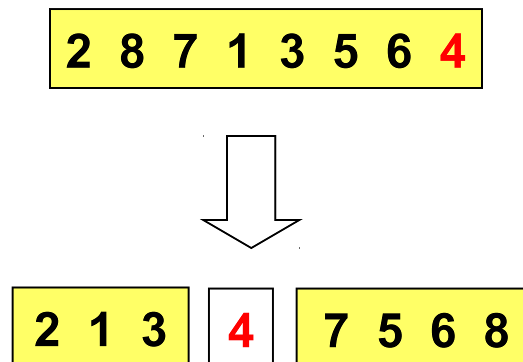


- *A parte mais delicada desse método se refere à divisão da partição*
  - Deve-se rearranjar o vetor na forma A[Esq..Dir] através da escolha arbitrária de um item x do vetor chamado pivô
  - Ao final, o vetor A deverá ter duas partes, uma esquerda com chaves menores ou iguais que x e a direita com valores de chaves maiores ou iguais que x

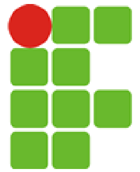
QuickSort – Partição

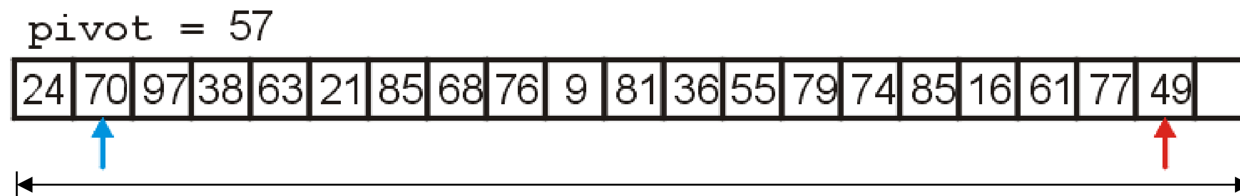


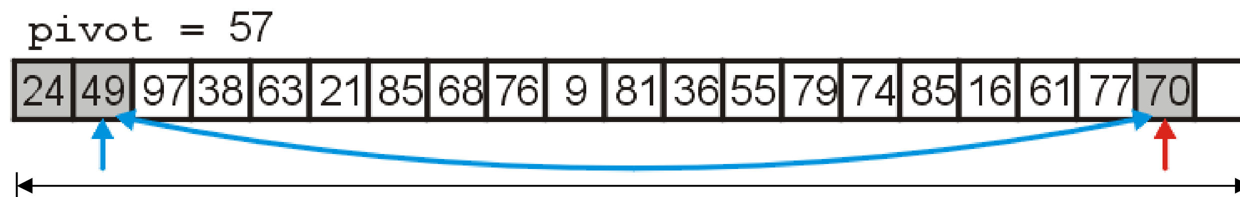
- *Ao final do processo, o vetor A[Esq..Dir] está particionado de tal forma que:*
  - ✓ A[Esq], A[Esq+1],..., A[j] ≤ x
  - ✓ A[i], A[i+1],..., A[Dir] ≥ x

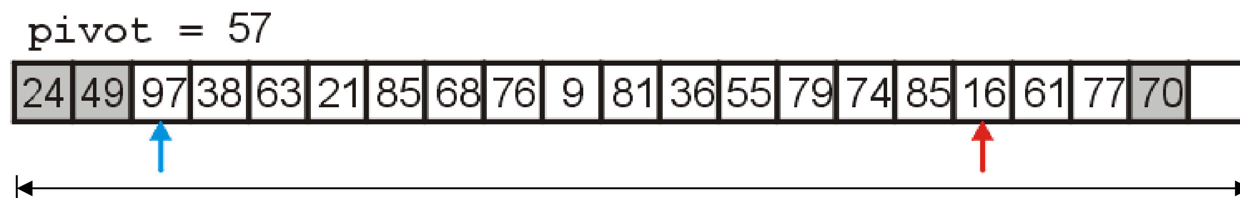


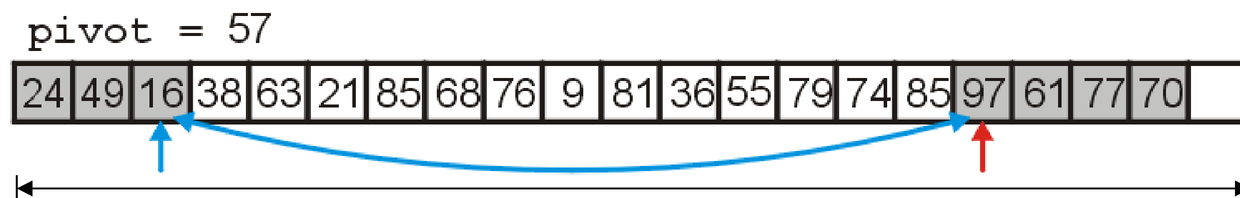
Partição – Exemplo



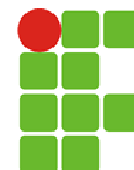








Partição – Exemplo



pivot = 57

24	49	16	38	63	21	85	68	76	9	81	36	55	79	74	85	97	61	77	70	

pivot = 57

24	49	16	38	55	21	85	68	76	9	81	36	63	79	74	85	97	61	77	70	

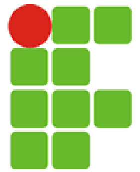
pivot = 57

24	49	16	38	55	21	85	68	76	9	81	36	63	79	74	85	97	61	77	70	

pivot = 57

24	49	16	38	55	21	36	68	76	9	81	85	63	79	74	85	97	61	77	70	

Partição – Exemplo



pivot = 57

24	49	16	38	55	21	36	68	76	9	81	85	63	79	74	85	97	61	77	70	

pivot = 57

24	49	16	38	55	21	36	9	76	68	81	85	63	79	74	85	97	61	77	70	

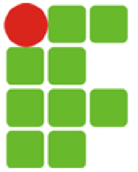
pivot = 57

24	49	16	38	55	21	36	9	76	68	81	85	63	79	74	85	97	61	77	70	

pivot = 57

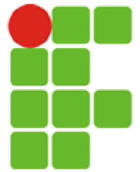
24	49	16	38	55	21	36	9	57	68	81	85	63	79	74	85	97	61	77	70	76

QuickSort

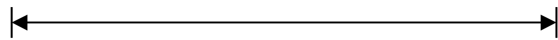


- *Procedimento QuickSort*
  - Escolher arbitrariamente um item x (pivô) do vetor
  - Percorrer o vetor a partir da esquerda até que um item A[i] ≥ x é encontrado; da mesma maneira, percorrer o vetor a partir da direita até que um item A[j] ≤ x é encontrado;
  - Como os itens A[i] e A[j] não estão na ordem correta no vetor final, eles devem ser trocados
  - Continuar o processo até que os índice i e j se cruzem em algum ponto do vetor

Quicksort – Exemplo

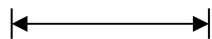


24	49	16	38	55	21	36	9	57	68	81	85	63	79	74	85	97	61	77	70	76

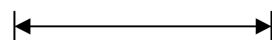


9	21	16	24	55	49	36	38	57	68	81	85	63	79	74	85	97	61	77	70	76

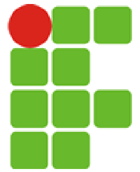
9	16	21	24	55	49	36	38	57	68	81	85	63	79	74	85	97	61	77	70	76

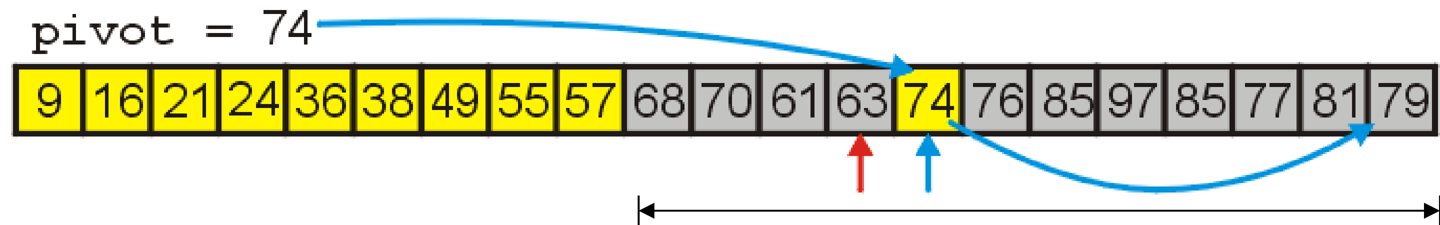


9	16	21	24	36	38	49	55	57	68	81	85	63	79	74	85	97	61	77	70	76



Quicksort – Exemplo



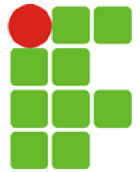


•

•

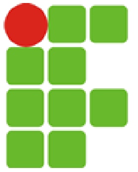
•

Quicksort – Implementação



```
void quicksort(T *item, int left, int right) {
        int i, j;
        T x, y;
        i = left;
        j = right;

        x = item[(left + right) / 2]; /* elemento pivo */

        /* partição das listas */
        do {
                /* procura elementos maiores que o pivô na primeira parte*/
                while (item[i] < x && i < right) {
                        i++;
                }
                /* procura elementos menores que o pivô na segunda parte */
                while (x < item[j] && j > left) {
                        j--;
                }
                if (i <= j) {
                        /* processo de troca (ordenação) */
                        y = item[i];
                        item[i] = item[j];
                        item[j] = y;
                        i++;
                        j--;
                }
        } while (i <= j);

        /* chamada recursiva */
        if (left < j) {
                quicksort(item, left, j);
        }
        if (i < right) {
                quicksort(item, i, right);
        }
}
```

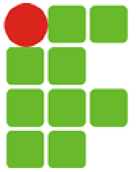
QuickSort – Observações



- *Apesar de possuir complexidade $O(n^2)$ no pior caso (quando o procedimento de particionamento produz um subproblema com n -1 elementos e um com 0 elementos), o QuickSort é considerado o melhor algoritmo de ordenação.*

- *Isso se deve à sua eficiência no melhor e no pior caso $O(n \log n)$.*

Bibliografia



- *M. T. GOODRICH, R. TAMASSIA, Estrutura de Dados e Algoritmos em Java, 2a Ed. Bookman, 2002.*
- *M. LAUREANO, Estrutura de Dados com Algoritmos e C, Brasport, 2008.*
- *CORMEN, Thomas et al. Algoritmos: teoria e prática. Rio de Janeiro: Elsevier, 2002.*
- *http://math.hws.edu/TMCM/java/xSortLab/*