PANC: Projeto e Análise de Algoritmos

Aula 11: Algoritmos de Ordenação II - Troca:
Bubble Sort, Shake Sort, Comb Sort e Quick Sort

Breno Lisi Romano

http://sites.google.com/site/blromano

Instituto Federal de São Paulo – IFSP São João da Boa Vista
Bacharelado em Ciência da Computação – 3º Semestre

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

Sumário

- Revisão de Conteúdo
- Introdução
- Algoritmos de Ordenação por Troca:
  - Bubble Sort
  - Shake Sort
  - Comb Sort
  - Quick Sort
- Exemplos Práticos

Recapitulando...

- Ordenação por Inserção:
  - Caracterizados pelo princípio no qual se divide o array em dois segmentos, sendo um já ordenado e o outro a ser ordenado
  - A partir disto, iterações são desenvolvidas sendo que, em cada uma delas, um elemento do segmento não ordenado é transferido para o segmento ordenado, na sua posição correta

- Métodos de Ordenação por Inserção:
  - Direta: $T(n) = O(n^2)$
  - Binária: $T(n) = \lg (n-1)!$
  - Shell Sort: $T(n)$ = Conjectura 1: $O(n^{1,25})$ e Conjectura 2: $O(n (\lg n)^2)$

Métodos de Ordenação por Troca

- Diferentemente dos Métodos de Ordenação por Inserção, os Métodos de Ordenação por Troca fazem a permutação dos valores de um array para buscar a ordenação do mesmo

- Os dois principais métodos por troca são:
  - Bubble Sort
  - Quick Sort

- Mas existem outros:
  - Shake Sort
  - Comb Sort

Detalhando...

ALGORITMOS DE ORDENAÇÃO POR TROCA

Bubble Sort, Shake Sort, Comb Sort e Quick Sort

Ordenação por Troca: Bubble Sort

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

Met. Ord. Troca: Bubble Sort (1)

- Este método efetuará tantas varreduras no array quantas forem necessárias para que todos os pares consecutivos se apresentem na ordem desejada
- O método é bastante trivial e, de forma geral, muito lento na ordenação
- O nome bolha deve-se ao fato de que, ao se fazer o acompanhamento até sua correta posição, como bolhas em um líquido, elas vão, naturalmente, se organizando
  - Cada bolha teria um diâmetro proporcional ao seu valor do array. Assim, as bolhas maiores subiriam com velocidades maiores, o que faria com que, após um certo tempo, elas se arranjassem em ordem de tamanho

Met. Ord. Troca: Bubble Sort (2)

- O algoritmo básico desse processo de ordenação é apresentado a seguir:

1. Em cada passo, cada elemento é comparado com o próximo
2. Se o elemento estiver fora de ordem, a troca é realizada
3. Realizam-se tantos passos quantos forem necessários, até que não ocorram mais trocas

Met. Ord. Troca: Bubble Sort (3)

- Suponha que se deseja classificar o seguinte array:

i	0	1	2	3	4	5	6	7
Vet[i]	44	55	12	42	94	18	06	67

- Comparamos todos os pares consecutivos, a partir do par mais a esquerda. Caso os elementos de um certo par se encontrarem fora da ordem desejada, efetuamos a troca das mesmas, de posição

Primeira Varredura

0	1	2	3	4	5	6	7	Resultado
44	55	12	42	94	18	06	67	Compara (44,55) => Não Troca
44	55	12	42	94	18	06	67	Compara (55,12) => Troca
44	12	55	42	94	18	06	67	Compara (55,42) => Troca
44	12	42	55	94	18	06	67	Compara (55,94) => Não Troca
44	12	42	55	94	18	06	67	Compara (94,18) => Troca
44	12	42	55	18	94	06	67	Compara (94,06) => Troca
44	12	42	55	18	06	94	67	Compara (94,67) => Troca
44	12	42	55	18	06	67	94	Fim da Primeira Varredura

Met. Ord. Troca: Bubble Sort (4)

- Como ainda existem pares desordenados, reiniciamos o processo desconsiderando a última posição do array, pois nele já se encontra o maior elemento

Segunda Varredura

0	1	2	3	4	5	6	7	Resultado
44	12	42	55	18	06	67	94	Compara (44,12) => Troca
12	44	42	55	18	06	67	94	Compara (44,42) => Troca
12	42	44	55	18	06	67	94	Compara (44,55) => Não Troca
12	42	44	55	18	06	67	94	Compara (55,18) => Troca
12	42	44	18	55	06	67	94	Compara (55,06) => Troca
12	42	44	18	06	55	67	94	Compara (55,67) => Não Troca
12	42	44	18	06	55	67	94	Fim da Segunda Varredura

Met. Ord. Troca: Bubble Sort (5)

- O processo se repete até ordenar todo o array, desconsiderando as últimas posições já ordenadas

Terceira Varredura

0	1	2	3	4	5	6	7	Resultado
12	42	44	18	06	55	67	94	Compara (12,42) => Não Troca
12	42	44	18	06	55	67	94	Compara (42,44) => Não Troca
12	42	44	18	06	55	67	94	Compara (44,18) => Troca
12	42	18	44	06	55	67	94	Compara (44,06) => Troca
12	42	18	06	44	55	67	94	Compara (44,55) => Não Troca
12	42	18	06	44	55	67	94	Fim da Terceira Varredura

Quarta Varredura

0	1	2	3	4	5	6	7	Resultado
12	42	18	06	44	55	67	94	Compara (12,42) => Não Troca
12	42	18	06	44	55	67	94	Compara (42,18) => Troca
12	18	42	06	44	55	67	94	Compara (42,06) => Troca
12	18	06	42	44	55	67	94	Compara (42,44) => Não Troca
12	18	06	42	44	55	67	94	Fim da Quarta Varredura

Met. Ord. Troca: Bubble Sort (6)

Quinta Varredura

0	1	2	3	4	5	6	7	Resultado
12	18	06	42	44	55	67	94	Compara (12,18) => Não Troca
12	18	06	42	44	55	67	94	Compara (18,06) => Troca
12	06	18	42	44	55	67	94	Compara (18,42) => Não Troca
12	06	18	42	44	55	67	94	Fim da Quinta Varredura

Sexta Varredura

0	1	2	3	4	5	6	7	Resultado
12	06	18	42	44	55	67	94	Compara (12,06) => Troca
06	12	18	42	44	55	67	94	Compara (12,18) => Não Troca
06	12	18	42	44	55	67	94	Fim da Sexta Varredura

Sétima Varredura

0	1	2	3	4	5	6	7	Resultado
06	12	18	42	44	55	67	94	Compara (06,12) => Não Troca
06	12	18	42	44	55	67	94	Fim da Sétima Varredura

- Como não existe mais pares ordenados, o processo finaliza no sétimo passo!

Met. Ord. Troca: Bubble Sort (7)

```
/*OrdenaBubbleSort(): Ordena a Sequencia de Numeros pelo Método de Ordenação de
                      Permutação das Chaves, também chamado Bubble Sort. Este algoritmo
                      empurra o maior valor para sua posição final. */
void OrdenaBubbleSort(int Vet[])
{
   int aux, i, j;

   for(j=N-1; j>=1; j--)
   {
      for(i=0; i<j; i++)
      {
         if(Vet[i]>Vet[i+1])
         {
            aux = Vet[i];
            Vet[i] = Vet[i+1];
            Vet[i+1] = aux;
         }
      }
   }
}
```

Met. Ord. Troca: Bubble Sort (8)

- Simule a ordenação do seguinte vetor utilizando o Bubble Sort:

i	0	1	2	3	4	5	6	7
Vet[i]	25	48	37	12	57	86	33	92

Solução:

Iteração	0	1	2	3	4	5	6	7
1	25	37	12	48	57	33	86	92
2	25	12	37	48	33	57	86	
3	12	25	37	33	48	57		
4	12	25	33	37	48			
5	12	25	33	37				
6	12	25	33					
7	12	25						
Vetor Ordenado	12	25	33	37	48	57	86	92

Met. Ord. Troca: Bubble Sort Melhorado (1)

- Existem algumas melhorias que podem ser efetuadas no algoritmo original do Bubble Sort
- Uma técnica óbvia para melhorar este algoritmo consiste em manter uma indicação informando se houve ou não a ocorrência de uma troca, para determinar, precocemente, o término do algoritmo
- Como ficaria???

Met. Ord. Troca: Bubble Sort Melhorado (2)

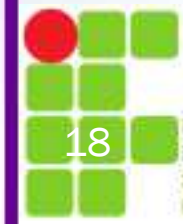
```
/*OrdenaBubbleSort(): Ordena a Sequencia de Numeros pelo Método de Ordenação de
                      Permutação das Chaves, também chamado Bubble Sort,
                      mas termina a execução quando nenhuma troca é realizada
                      após uma passada pelo vetor. */
void OrdenaBubbleSortMelhorado(int Vet[])
{
   int memoria, troca, i, j;

   troca=1; /*A variável "troca" será a verificação da troca em cada passada*/

   for(j=N-1; (j>=1) && (troca==1); j--)
   {
      troca=0; /*Se o valor continuar 0 na próxima passada quer dizer que não houve troca e a função é encerrada.*/
      for(i=0; i<j; i++)
      {
         if(Vet[i]>Vet[i+1])
         {
               memoria = Vet[i];
               Vet[i] = Vet[i+1];
               Vet[i+1] = memoria;
               troca=1; /*Se houve troca, "troca" recebe 1 para continuar rodando.*/
         }
      }
   }
}
```

Ordenação por Bubble Sort: Análise da Complexidade

- Principais pontos a se destacar:
  - Simples de entender e implementar
  - Uma desvantagem é que na prática ele tem execução lenta mesmo quando comparado a outros algoritmos quadráticos (n^2)
  - Tem um número muito grande de movimentação de elementos, assim não deve ser usado se a estrutura a ser ordenada for complexa

- Análise da Complexidade do Algoritmo:
  - **Qualquer caso (Pior, Melhor ou Médio):**
    - Definida pelo número de comparações envolvendo a quantidade de dados do Array
    - Número de comparações: (n-1) + (n-2) + (n-3) + ... + 2 + 1
    - Complexidade T(n): $\sum_{i}^{n-1} i = \frac{(n-1)n}{} \rightarrow O(n^2)$

Ordenação por Troca: Shake Sort



Met. Ord. Troca: Shake Sort (1)

- Um aperfeiçoamento do método Bubble Sort já melhorado ainda pode ser feito

- Podemos, ao final de cada varredura feita da direita para a esquerda, realizar outra da esquerda para a direita, de tal modo que tanto o maior como o menor elemento sejam deslocados para suas posições finais
  - Este método é chamado de Shake Sort, ou seja, ordenação por agitação

Met. Ord. Troca: Shake Sort (2)

- O procedimento básico desse processo de ordenação é apresentado a seguir:
  1. Em cada passo:
     - Cada elemento é comparado com o próximo, da esquerda para a direita, entre L(esquerda) e R (direita). Se o elemento estiver fora de ordem, a troca é realizada
     - Cada elemento é comparado com o próximo, da direita para a esquerda, entre R e L. Se o elemento estiver fora de ordem, a troca é realizada.
  2. Realizam-se tantos passos quantos forem necessários, até que não hajam trocas.

- Lembre-se que os elementos já colocados em ordem não precisam mais ser objetos de comparação

Met. Ord. Troca: Shake Sort (3)

- Suponha que se deseja classificar, em ordem crescente, o seguinte array:

i	0	1	2	3	4	5	6	7
Vet[i]	44	55	12	42	94	18	06	67
	⬆ L							⬆ R

- Inicialmente, comparamos todos os pares consecutivos, a partir do par mais a esquerda, com início em L. Caso os elementos se encontrarem fora da ordem desejada, os elementos são trocados de posição.

Met. Ord. Troca: Shake Sort (4)

i	0	1	2	3	4	5	6	7
Vet[i]	44	55	12	42	94	18	06	67

L R

Primeira Varredura – Esquerda para Direita

0	1	2	3	4	5	6	7	Resultado
44	55	12	42	94	18	06	67	Compara (44,55) => Não Troca
44	55	12	42	94	18	06	67	Compara (52,12) => Troca
44	12	55	42	94	18	06	67	Compara (55,42) => Troca
44	12	42	55	94	18	06	67	Compara (55,94) => Não Troca
44	12	42	55	94	18	06	67	Compara (94,18) => Troca
44	12	42	55	18	94	06	67	Compara (94,06) => Troca
44	12	42	55	18	06	94	67	Compara (94,67) => Troca
44	12	42	55	18	06	67	94	Fim da Primeira Varredura

Met. Ord. Troca: Shake Sort (5)

i	0	1	2	3	4	5	6	7
Vet[i]	44	12	42	55	18	06	67	94

L

R sempre aponta para a última posição que aconteceu uma troca com L

R

Primeira Varredura – Direita para Esquerda

0	1	2	3	4	5	6	7	Resultado
44	12	42	55	18	06	67	94	Compara (67,06) => Não Troca
44	12	42	55	18	06	67	94	Compara (06,18) => Troca
44	12	42	55	06	18	67	94	Compara (06,55) => Troca
44	12	42	06	55	18	67	94	Compara (06,42) => Troca
44	12	06	42	55	18	67	94	Compara (06,12) => Troca
44	06	12	42	55	18	67	94	Compara (06,44) => Troca
06	44	12	42	55	18	67	94	Fim da Primeira Varredura

Met. Ord. Troca: Shake Sort (6)

i	0	1	2	3	4	5	6	7
Vet[i]	06	44	12	42	55	18	67	94
		L					R	

Segunda Varredura – Esquerda para Direita

0	1	2	3	4	5	6	7	Resultado
06	44	12	42	55	18	67	94	Compara (44,12) => Troca
06	12	44	42	55	18	67	94	Compara (44,42) => Troca
06	12	42	44	55	18	67	94	Compara (44,55) => Não Troca
06	12	42	44	55	18	67	94	Compara (55,18) => Troca
06	12	42	44	18	55	67	94	Compara (55,67) => Não Troca
06	12	42	44	18	55	67	94	Fim da Segunda Varredura

Met. Ord. Troca: Shake Sort (7)

i	0	1	2	3	4	5	6	7
Vet[i]	06	12	42	44	18	55	67	94

L

R

R sempre aponta para a última posição que aconteceu uma troca com L

Segunda Varredura – Esquerda para Direita

0	1	2	3	4	5	6	7	Resultado
06	12	42	44	18	55	67	94	Compara (18,44) => Troca
06	12	42	18	44	55	67	94	Compara (18,42) => Troca
06	12	18	42	44	55	67	94	Compara (18,12) => Não Troca
06	12	18	42	44	55	67	94	Fim da Segunda Varredura

Met. Ord. Troca: Shake Sort (8)

i	0	1	2	3	4	5	6	7
Vet[i]	06	12	18	42	44	55	67	94
			↑ L		↑ R			

Terceira Varredura – Esquerda para Direita

0	1	2	3	4	5	6	7	Resultado
06	12	18	42	44	55	67	94	Compara (18,42) => Não Troca
06	12	18	42	44	55	67	94	Compara (42,44) => Não Troca
06	12	18	42	44	55	67	94	Fim da Terceira Varredura

- Na Terceira Varredura, nas comparações feitas da esquerda para a direita, nenhuma troca foi realizada. Isto significa que os dados já estão na ordem desejada e que, portanto, não é mais necessário realizar operações de comparação

Met. Ord. Troca: Shake Sort (9)

```
/*OrdenaShakeSort(): Ordena a Sequencia de Numeros pelo Método de Ordenação por
                     Agitação das Chaves, também chamado Shake Sort.*/
void OrdenaShakeSort(int Vet[])
{
    //Variaveis Locais
  int L = 0, R = Max-1, k = Max-1;
  int j, x;

  do
  {
    //Caminhando da Esquerda para a Direita no Vetor
    for(j=L; j<R; j++)
    {
       if(Vet[j] > Vet[j+1])
       {
         x = Vet[j];
         Vet[j] = Vet[j+1];
         Vet[j+1] = x;
         k = j;
       }
    }

    R = k;
    //Caminhando da Direita para a Esquerda no Vetor
    for(j=R; j>L; j--)
    {
      if(Vet[j-1] > Vet[j])
       {
         x = Vet[j-1];
         Vet[j-1] = Vet[j];
         Vet[j] = x;
         k = j;
       }
    }
    L = k;
  }while(L<R);
}
```

Entender o Funcionamento é Mais Importante que Decorar o Algoritmo!!!

Met. Ord. Troca: Shake Sort (10)

- Simule a ordenação do seguinte vetor utilizando o Shake Sort:

i	0	1	2	3	4	5	6	7
Vet[i]	25	48	37	12	57	86	33	92

Solução:

Iteração	0	1	2	3	4	5	6	7
1 (E->D)	25	37	12	48	57	33	86	92
1 (D->E)	12	25	33	37	48	57	86	92
2 (E->D)	12	25	33	37	48	57	86	92
Vetor Ordenado	12	25	33	37	48	57	86	92

Ordenação por Shake Sort: Análise da Complexidade

- Principais pontos a se destacar:
  - Para Knuth (1973), o ganho obtido com o algoritmo não é significativo
  - É considerado mais vantajoso quando se sabe, a priori que os elementos já estão em ordem, o que é um caso relativamente raro na prática

- Análise da Complexidade do Algoritmo:
  - Para Knuth (1973), a análise de desempenho do Shake Sort, a rigor, é a mesma do Bubble Sort:
    - $T(n) = O(n^2)$

Ordenação por Troca: Comb Sort

Met. Ord. Troca: Combo Sort (1)

- Um ganho expressivo no método Bubble Sort pode ser obtido usando uma estratégia de promover os elementos do array em direção as suas posições definitivas por saltos maiores que um.
- Comparar pares não consecutivos de elementos, localizados a uma distância h
  - h = N/1.3
- Quando h=10 ou h=9, deve-se utilizar h = 11 (tudo baseado em experimentos)

Met. Ord. Troca: Combo Sort (2)

```
/*OrdenaCombSort(): Ordena a Sequencia de Numeros pelo Método de Ordenação
                    conhecido como CombSort. */
void OrdenaCombSort(int Vet[])
{
   //Variaveis Locais
  double h = N;
  int x, i, Troca;

  do
  {
    h=h/1.3;
    if((h == 9)||(h == 10)) h=11;
    Troca=0;
    for(i=0; i<(N-h); i++)
    {
      if(Vet[(int)i] > Vet[(int)(i+h)])
      {
              x=Vet[i];
              Vet[(int)i]=Vet[(int)(i+h)];
              Vet[(int)(i+h)]=x;
              Troca=1;
      }
    }
  }while((Troca == 1)||(h >= 1));
}
```

Análise de Desempenho (Complexidade de Algoritmo)

- Comparando os desempenhos dos métodos Bubble Sort, Shake Sort e Comb Sort, para o seguinte array, obtemos a seguinte tabela que analisa o desempenho dos 3:

i	0	1	2	3	4	5	6	7
Vet[i]	44	55	12	42	94	18	06	67

Desempenho Comparativo entre os Métodos de Ordenação			
Métodos	**Num. Trocas**	**Num. Comparações**	**Num. Varreduras**
Bubble Sort	15	28	7
Shake Sort	15	23	3
Comb Sort	5	24	5

Ordenação por Troca: Quick Sort

Met. Ord. Troca: Quick Sort (1)

- Método muito eficiente baseado no princípio da permutação
  - Um dos melhores método de ordenação de arrays, o que justifica o seu nome

- Adota o princípio de que é mais rápido classificar dois arrays com N/2 elementos cada um, do que um com N elementos; ou seja, segue o paradigma de “Divisão e Conquista”

Met. Ord. Troca: Quick Sort (2)

Na prática, QuickSort é o algoritmo de ordenação mais rápido.

Também segue o paradigma da Divisão-e-Conquista.

Divisão:

Escolha um elemento p para ser o pivô em v;

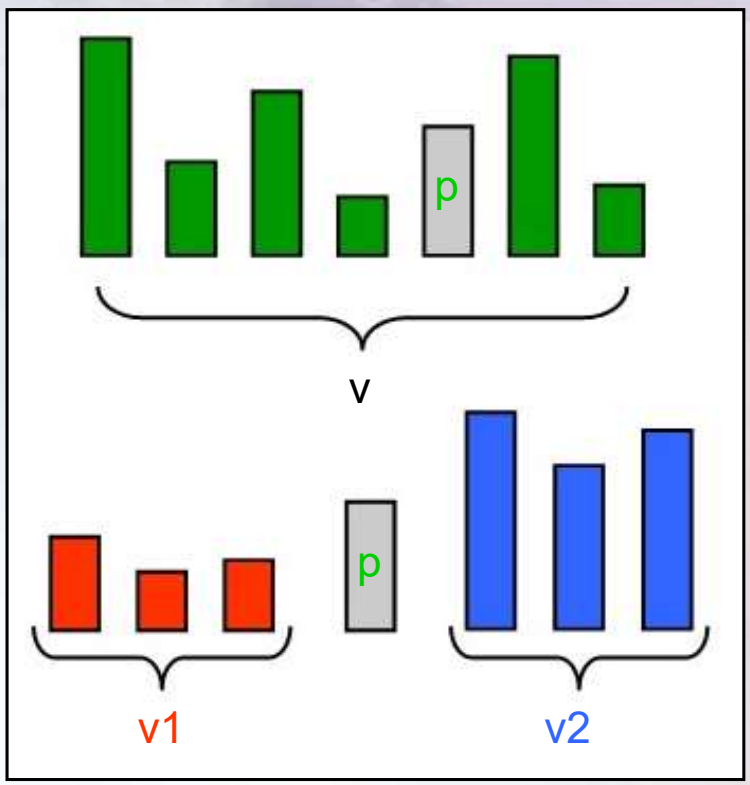
Particione o array v - {p} em dois grupos distintos:

$v1 = \{x \in v - \{p\} \mid x < p\}$

$v2 = \{x \in v - \{p\} \mid x \geq p\}$

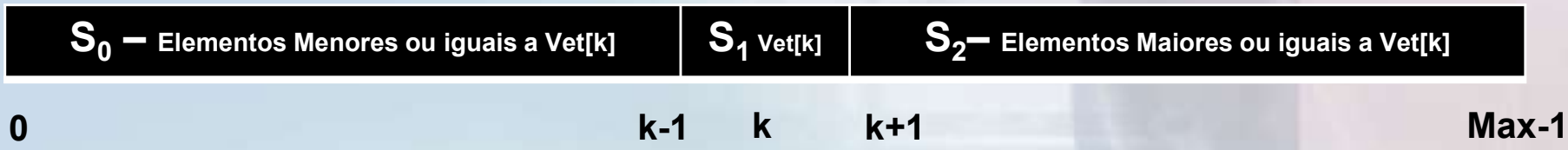
Conquista: ordene recursivamente v1 e v2;

Combinação: junte v1, p e v2 para obter v ordenado.



Met. Ord. Troca: Quick Sort (3)

- O ponto mais crítico deste algoritmo é a decisão de como fazer o particionamento do array. Suponha que o vet[] seja dividido em três segmentos S_i, i = 0, 1, 2



- A escolha do elemento vet[k], também chamado de pivô, que divide o array em três segmentos pode ser realizada de forma arbitrária, mas normalmente é escolhido o elemento da parte central, o que pode tornar a execução do algoritmo mais eficiente

- Em seguida, é realizada uma movimentação dos elementos de tal forma que em S_1 fiquem os elementos menores e no S_2 os maiores que vet[k]

Met. Ord. Troca: Quick Sort (4)

Algoritmo: Enquanto o array estiver desordenado, faça:

1. É escolhido, de forma arbitrária, um elemento X do Array, também chamado de pivô. No nosso caso, X = Parte Central do Array
2. Iniciar dois ponteiros (i, j), onde i=0 e j=N-1
3. O array é percorrido da esquerda para a direita, enquanto vet[i]<X; sendo então percorrido da direita para esquerda, enquanto vet[j]>X.
4. É efetuada a troca dos elementos vet[i] e vet[j], se i<=j; neste caso, incrementamos i e decrementamos j.
5. O processo de varredura continua até que i seja maior que j, condição ocorrida em algum ponto do array.
6. Depois realiza-se a ordenação recursivamente para o array da esquerda e da direita do pivô

Met. Ord. Troca: Quick Sort (5)

i	0	1	2	3	4	5	6	7
Vet[i]	44	55	12	42	94	18	06	67

L — pivô = vet[(L+R)/2] — R

Primeira Varredura (L=0 e R=7)

0	1	2	3	4	5	6	7	i	j	**Resultado** (pivô = vet[3]=42)
44	55	12	42	94	18	06	67	0	6	Troca (Vet[0], Vet[6]). i=1/ j=5
06	55	12	42	94	18	44	67	1	5	Troca (Vet[1], Vet[5]). i=2/ j=4
06	18	12	42	94	55	44	67	3	3	Troca (Vet[3], Vet[3]). i=4/ j=2
								4	2	Fim da Primeira Varredura (i>j)

- Terminamos uma varredura sempre que i > j; neste caso, i=4 e j=2. Devemos agora repetir o processo para o array da esquerda (Vet[L, j]) e o da direita (Vet[i, R]) do pivô. Sendo assim, temos os seguintes arrays:
  - Vet => L=0 e R=2
  - Vet => L=4 e R=7

Met. Ord. Troca: Quick Sort (6)

Segunda Varredura (L=0 e R=2)

0	1	2	i	j	Resultado (pivô = vet[1]=18)
06	18	12	1	2	Troca (Vet[1], Vet[2]). i=2/ j=1
06	12	18	2	1	Fim da Segunda Varredura (i>j)

- Similarmente à primeira varredura, terminamos a segunda com i > j; neste caso, i=2 e j=1. Desta vez, não precisamos repetir o processo para o array da direita do pivô (Vet[i, R]), pois ele possui um único elemento (Vet[2]). Para o vetor da esquerda (Vet[L, j]) , temos:
  - Vet => L=0 e R = 1

i	0	1	2	3	4	5	6	7
Vet[i]	06	12	18	42	94	55	44	67

Vetor Parcialmente Ordenado

Met. Ord. Troca: Quick Sort (7)

Terceira Varredura (L=0 e R=1)

0	1	i	j	Resultado (pivô = vet[0]=6)
06	12	0	0	Troca (Vet[0], Vet[0]). i=1/ j=0
06	12	1	0	Fim da Segunda Varredura (i>j)

- Similarmente à segunda varredura, terminamos a terceira com i > j; neste caso, i=1 e j=0. Desta vez, não precisamos repetir o processo para nenhum dos arrays (esquerda e direita), pois eles possuem um único elemento no array (Esquerda = Vet[0] / Direita = Vet[1])

i	0	1	2	3	4	5	6	7
Vet[i]	06	12	18	42	94	55	44	67

Vetor Parcialmente Ordenado

Met. Ord. Troca: Quick Sort (8)

Quarta Varredura (L=4 e R=7)

4	5	6	7	i	j	Resultado (pivô = vet[5]=55)
94	55	44	67	4	6	Troca (Vet[4], Vet[6]). i=5/ j=5
44	55	94	67	5	5	Troca (Vet[5], Vet[5]). i=6/ j=4
				6	4	Fim da Terceira Varredura (i>j)

- Terminamos a quarta varredura, pois i=6 e j=4. Devemos agora repetir o processo somente para o array da direita, pois o array da esquerda do pivô possui apenas um elemento (Vet[4]). Sendo assim, o processo deve ser repetido para o seguinte array :
  - Vet => L=6 e R=7

i	0	1	2	3	4	5	6	7
Vet[i]	06	12	18	42	44	55	94	67

Vetor Parcialmente Ordenado

Met. Ord. Troca: Quick Sort (9)

Quinta Varredura (L=6 e R=7)

6	7	i	j	Resultado (pivô = vet[6]=94)
94	67	6	7	Troca (Vet[6], Vet[7]). i=7/ j=6
67	94	7	6	Fim da Quarta Varredura (i>j)

- Terminamos a quinta varredura, pois i=7 e j=6. Como não existem mais arrays com mais de 1 elemento para ser ordenado, o array já se encontra ordenado.

i	0	1	2	3	4	5	6	7
Vet[i]	06	12	18	42	44	55	67	94

Vetor Ordenado!!!

Met. Ord. Troca: Quick Sort (10)

```
/*OrdenaQuickSortRecursivo(): Ordena, de forma recursiva, a Sequencia de Numeros
                              pelo Método de Ordenação por Partição e Troca
                              conhecido como QuickSort. */
void OrdenaQuickSortRecursivo(int Vet[], int inicio, int fim)
{
  int pivo, aux, i, j, meio;

  i = inicio;
  j = fim;

  meio = (int) ((i + j) / 2);
  pivo = Vet[meio];

  do{
    while (Vet[i] < pivo) i = i + 1;
    while (Vet[j] > pivo) j = j - 1;

    if(i <= j){
      aux = Vet[i];
      Vet[i] = Vet[j];
      Vet[j] = aux;
      i = i + 1;
      j = j - 1;
    }
  }while(j > i);

  if(inicio < j) OrdenaQuickSortRecursivo(Vet, inicio, j);
  if(i < fim) OrdenaQuickSortRecursivo(Vet, i, fim);
}
```

Ordenar o seguinte array utilizando o Quick Sort:

i	0	1	2	3	4	5	6	7
Vet[i]	12	48	37	33	57	86	25	92

[L=0/R=7] – Pivô=Vet[3]=33

Vet[] =	12	48	37	**33**	57	86	25	92	Troca: i (1) / j(6) => i=2/j=5
Vet[] =	12	25	37	**33**	57	86	48	92	Troca: i (2) / j(3) => i=3/j=2
Vet[] =	12	25	**33**	37	57	86	48	92	**(i>j): PAROU**

Fazer para os seguintes vetores: Vet[L=0/R=2] e Vet[L=3/R=7]

[L=0/R=2] – Pivô=Vet[1]=25

Vet[] =	12	**25**	33	33	57	86	48	92	Troca: i (1) / j(1) => i=2/j=0
Vet[] =	12	**25**	33	33	57	86	48	92	**(i>j): PAROU**

Fazer para os seguintes vetores: Vet[L=0/R=0] e Vet[L=2/R=2]: Não Precisa Fazer, Vetor com Elemento Único.

[L=3/R=7] – Pivô=Vet[5]=25

Vet[] =	12	25	33	37	57	**86**	48	92	Troca: i (5) / j(6) => i=6/j=5
Vet[] =	12	25	33	37	57	48	**86**	92	**(i>j): PAROU**

Fazer para os seguintes vetores: Vet[L=3/R=5] e Vet[L=6/R=7].

[L=3/R=5] – Pivô=Vet[4]=57

Vet[] =	12	25	33	37	**57**	48	86	92	Troca: i (4) / j(5) => i=5/j=4
Vet[] =	12	25	33	37	48	**57**	86	92	**(i>j): PAROU**

Fazer para os seguintes vetores: Vet[L=3/R=4] (Precisa ordenar) e Vet[L=5/R=5] (Não precisa, único elemento).

[L=3/R=4] – Pivô=Vet[3]=37

Vet[] =	12	25	33	**37**	48	57	86	92	Troca: i (3) / j(3) => i=4/j=2
Vet[] =	12	25	33	**37**	48	57	86	92	**(i>j): PAROU**

Fazer para os seguintes vetores: Vet[L=3/R=2] (Não precisa, pois R<L) e Vet[L=4/R=4] (Não precisa, único elemento).

[L=6/R=7] – Pivô=Vet[6]=86

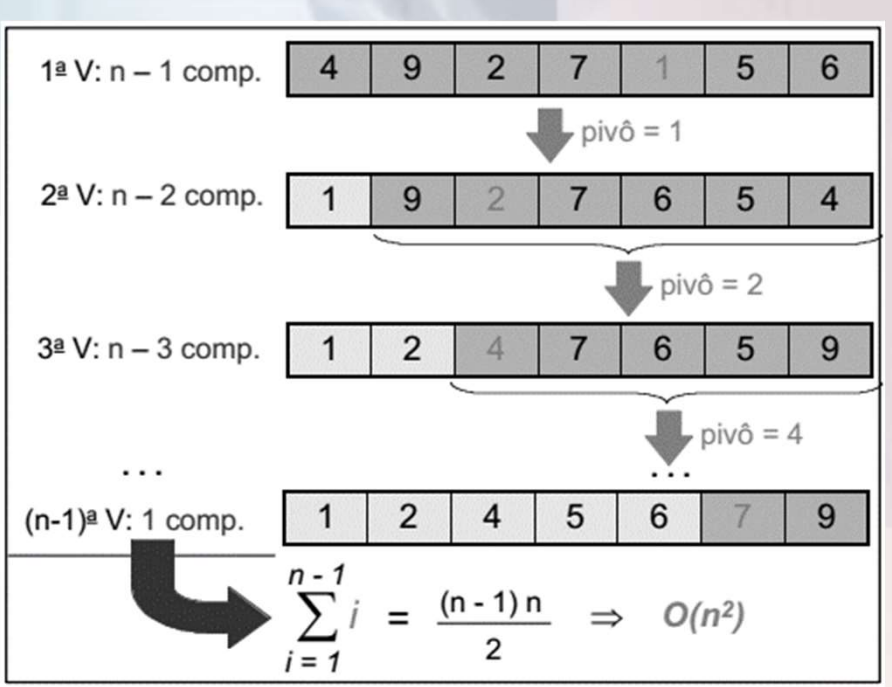
Vet[] =	12	25	33	37	48	57	**86**	92	Troca: i (6) / j(6) => i=7/j=5
Vet[] =	12	25	33	37	48	57	**86**	92	**(i>j): PAROU**

Fazer para os seguintes vetores: Vet[L=6/R=5] (Não precisa, pois R<L) e Vet[L=7/R=7] (Não precisa, único elemento).

Array Ordenado [] = 12 25 33 37 48 57 86 92

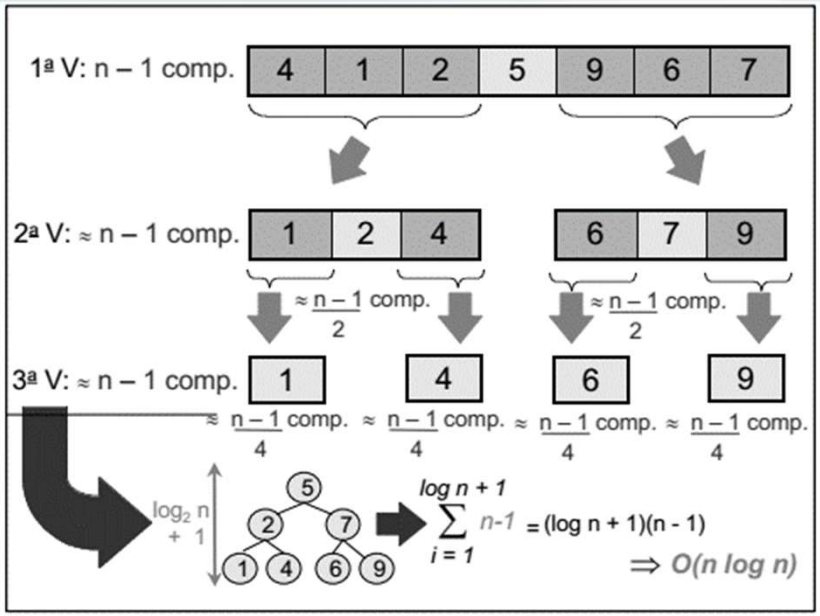
Ordenação por Quick Sort: Análise da Complexidade – Pior Caso

- **Análise para o Pior Caso:**
  - Segmentos totalmente desbalanceados
  - O elemento pivô divide o array de forma desbalanceada, ou seja, divide a lista em duas sub listas: uma com tamanho 0 e outra com tamanho n - 1 (no qual n se refere ao tamanho da lista original)
    - Isso pode ocorrer quando o elemento pivô é o maior ou menor elemento da lista, ou seja, quando a lista já está ordenada, ou inversamente ordenada
  - Se isso acontece, em todas as chamadas, cada etapa recursiva chamará a listas de tamanho igual à lista anterior – 1
  - Recorrência:
    - $T(n) = T(n-1) + T(0) + \theta(n)$
    - $T(n)$: $\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \rightarrow O(n^2)$
      - Série Aritmética



Ordenação por Quick Sort: Análise da Complexidade – Melhor Caso

- **Análise para o Melhor Caso:**
  - Segmentos balanceados
  - Pivô é sempre o elemento com valor mais próximo da média
  - Produz duas listas de tamanho não maior que n/2, uma vez que uma lista terá tamanho [n/2] e outra tamanho [n/2] – 1
  - Nesse caso, o quicksort é executado com maior rapidez
  - Recorrência:
    - $T(n) \leq 2\,T(n/2) + \theta(n)$
    - Resolvendo pelo Teorema Mestre: $T(n) = O(n \lg n)$



PANC: Projeto e Análise de Algoritmos

Aula 11: Algoritmos de Ordenação II - Troca:

Bubble Sort, Shake Sort, Comb Sort e Quick Sort

Breno Lisi Romano

Dúvidas???

http://sites.google.com/site/blromano

Instituto Federal de São Paulo – IFSP São João da Boa Vista
Bacharelado em Ciência da Computação – 3º Semestre

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista