SÃO
PAULO
TECH
SCHOOL

ED

Estrutura de Dados e Armazenamento

Ordenação

(Merge Sort, Quick Sort)

Profa. Célia Taniwaki

Algoritmos de ordenação

Existem inúmeros algoritmos de ordenação

Em inglês a palavra sort significa ordenar ou classificar

Os algoritmos realçados serão os que veremos com detalhes.

1. SelectionSort
2. Shell Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort
6. Heap Sort
7. Bubble Sort
8. Comb Sort
9. Cocktail Sort

Vídeo - Comparação

O link abaixo é de um vídeo que compara 9 algoritmos de ordenação:

- https://www.youtube.com/watch?v=ZZuD6iUe3Pc
- Esse vídeo utiliza 4 tipos de dados de entrada:
  - Random (aleatório)
  - Few unique (muitos dados duplicados)
  - Reversed (ordem inversa)
  - Almost sorted (praticamente ordenado)
- Assista o vídeo para ver quais algoritmos são mais eficientes e quais são menos

Merge Sort

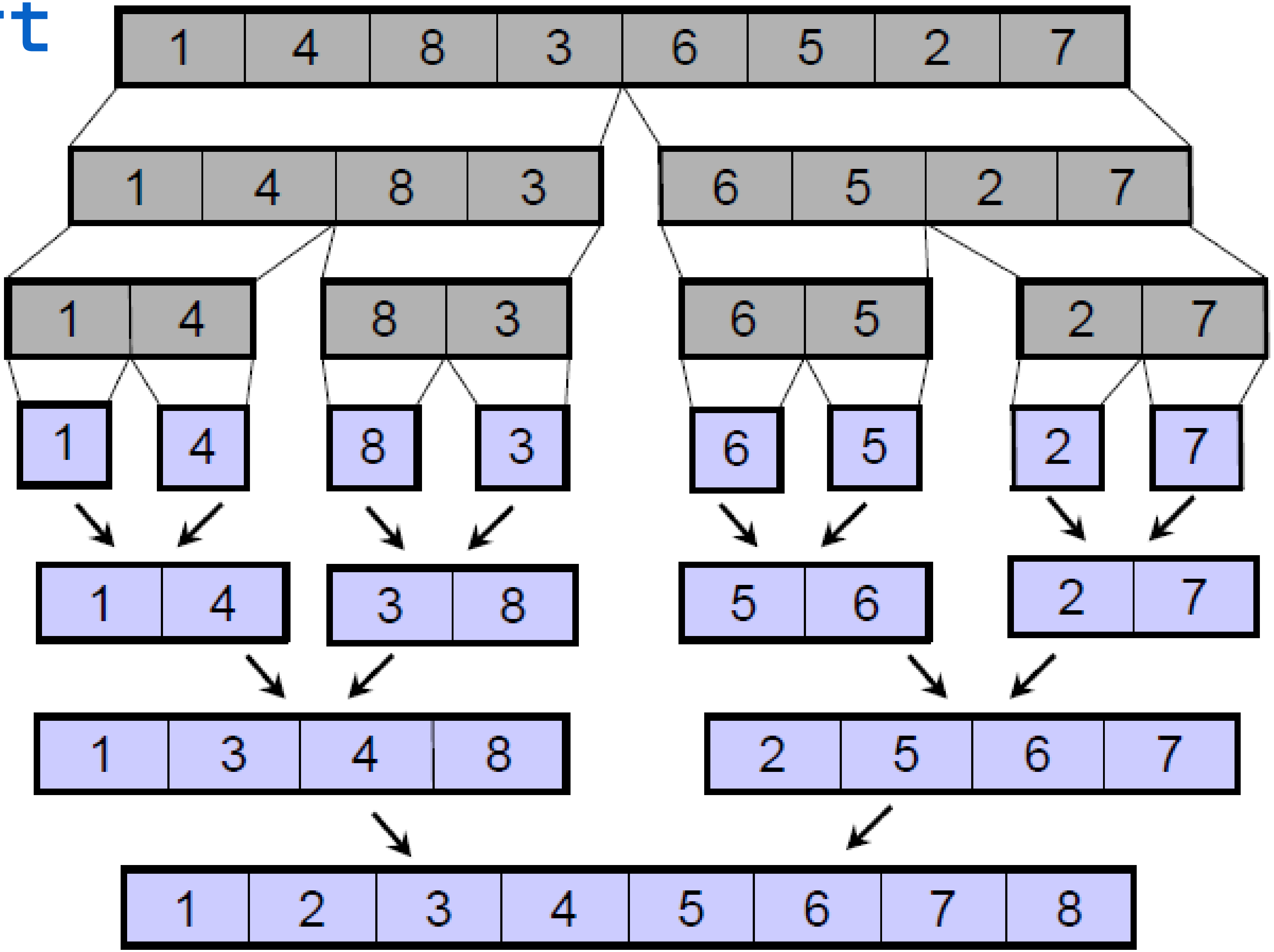
Método particular de ordenação

- Baseia-se em intercalações sucessivas de 2 sequências ordenadas em uma única sequência ordenada

Aplica o método "dividir para conquistar"

- Divide o vetor de n elementos em 2 segmentos de comprimento n/2
- Ordena recursivamente cada segmento
- Intercala os 2 segmentos ordenados para obter o vetor ordenado completo

Merge Sort Exemplo



Merge Sort – Algoritmo recursivo

```
mergeSort (int p, int r, int[] v)
/* p = índice inicial do vetor a ser ordenado */
/* r = índice final + 1 do vetor a ser ordenado */
/* vetor v [p ..... r-1]  */
início
  se (p < r-1)
  início
    int q ← (p + r) / 2;   /* q = índice do meio */
    mergeSort(p, q, v);   /* ordena 1a metade */
    mergeSort(q, r, v);     /* ordena 2a metade */
    intercala(p, q, r, v);  /* intercala 2 metades */
  fim
fim
```

Merge Sort – Algoritmo Intercala

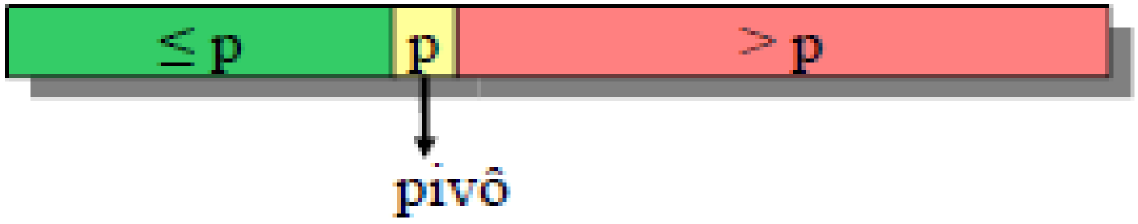
```
intercala (int p, int q, int r, int[] v)
/*  1a metade do vetor  v[ p ... q-1] */
/*  2a metade do vetor  v[ q ... r-1] */
início
  inteiro i, j, k, w[];
  /* criar vetor auxiliar w de tamanho r-p */

  i ← p;  j ← q;  k ← 0;
  enquanto (i < q) e (j < r) faça
    inicio
      se (v[i] <= v[j])
       então  w[k++] ← v[i++];
       senão  w[k++] ← v[j++];
    fim
  enquanto (i < q) faça w[k++] ← v[i++];
  enquanto (j < r) faça w[k++] ← v[j++];
  para i de p enquanto i < r faça  v[i] ← w[i - p];
fim
```

QuickSort

- Proposto por Hoare em 1960 e publicado em 1962.
- Em geral, o algoritmo é muito mais rápido que os algoritmos elementares.
- Também é um algoritmo de troca: ordena através de sucessivas trocas entre pares de elementos do vetor.
- Aplica o método "dividir para conquistar":
  - A idéia básica é dividir o problema de ordenar um conjunto com n itens em dois problemas menores.
  - Os problemas menores são ordenados independentemente.
  - Os resultados são combinados para produzir a solução final.

QuickSort

- A parte mais delicada do método é o processo de partição.
- O vetor A[Esq..Dir] é rearranjado por meio da escolha arbitrária de um **pivô** x.
- O vetor A é particionado em duas partes:
  - Parte esquerda: elementos $\leq$ x.
  - Parte direita: elementos $\geq$ x.



QuickSort – Escolha do pivô

- Particionamento pode ser feito de diferentes formas.
- Principal decisão é escolher o pivô
  - Primeiro elemento do vetor
  - Último elemento do vetor
  - Elemento do meio do vetor
  - Elemento que mais ocorre no vetor
  - Elemento mais próximo da média aritmética dos elementos do vetor
- Logicamente, o algoritmo é diferente, dependendo da forma de escolher o pivô

QuickSort –[pivô = elemento do meio]

- Seja o vetor abaixo.
- Considerando que 4 seja o pivô, o primeiro passo do Quicksort rearranjaria o vetor da forma abaixo:

6	5	**4**	1	3	2

pivô

2	3	1	**4**	5	6

pivô

QuickSort [pivô = elemento do meio]

1º Passo

Pivô
4

6	5	4	1	3	**2**
i					*j*

6>4 e 2<4, então, troca 6 com 2

Incrementa i e decrementa j

2	**5**	4	1	**3**	6
	i			*j*	

5>4 e 3<4, então, troca 5 com 3

Incrementa i e decrementa j

2	3	**4**	**1**	5	6
		i	*j*		

4=4 e 1<4, então, troca 4 com 1

Incrementa i e decrementa j

2	3	1	4	5	6
		j	*i*		
Partição 1			Partição 2		

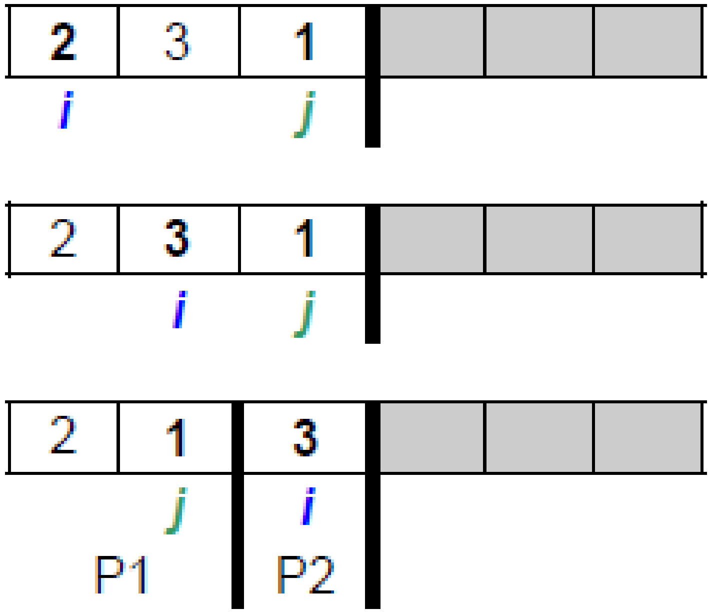
i > j → PARTICIONA

i é diferente de final e *j* é diferente de início.

Faz a chamada recursiva para os dois casos.

QuickSort – Explo (continuação)

2º Passo



2<3 → incrementa i

Pivô
3

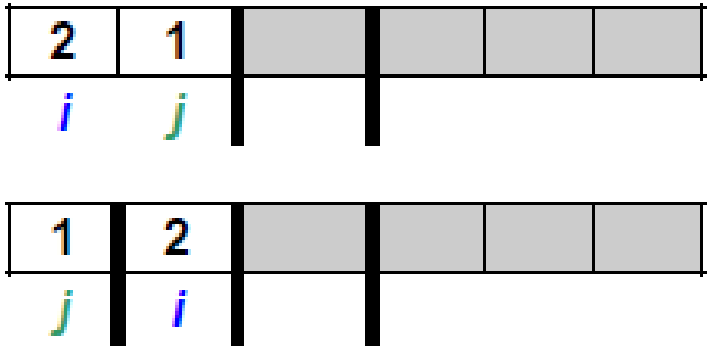
3=3 e 1<3, então, troca 3 com 1

Incrementa i e decrementa j

i > j → PARTICIONA

Chama a recursão apenas para a Partição 1 (P1), pois P2 é de tamanho igual a 1 (i = fim)

3º Passo



2=2 e 1<2, então, troca 2 com 1

Incrementa i e decrementa j

Pivô
2

i > j → j = inicio e i= fim → Finaliza

Não particiona mais

QuickSort [pivô = elem do meio]

```
particiona (int[] v, int indInicio, int indFim)
início
   inteiro i, j, pivo;
   i ← indInicio;  j ← indFim;
   pivo ← v[(indInicio+indFim)/2];
   enquanto i <= j faça
    início
      enquanto i < indFim e v[i] < pivo
         i ← i + 1;
      enquanto j > indInicio e v[j] > pivo
         j ← j - 1;
      se i <= j
      entao inicio
              troca (v[i], v[j]);
               i ← i + 1;
               j ← j - 1;
            fim
    fim
   se indInicio < j então particiona (v, indInicio, j);
   se i < indFim então particiona (v, i, indFim);
fim
```

QuickSort [pivô = último elemento]

```
particiona (int[] v, int indInicio, int indFim)
início
   inteiro i, j, pivo;
   pivo ← v[indFim];
   i ← indFim;
   para j de indFim – 1 enquanto j >= indInicio faça
    início
      se v[j] > pivo
      então inicio
                i ← i - 1;
               troca (v[i], v[j]);
            fim
      fim
   troca (v[indFim], v[i]);
   se indInicio < i então particiona (v, indInicio, i-1);
   se i < indFim  então particiona (v, i+1, indFim);
fim
```

Notação O-grande ou Big-O

- Em computação, utiliza-se a notação O-grande ou Big-O para representar a complexidade de tempo de execução de um algoritmo
- Leva-se em conta a quantidade de operações que o algoritmo executa em função da quantidade de dados que ele manipula

- A pesquisa sequencial tem complexidade O(n), sendo n o tamanho do vetor.
- Num vetor de 8 posições, são necessários 8 iterações para concluir que o elemento não está no vetor

- A pesquisa binária tem complexidade $O(\log_2 n)$
- Quando n=8, $\log_2 n = 3$. Vimos que em 3 iterações, encontramos o elemento ou concluímos que ele não está no vetor.

Complexidade dos algoritmos de ordenação

- Selection Sort, Bubble Sort e Insertion Sort tem **complexidade $O(n^2)$**, pois seus algoritmos tem um for dentro de outro for.
- Nesses algoritmos, para posicionar um elemento na posição correta, é preciso percorrer todo o vetor. Assim, como para cada elemento do vetor, precisa fazer aproximadamente n iterações, a quantidade total de iterações é aproximadamente $n * n = n^2$.
- Para um vetor com n = 8, isso dá aproximadamente 64 iterações.

- O Merge Sort tem **complexidade $O(n \log_2 n)$**, porque ele tem um comportamento semelhante ao da Pesquisa Binária, ao ir dividindo o vetor ao meio, e depois ir juntando fazendo as intercalações.
- Quando n=8, $n\log_2 n = 8 * 3 = 24$.
- O Quick Sort também tem, no geral, a mesma complexidade do Merge.

Complexidade dos algoritmos de ordenação

- Por isso, para quantidades muito grande de dados a serem ordenados, é recomendável utilizar o MergeSort ou o QuickSort, ao invés dos 3 primeiros algoritmos vistos neste material.

- Assista o vídeo que há no início do material para ver quais algoritmos são mais eficientes e quais são menos eficientes.

Outras animações interessantes

- MergeSort vs QuickSort: https://www.youtube.com/watch?v=es2T6KY45cA
- InsertionSort vs BubbleSort: https://www.youtube.com/watch?v=TZRWRjq2CAg
- Selection Sort:
  - Dança: https://www.youtube.com/watch?v=Ns4TPTC8whw
- Bubble Sort:
  - Lego: https://www.youtube.com/watch?v=iKQ461P0m2k
  - Dança: https://www.youtube.com/watch?v=lyZQPjUT5B4
- Explicação sobre HeapSort https://www.youtube.com/watch?v=H5kAcmGOn4Q

Agradeço
a sua atenção!

SÃO
PAULO
TECH
SCHOOL