# PAA - Aula ~~07~~ 08 - Bônus

# QuickSort e Problema da Separação

QuickSort é um algoritmo de ordenação que usa a técnica de divisão-e-conquista.

Principal diferença para o mergeSort:

- Ao invés de dividir o vetor ao meio,
- divide ele entre os elementos menores e maiores,
    - usando como referência um elemento chamado pivô.

Exemplo/ideia do quickSort:

- vetor fora de ordem  v 5 8 4 1 3 6 2 7
- escolher um pivô, digamos p = v[1] = 5
- dividir o vetor em dois subvetores, um com os menores que o pivô e outro com os maiores

  vl 4 1 3 2  vr 8 6 7
- ordenar recursivamente os dois subvetores, sendo que subvetores com 0 ou 1 elementos já estão ordenados

  vl 1 2 3 4  vr 6 7 8
- concatenar os subvetores ordenados com o pivô

  v = vl + p + vr = 1 2 3 4 5 6 7 8

Pseudocódigo do algoritmo quickSort recursivo:

```
quickSort(vetor v):
    se tam(v) <= 1: retorne
    p = escolhePivo(v) // por enquanto, pense que p = v[1]
    (vl, vr) = particiona(v, p)
    quickSort(vl)
    quickSort(vr)
    v = vl + p + vr // concatenação das partes ordenadas
```

Pseudocódigo do algoritmo particiona:

```
particiona(vetor v, pivô p): // supomos que p = v[1], para simplificar
    i = j = 1
    para k = 2 até tam(v):
        se v[k] <= p: vl[i++] = v[k]
        senão vr[j++] = v[k]
    devolve (vl, vr)
```

Eficiência do intercala: algoritmo executa em tempo linear no tamanho do vetor v,

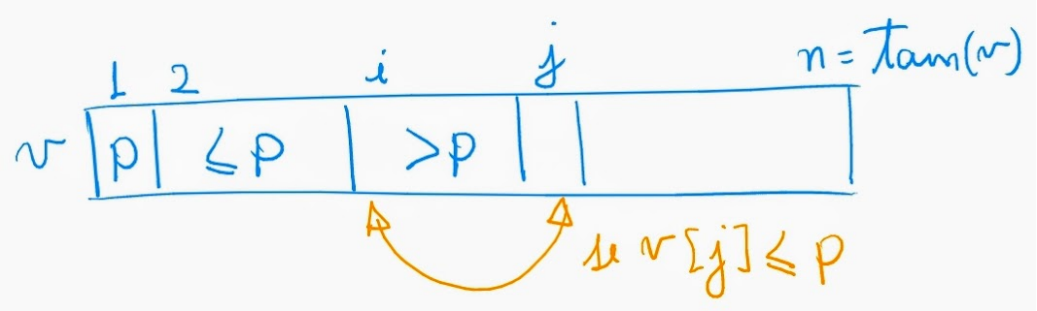
- i.e., O(n) para n = tam(v).
- Para verificar, note que em toda iteração o valor de k é incrementado.

Corretude do intercala: a principal propriedade invariante é que

- no início da iteração k os elementos no subvetor v[2..k-1]
    - que são menores ou iguais a p estão em vl[1..i-1]
    - e os que são maiores estão em vr[1..j-1].
- A prova segue por indução no número da iteração.

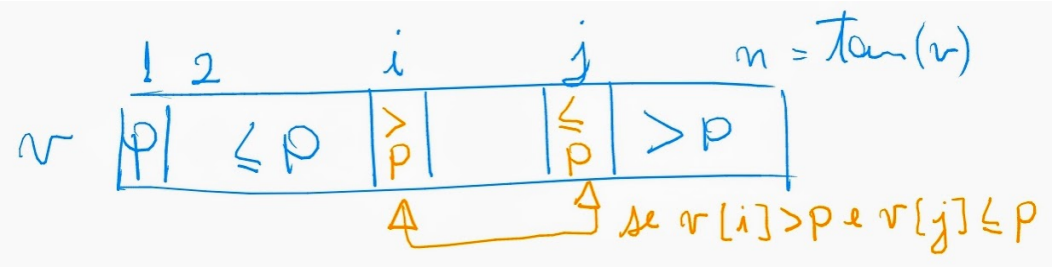
Conseguimos implementar sem usar um vetor auxiliar, i.e., in place?

- Conheço duas maneiras, em uma percorremos v da esquerda para a direita, mantendo o vetor dividido em três partes, como sugere a figura.




- A cada novo elemento considerado, decidimos se ele deve ir
    - para a 1ª ou 2ª parte, dependendo se é menor ou maior que o pivô.
- Quiz1: detalhe a implementação desta versão.
    - Atente para onde colocar o pivô quando terminar de percorrer v.

Na outra maneira percorremos o vetor a partir das pontas, como sugere a figura.




- Mantemos os menores que o pivô na esquerda e os maiores na direita.
- Quando encontramos um elemento maior no início e um menor no final,
    - trocamos ambos de posição.
- Quiz2: detalhe a implementação desta versão.
    - Atente para onde colocar o pivô quando terminar de percorrer v.

Conseguimos implementar o intercala sem inverter a posição relativa de elementos com o mesmo valor, i.e., de modo estável?

- R.: Não nas versões in place.

Pseudocódigo do algoritmo quickSort recursivo:

```
quickSort(vetor v):
    se tam(v) <= 1: retorne
    p = escolhePivo(v, n) // vamos analisar a relevância dessa escolha
    (vl, vr) = particiona(v, p)
    quickSort(vl)
    quickSort(vr)
    v = vl + p + vr // se particiona é in place não temos que concatenar
```

Qual o pior tipo de pivô que o quickSort pode escolher?

- R: Aquele que divide a entrada em um vetor vazio e outro com apenas um elemento a menos que o anterior.
- Qual a eficiência do quickSort neste caso? R: Theta(n^2).
- Isso porque o particiona percorre todo o vetor em cada chamada. Assim, no total temos n + n-1 + n-2 + ... + 3 + 2 + 1 = n(n-1)/2 iterações do particiona.

Qual o melhor tipo de pivô que o quickSort pode escolher?

- R: Aquele que divide a entrada exatamente na metade.
- Qual a eficiência do quickSort neste caso? R: Theta(n log n).
- Isso porque o quickSort fica com uma recorrência igual a do mergeSort, i.e.,

$$T(n) = 2T(n/2) + O(n)$$

O que seria um bom pivô?

- R.: Aquele que divide a entrada numa proporção 25%-75%.
- Por que isso caracteriza um bom pivô (ou uma boa divisão)?
    - R: Porque a altura de uma árvore de recursão em que só ocorram partições boas é limitada por O(log n).
- Para deduzir esse valor, seja tam_sp(j) o tamanho do maior subproblema depois de j divisões boas.
- Como em cada divisão boa o maior subproblema diminui para pelo menos 3/4, temos que tam_sp(j) <= n (3/4)^j

- Sendo h o maior número de divisões boas, quando mesmo o maior problema caiu no caso base, temos

$1 <= n*(3/4)^h \rightarrow (4/3)^h <= n \rightarrow h <= \log_{(4/3)} n \rightarrow h <= (1 / \lg (4/3)) \lg n = O(\lg n)$

Como encontrar bons pivôs? R.: Usando aleatoriedade.

- O algoritmo escolhe o pivô de modo uniforme e aleatório.
- Qual a probabilidade de obter uma boa divisão, ou seja, um bom pivô?
    - R.: 50%, pois qualquer elemento cuja posição final está no "meio" (entre 25%-75%) do vetor gera essa divisão.

Para concluir a análise, vamos usar o conceito de fase.

- Definimos uma variável aleatória X que nos diz quantos níveis de recursão (ou quantas partições) são necessários para que ocorra uma boa divisão.
    - A duração de uma fase é o valor de X.
- Note que o valor de X pode ser muito grande, se tivermos muito azar nas escolhas do pivô. Mas, qual é o valor esperado de X, i.e., E[X]? Sendo p a probabilidade de uma boa divisão, temos

$$E[X] = 1*p + (1-p)*(1 + E[X]) = p + 1 - p + E[X] - pE[X] = 1 + E[X] - pE[X]$$

$$E[X] = 1 + E[X] - pE[X] \rightarrow pE[X] = 1 \rightarrow E[X] = 1/p$$

No nosso caso, p = 1/2. Portanto E[X] = 1/(1/2) = 2

- Assim, em média, a cada dois níveis da recursão temos um pivô bom.
- Como o ramo mais profundo da árvore de recursão tem no máximo
    - (1 / lg (4/3)) lg n = O(lg n) partições com pivôs bons,
- o número de níveis esperados é (2 / lg (4/3)) lg n que também é O(lg n).
- Dado que o trabalho por nível da árvore é O(n)
    - chegamos ao tempo esperado O(n log n).

Importante: note que o tempo esperado depende apenas

- das escolhas do algoritmo, e não dos valores da entrada.