Projeto e Análise de Algoritmos (PAA)

QuickSort e o 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.

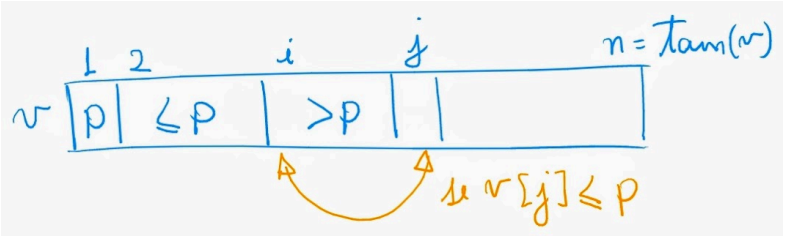
Exemplo/ideia do quickSort:

- vetor fora de ordem v[1 .. n] = 5 8 4 1 3 6 2 7
- escolher um pivô, digamos p = v[1] = 5
- separa o vetor entre os menores e os maiores que o pivô, que fica em v[k]
  v[1 .. k-1] = 4 1 3 2 v[k+1 .. n] = 8 6 7
- ordenar recursivamente os dois subvetores, sendo que subvetores com 0 ou 1 elementos já estão ordenados
  v[1 .. k-1] = 1 2 3 4 v[k+1 .. n] = 6 7 8
- concatenar os subvetores ordenados com o pivô
  v[1 .. n] = v[1 .. k-1] + v[k] + v[k+1 .. n] = 1 2 3 4 5 6 7 8

Pseudocódigo do algoritmo quickSort recursivo:

```
quickSort(v[e .. d]): // na primeira chamada e = 1 e d = n
    se d - e + 1 <= 1: retorne // d - e + 1 é o tamanho do vetor atual
    escolhePivo(v[e .. d]) // escolhe o pivô p e o deixa na posição v[e]
    k = separa(v[e .. d]) // devolve v[e .. k-1] <= v[k] = p < v[k+1 .. d]
    quickSort(v[e .. k-1])
    quickSort(v[k+1 .. d)
```

Ideia do separa: percorrer 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.

Exemplo do separa:

- v[1 .. n] = 5 8 4 1 3 6 2 7
- p = v[1]

Pseudocódigo do algoritmo separa:

```
separa(v[e .. d]):
        p = v[e]
        i = e + 1
        para j = e + 1 até d:
                se v[j] <= p:
                        troca(v, i, j); i += 1
        troca(v, e, i - 1) // Quiz1: por que essa troca?
        devolve i - 1
```

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

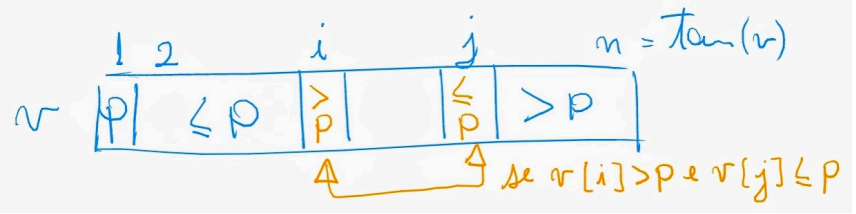
- i.e., O(n) para n = d - e + 1
- Para verificar, note que j começa em e + 1,
    - e é incrementado em toda iteração até d.

Corretude do separa: a principal propriedade invariante é que

- no início da iteração j os originalmente no subvetor v[e+1 .. j-1]
    - que são menores que p estão em v[e+1 .. i-1]
    - e os que são maiores estão em v[i .. j-1].
- A prova segue por indução no valor de j.

Bônus: em outra maneira de implementar o separa

- 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.

Quiz3: Conseguimos implementar o separa 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(v[e .. d]): // na primeira chamada e = 1 e d = n
    se d - e + 1 <= 1: retorne // d - e + 1 é o tamanho do vetor atual
    escolhePivo(v[e .. d]) // escolhe o pivô p e o deixa na posição v[e]
    k = separa(v[e .. d]) // devolve v[e .. k-1] <= v[k] = p < v[k+1 .. d]
    quickSort(v[e .. k-1])
    quickSort(v[k+1 .. d)
```

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: $T(n) = T(n-1) + cn = \Theta(n^2)$.
- Isso porque o separa percorre todo o vetor em cada chamada. Assim, no total temos $n + n-1 + n-2 + \ldots + 3 + 2 + 1 = n(n-1)/2$ iterações do separa.

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.,

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

O que seria um bom pivô? R.: Um 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.
- Em média, a cada 2 pivôs teremos 1 bom. Assim, nossa árvore de recursão deve ter altura $h <= (2 / \lg (4/3)) \lg n = O(\lg n)$

Importante: note que o tempo esperado depende apenas

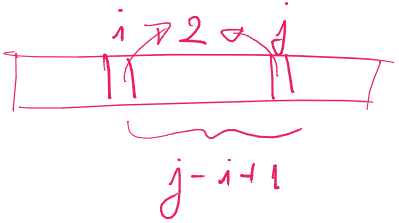
- das escolhas do algoritmo, e não dos valores da entrada.

Para uma análise mais precisa, seja C uma variável aleatória,

- que conta o total de comparações feitas pelo separa em todas as chamadas da função quickSort
- Note que a eficiência do quickSort corresponde assintoticamente a C
- Seja X_{ij} o # de comparações entre o i-ésimo menor valor (z_i) e o j-ésimo menor (z_j), c/ $i < j$
- Observe que $X_{ij} \in \{0,1\}$, pois eles só são comparados se z_i ou z_j forem escolhidos como pivôs, antes de algum valor z_k, c/ $i < k < j$ ser escolhido como pivô
- Note que $C = \sum_{j=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$

Temos que $E[X_{ij}] = 0 \cdot Pr[X_{ij} = 0] + 1 \cdot Pr[X_{ij} = 1]$

$Pr[X_{ij} = 1] = \frac{2}{j - i + 1}$, pois enquanto pivôs menores que z_i ou maiores que z_j forem sorteados, z_i e z_j não são comparados, e na 1ª iteração em que for escolhido um pivô entre z_i e z_j (inclusive), eles têm 2 chances em $j - i + 1$ de ser comparados



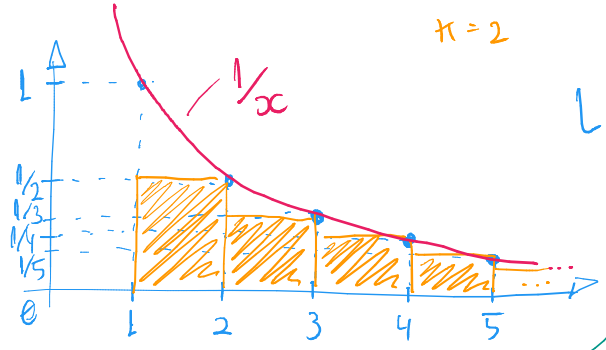
linearidade da esperança

$$E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} = 2 \sum_{i=1}^{n-1} \left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n - i + 1} \right)$$

$$E[C] \leq 2 \sum_{i=1}^{n-1} \sum_{k=2}^{n} \frac{1}{k} \leq 2 \sum_{i=1}^{n-1} \ln n = 2(n-1)\ln n$$

$$= \Theta(n \lg n)$$

Para verificar que $\sum_{k=2}^{n} \frac{1}{k} \leq \ln n$ observe que:



Logo, $\sum_{k=2}^{n} \frac{1}{k} \leq \int_{1}^{n} \frac{1}{x} = \ln n - \ln 1 = \ln n$

Outra maneira de deduzir é:

$$\sum_{k=2}^{n} \frac{1}{k} = \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} + \cdots \leq \underbrace{\frac{1}{2} + \frac{1}{2}}_{=1} + \underbrace{\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}}_{=1} + \underbrace{\frac{1}{8} + \cdots}_{=1}$$

$$= \sum_{i=1}^{h} \sum_{j=1}^{2^i} \frac{1}{2^i} = \sum_{i=1}^{h} 2^i \cdot \frac{1}{2^i} \leq \sum_{i=1}^{\lg n} 1 = \lg n$$

Note que:

$n = 2^1 + 2^2 + \cdots + 2^h$

$= 2^{h+1} - 2^1$

$h+1 = \lg(n+2)$

$h = \lg(n+2) - 1$

$\leq \lg n$