Construção e Análise de Algoritmos

aula 03: O algoritmo de ordenação *Quicksort*

1 Introdução

Na aula passada, nós estudamos o algoritmo de ordenação Mergesort e vimos que ele executa em tempo $O(n \log n)$, o que é muito bom.

Mas, apesar da sua eficiência em termpos de tempo, o algoritmo Mergesort não é muito econômico em termos de memória: para ordenar uma lista com n elementos é preciso reservar o espaço equivalente a uma outra lista de n elementos para realizar as operações de intercalação.

Em alguns casos isso não é realmente um problema, mas em outros casos isso pode ser.

Na aula de hoje, nós vamos apresentar um outro algoritmo de ordenação que não sofre desse problema.

2 Ordenação por partição

Considere mais uma vez o velho, mas não tão bom assim, algoritmo da bolha.

Na aula passada nós vimos que a estratégia de quebrar a lista no meio, ordenar as duas metades em separado, e intercalar as metades ordenadas resultantes, permite reduzir o seu tempo de execução essencialmente à metade

$$2 \cdot \left(\frac{n}{2}\right)^2 \;+\; n \quad = \quad \frac{n^2}{2} \;+\; n$$

A primeira observação de hoje é que mesmo que a divisão não seja exatamente no meio, ainda assim o ganho pode ser significativo.

Por exemplo, suponha que nós dividimos a lista de tamanho n em um porção de tamanho $n/3$ e uma porção de tamanho $2n/3$.

Então, o tempo de execução associado à estratégia acima é dado por

$$\left(\frac{n}{3}\right) \;+\; \left(\frac{2n}{3}\right) \;+\; n \quad = \quad \frac{5n^2}{9} \;+\; n$$

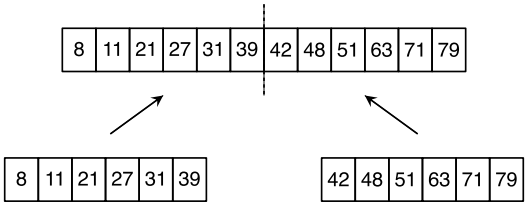
(Note que o tempo da intercalação na muda — porque?)

Essa observação é a chave para permitir que nós eliminemos a operação de intercalação sem uma perda significativa de eficiência.

A ideia é a seguinte.

O papel da intercalação é combinar os elementos das duas porções ordenadas para produzir uma única lista completamente ordenada.

Mas, se todos os elementos da primeira porção já são menores do que todos os elementos da segunda porção, então não há realmente nada a fazer.



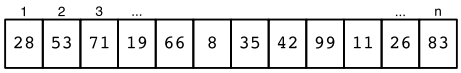
É claro que é preciso ter muita sorte para que uma situação dessas aconteça por acaso.

Mas, não é tão difícil escrever um procedimento que separe os menores elementos de uma lista no lado esquerdo e os maiores elementos no lado direito.

Esse procedimento é chamado de *partição*.

Tipicamente, a partição é realizada escolhendo um elemento qualquer da lista e movendo todos os elementos menores do que ele para a sua esquerda, e todos os elementos maiores do que ele para a sua direita.

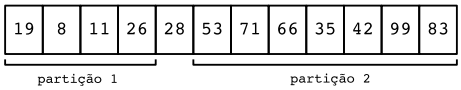
Por exemplo, considere a seguinte lista de números



O elemento que é utilizado para realizar a partição é chamado de *pivot*.

Nesse caso, nós vamos utilizar o primeiro elemento como pivot.

Fazendo isso, nós obtemos o seguinte resultado



Agora, basta ordenar as duas partições e a lista ficará completamente ordenada.

Existem muitas maneiras de implementar o procedimento de partição.

Abaixo nós temos uma implementação que sempre utiliza o primeiro elemento como pivot.

```
Procedimento Partição ( V[1..n] )
    {
 1.     pivot ← V[1]
 2.     j ← 2
 3.     k ← n
 4.     Enquanto ( j ≤ k )
        {
 5.         Se ( V[j] > pivot e V[k] ≤ pivot )
            {
 6.             aux ← V[j];   V[j] ← V[k];   V[k] ← aux
            }
 7.         Se ( V[j] ≤ pivot )   j++
 8.         Se ( V[k] > pivot )   k++
        }

        // Ao final do laço, V[k] é o último elemento menor ou igual ao pivot
 9.     aux ← V[1];   V[1] ← V[k];   V[k] ← aux
```

```
10.     Retorna (k)   // a posição do pivot
    }
```

Como usual, nós vamos assumir que o bloco no interior do laço (linhas `4-7`) executa em tempo $O(1)$; e também vamos assumir que as linhas `8` e `9` executam em tempo $O(1)$.

Dessa maneira, o tempo de execução do procedimento é dado pelo número de iterações do laço.

No início, `j` recebe o valor 2 e `k` recebe o valor n.

A cada iteração, ao menos uma das seguintes coisas acontece

- `j` tem seu valor incrementado
- `k` tem seu valor decrementado

(Porque?)

Agora, não é difícil ver que o laço realiza no máximo $n-1$ iterações, e portanto o procedimento `Partição` executa em tempo $O(n)$.

Ou seja, o procedimento *Partição* executa (aprox.) no mesmo tempo que o procedimento `Intercalação`.

Mas, como ilustrado no exemplo acima, a partição não divide a lista necessariamente no meio.

E isso significa que o ganho de eficiência da estratégia de divisão não será o maior possível.

A seguir, nós vamos examinar o impacto desse fato no tempo de execução de um algoritmo de ordenação baseado exclusivamente no procedimento de partição.

3 O algoritmo *Quicksort*

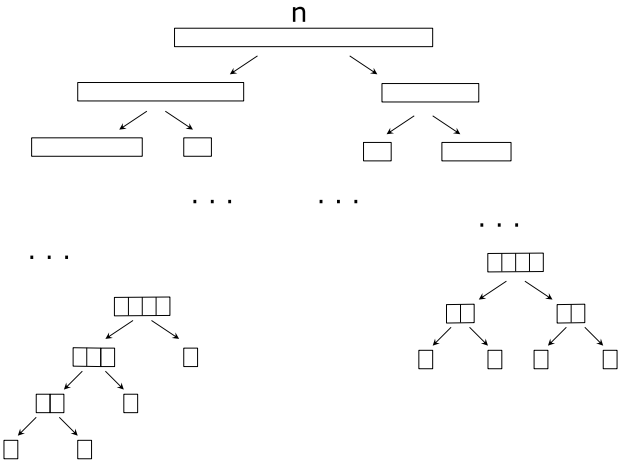
O algoritmo Quicksort é muito semelhante ao algoritmo Mergesort.

Tanto do ponto de vista do pseudo-código

```
Procedimento Quicksort-Rec ( V[i..j] )
  {

     Se (i >= j)   Retorna;

     k  <--  Partição ( V[i..j] )

     Quicksort-Rec ( V[i..k-1] )

     Quicksort-Rec ( V[k+1..j] )

  }
```

```
Procedimento Mergesort-Rec ( V[i..j] )
  {

     Se (i = j)    Retorna;

     k  <--  (i+j) / 2

     Mergesort-Rec ( V[i..k] )

     Mergesort-Rec ( V[k+1..j] )

     Intercalacao (V[1..k],V[k+1..j]);

  }
```

Quando do ponto de vista da sua árvore típica de execução



Mas, a diferença que existe entre eles pode acarretar consequências bastante significativas em alguns casos particulares.

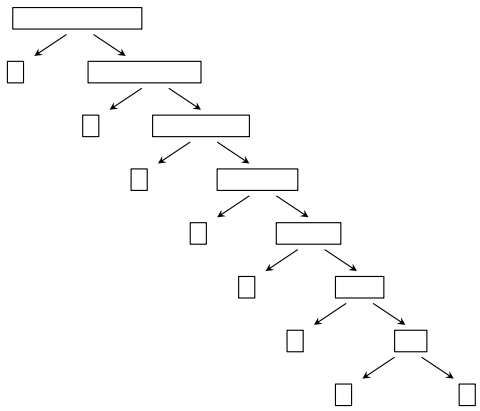
A diferença mais importante entre os dois algoritmos está no fato de que o procedimento `Intercalação` sempre divide a lista exatamente no meio, enquanto que o procedimento Partição pode produzir partições desbalanceadas.

Os dois casos extremos são aqueles em que um dos lados da partição contém apenas um elemento, e o outro lado contém todos os outros.

o que acontece quando o pivot é o menor ou o maior elemento da lista.

É claro que a probabilidade disso ocorrer é muito baixa, mas isso certamente é um caso possível.

De fato, a coisa pode ser ainda muito pior: é possível que em todas as partições realizadas pelo algoritmo, o pivot utilizado seja sempre o menor elemento da lista



Nesse caso, o tempo de execução do algoritmo, que é dado (aprox.) pela soma dos tempos das chamadas ao procedimento `Partição` é dado por

$$n \ + \ n-1 \ + \ \ldots \ + \ 3 \ + \ 2 \ + \ 1 \quad = \quad O(n^2)$$

Ou seja, nesse caso, o algoritmo Quicksort se comporta como o algoritmo da bolha...

Mas, como já indicamos acima, é preciso mesmo ter muito azar para que esse tipo de coisa aconteça, e na prática isso nunca acontece.

Na prática, o algoritmo Quicksort tem um desempenho similar ao do algoritmo Mergesort.

Mas, porque é assim?

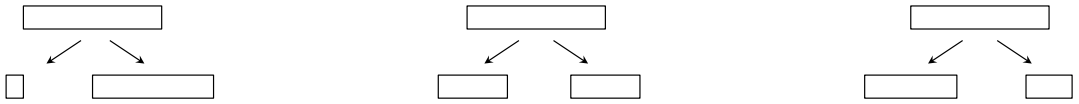
Isto é, porque, apesar do algoritmo Quicksort não controlar o tamanho das suas partições, o seu desempenho é tão bom?

A seguir, nós vamos ver porque isso acontece.

A primeira observação é a seguinte.

Suponha que o pivot é escolhido aleatoriamente dentre os elementos da lista.

Então, é possível que ele produza uma partição muito ruim, ou uma partição muito boa, ou uma partição regular



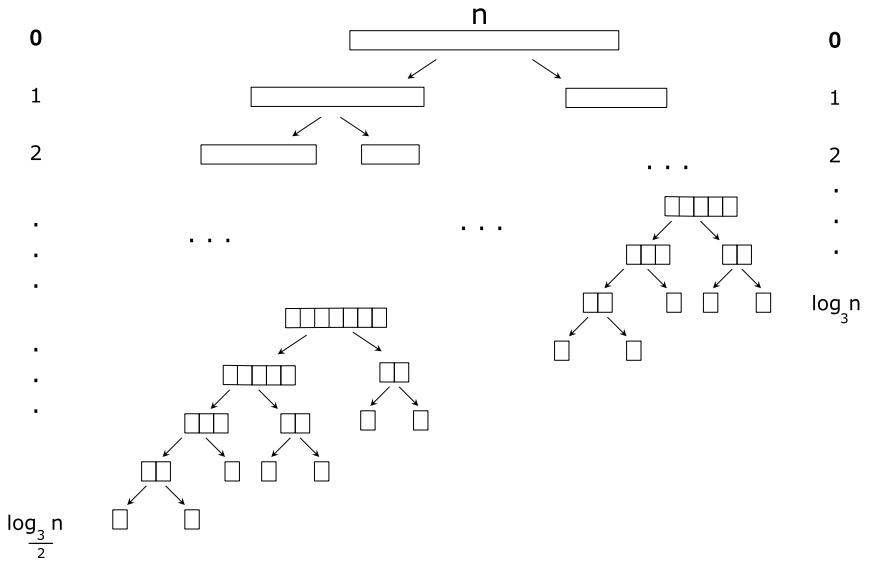
Agora, suponha que uma partição regular é uma partição onde

- o menor lado contém $n/3$ elementos
- o maior lado contém $2n/3$ elementos

Então, a probabilidade de que um pivot aleatório produza uma partição pelo menos regular (i.e., regular, boa ou muito boa)) é igual a 1/2.

(Porque?)

A próxima observação é que, mesmo que todas as partições realizadas pelo algoritmo seja regulares, ainda assim o seu desempenho é muito bom.



Vejamos.

Nós já sabemos que a partição que é feita no nível 1 da árvore leva tempo n, e ela produz um lado com $2n/3$ elementos e outro lado com $n/3$ elementos.

A seguir, a partição que acontece no lado esquerdo do nível 2 da árvore leva tempo $2n/3$, e a partição no lado direito do nível 2 leva tempo $n/3$, em um total novamente de n.

Além disso, os fragmentos que aparecem no nível 2 tem tamanhos $4n/9$, $2n/9$, $2n/9$ e $n/9$.

Raciocinando de maneira análoga, as partições que acontecem no nível 3 levam um tempo total de n, e produzem mais uma vez fragmentos cujo tamanho total é n.

A coisa continua assim até que chegamos ao nível $\log_2 n$, onde começam a aparecer fragmentos de tamanho 1 que não são mais particionados.

A partir desse ponto, o tempo total gasto pelo procedimento de partição em cada nível é menor do que n.

Finalmente, quando chegamos ao nível $\log_{3/2} n$ todos os fragmentos da lista já foram reduzidos a um único elemento e o algoritmo pára.

Como base nessas observações, é fácil ver que o tempo de execução do algoritmo Quicksort nesse caso é menor que

$$\underbrace{n \ + \ n \ + \ n \ + \ \ldots \ + \ n}_{\log 3/2n} \quad = \quad O(n \log n)$$

Agora nós já começamos a entender porque o desempenho do algoritmo Quicksort é tão bom na prática: mesmo em uma situação em que todas as partições são regulares, o seu tempo de execução é da mesma ordem do tempo do algoritmo Mergesort.

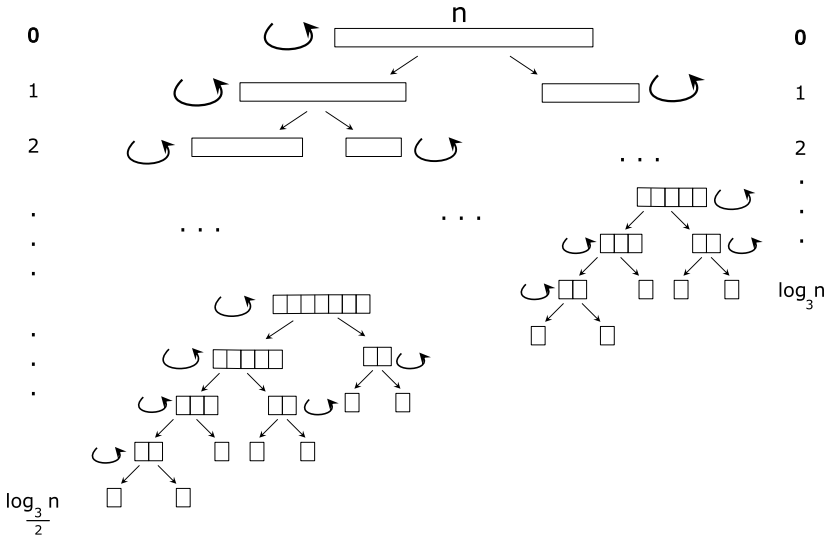
Mas, essa suposição também não é realista.

Quer dizer, em uma execução típica do algoritmo Quicksort algumas partições serão aproximadamente regulares, mas outras serão boas e outras ainda serão ruins.

O que é que nós podemos dizer sobre um caso desses?

Para entender o que acontece nesse caso, nós vamos introduzir uma modificação artificial no algoritmo. Nós vamos assumir que

- *sempre que o procedimento* `Partição` *produz uma partição ruim, ele refaz o seu trabalho, e continua refazendo até que ele consiga produzir uma partição ao menos regular*



É fácil ver que, após essa modificação, a árvore de execução do algoritmo é ao menos tão boa quanto a árvore com partições regulares.

Mas, agora é preciso levar em conta o laço que foi introduzido no procedimento `Partição`.

A primeira observação é que os laços só atrapalham a evolução do algoritmo; ao jogar trabalho útil fora, eles só podem aumentar o tempo de execução.

(Porque?)

A seguir, nós relembramos que, ao selecionar o pivot aleatoriamente, o algoritmo tem probabilidade 1/2 de produzir uma partição ao menos regular.

Isso é como lançar uma moeda: com probabilidade 1/2 nós obtemos Cara (uma partição ao menos regular), e com probabilidade 1/2 nós obtemos Coroa (uma partição ruim).

Utilizando essa imagem, nós vemos que os laços correspondem então a um experimento em que uma moeda é lançada até que se obtenha o resultado Cara pela primeira vez.

Quem quer que já tenha feito esse experimento algumas vezes (ou que já tenha pensado um pouco sobre ele) sabe que a moeda precisa ser lançada em média 2 vezes para se obter a Cara.

(Porque?)

Portanto, a consequência de se introduzir os laços no algoritmo é que agora cada partição é realizada em média duas vezes.

Mas isso significa que o tempo (médio) de execução do algoritmo modificado é menor que

$$\underbrace{2n \;+\; 2n \;+\; 2n \;+\; \ldots \;+\; 2n}_{\leq\ \log 3/2n} \quad = \quad O(n \log n)$$

Exercícios

1. **3-Partição**

   Apresente o pseudocódigo do procedimento `3-Partição` que divide uma lista desordenada em 3 partes utilizando dois elementos como pivôs.

   Estime o tempo de execução do seu algoritmo.

2. **Seleção do k-ésimo elemento**

   Apresente um algoritmo que encontra o k-ésimo menor elemento de um vetor realizando sucessivas chamadas ao procedimento `Partição`.

   Analise o tempo de execução do seu algoritmo.

3. **Partição estável** (OPCIONAL)

   O que você acha dessa ideia:

   - Para escolher um bom pivot para o procedimento de partição, nós podemos selecionar $\sqrt{n}$ elementos, ordená-los com o algoritmo da bolha, e a seguir selecionar o elemento central para utilizar como pivô.

   Qual o tempo de execução do algoritmo Quicksort quando utilizamos essa ideia, no pior caso e no caso médio?

   (E se utilizássemos o próprio algoritmo Quicksort para ordenar o subconjunto? qual seria o desempenho do algoritmo nesse caso?)

4. **Ordenação de listas encadeadas**

   Você consegue escrever uma implementação do algoritmo Quicksort que ordena uma lista encadeada com n elementos em tempo (médio) $O(n \log n)$?

   E o algoritmo Mergesort?

   Você consegue escrever uma implementação desse algoritmo que ordena uma lista encadeada com n elementos em tempo $O(n \log n)$?