01

Análise assimptótica das ordenações e questionando algoritmos

Transcrição

Depois de entender como cresce o particiona, veremos a ordenação baseada nesta função: o `TestaOrdenacaoRapida`.

O `TestaOrdenacaoRapida` chama o método `ordena`.

```
private static void ordena(Nota[] notas, int de, int ate) {
    int elementos = ate - de;
    if(elementos > 1) {
        int posicaoDoPivo = particiona(notas, de, ate);
        ordena(notas, de, posicaoDoPivo);
        ordena(notas, posicaoDoPivo + 1, ate);
```

O que ele fazia? Ele primeiro particionava o *array* inteiro, ou seja, é linear e realizava *n* operações. Logo, ele chamava o próprio algoritmo para uma das metades, depois, para outra. Cada vez que ele chamava o algoritmo, ele também executava o `particiona`. Ou seja, é o linear (*n*) multiplicado pelo número de vezes que chamamos o `ordena`. E quantas vezes chamamos o `ordena`? Como calculamos o número de repetições do processo em que dividimos o *array* em dois trechos e selecionamos apenas uma parte? Nós já vimos um cálculo parecido anteriormente, duas vezes. O que podemos imaginar é que o número de operações da ordenação irá crescer de acordo que realizarmos *n* operações - devido ao `particiona` - multiplicado pela quantidade de vezes que chamamos o `ordena` (*log n*). Este algoritmo crescerá como um *Merge Sort*, também será **n * log(n)**, ou seja, *Θ(n log(n))*.

Comparando o sort rápido com o Merge Sort

Vamos comparar o nosso *Merge Sort* com a implementação de ordenação baseada no `particiona`, em posicionamos o pivô e particionamos. Um deles é o **n * log n** (o *Merge Sort*).

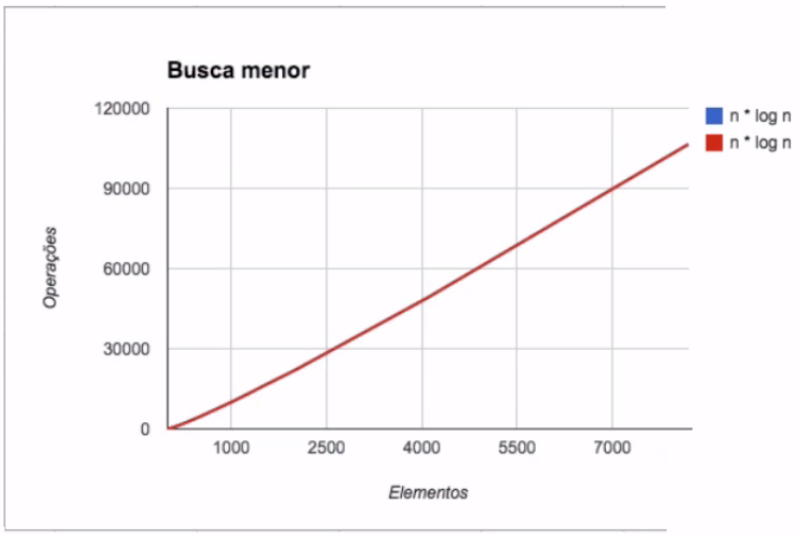
Elementos	n^2	n * log n
1	1	0
2	4	2
4	16	8
8	64	24
16	256	64
32	1024	160
64	4096	384
128	16384	896
256	65536	2048
512	262144	4608
1024	1048576	10240
2048	4194304	22528
4096	16777216	49152
8192	67108864	106496

O outro a algoritmo será **n * log n**, que será o nosso *sort* novo.

Elementos	n * log n	n * log n
1	0	0
2	2	2
4	8	8
8	24	24
16	64	64
32	160	160
64	384	384
128	896	896
256	2048	2048
512	4608	4608
1024	10240	10240
2048	22528	22528
4096	49152	49152
8192	106496	106496

Então, um algoritmo crescerá **n * log n** e o outro crescerá **n * log n**.

Elementos	n * log n	n * log n
1	0	0
2	2	2
4	8	8
8	24	24
16	64	64
32	160	160
64	384	384
128	896	896
256	2048	2048
512	4608	4608
1024	10240	10240
2048	22528	22528
4096	49152	49152
8192	106496	106496

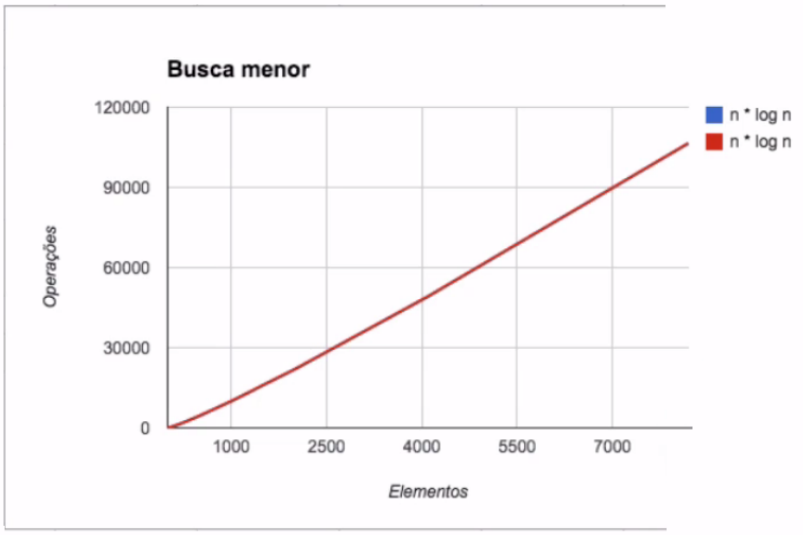


Vamos comparar os dois algoritmos no gráfico. As linhas de ambos estão emparelhadas. Os dois crescem da mesma maneira! Como poderemos comparar o *Merge Sort* e o novo algoritmo, se eles crescem da mesma maneira?

Quicksort

Temos dois algoritmos de ordenação novos: o *Merge Sort* e o novo *Sort*. Os dois crescem de maneira *n log(n)*. Queremos descobrir qual é o melhor para ser usado...

Elementos	n * log n	n * log n
1	0	0
2	2	2
4	8	8
8	24	24
16	64	64
32	160	160
64	384	384
128	896	896
256	2048	2048
512	4608	4608
1024	10240	10240
2048	22528	22528
4096	49152	49152
8192	106496	106496

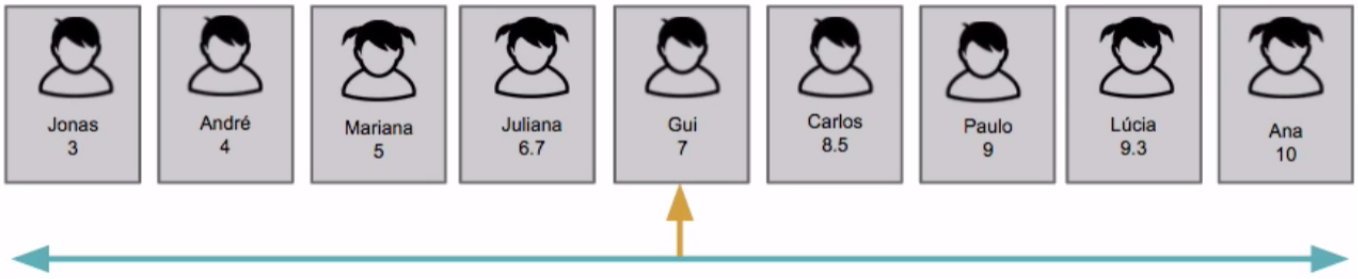


Na prática, costumamos comparara quantas operações os algoritmos fazem em média. Porém, como eles crescem da mesma maneira, teremos que analisar outros detalhes, por exemplo, quantas trocas eles fazem? Então, ele é *2(n log(n)* ou *3(n log(n)*? Ele é 2(n log(n) + 5) ou 2(n log(n) + 17)? Em média, trabalhando com dados reais, o que acontecerá com o *Merge Sort* e o novo *sort*? Quais dos dois irá terá um melhor comportamento? Este é o tipo de análise feita nesta situação, pelos cientistas da computação.

A resposta dos especialistas é que, em média, o novo algoritmo é mais rápido do que o algoritmo de intercalação, o *Merge Sort*. O novo algoritmo mais rápido de ordenação nós chamamos de ***Quicksort***. Ele usa o pivô para particionar e ordenar. Na prática, ele executa um número menor de operações do que o *Merge Sort* - ainda que ambos cresçam de forma parecida.

Parabéns

Conhecemos diversos algoritmos em que trabalhávamos com uma quantidade de elementos dentro do *array*, depois a dividíamos em partes menores, e depois atacá-las separadamente.



Por exemplo, com o ***Merge Sort***, nós chamamos o algoritmo para um trecho, e depois, par ao outro. No ***Quicksort*** igualmente, chamávamos o algoritmo para um trecho e em seguida, para o outro. No **busca binária** o que fazíamos? Chamávamos o algoritmo para uma parte, depois, para a outra. Mas a base do que estamos fazendo é: nós dividimos um problema e depois, juntamos as partes e assim teremos um resultado final. No fim do *Quicksort*, como resultado nós tínhamos o *array* ordenado. No *Merge Sort*, nós intercalávamos os dois trechos separados. No *busca binária*, separávamos o resultado do lado em que continuávamos buscando. Ou seja, nós sempre dividimos o nosso problema e depois, juntávamos o resultado das divisões. Assim, chegávamos a um resultado final. Este tipo de solução ou técnica é chamada de **divisão e conquista**. Se não conseguimos conquistar diretamente o problema, porque tornará a conquista demorada (como o *Selection Sort* e o *Insertion Sort*). Então, o que fazemos é dividir o problema em partes. A estratégia pode ser melhor como foi nos três casos, em que dividimos e conquistamos. Com isto, nós criamos algoritmos que são **logarítmicos** - *n log(n)* ou *log* - que se saíram melhor do que os anteriores.

No mundo real, sempre iremos questionar qual algoritmo vale a pena implementar, quando ele ainda não foi implementado. Mas a base do nosso curso é compreender que existem diversos algoritmos para resolver os mesmo tipos de problemas, em cada situação cada um será o mais apropriado para ser usado. Alguns funcionarão melhor em determinadas situações e outros funcionarão em qualquer situação. Uns podem ser mais lentos, outros mais rápidos, alguns consumirão mais ou menos memória. Todos estes algoritmos são usados para resolver problemas computacionais, como por exemplo:

Quero ordenar.

Quem é o menor? Quem é o maior?

Quero saber se um elemento está dentro da lista.

Em qual posição um determinado elemento ficou?

Problemas como estes, podemos resolver usando diversos tipos de algoritmos. Nestes primeiros cursos , nós analisamos e entendemos para que servem os algoritmos e como podemos compará-los. A partir de agora, quando conhecermos novos algoritmos, sempre poderemos observar se eles são lentos ou rápidos, se são apropriados para uma situação ou se são complicados de serem aplicados em outra.