01

Análise assimptótica das divisões

Transcrição

E os outros algoritmos de ordenação que nós já vimos? O quão rápido são eles e o que eles fazem exatamente? É o que iremos descobrir.

O primeiro algoritmo que conhecemos, tentava intercalar os elementos que tínhamos.

O que fazia a função `intercala` ? Vamos analisar o código:

```
private static void intercala(Nota[] notas, int inicial, int miolo, int termino) {
    Nota[] resultado = new Nota(termino - inicial);

    int atual = 0;
    int atual1 = inicial;
    int atual2 = miolo;
    while(atual1 < miolo &&
          atual2 < termino) {
        Nota nota1 = notas[atual1];
        Nota nota2 = notas[atual2];
        if(nota1.getValor() < nota2.getValor()) {
            resultado[atual] = nota1
            atual++;
        } else {
            resultado[atual] = nota2;
            atual2++;
        }
        atual++;
    }

    }
```

Se o *array* tivesse *n* elementos, o `intercala` passava por cada um dos itens. Isto significa que ele fará *n* operações. Talvez, ele não faça todas as operações no trecho do código acima, mas ele terminará no `while` que está logo abaixo desta parte.

```
while(atual1 < miolo &&
          atual2 < termino) {
        Nota nota1 = notas[atual1];
        Nota nota2 = notas[atual2];
        if(nota1.getValor() < nota2.getValor()) {
            resultado[atual] = nota1
            atual++;
        } else {
            resultado[atual] = nota2;
            atual2++;
        }
        atual++;
}
while(atual1 < miolo) {
    resultado[atual] = notas[atual1];
```

```
        atual1++;
        atual++;
    }

    while(atual2 < termino) {
        resultado[atual] = notas[atual2];
        atual2++;
        atual++;
    }
```

Para intercalarmos dois trechos de um *array*, ele precisará fazer *n* operações. Depois, com o `for` faremos mais *n* operações novamente.

```
    for(int contador = 0; contador < atual, contador++) {
        notas[inicial + contador] = resultado[contador];
    }
```

Porém, *2n* não fará diferença na nossa análise. Da maneira como estamos analisando, `2n` , `2n + 17` , `2n - 35` , todos se comportam igualmente a *n*. Por quê? Porque todos crescerão de forma linear no gráfico. Eles não crescerão exponencialmente ou quadraticamente... Ele crescerá linearmente. Para nós, é um fator interessante, considerando que buscamos algoritmos mais rápidos do que o linear.

Então, com o `intercala` teremos que fazer *n* operações. O algoritmo que intercala dois trechos de um *array* é linear.

E o desempenho de merge sort

Sabemos que para intercalar o trecho de um *array*, executaremos uma operação linear. Isto significa que precisaremos realizar *n* operações. A ordenação que fizemos anteriormente usava o `intercala` diversas vezes, ou seja, executava várias vezes a quantidade *n* de operações. Se *n* for um número fixo como três ou quatro vezes, tudo bem. O problema é que não trabalharemos com números fixos. Como funcionará o algoritmo de ordenar neste caso?

```
    private static void ordena(Nota[] notas, int inicial, int termino) {
        int quantidade = termino - inicial;
        if(quantidade > 1) {
            int meio = (inicial + termino) / 2;
            System.out.println(inicial + " " + termino + " " + meio);

        ordena(notas, inicial, meio);
        ordena(notas, meio, termino);
        intercala(notas, inicial, meio, termino);

        }
    }
```

Ele separava metade do *array*, ordenava um trecho e depois, intercalava todos os elementos. Ou seja, ele dividia por 2, depois, dividia por 2 novamente e seguia repetindo a divisão várias vezes. Em cada uma delas, ele intercalava. Quantas vezes ele executará a divisão por 2, até chegar a quantidade de um elemento?

Da mesma forma que na busca binária nós repetimos diversas vezes a divisão por 2, nós queremos descobrir qual a potência de 2 que resultará no número total de elementos. Nós já conhecemos este algoritmo! Assim como na busca binária, nós

repetiremos o processo de divisão até restar um elemento. Quantas vezes precisaremos repetir a operação? A resposta é: ***log*** **do número, na** ***base 2***.

Se utilizarmos o `ordena`, ele irá executar o `intercala` diversas vezes, o que será a quantidade de *n* operações. Então, nosso algoritmo será *n* multiplicado pelo número de vezes que a operação será executada (*log n*). Ou seja, ele será *n log n*. Em seguida, iremos compará-lo com os outros algoritmos de ordenação que conhecemos.

Como funciona o algoritmo *n log n*? Ele quebra o *array* e intercala as partes menores. Assim, temos um algoritmo que executa *n log n*.

Comparando o mergesort com outros sorts

Veremos a comparação do algoritmo do tipo *selection sort* e *insertion sort* que são quadráticos, ou seja, são **n^2** (*n* multiplicado por ele mesmo).

Elementos	n^2	log n
1	1	0
2	4	1
4	16	2
8	64	3
16	256	4
32	1024	5
64	4096	6
128	16384	7
256	65536	8
512	262144	9
1024	1048576	10
2048	4194304	11
4096	16777216	12
8192	67108864	13

Nós iremos comparar o número de operações de um **quadrático** com um algoritmo que é **n * log n**.

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 *sort* novo divide e mergeia (intercala). Se observarmos a tabela, com 64 elementos, a ordenação **quadrática** fará 4096 operações, enquanto, a ordenação **n * log n** fará 384.

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

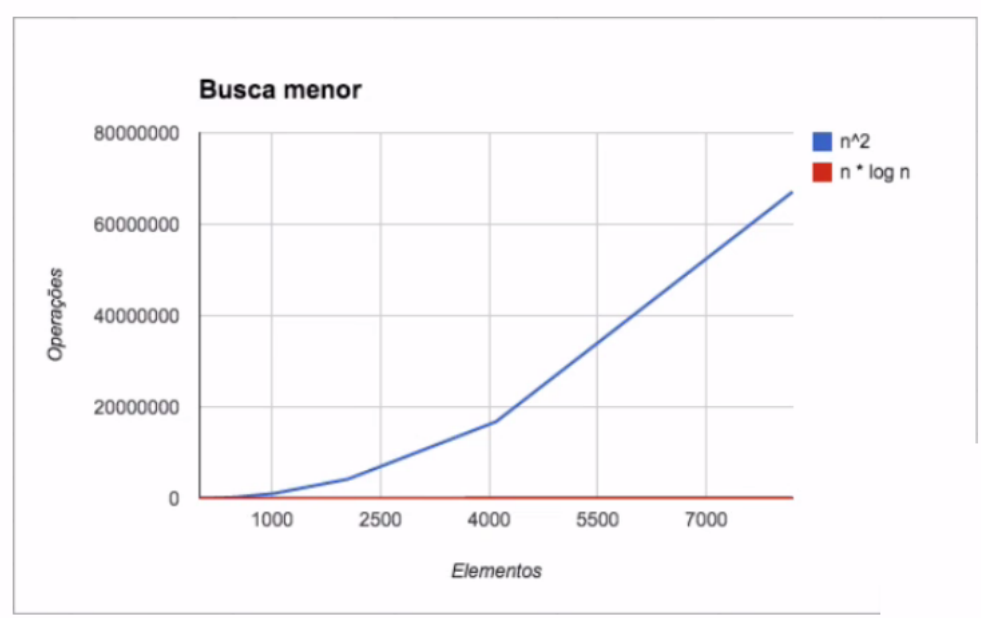
Talvez, uma operação dez vezes mais rápida não pareça o suficiente boa. Mas se compararmos os algoritmos utilizados em um *array* com 8.192 de elementos, o **quadrático** fará 67.108.864 operações para ordenar o *selection sort* e o *insertion sort*. Não parece um desempenho bom.

A	B	C
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

Se tivéssemos um baralho com 8192 cartas e precisássemos ordená-las, o que faríamos? Dividiríamos o monte com outras pessoas, porque o número de operações que faríamos seria muito menor. Nós iríamos dividir e intercalar. Por exemplo, para a mesma quantidade de elementos, com o algoritmo novo faríamos 106.496 operação. Uma diferença grande na quantidade de operações.

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

Podemos ver no gráfico, a diferença de crescimento dos algoritmos.

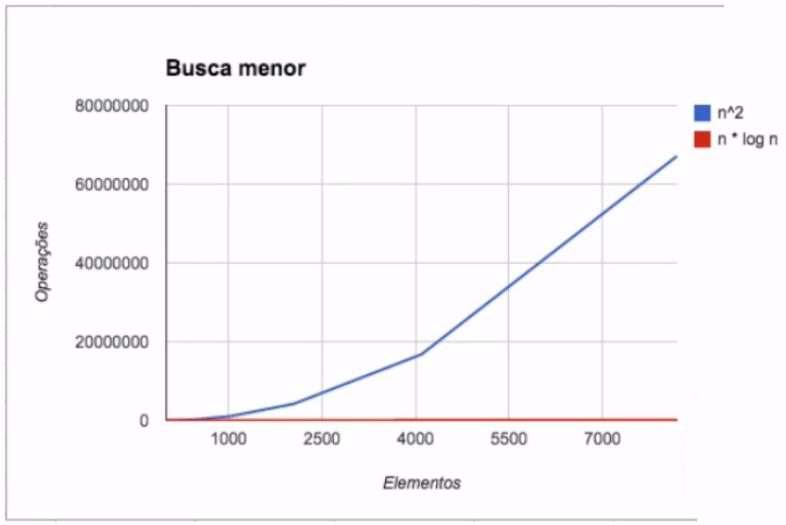


Desejo boa sorte para quem quiser ordenar um *array* com um algoritmo quadrático.

Analisando o mergesort

Nós vimos que o algoritmo novo de ordenação é **n * log n**, que primeiro divide e depois intercala. Como ele é baseado em **intercalar** os elementos, o nome do algoritmo será relacionado com fundir dois trechos intercalando: ***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



Trabalhar com *merge sort* é mais rápido do que com o *Selection Sort* ou *Insertion Sort*. Quando trabalhamos com uma quantidade pequena de elementos, não fará muita diferença para o computador.

Elementos	n^2	n * log n
1	1	0
2	4	2
4	16	8
8	64	24

De 64 para 24 operações, a diferença é irrelevante. Mesmo que um algoritmo seja dez vezes mais lento para 64 elementos, a diferença também é irrelevante.

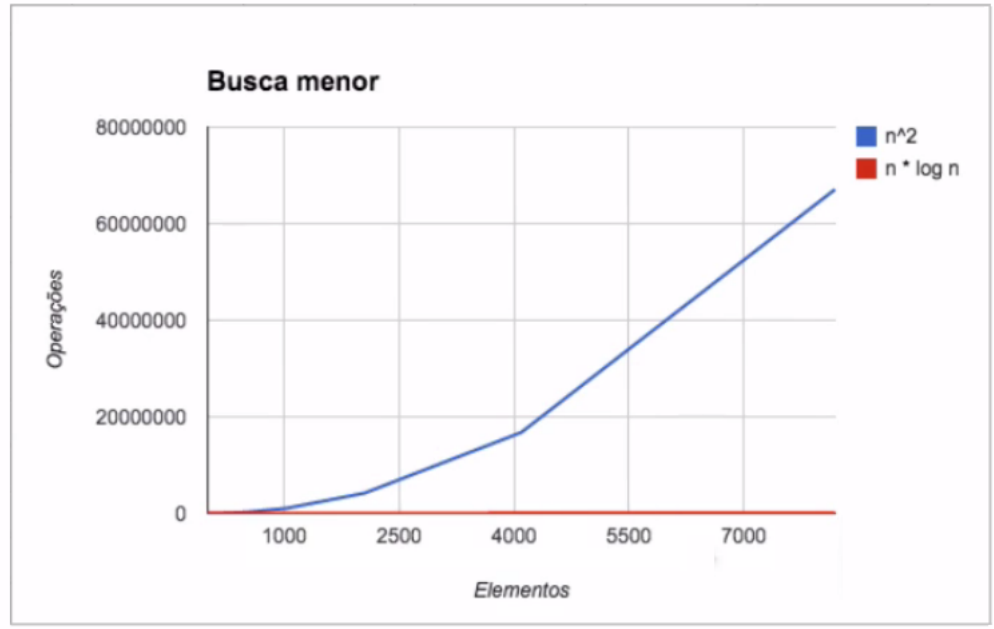
Elementos	n^2	n * log n
1	1	0
2	4	2
4	16	8
8	64	24
16	256	64
32	1024	160
64	4096	384

No cotidiano, se você tiver que fazer 4 mil ou 400 operações, o computador executará em um piscar de olhos. Porém, se aumentamos significantemente o número de elementos, você terá dificuldades em conseguir executar todas.

Se aumenta o número de usuários acessando a mesma máquina simultaneamente, compartilhando o mesmo processador, não iremos conseguir. Poderíamos usar uma ordenação quadrática para um número baixo de elementos. No entanto, com um número de elementos elevado, a diferença será grotesca.

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

Se aumentarmos o número de pessoas querendo acessar simultaneamente o processador, iremos sobrecarregá-lo. A ordem de grandeza de um algoritmo irá influenciar bastante na rapidez. Quanto maior for o número de elementos, maior será a distancia das linhas no gráfico.



A análise que fazemos considera o número de *sort*s no *Selection Sort* e no *Merge Sort*, verificando o crescimento a longo prazo. A linha azul cresce quadraticamente e ficará inviável. A linha vermelha do *Merge Sort* crescerá *n·log(n)*.

Analisando o particiona

Da mesma maneira que analisamos o algoritmo do `Merge Sort`, vamos analisar também o outro algoritmo de ordenação que implementamos. Iremos analisar sem nos focarmos no "melhor" ou no "pior" caso, mas como o algoritmo cresce em geral.

Nós fizemos uma função da classe `TestaPivota`, que pivotava. Como ela funcionava?

```
private static int particiona(Nota[] notas, int inicial, int termino) {
    int menoresEncontrados = 0;

    Nota pivo = notas[termino - 1];
    for(int analisando = 0; analisando < termino - 1; analisando++) {
        Nota atual = notas[analisando];
        if(atual.getValor() <= pivo.getValor()) {
            troca(notas, analisando, menoresEncontrados);
            menoresEncontrados++;
        }
    }
    troca(notas, termino -1, menoresEncontrados);
    return menoresEncontrados;
}
```

Ele primeiro executava um `for` que passava por todos os elementos entre o `inicio` e o `termino`.

```
for(int analisando = 0; analisando < termino - 1; analisando++) {
        Nota atual = notas[analisando];
        if(atual.getValor() <= pivo.getValor()) {
            troca(notas, analisando, menoresEncontrados);
            menoresEncontrados++;
```

```
        }
    }
```

Isto significa que o nosso `for` realiza quantas operações? A resposta é ***n***. Poderia ser também *2n*, *5n*, *5n* - 3, porém, nós estamos interessados na grandeza, na maneira como o algoritmo cresce. Neste caso, ele irá crescer de forma **linear**.

Para nós particionarmos o *array* com o pivô, a quantidade de operações realizadas crescerá de acordo com o número de elementos analisados. Se temos dez elementos, teremos que passar por cada um deles para encontrar a posição correta do pivô. O mesmo acontecerá se tivermos vinte elementos: igualmente teremos que passar por todos os itens, antes de descobrirmos a posição do pivô. Isto acontece, porque teremos que identificar todos os elementos menores do que o pivô. Então, passamos pelo *array* inteiro, do `início` ao `término` do trecho analisado... Logo, o `particiona` crescerá de acordo com o número de elementos. Ele é linear.