Construção e Análise de Algoritmos

aula 06: Notação assintótica

1 Introdução

Nós temos dito diversas vezes ao longo do curso que os nossos algoritmos executam em tempo $\Theta(n^2)$ ou $\Theta(n \log n)$, explicando que isso é a "ordem de magnitude"do seu tempo de execução.

Na aula de hoje, nós vamos ver com precisão o que isso quer dizer.

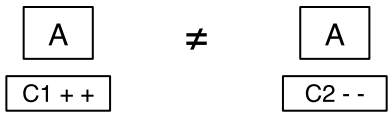
Mas, vejamos primeiramente o que isso não quer dizer.

Suponha, por exemplo, que um certo algoritmo A possui tempo de execução $\Theta(n \log n)$

- *Isso quer dizer que o seu tempo de execução é (aproximadamente) $n \log n$?*

Não, isso não é verdade.

Se o algoritmo é executado em computadores diferentes, um bem rápido e outro bem lento

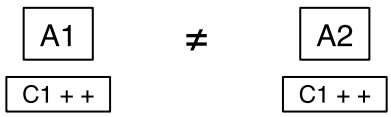


então os tempos de execução nos dois casos serão bem diferentes (e não podem ambos ser $n \log n$).

Suponha agora que nós temos dois algoritmos A_1 e A_2, ambos com tempo de execução $\Theta(n \log n)$.

- *Isso quer dizer que os dois algoritmos rodam (aproximadamente) no mesmo tempo, quando executados no mesmo computador?*

Não, isso não é verdade (necessariamente).

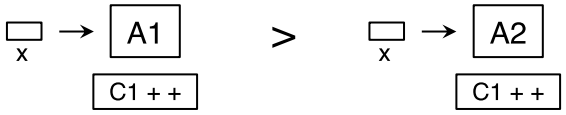


É possível que A_1 seja, digamos, 20 vezes mais rápido que A_2, e ainda assim a análise indique que ambos possuem tempo $\Theta(n \log n)$.

Suponha, finalmente, que o algoritmo A_1 possui tempo $\Theta(n \log n)$, e que o algoritmo A_2 possui tempo $\Theta(n^2)$.

- *Isso quer dizer que o algoritmo A_1 roda mais rápido que o algoritmo A_2, quando ambos executam no mesmo computador?*

Não, isso não é verdade (necessariamente).



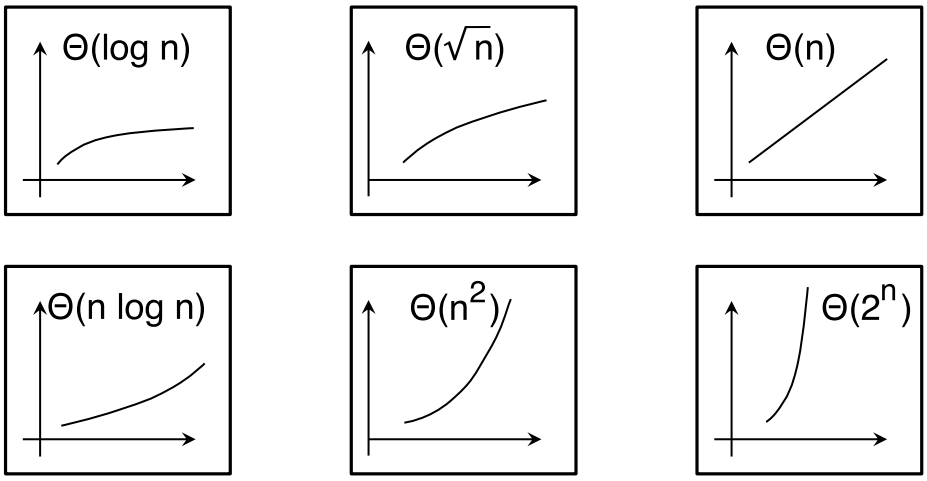
É possível que, para algumas entradas específicas, o tempo de execução de A_1 seja maior do que o tempo de execução de A_2.

Mas, o que a notação Θ significa então?

2 O gráfico do tempo de execução

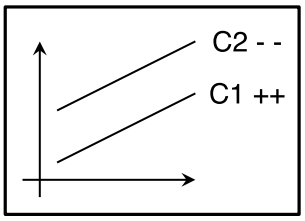
Quando dizemos que um algoritmo executa em tempo $\Theta(n \log n)$ ou $\Theta(n^2)$, isso nos dá uma indicação de como é o gráfico do seu tempo de execução — o formato da curva.

Abaixo nós temos diversos exemplos



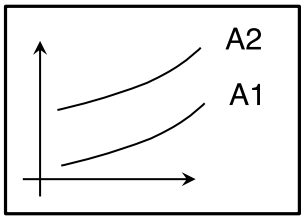
Agora, nós podemos entender as respostas que foram dadas na seção anterior.

Se um mesmo algoritmo de tempo $\Theta(n)$ é executado em dois computadores diferentes, um bem rápido e outro bem lento, os gráficos dos respectivos tempos de execução seriam algo como



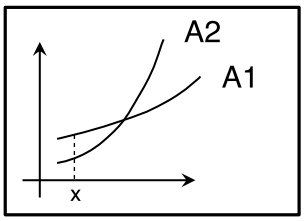
Isto é, os tempos reais de execução são bem diferentes, mas em ambos os casos o gráfico tem a forma linear — o que reflete o fato de que o algoritmo possui tempo $\Theta(n)$.

De maneira análoga, se dois algoritmos A_1 e A_2 com tempo $\Theta(n \log n)$ são executados no mesmo computador, os gráficos dos respectivos tempos de execução poderiam ser algo como



Novamente, os tempos de execução podem ser bem diferentes, mas as duas curvas possuem o mesmo formato — o que reflete o fato de que os dois algoritmos possuem tempo $\Theta(n \log n)$.

Finalmente, um algoritmo $\Theta(n^2)$ pode ser mais rápido que um algoritmo $\Theta(n \log n)$, em alguns casos, pois as respectivas podem se cruzar no gráfico



Isto é, no exemplo acima, para a entrada x, o algoritmo $\Theta(n^2)$ é mais rápido que o algoritmo $\Theta(n \log n)$.

3 Taxas de crescimento

Apesar da última observação acima, parece intuitivamente claro que é melhor ter um algoritmo de tempo $\Theta(n \log n)$ do que um algoritmo de tempo $\Theta(n^2)$.

Mas, porque?

Para respoder essa pergunta, considere um algoritmo A de tempo $\Theta(n)$.

Mais especificamente, suponha que o tempo de execução do algoritmo A é dado por

$$T(n) \quad = \quad 7n$$

Isso significa que, quando fornecemos uma entrada de tamanho $n = 100$, por exemplo, a execução do algoritmo A leva tempo

$$T(100) \quad = \quad 7 \cdot 100 \quad = \quad 700$$

Agora, veja o que acontece quando nós dobramos o tamanho da entrada

$$T(200) \quad = \quad 7 \cdot 200 \quad = \quad 1400 \quad = \quad 2 \cdot 700$$

Ou seja, o tempo de execução dobra também.

A observação interessante, a seguir, é que a mesma coisa acontece quando o tempo de execução do algoritmo é, digamos,

$$T(n) \quad = \quad 7n$$

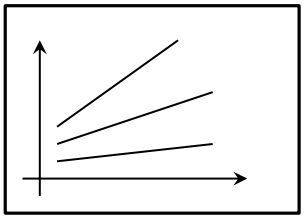
Veja só

$$T(100) \quad = \quad 5 \cdot 100 \quad = \quad 500 \qquad\qquad T(200) \quad = \quad 5 \cdot 200 \quad = \quad 1000$$

De fato, esse comportamento é característico de qualquer algoritmo cujo tempo de execução tenha a forma

$$T(n) \quad = \quad k \cdot n$$

E é isso basicamente o que significa dizer que um algoritmo possui tempo $\Theta(n)$ — isto é, o seu tempo de execução cresce de maneira linear com o tamanho da entrada.



Agora, considere um algoritmo que executa em tempo

$$T(n) \quad = \quad 7n^2$$

Quando esse algoritmo recebe uma entrada de tamanho $n = 10$, a sua execução leva tempo

$$T(10) \quad = \quad 7 \cdot 10^2 \quad = \quad 700$$

Agora, observe o que acontece quando nós dobramos o tamanho da entrada

$$T(20) \quad = \quad 7 \cdot 20^2 \quad = \quad 2800 \quad = \quad 4 \cdot 700$$

Ou seja, o tempo de execução quadruplicou.

E se, ao invés de duplicarmos o tamanho da entrada, nós triplicamos

$$T(30) \quad = \quad 7 \cdot 30^2 \quad = \quad 6300 \quad = \quad 9 \cdot 700$$

o tempo de execução é multiplicado por 9.

O que está acontecendo aqui é que o tempo de execução está variando de maneira quadrática

com relação à variação no tamanho da entrada.

Não é difícil ver que qualquer algoritmo com tempo de execução da forma

$$T(n) \quad = \quad k \cdot n^2$$

tem esse comportamento.

(Verifique isso!)

E isso é o que significa dizer que um algoritmo possui tempo $\Theta(n^2)$.

Finalmente, considere um algoritmo que executa em tempo $\Theta(\log n)$.

Mais especificamente, suponha que o seu tempo de execução é exatamente dado por

$$T(n) \quad = \quad \log_2 n$$

Quando esse algoritmo recebe uma entrada de tamanho $n = 1024$, a sua execução leva tempo

$$T(1024) \quad = \quad \log_2 1024 \quad = \quad 10$$

Agora, observe o que acontece quando nós dobramos o tamanho da entrada

$$T(2048) \quad = \quad \log_2 2048 \quad = \quad 11$$

Ou seja, o tempo de execução aumenta de apenas 1 unidade.

E veja o que acontece quando nós multiplicamos o tamanho da entrada por 4 e por 8:

$$T(4096) \quad = \quad \log_2 4096 \quad = \quad 12 \qquad\qquad T(8192) \quad = \quad \log_2 8192 \quad = \quad 13$$

Isto é, a cada vez que nós dobramos o tamanho da entrada o tempo de execução aumenta de 1 unidade.

Caso o tempo de execução do algoritmo seja dado por

$$T(n) \quad = \quad 7 \cdot \log_2 n$$

o comportamento que obtemos é o seguinte

$$T(1024) \quad = \quad 7 \cdot \log_2 1024 \quad = \quad 70 \qquad\qquad T(2048) \quad = \quad 7 \cdot \log_2 2048 \quad = \quad 77$$

Isto é, a cada vez que nós dobramos o tamanho da entrada o tempo de execução do algoritmo aumenta de 7 unidades.

Em geral, se o tempo de execução do algoritmo é dado por

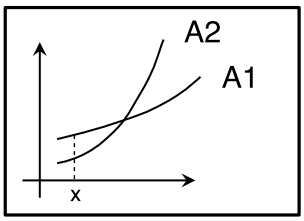
$$T(n) \quad = \quad k \cdot \log_2 n$$

ele possui o seguinte comportamento: a cada vez que nós dobramos o tamanho da entrada o tempo de execução aumenta de um valor constante k.

E é isso, basicamente, o que significa dizer que um algoritmo possui tempo $\Theta(\log n)$.

Agora que nós temos uma ideia mais clara do significado da notação Θ, nós podemos explicar porque, em geral, é preferível ter um algoritmo A_1 de tempo $\Theta(n \log n)$ do que um algoritmo A_2 de tempo $\Theta(n^2)$:

- pode até ser que, para entradas pequenas, o tempo do algoritmo A_1 seja menor do que o tempo do algoritmo A_2
- mas, quando o tamanho das entradas aumenta, o tempo do algoritmo A_2 aumenta mais rápido do que o tempo do algoritmo A_1
- e isso significa que, a partir de um certo ponto, o tempo de execução de A_1 será sempre menor do que o tempo de execução de A_2



4 Termos de menor ordem

Na seção anterior nós aprendemos, entre outras coisas, que o coeficiente que aparece na frente da expressão do tempo de execução de um algoritmo não é relevante para a notação Θ.

Isto é,

- qualquer algoritmo com tempo de execução $k \cdot n$ é descrito como $\Theta(n)$
- qualquer algoritmo com tempo de execução $k \cdot n^2$ é descrito como $\Theta(n^2)$
- e assim por diante ...

A seguir, nós vamos ver que os termos de menor ordem também não são muito relevantes.

Para entender isso, considere um algoritmo A cujo tempo de execução é dado por

$$T(n) \quad = \quad 7n \, + \, 80$$

Quando nós fornecemos uma entrada de tamanho $n = 10$ para esse algoritmo, a sua execução

leva tempo

$$T(10) \quad = \quad 7 \cdot 10 \; + \; 80 \quad = \quad 150$$

E, quando nós dobramos o tamanho da entrada, o tempo de execução passa a ser

$$T(20) \quad = \quad 7 \cdot 20 \; + \; 80 \quad = \quad 220$$

Isto é, desta vez, ao dobrar o tamanho da entrada, o tempo de execução do algoritmo não dobrou — e, assim, o algoritmo não parece ser $\Theta(n)$.

Mas, veja o que acontece quando nós consideramos uma entrada muito grande, por exemplo $n = 10000$

$$T(10000) \quad = \quad 7 \cdot 10000 \; + \; 80 \quad = \quad 70080$$

Agora, dobrando o valor da entrada, nós obtemos

$$T(20000) \quad = \quad 7 \cdot 20000 \; + \; 80 \quad = \quad 140080$$

Ou seja, o tempo de execução aproximadamente dobrou.

E, quando consideramos entradas ainda maiores, a influência do termo 80 fica ainda menos relevante.

Portanto, mesmo nesse caso, nós descrevemos o tempo de execução do algoritmo como $\Theta(n)$.

Mais precisamente, tudo o que importa para a notação Θ é o comportamento do tempo de execução do algoritmo para entradas muito grandes.

Assim,

- se um algoritmo possui tempo $\Theta(n)$, isso significa que o seu tempo de execução cresce de maneira (aprox.) linear com o tamanho da entrada (para entradas muito grandes)

Vejamos outro exemplo.

Considere um algoritmo que executa em tempo

$$T(n) \quad = \quad n^2 \; + \; 8n \; + \; 80$$

Examinando entradas pequenas

$$T(10) \quad = \quad 10^2 \; + \; 8 \cdot 10 \; + \; 80 \quad = \quad 260 \qquad\qquad T(20) \quad = \quad 20^2 \; + \; 8 \cdot 20 \; + \; 80 \quad = \quad 640$$

nós não detectamos a variação quadrática no tempo de execução.

Mas, considerando entradas muito grandes, esse comportamento aparece

$T(1000) = 1000^2 + 8 \cdot 1000 + 80 = 1008080$ $\qquad$ $T(2000) = 2000^2 + 8 \cdot 2000 + 80 = 4016080$

Portanto, ao analisar o tempo de execução de um algoritmo utilizando a notação Θ, nós

- não apenas desconsideramos o coeficiente do termo de maior ordem
- como também desconsideramos todos os termos de ordem menor

Dessa maneira, um algoritmo com tempo de execução

$$T(n) \quad = \quad 7n^2 \, + \, 5n \, + \, 180$$

é descrito simplesmente como $\Theta(n^2)$

- isto é, um algoritmo cujo tempo de execução cresce de maneira (aprox.) quadrática com o tamanho da entrada (para entradas muito grandes)

5 Analisando algoritmos com a notação Θ

Nessa seção, nós vamos ver que a notação Θ é muito conveniente para analisar o tempo de execução de algoritmos.

Considere o seguinte esquema de algoritmo:

```
Procedimento  Algoritmo1 (E: entrada de tamanho n)
 {
     cmd;    |
     ...     |   bloco 1
     cmd;    |

     Para  i <- 1  Até n
        Para  j <- 1  Até n
         {
             cmd;    |
             ...     |   bloco 2
             cmd;    |
         }
 }
```

A ideia aqui é que cada linha com `cmd;` corresponde a uma instrução simples, que executa em tempo constante (i.e., que não depende de n).o

Como se pode ver, o algoritmo possui um bloco de instruções desse tipo no início, e um outro

bloco no interior do laço mais interno.

Nós podemos assumir que o primeiro bloco de instruções executa, digamos, em tempo l, e que o segundo bloco executa, digamos, em tempo k.

Não é difícil ver que o primeiro bloco é executado apenas uma vez, enquando que o segundo bloco é executado n^2 vezes (que é o número total de voltas realizadas pelo laço mais interno).

Portanto, o tempo de execução do algoritmo é dado por

$$T(n) \quad = \quad kn^2 \; + \; l$$

Mas, nós já sabemos que, do ponto de vista da notação Θ, o coeficiente k e o termo l não importam, e o tempo de execução é descrito simplesmente por $\Theta(n^2)$.

Em certo sentido, esse passo de raciocínio corresponde a assumir que todo bloco de instruções simples executa em tempo $\Theta(1)$ (i.e., uma constante que não nos interessa).

Com essa suposição, o tempo de execução do algoritmo pode ser descrito como

$$T(n) \quad = \quad \Theta(1) \; + \; n^2 \cdot \Theta(1) \quad = \quad \Theta(n^2)$$

Agora, suponha que após os dois laços aninhados foi acrescentado um terceiro bloco de instruções simples

```
cmd;    |
...     |   bloco 3
cmd;    |
```

Nesse caso, o tempo de execução do algoritmo passa a ser

$$T(n) \quad = \quad \Theta(1) \; + \; n^2 \cdot \Theta(1) \; + \; \Theta(1) \quad = \quad \Theta(n^2)$$

Ou seja, o novo bloco de instruções não afeta o tempo de execução do algoritmo de maneira relevante (do ponto de vista da notação assintótica).

A seguir, assumindo que o terceiro bloco de instruções aparece dentro de um laço

```
Para  h <- 1  Até n
 {
     cmd;    |
     ...     |   bloco 3
     cmd;    |
 }
```

o tempo de execução passa a ser

$$T(n) \quad = \quad \Theta(1) \ + \ n^2 \cdot \Theta(1) \ + \ n \cdot \Theta(1) \quad = \quad \Theta(n^2)$$

Mesmo se o terceiro bloco aparece no interior de dois laços aninhados

```
Para  g <- 1  Até n
   Para  h <- 1  Até n
    {
        cmd;    |
        ...     |   bloco 3
        cmd;    |
    }
```

isso não afeta o tempo de execução do algoritmo (do ponto de vista da notação Θ):

$$T(n) \quad = \quad \Theta(1) \ + \ n^2 \cdot \Theta(1) \ + \ n^2 \cdot \Theta(1) \quad = \quad \Theta(n^2)$$

Mas, adicionando mais um laço

```
Para  f <- 1  Até n
   Para  g <- 1  Até n
      Para  h <- 1  Até n
       {
           cmd;    |
           ...     |   bloco 3
           cmd;    |
       }
```

a estimativa assintótica do tempo de execução muda:

$$T(n) \quad = \quad \Theta(1) \ + \ n^2 \cdot \Theta(1) \ + \ n^3 \cdot \Theta(1) \quad = \quad \Theta(n^3)$$

Note que existe uma álgebra curiosa para a manipulação de termos Θ. Por exemplo,

- $\Theta(1) \ + \ \Theta(1) \ = \ \Theta(1)$
- $\Theta(1) \ + \ \Theta(n) \ = \ \Theta(n)$
- $n \cdot \Theta(1) \ = \ \Theta(n)$

Mas, isso corresponde apenas ao fato de que nós descartamos todos os termos de ordem menor, e desconsideramos o coeficiente do termos dominante.

Do ponto de vista concreto, isso significa que, no momento da análise de um algoritmo, basta identificar o trecho de código que corresponde ao termo dominante e estimar a sua contribuição para o tempo de execução do algoritmo.

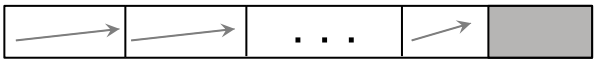
6 Estudo de caso: Multilistas

Quando utilizamos um vetor `V[1..n]` para armazenar uma lista de números, surge o seguinte dilema: devemos manter o vetor ordenado ou desordenado?

Quando mantemos o vetor desordenado, a operação de inserção leva tempo $\Theta(1)$ (basta inserir o novo elemento no final), mas a busca e a remoção levam tempo $\Theta(n)$ (pois, no pior caso, é preciso examinar a lista inteira para localizar o elemento).

Quando mantemos o vetor ordenado a situação mais ou menos se inverte: a operação de busca passa a levar tempo $\Theta(\log n)$ (utilizando o procedimento de busca binária), e a inserção passa a levar tempo $\Theta(n)$ (pois, no pior caso, é preciso deslocar a lista inteira para abrir espaço para o novo elemento) — a remoção também leva tempo $\Theta(n)$.

A ideia das multilistas consiste em dividir o vetor em um conjunto de pequenas listas ordenadas



Dessa maneira, nós conseguimos reduzir o tempo da inserção e da remoção, aumentando um pouco o tempo da busca, e obtemos um equilíbrio entre as 3 operações.

Vejamos.

Suponha que cada sublista contém $\sqrt{n}$ elementos, de modo que existem no máximo $\sqrt{n}$ delas

Para buscar um elemento x nessa estrutura, nós basicamente realizamos a busca binária em cada sublista — no pior caso, é preciso examinar todas as $\sqrt{n}$ sublistas.

Esse procedimento leva tempo

$$T(n) \quad = \quad \sqrt{n} \cdot \Theta(\log n) \quad = \quad \Theta(\sqrt{n} \cdot \log n)$$

Para inserir um novo elemento na estrutura, basta inseri-lo na última sublista (se houver espaço, e na sua posição correta), ou iniciar uma nova sublista.

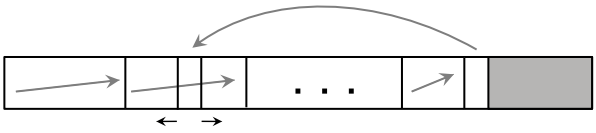
No pior caso, esse procedimento envolve deslocar todos os elementos de uma sublista, o que leva tempo

$$T(n) \quad = \quad \Theta(\sqrt{n})$$

A operação de remoção é realizada em duas etapas:

- primeiro é preciso fazer uma busca para localizar o elemento
- depois o elemento deve ser removido da estrutura

Para remover o elemento da estrutura, nós movemos o último elemento da última sublista para a sua posição, e depois o deslocamos para a sua posição correta nessa sublista



Essa operação leva tempo $\Theta(\sqrt{n})$ no pior caso — que é o número máximo de posições que o elemento é deslocado.

Portanto, o tempo total da operação de remoção é

$$T(n) \quad = \quad \Theta(\sqrt{n} \cdot \log n) \; + \; \Theta(\sqrt{n}) \quad = \quad \Theta(\sqrt{n} \cdot \log n)$$

Abaixo nós temos uma comparação do desempenho das 3 estruturas de dados

	Busca	**Inserção**	**Remoção**
Vetor desordenado	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
Vetor ordenado	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
Multilistas	$\Theta(\sqrt{n} \cdot \log n)$	$\Theta(\sqrt{n})$	$\Theta(\sqrt{n} \cdot \log n)$