Estruturas de Dados II

Notação Assintótica

Prof. Bruno Azevedo

Instituto Federal de São Paulo

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus Catanduva

Análise do Tempo de Execução de Algoritmos

- Se formos analisar o tempo de execução de um algoritmo, como fizemos em aulas prévias, podemos obter algo assim:
- "Em uma entrada de tamanho n, o algoritmo é executado em no máximo $1,9n^2 + 4,7n + 5$ passos".
- Essa é uma análise bem precisa do tempo de execução de um algoritmo.
- Entretanto, existem problemas com esse tipo de análise.

Análise do Tempo de Execução de Algoritmos

- Obter um limite tão preciso pode ser uma atividade exaustiva e fornecer mais detalhes do que precisamos de fato.
- E o mais importante, considerando uma entrada grande o suficiente, tais detalhes se tornam irrelevantes quando comparando o tempo de execução de algoritmos.
- Por exemplo, se temos um algoritmo que executa em $2n$ passos e outro que executa em $2n + 20$ passos, isso pode parecer relevante se temos uma entrada de $n = 4$.
- Mas, considerando o poder computacional de máquinas modernas, na prática ambos executarão de forma instantânea de qualquer forma com $n = 4$.
- E se tivermos uma entrada mais significativa, como $n = 10.000$, a constante aditiva na análise do segundo algoritmo se torna irrelevante no tempo final.
- Ou seja, tanta precisão acaba se tornando irrelevante na prática.

Análise do Tempo de Execução de Algoritmos

- Por todas essas razões, queremos expressar a taxa de crescimento dos tempos de execução de uma maneira que seja insensível a fatores constantes e termos de baixa ordem.
- Vamos aprender um modo preciso de fazer isso.

Notação-O

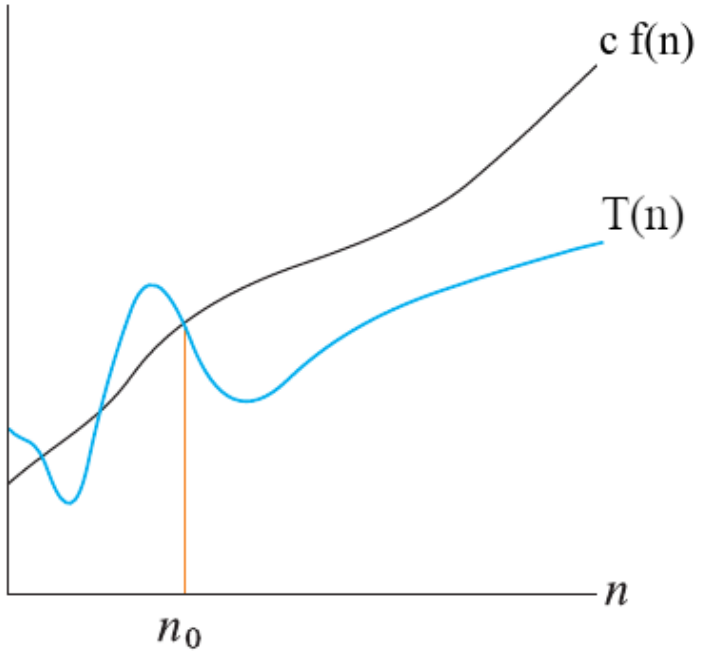
- Seja T(n) uma função. Por exemplo, o tempo de execução no pior caso de um certo algoritmo, dada uma entrada de tamanho n.
- Dada outra função f(n), dizemos que T(n) é O(f(n)) se, para valores suficientemente grandes de n, a função T(n) é limitada superiormente por um múltiplo constante de f(n).
- Também podemos escrever que T(n) = O(f(n)). Leia-se: "O de f de n" ou "ordem de f de n".
- Vamos definir O(f(n)) de modo mais preciso.

Notação-O

Definição de $O(f(n))$

$T(n)$ é $O(f(n))$ se existirem constantes $c > 0$ e $n_0 \geq 0$ de modo que, para todo $n \geq n_0$, tenhamos $T(n) \leq c \cdot f(n)$.

- n_0 é um valor mínimo de n a partir do qual a função $T(n)$ começa a satisfazer a condição $T(n) \leq c \cdot f(n)$.
- Em outras palavras, para que $T(n)$ seja $O(f(n))$, deve existir um ponto de corte n_0 a partir do qual a função $T(n)$ se mantém abaixo de $c \cdot f(n)$ para todos os valores de n maiores ou iguais a n_0.
- Por exemplo, se tivermos $T(n) = 2n^2$ e $f(n) = n^2$, podemos escolher $c = 2$ e $n_0 = 1$. Isso significa que a partir de $n = 1$, $T(n)$ sempre será menor ou igual a $2 \cdot n^2$, o que satisfaz a condição para $T(n)$ ser $O(n^2)$.
- A notação O, portanto, expressa um **limitante superior** na taxa de crescimento de uma função.



Notação-O

- Vamos entender isso aplicado a algoritmos.
- Estamos definindo $O(f(n))$, que é uma notação para expressar a taxa de crescimento dos tempos de execução de uma maneira que seja insensível a fatores constantes e termos de baixa ordem.
- A notação $T(n) = O(f(n))$ significa que o tempo de execução $T(n)$ de um algoritmo é limitado superiormente por uma função $f(n)$.
- Para satisfazer essa definição, devem existir constantes positivas c e n_0 tais que para todo $n \geq n_0$, $T(n)$ seja menor ou igual a $c \cdot f(n)$.
- Nesse caso, dizemos que T é limitada superiormente de forma assintótica por f.
- Vamos ver um exemplo de como essa definição nos permite expressar limites superiores nos tempos de execução de um algoritmo.

Notação-O

- $T(n) = pn^2 + qn + r$, onde p, q e r são constantes positivas.
- Podemos afirmar que este tempo de execução é $O(n^2)$?

Definição de $O(f(n))$

$T(n)$ é $O(f(n))$ se existirem constantes $c > 0$ e $n_0 \geq 0$ de modo que, para todo $n \geq n_0$, tenhamos $T(n) \leq c \cdot f(n)$.

- Vamos aumentar os termos até que nosso polinômio se torne uma constante vezes n^2.

Notação-O

- É direto que $T(n) = pn^2 + qn + r \leq pn^2 + qn^2 + rn^2$ para $n \geq 1$, dado que $qn \leq qn^2$ e $r \leq rn^2$.
- Ou seja, podemos dizer que $T(n) \leq pn^2 + qn^2 + rn^2$.
- Agora vamos isolar n^2, obtendo $T(n) \leq (p + q + r)n^2$.
- Se definimos $c = p + q + r$, obtemos $T(n) \leq c \cdot n^2$
- Essa desigualdade é exatamente o que a definição de $O(f(n))$, requer.
- Ou seja, é correto afirmar que $T(n)$ é limitado superiormente por n^2, com uma constante $c = p + q + r$ e $n \geq 1$.
- Sucintamente, $T(n)$ é $O(n^2)$.

Notação-O

- Observe que a notação-O expressa apenas um limite superior na taxa de crescimento de uma função e não a taxa exata de crescimento.
- Por exemplo, assim como afirmamos que a função $T(n) = pn^2 + qn + r$ é $O(n^2)$, também é correto dizer que ela é $O(n^3)$.
- Basta aplicar a definição, como fizemos anteriormente, para comprovar que isso também é verdade.

Notação-O

- Dizer que a função $T(n) = pn^2 + qn + r$ é $O(n^2)$ é mais preciso que dizer que é $O(n^3)$ ou $O(n^5)$, embora os três sejam limitantes superiores válidos.
- As vezes um algoritmo pode ser conhecido por ter um tempo de execução de $O(n^3)$, mas depois de uns anos podem reanalisar o mesmo algoritmo e descobrir que na verdade seu tempo de execução é de $O(n^2)$.
- O primeiro resultado era correto, apenas não era tão "justo" quanto poderia ser.

Notação-Ω

- Notação O provê um limitante superior para uma função.
- A notação Ω provê uma notação complementar à notação O.
- Por exemplo, se conseguimos provar que um algoritmo possui complexidade computacional de $O(n^2)$, podemos querer demonstrar que esse limitante superior é o **melhor possível**.

Notação-Ω

- Queremos expressar a ideia de que, para entradas de tamanho n arbitrariamente grandes, a função $T(n)$ é **pelo menos** um múltiplo constante de alguma função $f(n)$.
- Caso esteja confusos, lembrem que podemos ter inúmeras entradas de tamanho um tamanho n específico. Por exemplo, pensem em uma lista de 1000 números que desejamos ordenar.
- A notação Ω pode ser usada para verificar se o limitante superior dado por O é o mais justo possível. Vamos a definição:

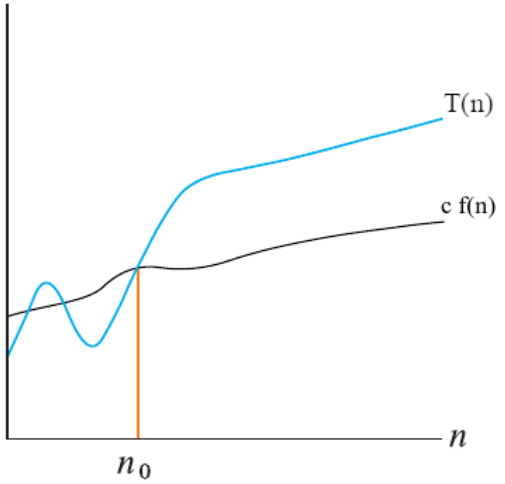
Definição de $\Omega(f(n))$

$T(n)$ é $\Omega(f(n))$ se existirem constantes $c > 0$ e $n_0 \geq 0$ de modo que, para todo $n \geq n_0$, tenhamos $\mathrm{T(n)} \geq c \cdot f(n)$.

Notação-Ω

Definição de $\Omega(f(n))$

$T(n)$ é $\Omega(f(n))$ se existirem constantes $c > 0$ e $n_0 \geq 0$ de modo que, para todo $n \geq n_0$, tenhamos $\mathrm{T(n)} \geq c \cdot f(n)$.



- Dizemos que T é **assintoticamente limitado inferiormente** por f.
- Exemplificando, se $T(n)$ é $\Omega(n^2)$, sabemos que **existe pelo menos uma instância que executa em n^2!**
- Podem existir outras instâncias de tamanho n que executam em menos passos.
- Temos, portanto, o tempo mínimo garantido em certos cenários.

Notação-Ω

Definição de $\Omega(f(n))$

$T(n)$ é $\Omega(f(n))$ se existirem constantes $c > 0$ e $n_0 \geq 0$ de modo que, para todo $n \geq n_0$, tenhamos T(n) $\geq c \cdot f(n)$.

- E como demonstrar que um algoritmo é $\Omega(f(n))$? Como devemos proceder?
- Por exemplo, se queremos demonstrar que um algoritmo é $\Omega(n^2)$.
- Basta encontrar uma entrada generica de tamanho n, para o qual o algoritmo levará pelo menos n^2 operações.
- O objetivo é mostrar que há pelo menos um cenário, de tamanho n onde o tempo de execução é n^2.
- Mas tendo tal análise, ainda precisamos encaixar esse $T(n)$ encontrado em nossa definição de $\Omega(f(n))$.

Notação-Ω

Definição de $\Omega(f(n))$

$T(n)$ é $\Omega(f(n))$ se existirem constantes $c > 0$ e $n_0 \geq 0$ de modo que, para todo $n \geq n_0$, tenhamos $\mathrm{T(n)} \geq c \cdot f(n)$.

- Para encaixarmos uma análise de tempo de execução na definição é simples.
- Precisamos encontrar valores de c e n_0 adequados.
- Por exemplo, vamos supor que encontramos uma entrada adequada e fizemos a sua análise de tempo.
- Descobrimos que $T(n) = 7n^2 + 3n + 200$.
- Agora vamos demonstrar que $7n^2 + 3n + 200$ é $\Omega(n^2)$.

Notação-Ω

Definição de $\Omega(f(n))$

$T(n)$ é $\Omega(f(n))$ se existirem constantes $c > 0$ e $n_0 \geq 0$ de modo que, para todo $n \geq n_0$, tenhamos $\mathrm{T(n)} \geq c \cdot f(n)$.

- Vamos demonstrar que $7n^2 + 3n + 200$ é $\Omega(n^2)$.
- Para este fim, precisamos demonstrar que $7n^2 + 3n + 200 \geq c \cdot n^2$.
- Vamos dividir os dois lados por n^2, resultando em $7 + 3/n + 200/n^2 \geq c$.
- Agora é simples: se n_0 é um inteiro positivo, temos que a desigualdade é válida com $c = 7$.

Notação-O e Notação-Ω

- Temos então uma **possível** conclusão direta considerando os resultados obtidos com $O(f(n))$ e $\Omega(f(n))$.
- Se conseguirmos mostrar que $T(n)$ é tanto $O(f(n))$ quanto $\Omega(f(n))$, então encontramos o limitante justo.
- $T(n)$ cresce exatamente como $f(n)$, até um fator constante, no pior caso.
- Há uma notação para expressar isso.

Notação-Θ

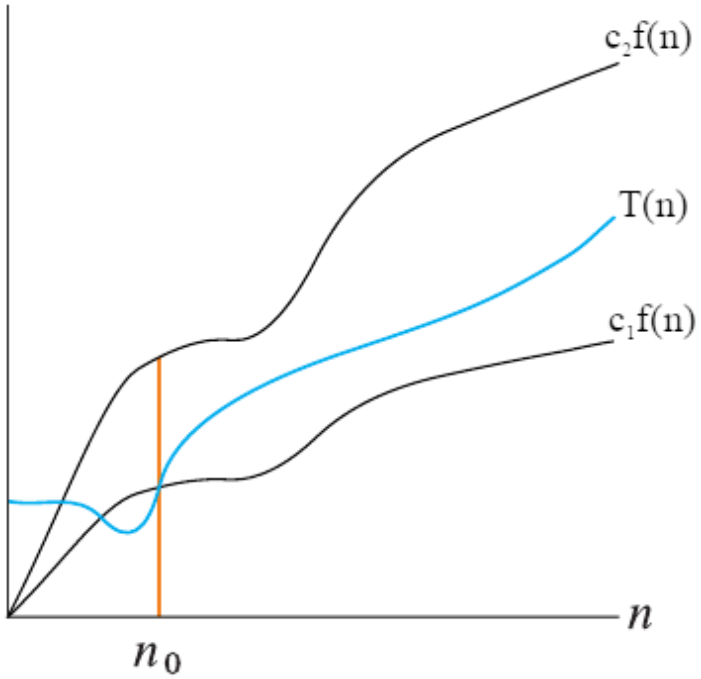
- Se uma função $T(n)$ é tanto $O(f(n))$ quanto $\Omega(f(n))$, dizemos que $T(n)$ é $\Theta(f(n))$.
- Nesse caso, dizemos que $f(n)$ **é um limitante assintoticamente justo para** $T(n)$.
- Limitantes assintoticamente justos sobre tempos de execução no pior caso são bons de se ter, pois **caracterizam o desempenho no pior caso de um algoritmo precisamente até fatores constantes**.

Notação-Θ

- Vamos conhecer a definição formal:

Definição de $\Theta(f(n))$

$T(n)$ é $\Theta(f(n))$ se existirem constantes $c_1 > 0$, $c_2 > 0$ e $n_0 \geq 0$ de modo que, para todo $n \geq n_0$, tenhamos $c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$.



- Mas essa definição de fato garante que se uma função $T(n)$ é tanto $O(f(n))$ quanto $\Omega(f(n))$, então $T(n)$ é $\Theta(f(n))$?
- Precisamos demonstrar isso com o rigor matemático necessário.

Notação-Θ

- Primeiro, vamos transformar a afirmação "Se uma função $T(n)$ é tanto $O(f(n))$ quanto $\Omega(f(n))$, dizemos que $T(n)$ é $\Theta(f(n))$" em um teorema.

Teorema

Para quaisquer duas funções $T(n)$ e $f(n)$, temos que $T(n) = \Theta(f(n))$ se e somente se $T(n) = O(f(n))$ e $T(n) = \Omega(f(n))$.

- Vamos provar esse teorema, é extremamente simples.

Notação-Θ

Teorema

Para quaisquer duas funções $T(n)$ e $f(n)$, temos que $T(n) = \Theta(f(n))$ se e somente se $T(n) = O(f(n))$ e $T(n) = \Omega(f(n))$.

- Dado que $T(n) = \Theta(f(n))$, temos que

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n) \text{ para } n \geq n_0.$$

- Dado que $T(n) = \Omega(f(n))$ e $T(n) = O(f(n))$, temos que

$$c_3 \cdot f(n) \leq T(n) \text{ para todo } n \geq n_1 \quad \text{e}$$

$$T(n) \leq c_4 \cdot f(n) \text{ para todo } n \geq n_2.$$

- Se definirmos $n_3 = \max(n_1, n_2)$ e combinarmos as desigualdades, obtemos

$$c_3 \cdot f(n) \leq T(n) \leq c_4 \cdot f(n) \text{ para todo } n \geq n_3.$$

- O que é a definição de Θ.

Notações Assintóticas

- Finalmente, para garantir o rigor de nossas definições, devemos fazer uma sucinta alteração nas definições.
- Lembrem que estamos definindo funções que nunca serão negativas, são análises de complexidade de tempo (e espaço).
- Portanto, para garantir a consistência matemática:

Definição de $O(f(n))$

$T(n)$ é $O(f(n))$ se existirem constantes $c > 0$ e $n_0 \geq 0$ de modo que, para todo $n \geq n_0$, tenhamos $0 \leq T(n) \leq c \cdot f(n)$.

Definição de $\Omega(f(n))$

$T(n)$ é $\Omega(f(n))$ se existirem constantes $c > 0$ e $n_0 \geq 0$ de modo que, para todo $n \geq n_0$, tenhamos $0 \leq c \cdot f(n) \leq T(n)$.

Definição de $\Theta(f(n))$

$T(n)$ é $\Theta(f(n))$ se existirem constantes $c_1 > 0$, $c_2 > 0$ e $n_0 \geq 0$ de modo que, para todo $n \geq n_0$, tenhamos $0 \leq c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$.

Notações o e ω

- Podemos utilizar as notações o e ω (leia-se pequeno-o e pequeno-ômega) para quando temos limitantes que sabemos não ser justos.
- O limitante $7n^2 = O(n^2)$ é assintoticamente justo, mas o limitante $3n = O(n^2)$ não é.
- Usamos a notação o para denotar um limitante superior que não é assintoticamente justo.
- Da mesma forma, usamos a notação ω para denotar um limitante inferior que não é assintoticamente justo.

Notações o e ω

- Definindo formalmente as notações o e ω.

Definição de $o(f(n))$

$T(n)$ é $O(f(n))$ se existirem constantes $c > 0$ e $n_0 \geq 0$ de modo que, para todo $n \geq n_0$, tenhamos $0 \leq T(n) < c \cdot f(n)$.

Definição de $\omega(f(n))$

$T(n)$ é $\Omega(f(n))$ se existirem constantes $c > 0$ e $n_0 \geq 0$ de modo que, para todo $n \geq n_0$, tenhamos $0 \leq c \cdot f(n) < T(n)$.

Tempo Constante

- Até o momento, aprenderam a descrever a taxa de crescimento dos tempos de execução de algoritmos dado o tamanho da entrada.
- Tempo constante é um conceito em análise de algoritmos que descreve operações que levam um tempo fixo para serem executadas, independentemente do tamanho da entrada.
- Por exemplo, acessar um elemento de um vetor de qualquer tamanho (1 ou 1.000.000) leva o mesmo tempo.
- Outro exemplo, atribuir um valor a uma variável (por exemplo, ‘x = 5‘) são consideradas como tempo constante.
- Independente da notação assintótica utilizada, tempo constante é denotado por ’1’.
- Por exemplo, se dizemos que um algoritmo executa em $O(1)$, significa que este algoritmo sempre executa em tempo constante, **independente** do tamanho da entrada.

Tempo Constante

- Veremos alguns exemplos de algoritmos de tempo constante que já conhecem.
- Dado que temos uma pilha de tamanho n, no pior caso, quanto tempo leva a operação de empilhar, exibida abaixo?

```
EMPILHAR(P, TopoPilha, topoMax, elemento)
    if TopoPilha = topoMax then
        "Erro de Overflow"
    else
        TopoPilha ← TopoPilha + 1
        P[TopoPilha] ← elemento
```

- E a operação de enfileirar, exibida abaixo?

```
ENFILEIRAR(F, inicio, fim, comp, elemento)
    F[fim] ← elemento
    if fim = comp - 1 then
        fim = 0
    else
        fim = fim + 1
```

- Ou seja, existem algoritmos que executam em **tempo constante**.
- Ambos algoritmos acima executam em $O(1)$.