SIN110 Algoritmos e Grafos

aula 02

Análise de Algoritmos

- correção e complexidade
- comportamento assintótico
- algoritmos iterativos

Análise de Algoritmos

O que se analisa ...

- Como estimar a quantidade de **recursos** (tempo, memória) que um algoritmo consome/gasta = análise de complexidade
- Como provar a "corretude" de um algoritmo
- Como projetar algoritmos eficientes (= rápidos) para vários problemas computacionais

Análise de Algoritmos

- Não existe um conjunto completo de regras para analisar algoritmos.
- Aho, Hopcroft e Ullman, em *Data Structure and Algorithms – 1983,* enumeram alguns princípios a serem seguidos:

1) O tempo de execução de um comando de atribuição, leitura ou escrita pode ser considerado constante. Há exceções para as linguagens que permitem a chamada de funções em comandos de atribuição, ou atribuições envolvendo arranjos muito grandes.

Análise de Algoritmos

2) O tempo para executar uma estrutura de repetição (um looping) é a soma do tempo de execução do corpo do comando mais o tempo de avaliar a condição para conclusão, em geral constante, multiplicado pelo número de iterações da repetição.

3) O tempo de execução de um comando de decisão é composto pelo tempo de execução dos comandos executados dentro do comando condicional, mais o tempo de avaliar a condição, que é uma constante.

4) O tempo para executar uma estrutura de repetição (um looping) é a soma do tempo de execução do corpo do comando mais o tempo de avaliar a condição para conclusão, em geral constante, multiplicado pelo número de iterações da repetição.

Análise de Algoritmos

5) Se o programa possui vários procedimentos não recursivos, o tempo de execução de cada procedimento deve ser computado separadamente um a um, iniciando com os procedimentos que não chamam outros procedimentos. Em seguida, devem ser avaliados os procedimentos que chamam os procedimentos que não chamavam outros procedimentos, agregando os tempos já computados dos procedimentos. Este processo é repetido até chegar ao programa principal.

6) Quando o programa possui procedimentos recursivos, para cada procedimento é associada uma função de complexidade $f(n)$ desconhecida, onde n mede o tamanho dos argumentos para o procedimento.

Complexidade de algoritmos: um exemplo

Problema: ordenar um vetor em ordem crescente

Entrada: um vetor $A[1 \dots n]$

Saída: vetor $A[1 \dots n]$ rearranjado em ordem crescente

Vamos começar analisando o algoritmo de ordenação baseado no método de inserção (Insertion sort).
Isto nos permitirá destacar alguns dos aspectos mais importantes no estudo de algoritmos para esta disciplina.

Complexidade de algoritmos: um exemplo

Inserção em um vetor ordenado

1						j				n
20	25	35	40	44	55	38	99	10	65	50

- O subvetor $A[1 \ldots j-1]$ está ordenado.
- Queremos inserir a $chave = 38 = A[j]$ em $A[1 \ldots j-1]$ de modo que no final tenhamos:

1						j				n
20	25	35	38	40	44	55	99	10	65	50

- Agora $A[1 \ldots j]$ está ordenado.

Complexidade de algoritmos: um exemplo

Pseudocódigo

```
ORDENA-POR-INSERÇÃO(A, n)
  para j ← 2 até n faça
      chave ← A[j]
      ▷ Insere A[j] no subvetor ordenado A[1..j − 1]
      i ← j − 1
      enquanto i ≥ 1  e A[i] > chave faça
          A[i + 1] ← A[i]
          i ← i − 1
      A[i + 1] ← chave
```

Complexidade de algoritmos: análise do exemplo

O algoritmo pára

ORDENA-POR-INSERÇÃO(A, n)

1 **para** $j \leftarrow 2$ **até** n **faça**

  . . .

4   $i \leftarrow j - 1$

5   **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**

6   . . .

7     $i \leftarrow i - 1$

8   . . .

No laço **enquanto** na linha 5 o valor de i diminui a cada iteração e o valor inicial é $i = j - 1 \geq 1$. Logo, a sua execução pára em algum momento por causa do teste condicional $i \geq 1$.

O laço na linha 1 evidentemente pára (o contador j atingirá o valor $n + 1$ após $n - 1$ iterações).

Portanto, o algoritmo **pára**.

Complexidade de algoritmos: análise do exemplo

Ordena – por - inserção

```
ORDENA-POR-INSERÇÃO(A, n)
para j ← 2 até n faça
    chave ← A[j]
    ▷ Insere A[j] no subvetor ordenado A[1..j − 1]
    i ← j − 1
    enquanto i ≥ 1 e A[i] > chave faça
        A[i + 1] ← A[i]
        i ← i − 1
    A[i + 1] ← chave
```

O que falta fazer?

- Verificar se ele produz uma resposta correta.
- Analisar sua complexidade de tempo.

Complexidade de algoritmos: análise do exemplo

Invariante de laço e provas de corretude

- Definição: um invariante de um laço é uma propriedade que relaciona as variáveis do algoritmo a cada execução completa do laço.
- Ele deve ser escolhido de modo que, ao término do laço, tenha-se uma propriedade útil para mostrar a corretude do algoritmo.
- A prova de corretude de um algoritmo requer que sejam encontrados e provados invariantes dos vários laços que o compõem.
- Em geral, é mais difícil descobrir um invariante apropriado do que mostrar sua validade se ele for dado de bandeja. . .

Complexidade de algoritmos: análise do exemplo

Exemplo de invariante

```
ORDENA-POR-INSERÇÃO(A, n)
1  para j ← 2 até n faça
2      chave ← A[j]
3      ▷ Insere A[j] no subvetor ordenado A[1..j − 1]
4      i ← j − 1
5      enquanto i ≥ 1  e A[i] > chave faça
6          A[i + 1] ← A[i]
7          i ← i − 1
8      A[i + 1] ← chave
```

No começo de cada iteração do laço **para** das linha 1–8, o subvetor $A[1 \ldots j-1]$ está ordenado.

Complexidade de algoritmos: análise do exemplo

Corretude de algoritmos por invariante

A estratégia "típica" para mostrar a corretude de um algoritmo iterativo através de invariantes segue os seguintes passos:

1. Mostre que o invariante vale no início da primeira iteração (trivial, em geral)
2. Suponha que o invariante vale no início de uma iteração qualquer e prove que ele vale no ínicio da próxima iteração
3. Conclua que se o algoritmo pára e o invariante vale no ínicio da última iteração, então o algoritmo é correto.

Note que (1) e (2) implicam que o invariante vale no início de qualquer iteração do algoritmo. Isto é similar ao método de indução matemática ou indução finita!

Complexidade de algoritmos: análise do exemplo

Corretude da Ordenação-por-inserção

Vamos verificar a corretude do algoritmo de ordenação por inserção usando a técnica de **prova por invariantes de laços**.

No começo de cada iteração do laço **para** das linha 1–8, o subvetor $A[1 \dots j-1]$ está ordenado.

1						j				n
20	25	35	40	44	55	38	99	10	65	50

- Suponha que o invariante vale.
- Então a corretude do algoritmo é “evidente”. **Por quê?**
- No ínicio da última iteração temos $j = n + 1$. Assim, do invariante segue que o (sub)vetor $A[1 \dots n]$ está ordenado!

Complexidade de algoritmos: análise do exemplo

Análise do algoritmo

O que é importante analisar/considerar?

- Corretude do algoritmo

- Complexidade de tempo do algoritmo: quantas intruções são necessárias no pior caso para ordenar os *n* elementos?

Complexidade de algoritmos: análise do exemplo

Vamos contar ...

ORDENA-POR-INSERÇÃO(A, n)	Custo	Vezes
1 **para** $j \leftarrow 2$ **até** n **faça**	c_1	n
2 $\quad chave \leftarrow A[j]$	c_2	$n-1$
3 $\quad \triangleright$ Insere $A[j]$ em $A[1\dots j-1]$	0	$n-1$
4 $\quad i \leftarrow j-1$	c_4	$n-1$
5 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**	c_5	$\sum_{j=2}^{n} t_j$
6 $\qquad A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $\qquad i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $\quad A[i+1] \leftarrow chave$	c_8	$n-1$

O valor c_k representa o custo (tempo) de cada execução da linha k.

Denote por t_j o número de vezes que o teste no laço **enquanto** na linha 5 é feito para aquele valor de j.

Complexidade de algoritmos: análise do exemplo

Tempo de execução total

Logo, o tempo total de execução $T(n)$ de Ordena-Por-Inserção é a soma dos tempos de execução de cada uma das linhas do algoritmo, ou seja:

$$
\begin{aligned}
T(n) \quad = \quad & c_1 n + c_2(n-1) + c_4(n-1) + c_5 \textstyle\sum_{j=2}^{n} t_j \\
& + c_6 \textstyle\sum_{j=2}^{n}(t_j - 1) + c_7 \textstyle\sum_{j=2}^{n}(t_j - 1) \\
& + c_8(n-1)
\end{aligned}
$$

Como se vê, entradas de tamanho igual (i.e., mesmo valor de n), podem apresentar tempos de execução diferentes já que o valor de $T(n)$ depende dos valores dos t_j.

Complexidade de algoritmos: análise do exemplo

O pior caso ...

Quando o vetor A está em ordem decrescente, ocorre o pior caso para Ordena-Por-Inserção. Para inserir a *chave* em $A[1 \ldots j-1]$, temos que compará-la com todos os elementos neste subvetor. Assim, $t_j = j$ para $j = 2, \ldots, n$.

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\
&\quad - (c_2 + c_4 + c_5 + c_8)
\end{aligned}
$$

O tempo de execução no pior caso é da forma $an^2 + bn + c$ onde a, b, c são constantes que dependem apenas dos c_i.

Portanto, **no pior caso**, o tempo de execução é uma **função quadrática** no **tamanho da entrada**.

Complexidade de algoritmos: análise do exemplo

... e o melhor caso

O melhor caso de Ordena-Por-Inserção ocorre quando o vetor A já está ordenado. Para $j = 2, \ldots, n$ temos $A[i] \leq chave$ na linha 5 quando $i = j - 1$. Assim, $t_j = 1$ para $j = 2, \ldots, n$.

Logo,

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

Este tempo de execução é da forma $an + b$ para constantes a e b que dependem apenas dos c_i.

Portanto, **no melhor caso**, o tempo de execução é uma **função linear** no **tamanho da entrada**.

Análise de algoritmos: outro exemplo

Algoritmo Ordena – por - Seleção

```
Ordena – por - Seleção(A,n)
para i ← 1 até n-1 faça
        min ← i
        para j ← i + 1 até n faça
                se A[j] <  A[min]
                        então min ← j
        aux  ← A[min]
        A[min]  ←  A[i]
        A[i]  ← aux
```

Análise de algoritmos: outro exemplo

```
Ordena – por - Seleção(A,n)
1.  para i ← 1 até n-1 faça
2.          mini ← i
3.          para j ← i + 1 até n faça
4.                  se A[j] <  A[min]
5.                       então min ← j
6.          aux  ← A[min]
7.          A[min]  ←  A[i]
8.          A[i]  ← aux
```

1. Correção

- O algoritmo pára: observando o laço principal (linhas 1-8) e o interno (lihas 3-5) verificamos o controle dos contadores i e j que garantem m número finito de execuções.
- O algoritmo ordena corretamente o vetor A[1..n] em ordem crescente ao finalizar a execução do algoritmo.
- Em cada iteração, após a execução das linhas 2-8, o subvetor A[1..i] está ordenado.

Vide simulação a seguir.

Análise de algoritmos: outro exemplo

Ordena – por - Seleção(A,n) : simulação

Em cada iteração o item de menor valor é posicionado (destacado em **negrito**)

15	25	57	13	**9**	18	48	37	12	92	86	33
9	25	57	13	15	18	48	37	12	92	86	33
9	**12**	57	**13**	15	18	48	37	25	92	86	33
9	**12**	**13**	57	**15**	18	48	37	25	92	86	33
9	**12**	**13**	**15**	57	**18**	48	37	25	92	86	33
9	**12**	**13**	**15**	**18**	57	48	37	**25**	92	86	33
9	**12**	**13**	**15**	**18**	**25**	48	37	57	92	86	**33**
9	**12**	**13**	**15**	**18**	**25**	**33**	**37**	57	92	86	48
9	**12**	**13**	**15**	**18**	**25**	**33**	**37**	57	92	86	**48**
9	**12**	**13**	**15**	**18**	**25**	**33**	**37**	**48**	92	86	**57**
9	**12**	**13**	**15**	**18**	**25**	**33**	**37**	**48**	**57**	**86**	92
9	**12**	**13**	**15**	**18**	**25**	**33**	**37**	**48**	**57**	**86**	**92**
9	**12**	**13**	**15**	**18**	**25**	**33**	**37**	**48**	**57**	**86**	**92**

Análise de algoritmos: outro exemplo

2. Complexidade

Ordena-por-Selação(A,n)	***nº execuções***
1. para i ← 1 até n-1 faça	n
2. min ← i	$n-1$
3. para j ← i + 1 até n faça	$n + (n-1)+\ldots+2 = (n^2+n-2)/2$
4. se A[j] < A[min]	$(n-1)+(n-2)+\ldots+1 = (n^2-n)/2$
5. então min ← j	$(n-1)+(n-2)+\ldots+1 = (n^2-n)/2$
6. aux ← A[min]	$n-1$
7. A[min] ← A[i]	$n-1$
8. A[i] ← aux	$n-1$

total: $\boldsymbol{T(n) = (3n^2 + 9n - 10)/2}$

→ *Também uma função quadrática em n*

Medida de complexidade de algoritmos

- A complexidade de tempo (= eficiência) de um algoritmo é o número de instruções básicas que ele executa em função do tamanho da entrada.
- Adotamos uma "atitude pessimista" e em geral fazemos uma análise de pior caso.
  Determinamos o tempo máximo necessário para resolver uma instância de um certo tamanho.
- Além disso, a análise concentra-se no comportamento do algoritmo para entradas de tamanho GRANDE = análise assintótica.

Medida de complexidade de algoritmos

- Um algoritmo é chamado eficiente se a função que mede sua complexidade de tempo é limitada por um **polinômio** no tamanho da entrada.
  Por exemplo: n, $3n - 7$, $n \log n$, $4n^2$, $143n^2 - 4n + 2$, n^5.
- Mas por que polinômios?
  Resposta padrão: (polinômios são funções bem "comportadas").

Medida de complexidade de algoritmos: exemplo

Quanto tempo consome o algoritmo abaixo que opera sobre um vetor A[1..n]?

```
Alg2(A,n)
1.  S ← 0
2.  para i ← 1 até n faça
3.        S ← S + A[i]
4.  m ← S/n
5.  k ← 1
6.  para i ← 2 até n faça
7.        se( (A[i] – m)² < (A[k] – m)²)
8.             k ← i
9. devolve k
```

Medida de complexidade de algoritmos: exemplo

Quanto tempo consome o algoritmo abaixo que opera sobre um vetor A[1..n]?

Alg2(A,n)	nº de execuções
S ← 0	1
para i ← 1 até n faça	n+1
S ← S + A[i]	n
m ← S/n	1
k ← 1	1
para i ← 2 até n faça	n
se((A[i] – m)^2 < (A[k] – m)^2)	n-1
k ← i	n-1
devolve k	1
total:	$T(n) = 5n + 3$

SIN110 Algoritmos e Grafos

Comportamento Assintótico

Comportamento assintótico

*Na análise de algoritmos, considera-se a análise de pior caso e o comportamento assintótico de um algoritmo. (**instâncias de tamanho grande**).*

O algoritmo Ordena-por-Inserçao tem como complexidade (pior caso) uma função quadrática $An^2 + Bn + C$, onde A, B e C são constantes absolutas que dependem dos custos (associados ao número de execuções de cada linha). O algoritmo do exemplo anterior, tem complexidade dada por uma função linear $Dn + E$ (determinados $5n + 3$).

O estudo do comportamento assintótico permite-nos ignorar os valores dessas constantes, isto é, aquilo que independe do tamanho da entrada (valores de A, B e C ou, D e E citados).

Comportamento assintótico

Análise assintótica de funções quadráticas

Considere a função quadrática $3n^2 + 10n + 50$:

n	$3n^2 + 10n + 50$	$3n^2$
64	12978	12288
128	50482	49152
512	791602	786432
1024	3156018	3145728
2048	12603442	12582912
4096	50372658	50331648
8192	201408562	201326592
16384	805470258	805306368
32768	3221553202	3221225472

Como se vê, $3n^2$ é o termo dominante quando n é grande.

De um modo geral, podemos nos concentrar nos termos dominantes e esquecer os demais.

Notação assintótica

Usando a notação assintótica, dizemos que o algoritmo Ordena-por-Inserção tem complexidade de tempo de pior caso $O(n^2)$.

Isto tem dois significados:

- A complexidade de tempo é limitada (superiormente) assintoticamente por algum polinômio da forma An^2 para alguma constate A.

- Para todo n suficientemente grande, existe alguma instância de tamanho n que consome tempo pelo menos Dn^2, para alguma constante positiva D,

Comparando funções

Vamos comparar funções assintoticamente, ou seja, para valores grandes, desprezando constantes multiplicativas e termos de menor ordem

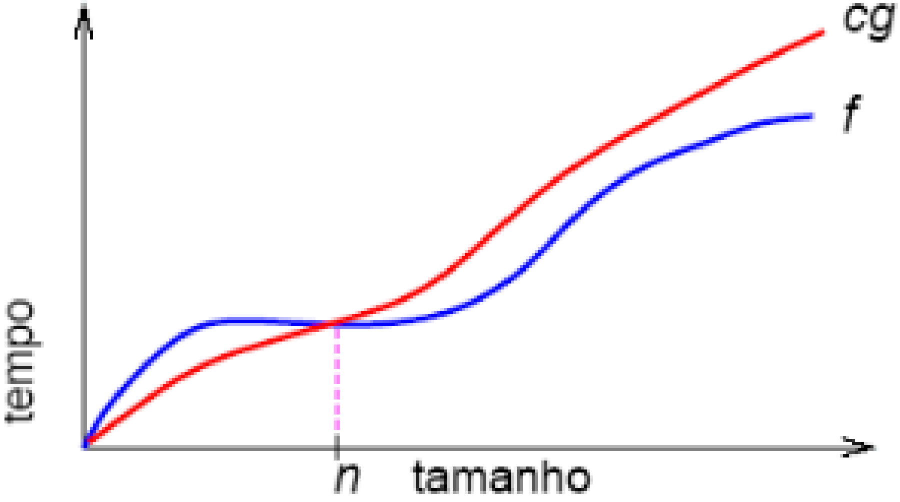
	$n = 100$	$n = 1000$	$n = 10^4$	$n = 10^6$	$n = 10^9$
$\log n$	2	3	4	6	9
n	100	1000	10^4	10^6	10^9
$n \log n$	200	3000	$4 \cdot 10^4$	$6 \cdot 10^6$	$9 \cdot 10^9$
n^2	10^4	10^6	10^8	10^{12}	10^{18}
$100n^2 + 15n$	$1,0015 \cdot 10^6$	$1,00015 \cdot 10^8$	$\approx 10^{10}$	$\approx 10^{14}$	$\approx 10^{20}$
2^n	$\approx 1,26 \cdot 10^{30}$	$\approx 1,07 \cdot 10^{301}$	?	?	?

Classe O

Definição:

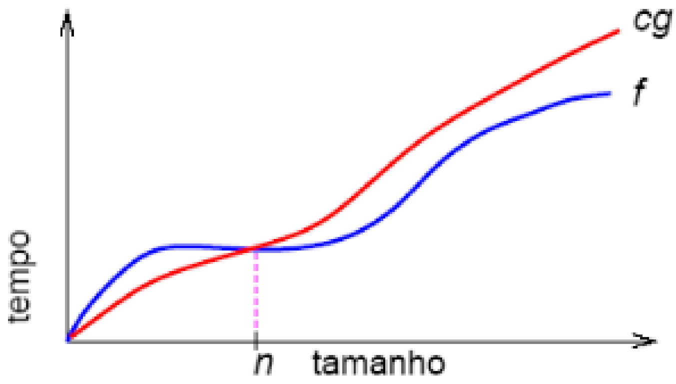
$O(g(n))$ = {$f(n)$: existem constantes positivas c e n_0 tais que $0 \leq f(n) \leq cg(n)$, para todo $n \geq n_0$}.

Informalmente diz-se que, $\boldsymbol{f(n) \in O(g(n))}$, então f(n) cresce no máximo tão rapidamente quanto *g(n).*



Classe O

$O(g(n)) = \{f(n)$: existem constantes positivas c e n_0 tais que $0 \le f(n) \le cg(n)$, para todo $n \ge n_0\}$



Exemplo:

$\frac{1}{2}n^2 - 3n \in O(n^2)$

Valores de c e n_0 que satisfazem a definição são

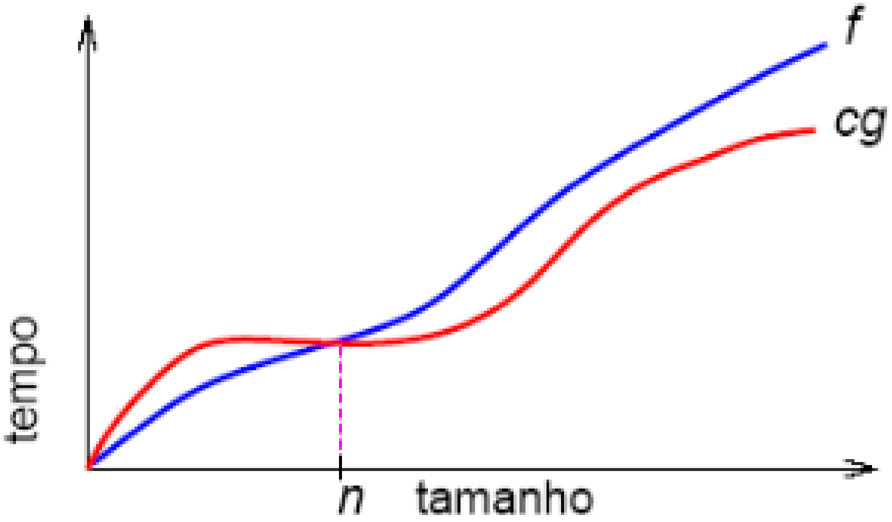
$$c = \frac{1}{2} \text{ e } n_0 = 7.$$

Classe Ω

Definição:

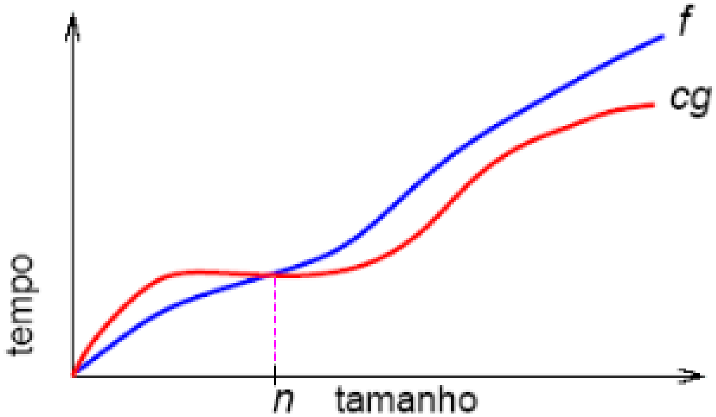
$\Omega(g(n))$ = {f(n): existem constantes positivas c e n_0 tais que $0 \leq cg(n) \leq f(n)$, para todo $n \geq n_0$}.

Informalmente diz-se que, $f(n) \in \Omega(g(n))$, então f(n) cresce no mínimo tão lentamente quanto *g(n).*



Classe Ω

$\Omega(g(n)) = \{f(n)$: existem constantes positivas c e n_0 tais que $0 \le cg(n) \le f(n)$, para todo $n \ge n_0\}$.



Exemplo:

$\frac{1}{2}n^2 - 3n \in \Omega(n^2)$

Valores de c e n_0 que satisfazem a definição são

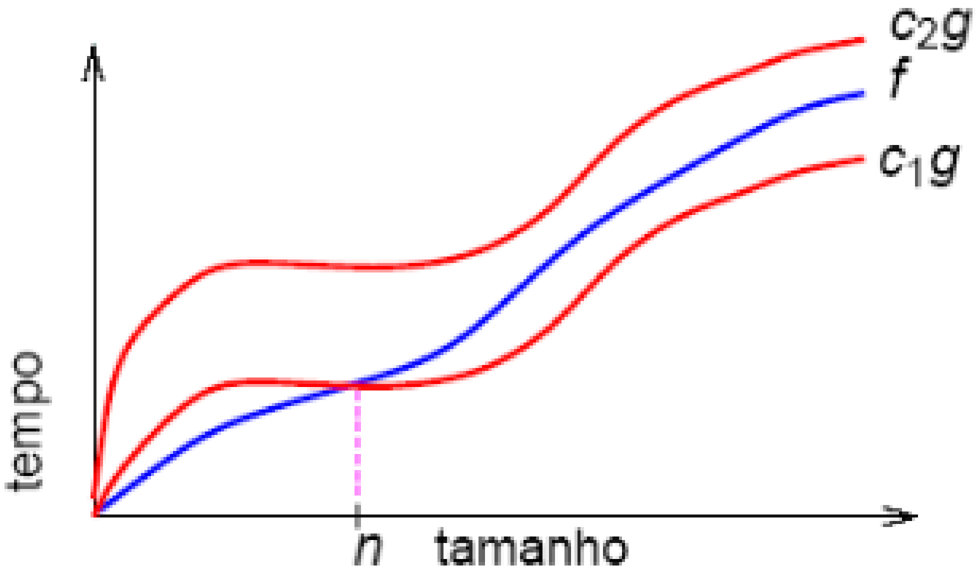
$$c = \frac{1}{14} \text{ e } n_0 = 7.$$

Classe Θ

Definição:

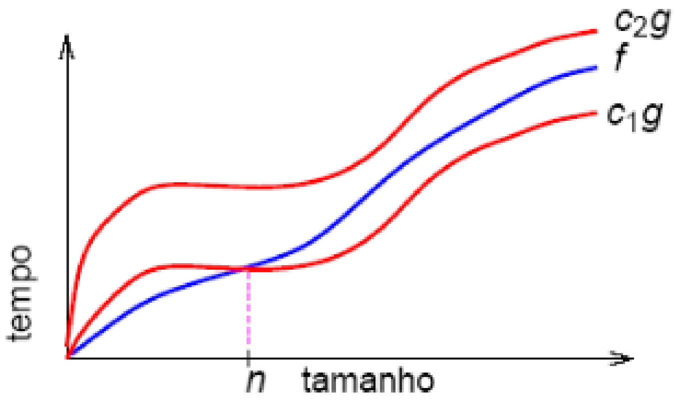
$\Theta(g(n)) = \{f(n)$: existem constantes positivas c_1, c_2 e n_0 tais que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$, para todo $n \geq n_0\}$.

Informalmente diz-se que, ***$f(n) \in \Theta(g(n))$***, então f(n) cresce tão rapidamente quanto *g(n).*



Classe Θ

$\Theta(g(n)) = \{f(n)$: existem constantes positivas c_1, c_2 e n_0 tais que $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$, para todo $n \ge n_0\}$.



Exemplo:

$\frac{1}{2}n^2 - 3n \in \Theta(n^2)$

Valores de c_1, c_2 e n_0 que satisfazem a definição são

$$c_1 = \frac{1}{14},\ c_2 = \frac{1}{2} \text{ e } n_0 = 7.$$

Alguns nomes de Classes ... Θ

$\Theta(1)$	constante
$\Theta(\log n)$	logarítmica
$\Theta(n)$	linear
$\Theta(n \log n)$	$n \log n$
$\Theta(n^2)$	quadrática
$\Theta(n^3)$	cúbica
$\Theta(2^n)$	exponencial

Algoritmos Iterativos

Exemplos de aplicação

Classificação por Inserção:

	pior caso	
Inserção (A, n)	**contagem**	**consumo**
para j ← 2 **até** n **faça**	n	$O(n)$
x ← A[j]	(n-1)	$O(n)$
i ← j - 1	(n-1)	$O(n)$
enquanto i>0 **e** A[i]>x **faça**	n+(n-1)+...+2	$nO(n) = O(n^2)$
A[i+1] ← A[i]	(n-1)+(n-2)+...+1	$nO(n) = O(n^2)$
i ← i - 1	(n-1)+(n-2)+...+1	$nO(n) = O(n^2)$
A[i+1] ← x	(n-1)	$O(n)$
	$(3n^2+7n-8)/2$	**$O(3n^2+4n) = O(n^2)$**

Exemplos de aplicação

Outra análise:

	melhor caso	
Inserção (A, n)	**contagem**	**consumo**
para j ← 2 **até** n **faça**	n	$\Omega(n)$
x ← A[j]	(n-1)	$\Omega(n)$
i ← j - 1	(n-1)	$\Omega(n)$
enquanto i>0 e A[i]>x **faça**	(n-1)	$\Omega(n)$
A[i+1] ← A[i]	0	0
i ← i - 1	0	0
A[i+1] ← x	(n-1)	$\Omega(n)$
	5n-4	$\boldsymbol{\Omega(5n) = \Omega(n)}$

Exemplo de Análise

Intercalação de dois vetores:

Supondo A[e..m] e A[m+1..d] em ordem crescente; queremos colocar A[e..d] em ordem crescente:

e					m	m+1			d
11	33	33	55	55	77	22	44	66	88

Intercala (A, e, m, d)	contagem	consumo
crie vetor B[e..d]	n=d-e+1	$\Theta(n)$
para i ← e **até** m **faça**	2n+2	$\Theta(n)$
B[i] ← A[i]		
para j ← m+1 **até** d **faça**		
B[d+m+1-j] ← A[j]		
i ← e	2	$\Theta(1)$
j ← d		
para k ← e **até** d **faça**	n	$n\Theta(1) = \Theta(n)$
se B[i] ≤ B[i]	n	$n\Theta(1) = \Theta(n)$
então A[k] ← B[i]	2n	$n\Theta(1) = \Theta(n)$
i ← i+1		
senão A[k] ← B[j]		
j ← j-1		
Total:	**7n+4**	$\boldsymbol{\Theta(5n+1) = \Theta(n)}$

+ Exemplo de Análise

Problema da Seqüência de soma máxima

Dada uma seqüência de números inteiros a_1, a_2,..., An determine a subseqüência a_i,..., a_j , com $0 \leq i \leq j \leq n$, que tem o valor máximo para a soma $\sum_{k=i}^{j} a_k$.

Como exemplo considere a seqüência:

-2, 11, -4, 13, -5, -2.

A solução para essa entrada tem valor 20 com soma no segmento: a_2, ..., a_4..

Se obtivermos um valor negativo na resposta, consideramos o valor nulo.

+ Exemplo de Análise

Esse problema é interessante por apresentar soluções que apresentam variações drásticas no tempo de processamento como ilustra a tabela com tempos anotados em segundos para os algoritmos, que apresentaremos a seguir, rodando em uma máquina real:

algoritmo	Somax1	Somax2	Somax3	Somax4
tempo	$O(n^3)$	$O(n^2)$	$O(n \lg n)$	$O(n)$
n = 10	0.00103	0.00045	0.00066	0.00034
n = 100	0.47015	0.01112	0.00486	0.00063
n = 1000	448.77	1.1233	0.05843	0.00333
n = 10000	427913.5	111.13	0.68631	0.03042
n = 100000	4×10^9	10994.2	8.0113	0.29832

+ Exemplo de Análise

```
Somax1(A, n)
max ← 0
para i ← 1 até n faça
    para j ← i até n faça
        aux ← 0
        para k ← i até j faça
            aux ← aux + A[k]
            se aux > max
               então max ← aux;
devolve max
```

+ Exemplo de Análise

```
Somax2(A, n)
max ← 0
para i ← 1 até n faça
    aux ← 0
    para j ← i até n faça
        aux ← aux + A[j]
        se aux > max
            então max ← aux;
devolve max
```

+ Exemplo de Análise

```
Somax4(A, n)
max ← aux ← 0
para i ← 1 até n faça
    aux  ←  aux + A[i]
    se aux > max
       então max ← aux
       senão se aux < 0
                   então aux ← 0
devolve max
```

Exercícios

1) Problema: dado um inteiro x e um vetor de inteiros distintos A[e..d], em ordem crescente, encontre j tal que A[j] ≤ x < A[j+1]. Escreva e analise (correção e o consumo de tempo) de um algoritmo iterativo de uma busca: (i) "linear" , (ii) "binária".

2) Seja M uma matriz n × m de números reais tal que:

(a) cada linha de M está ordenada em ordem crescente (da esquerda para a direita) e,

(b) cada coluna de M está ordenada em ordem crescente (de cima para baixo).

Projete um algoritmo (baseado em comparações) que recebe M e um inteiro x e determina se x aparece em M (ou seja, se existem índices i, j tais que M[i, j] = x).

Analise a correção e complexidade de sua solução.