Invariantes 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...

Exemplo de invariante

ORDENA-POR-INSERÇÃO(A, n)
1 **para** $j \leftarrow 2$ **até** n **faça**
2 $\quad$ $chave \leftarrow A[j]$
3 $\quad$ ▷ Insere $A[j]$ no subvetor ordenado $A[1..j-1]$
4 $\quad$ $i \leftarrow j-1$
5 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**
6 $\quad\quad$ $A[i+1] \leftarrow A[i]$
7 $\quad\quad$ $i \leftarrow i-1$
8 $\quad$ $A[i+1] \leftarrow chave$

Invariante principal de ORDENA-POR-INSERÇÃO: (i1)

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

Corretude de algoritmos por invariantes

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!

Corretude da ordenação por inserção

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

Invariante principal: (i1)

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

Melhorando a argumentação

ORDENA-POR-INSERÇÃO(A, n)
1 **para** $j \leftarrow 2$ **até** n **faça**
2 $\quad$ $chave \leftarrow A[j]$
3 $\quad$ ▷ Insere $A[j]$ no subvetor ordenado $A[1\ldots j-1]$
4 $\quad$ $i \leftarrow j-1$
5 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**
6 $\quad\quad$ $A[i+1] \leftarrow A[i]$
7 $\quad\quad$ $i \leftarrow i-1$
8 $\quad$ $A[i+1] \leftarrow chave$

Um invariante mais preciso: (i1’)

No começo de cada iteração do laço **para** das linhas 1–8, o subvetor $A[1\ldots j-1]$ é uma permutação ordenada do subvetor original $A[1\ldots j-1]$.

Esboço da demonstração de (i1’)

1. Validade na primeira iteração: neste caso, temos $j = 2$ e o invariante simplesmente afirma que $A[1\ldots 1]$ está ordenado, o que é evidente.
2. Validade de uma iteração para a seguinte: segue da discussão anterior. O algoritmo empurra os elementos maiores que a chave para seus lugares corretos e ela é colocada no espaço vazio.

   Uma demonstração mais formal deste fato exige invariantes auxiliares para o laço interno **enquanto**.
3. Corretude do algoritmo: na última iteração, temos $j = n+1$ e logo $A[1\ldots n]$ está ordenado com os elementos originais do vetor. Portanto, o algoritmo é correto.

Invariantes auxiliares

No início da linha 5 valem os seguintes invariantes:

(i2) $A[1\ldots i]$ e $A[i+2\ldots j]$ contém os elementos de $A[1\ldots j]$ antes de entrar no laço que começa na linha 5.

(i3) $A[1\ldots i]$ e $A[i+2\ldots j]$ são crescentes.

(i4) $A[1\ldots i] \leq A[i+2\ldots j]$

(i5) $A[i+2\ldots j] > \mathit{chave}$.

$$\left.\begin{array}{l}\text{Invariantes (i2) a (i5)}\\ +\text{condição de parada na linha 5}\\ +\text{atribuição da linha 7}\end{array}\right\} \Longrightarrow \text{invariante (i1')}$$

Demonstração? Mesma que antes.

Complexidade do algoritmo

- Vamos tentar determinar o tempo de execução (ou complexidade de tempo) de ORDENA-POR-INSERÇÃO em função do tamanho de entrada.
- Para o problema de Ordenação vamos usar como tamanho de entrada a dimensão do vetor e ignorar o valores dos seus elementos (modelo RAM).
- A complexidade de tempo de um algoritmo é o número de instruções básicas (operações elementares ou primitivas) que executa a partir de uma entrada.
- Exemplo: comparação e atribuição entre números ou variáveis numéricas, operações aritméticas, etc.

Vamos contar ?

ORDENA-POR-INSERÇÃO(A, n)	Custo	# execuções
1 **para** $j \leftarrow 2$ **até** n **faça**	c_1	?
2 $\mathit{chave} \leftarrow A[j]$	c_2	?
3 $\triangleright$ Insere $A[j]$ em $A[1\ldots j-1]$	0	?
4 $i \leftarrow j-1$	c_4	?
5 **enquanto** $i \geq 1$ **e** $A[i] > \mathit{chave}$ **faça**	c_5	?
6 $A[i+1] \leftarrow A[i]$	c_6	?
7 $i \leftarrow i-1$	c_7	?
8 $A[i+1] \leftarrow \mathit{chave}$	c_8	?

A constante 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.

Vamos contar ?

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

A constante 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.

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:

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

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.

Melhor caso

O melhor caso de Ordena-Por-Inserção ocorre quando o vetor A já está ordenado. Para $j = 2, \ldots, n$ temos $A[i] \leq \mathit{chave}$ 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**.

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 $j = 2, \ldots, n$.

Lembre-se que:

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

e

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}.$$

Pior caso – continuação

Temos então que

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

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 assintótica de algoritmos

- Como dito anteriormente, na maior parte desta disciplina, estaremos nos concentrando na **análise de pior caso** e no **comportamento assintótico** dos algoritmos (instâncias de tamanho grande).
- O algoritmo Ordena-Por-Inserção tem como complexidade (de pior caso) uma função quadrática $an^2 + bn + c$, onde a, b, c são constantes absolutas que dependem apenas dos custos c_i.
- O estudo assintótico nos permite "jogar para debaixo do tapete" os valores destas constantes, i.e., aquilo que independe do tamanho da entrada (neste caso os valores de a, b e c).
- **Por que podemos fazer isso ?**

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 notação assintótica, dizemos que o algoritmo Ordena-Por-Inserção tem complexidade de tempo de pior caso $\Theta(n^2)$.
- Isto quer dizer duas coisas:
  - a complexidade de tempo é limitada (superiormente) assintoticamente por algum polinômio da forma an^2 para alguma constante a,
  - para todo n suficientemente grande, existe alguma instância de tamanho n que consome tempo pelo menos dn^2, para alguma contante positiva d.
- Mais adiante discutiremos em detalhes o uso da notação assintótica em análise de algoritmos.

Ordenação por intercalação

Q que significa intercalar dois (sub)vetores ordenados?

Problema: Dados $A[p\ldots q]$ e $A[q+1\ldots r]$ crescentes, rearranjar $A[p\ldots r]$ de modo que ele fique em ordem crescente.

Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

Saída:

	p				q				r
A	11	22	33	44	55	66	77	88	99

Intercalação

	p				q				r
A	22	33	55	77	99	11	44	66	88
B									

Intercalação

	k								
A									
	i								j
B	22	33	55	77	99	88	66	44	11

Intercalação

		k							
A	11								
	i							j	
B	22	33	55	77	99	88	66	44	11

Intercalação

			k						
A	11	22							
		i						j	
B	22	33	55	77	99	88	66	44	11

Intercalação

				k					
A	11	22	33						
			i					j	
B	22	33	55	77	99	88	66	44	11

Intercalação

					k				
A	11	22	33	44					
			i				j		
B	22	33	55	77	99	88	66	44	11

Intercalação

						k			
A	11	22	33	44	55				

				i			j		
B	22	33	55	77	99	88	66	44	11

Intercalação

							k		
A	11	22	33	44	55	66			

				i		j			
B	22	33	55	77	99	88	66	44	11

Intercalação

								k	
A	11	22	33	44	55	66	77		

					i	j			
B	22	33	55	77	99	88	66	44	11

Intercalação

									k
A	11	22	33	44	55	66	77	88	

					$i = j$				
B	22	33	55	77	99	88	66	44	11

Intercalação

A	11	22	33	44	55	66	77	88	99

					j	i			
B	22	33	55	77	99	88	66	44	11

Intercalação

Pseudo-código

INTERCALA(A, p, q, r)
para $i \leftarrow p$ **até** q **faça**
  $B[i] \leftarrow A[i]$
para $j \leftarrow q+1$ **até** r **faça**
  $B[r+q+1-j] \leftarrow A[j]$
$i \leftarrow p$
$j \leftarrow r$
para $k \leftarrow p$ **até** r **faça**
  se $B[i] \leq B[j]$
    então $A[k] \leftarrow B[i]$
      $i \leftarrow i+1$
    senão $A[k] \leftarrow B[j]$
      $j \leftarrow j-1$

Complexidade de Intercala

Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

Saída:

	p				q				r
A	11	22	33	44	55	66	77	88	99

Tamanho da entrada: $n = r - p + 1$

Consumo de tempo: $\Theta(n)$

Corretude de Intercala

Invariante principal de Intercala:

No começo de cada iteração do laço das linhas 7–12, vale que:

1. $A[p \ldots k-1]$ está ordenado,
2. $A[p \ldots k-1]$ contém todos os elementos de $B[p \ldots i-1]$ e de $B[j+1 \ldots r]$,
3. $B[i] \geq A[k-1]$ e $B[j] \geq A[k-1]$.

Exercício. Prove que a afirmação acima é de fato um invariante de INTERCALA.

Exercício. (fácil) Mostre usando o invariante acima que INTERCALA é correto.

Algoritmos recursivos

"To understand recursion, we must first understand recursion."
(anônimo)

- O que é o paradigma de **divisão-e-conquista**?
- Como mostrar a corretude de um algoritmo recursivo?
- Como analisar o consumo de tempo de um algoritmo recursivo?
- O que é uma **fórmula de recorrência**?
- O que significa *resolver* uma fórmula de recorrência?

Recursão e o paradigma de divisão-e-conquista

- Um algoritmo recursivo encontra a saída para uma instância de entrada de um problema chamando a si mesmo para resolver instâncias menores deste mesmo problema.
- Algoritmos de divisão-e-conquista possuem três etapas em cada nível de recursão:
  1. **Divisão:** o problema é dividido em subproblemas semelhantes ao problema original, porém tendo como entrada instâncias de tamanho menor.
  2. **Conquista:** cada subproblema é resolvido recursivamente a menos que o tamanho de sua entrada seja suficientemente "pequeno", quando este é resolvido diretamente.
  3. **Combinação:** as soluções dos subproblemas são combinadas para obter uma solução do problema original.

Exemplo de divisão-e-conquista: *Mergesort*

- Mergesort é um algoritmo para resolver o problema de ordenação e um exemplo clássico do uso do paradigma de divisão-e-conquista. (*to merge* = intercalar)
- Descrição do Mergesort em alto nível:
  1. **Divisão**: divida o vetor com n elementos em dois subvetores de tamanho $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$, respectivamente.
  2. **Conquista**: ordene os dois vetores recursivamente usando o Mergesort;
  3. **Combinação**: intercale os dois subvetores para obter um vetor ordenado usando o algoritmo Intercala.

Mergesort

	p				q				r
A	66	33	55	44	99	11	77	22	88

Mergesort

	p				q				r
A	66	33	55	44	99	11	77	22	88

	p		q		r				
A	66	33	55	44	99				

Mergesort

	p				q				r
A	66	33	55	44	99	11	77	22	88

	p		q		r				
A	66	33	55	44	99				

	p	q	r						
A	66	33	55						

Mergesort

	p				q				r
A	66	33	55	44	99	11	77	22	88

	p		q		r				
A	66	33	55	44	99				

	p	q	r						
A	66	33	55						

	p	r							
A	66	33							

Mergesort

	p				q				r
A	66	33	55	44	99	11	77	22	88

	p		q		r				
A	66	33	55	44	99				

	p	q	r						
A	66	33	55						

	p	r							
A	66	33							

	$p = r$								
A	66								

Mergesort

	p				q				r
A	66	33	55	44	99	11	77	22	88

	p		q		r				
A	66	33	55	44	99				

	p	q	r						
A	66	33	55						

	p	r							
A	66	33							

Mergesort

	p				q				r
A	66	33	55	44	99	11	77	22	88

	p		q		r				
A	66	33	55	44	99				

	p	q	r						
A	66	33	55						

	p	r							
A	66	33							

		$p = r$							
A		33							

Mergesort

	p				q				r
A	66	33	55	44	99	11	77	22	88
	p		q		r				
A	66	33	55	44	99				
	p	q	r						
A	66	33	55						
	p	r							
A	66	33							

Mergesort

	p				q				r
A	33	66	55	44	99	11	77	22	88
	p		q		r				
A	33	66	55	44	99				
	p	q	r						
A	33	66	55						
	p	r							
A	33	66							

Mergesort

	p				q				r
A	33	66	55	44	99	11	77	22	88
	p		q		r				
A	33	66	55	44	99				
	p	q	r						
A	33	66	55						

Mergesort

	p				q				r
A	33	66	55	44	99	11	77	22	88
	p		q		r				
A	33	66	55	44	99				
	p	q	r						
A	33	66	55						
			$p = r$						
A			55						

Mergesort

	p				q				r
A	33	66	55	44	99	11	77	22	88
	p		q		r				
A	33	66	55	44	99				
	p	q	r						
A	33	66	55						

Mergesort

	p				q				r
A	33	55	66	44	99	11	77	22	88
	p		q		r				
A	33	55	66	44	99				
	p	q	r						
A	33	55	66						

Mergesort

	p				q				r
A	33	55	66	44	99	11	77	22	88

	p		q		r				
A	33	55	66	44	99				

Mergesort

	p				q				r
A	33	55	66	44	99	11	77	22	88

	p		q		r				
A	33	55	66	44	99				

				p	r				
A				44	99				

Mergesort

	p				q				r
A	33	55	66	44	99	11	77	22	88

	p		q		r				
A	33	55	66	44	99				

				p	r				
A				44	99				

				$p = r$					
A				44					

Mergesort

	p				q				r
A	33	55	66	44	99	11	77	22	88

	p		q		r				
A	33	55	66	44	99				

				p	r				
A				44	99				

Mergesort

	p				q				r
A	33	55	66	44	99	11	77	22	88

	p		q		r				
A	33	55	66	44	99				

				p	r				
A				44	99				

					$p = r$				
A					99				

Mergesort

	p				q				r
A	33	55	66	44	99	11	77	22	88

	p		q		r				
A	33	55	66	44	99				

				p	r				
A				44	99				

Mergesort

	p				q				r
A	33	55	66	44	99	11	77	22	88

	p		q		r				
A	33	55	66	44	99				

Mergesort

	p				q				r
A	33	44	55	66	99	11	77	22	88

	p		q		r				
A	33	44	55	66	99				

Mergesort

	p				q				r
A	33	44	55	66	99	11	77	22	88

Mergesort

	p				q				r
A	33	44	55	66	99	11	77	22	88

						p			r
A						11	77	22	88

Mergesort

	p				q				r
A	33	44	55	66	99	11	77	22	88

						p			r
A						11	77	22	88

						p	r		
A						11	77		

Mergesort

	p				q				r
A	33	44	55	66	99	11	77	22	88

						p			r
A						11	77	22	88

						p	r		
A						11	77		

						$p = r$			
A						11			

Mergesort

	p				q				r
A	33	44	55	66	99	11	77	22	88

						p			r
A						11	77	22	88

						p	r		
A						11	77		

Mergesort

	p				q				r
A	33	44	55	66	99	11	77	22	88

						p			r
A						11	77	22	88

						p	r		
A						11	77		

							$p = r$		
A							77		

Mergesort

	p				q				r
A	33	44	55	66	99	11	77	22	88

						p			r
A						11	77	22	88

						p	r		
A						11	77		

Mergesort

	p				q				r
A	33	44	55	66	99	11	77	22	88

						p			r
A						11	77	22	88

Mergesort

	p				q				r
A	33	44	55	66	99	11	77	22	88

						p			r
A						11	77	22	88

								p	r
A								22	88

Mergesort

	p				q				r
A	33	44	55	66	99	11	77	22	88

						p			r
A						11	77	22	88

								p	r
A								22	88

								$p = r$	
A								22	

Mergesort

	p				q				r
A	33	44	55	66	99	11	77	22	88
						p			r
A						11	77	22	88
								p	r
A								22	88

Mergesort

	p				q				r
A	33	44	55	66	99	11	77	22	88
						p			r
A						11	77	22	88
								p	r
A								22	88
									$p = r$
A									88

Mergesort

	p				q				r
A	33	44	55	66	99	11	77	22	88
						p			r
A						11	77	22	88
								p	r
A								22	88

Mergesort

	p				q				r
A	33	44	55	66	99	11	77	22	88
						p			r
A						11	77	22	88

Mergesort

	p				q				r
A	33	44	55	66	99	11	22	77	88
						p			r
A						11	22	77	88

Mergesort

	p				q				r
A	33	44	55	66	99	11	22	77	88

Mergesort

		p				q				r
A	11	22	33	44	55	66	77	88	99

Mergesort

	p				q				r
A	11	22	33	44	55	66	77	88	99

Mergesort

Relembrando: o objetivo é rearranjar $A[p \dots r]$, com $p \leq r$, em ordem crescente.

```
MERGESORT(A, p, r)
   se p < r
      então q ← ⌊(p + r)/2⌋
            MERGESORT(A, p, q)
            MERGESORT(A, q + 1, r)
            INTERCALA(A, p, q, r)
```

	p				q				r
A	66	33	55	44	99	11	77	22	88

Mergesort

Relembrando: o objetivo é rearranjar $A[p \dots r]$, com $p \leq r$, em ordem crescente.

```
MERGESORT(A, p, r)
   se p < r
      então q ← ⌊(p + r)/2⌋
            MERGESORT(A, p, q)
            MERGESORT(A, q + 1, r)
            INTERCALA(A, p, q, r)
```

	p				q				r
A	33	44	55	66	99	11	77	22	88

Mergesort

Relembrando: o objetivo é rearranjar $A[p \dots r]$, com $p \leq r$, em ordem crescente.

```
MERGESORT(A, p, r)
   se p < r
      então q ← ⌊(p + r)/2⌋
            MERGESORT(A, p, q)
            MERGESORT(A, q + 1, r)
            INTERCALA(A, p, q, r)
```

	p				q				r
A	33	44	55	66	99	11	22	77	88

Mergesort

Relembrando: o objetivo é rearranjar $A[p \dots r]$, com $p \leq r$, em ordem crescente.

```
MERGESORT(A, p, r)
   se p < r
      então q ← ⌊(p + r)/2⌋
            MERGESORT(A, p, q)
            MERGESORT(A, q + 1, r)
            INTERCALA(A, p, q, r)
```

	p				q				r
A	11	22	33	44	55	66	77	88	99

Corretude do Mergesort

```
MERGESORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3               MERGESORT(A, p, q)
4               MERGESORT(A, q + 1, r)
5               INTERCALA(A, p, q, r)
```

O algoritmo está correto?

A corretude do algoritmo Mergesort apoia-se na corretude do algoritmo Intercala e pode ser demonstrada **por indução** em $n := r - p + 1$.

Aprenderemos como fazer provas por indução mais adiante.

Complexidade do Mergesort

```
MERGESORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3               MERGESORT(A, p, q)
4               MERGESORT(A, q + 1, r)
5               INTERCALA(A, p, q, r)
```

Qual é a complexidade de MERGESORT?

Seja $T(n) :=$ o consumo de tempo máximo (pior caso) em função de $n = r - p + 1$

Complexidade do Mergesort

```
MERGESORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3               MERGESORT(A, p, q)
4               MERGESORT(A, q + 1, r)
5               INTERCALA(A, p, q, r)
```

linha	consumo de tempo
1	?
2	?
3	?
4	?
5	?

$T(n) = ?$

Complexidade do Mergesort

```
MERGESORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3               MERGESORT(A, p, q)
4               MERGESORT(A, q + 1, r)
5               INTERCALA(A, p, q, r)
```

linha	consumo de tempo
1	$\Theta(1)$
2	$\Theta(1)$
3	$T(\lceil n/2 \rceil)$
4	$T(\lfloor n/2 \rfloor)$
5	$\Theta(n)$

$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) + \Theta(2)$

Complexidade do Mergesort

- Obtemos o que chamamos de fórmula de recorrência (i.e., uma fórmula definida em termos de si mesma).

  $T(1) = \Theta(1)$

  $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$ para $n = 2, 3, 4, \ldots$

- Em geral, ao aplicar o paradigma de divisão-e-conquista, chega-se a um algoritmo recursivo cuja complexidade $T(n)$ é uma fórmula de recorrência.
- É necessário então resolver a recorrência! Mas, o que significa resolver uma recorrência?
- Significa encontrar uma "fórmula fechada" para $T(n)$.
- No caso, $T(n) = \Theta(n \lg n)$. Assim, o consumo de tempo do Mergesort é $\Theta(n \lg n)$ no pior caso.
- Veremos mais tarde como resolver recorrências.

Crescimento de funções

Notação Assintótica

- Vamos expressar complexidade através de funções em variáveis que descrevam o tamanho de instâncias do problema. Exemplos:
  - Problemas de aritmética de precisão arbitrária: número de bits (ou bytes) dos inteiros.
  - Problemas em grafos: número de vértices e/ou arestas
  - Problemas de ordenação de vetores: tamanho do vetor.
  - Busca em textos: número de caracteres do texto ou padrão de busca.
- Vamos supor que funções que expressam complexidade são sempre positivas, já que estamos medindo número de operações.

Comparação de Funções

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

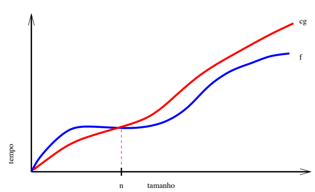
	$n = 100$	$n = 1000$	$n = 10^4$	$n = 10^6$	$n = 10^9$
$\log n$	2	3	4	6	9
n	100	1000	10^4	10^6	10^9
$n \log n$	200	3000	$4 \cdot 10^4$	$6 \cdot 10^6$	$9 \cdot 10^9$
n^2	10^4	10^6	10^8	10^{12}	10^{18}
$100n^2 + 15n$	$1,0015 \cdot 10^6$	$1,00015 \cdot 10^8$	$\approx 10^{10}$	$\approx 10^{14}$	$\approx 10^{20}$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

Classe O

Definição:

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

Informalmente, dizemos que, se $f(n) \in O(g(n))$, então $f(n)$ cresce no máximo tão rapidamente quanto $g(n)$.



Classe O

Definição:

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

Informalmente, dizemos que, se $f(n) \in O(g(n))$, então $f(n)$ cresce no máximo tão rapidamente quanto $g(n)$.

Exemplo:

$\frac{1}{2}n^2 - 3n \in O(n^2)$
Valores de c e n_0 que satisfazem a definição são

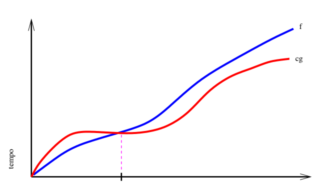
$$c = \frac{1}{2} \text{ e } n_0 = 7.$$

Classe Ω

Definição:

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

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



Classe Ω

Definição:

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

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

Exemplo:

$\frac{1}{2}n^2 - 3n \in \Omega(n^2)$
Valores de c e n_0 que satisfazem a definição são

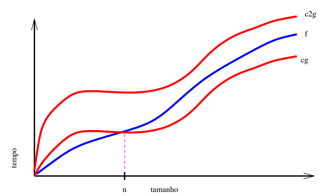
$$c = \frac{1}{14} \text{ e } n_0 = 7.$$

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, dizemos que, se $f(n) \in \Theta(g(n))$, então $f(n)$ cresce tão rapidamente quanto $g(n)$.



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, dizemos que, se $f(n) \in \Theta(g(n))$, então $f(n)$ cresce tão rapidamente quanto $g(n)$.

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

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

Classe o

Definição:

$o(g(n)) = \{f(n) :$ para toda constante positiva c, existe uma constante $n_0 > 0$ tal que $0 \leq f(n) < cg(n)$, para todo $n \geq n_0\}$.

Informalmente, dizemos que, se $f(n) \in o(g(n))$, então $f(n)$ cresce mais lentamente que $g(n)$.

Exemplo:

$1000n^2 \in o(n^3)$

Para todo valor de c, um n_0 que satisfaz a definição é

$$n_0 = \left\lceil \frac{1000}{c} \right\rceil + 1.$$

Classe ω

Definição:

$\omega(g(n)) = \{f(n) :$ para toda constante positiva c, existe uma constante $n_0 > 0$ tal que $0 \leq cg(n) < f(n)$, para todo $n \geq n_0.\}$

Informalmente, dizemos que, se $f(n) \in \omega(g(n))$, então $f(n)$ cresce mais rapidamente que $g(n)$.

Exemplo:

$\frac{1}{1000}n^2 \in \omega(n)$

Para todo valor de c, um n_0 que satisfaz a definição é

$$n_0 = \lceil 1000c \rceil + 1.$$

Definições equivalentes

$f(n) \in o(g(n))$ se $\displaystyle\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

$f(n) \in O(g(n))$ se $\displaystyle\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$.

$f(n) \in \Theta(g(n))$ se $\displaystyle 0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$.

$f(n) \in \Omega(g(n))$ se $\displaystyle\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$.

$f(n) \in \omega(g(n))$ se $\displaystyle\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

Propriedades das Classes

Transitividade:

Se $f(n) \in O(g(n))$ e $g(n) \in O(h(n))$, então $f(n) \in O(h(n))$.
Se $f(n) \in \Omega(g(n))$ e $g(n) \in \Omega(h(n))$, então $f(n) \in \Omega(h(n))$.
Se $f(n) \in \Theta(g(n))$ e $g(n) \in \Theta(h(n))$, então $f(n) \in \Theta(h(n))$.
Se $f(n) \in o(g(n))$ e $g(n) \in o(h(n))$, então $f(n) \in o(h(n))$.
Se $f(n) \in \omega(g(n))$ e $g(n) \in \omega(h(n))$, então $f(n) \in \omega(h(n))$.

Propriedades das Classes

Reflexividade:
$f(n) \in O(f(n))$.
$f(n) \in \Omega(f(n))$.
$f(n) \in \Theta(f(n))$.
Simetria:
$f(n) \in \Theta(g(n))$ se, e somente se, $g(n) \in \Theta(f(n))$.
Simetria Transposta:
$f(n) \in O(g(n))$ se, e somente se, $g(n) \in \Omega(f(n))$.
$f(n) \in o(g(n))$ se, e somente se, $g(n) \in \omega(f(n))$.

Exemplos

Quais as relações de comparação assintótica das funções:

- 2^π
- $\log n$
- n
- $n \log n$
- n^2
- $100n^2 + 15n$
- 2^n

Demonstração Direta

A *demonstração direta* de uma implicação $p \Rightarrow q$ é uma sequência de passos lógicos (implicações):

$$p \Rightarrow p_1 \Rightarrow p_2 \Rightarrow \cdots \Rightarrow p_n \Rightarrow q,$$

que resultam, por transitividade, na implicação desejada. Cada passo da demonstração é um axioma ou um teorema demonstrado previamente.

Exemplo:
Provar que $\sum_{i=1}^{k} 2i - 1 = k^2$.

Demonstração pela Contrapositiva

A *contrapositiva* de $p \Rightarrow q$ é $\neg q \Rightarrow \neg p$.
A contrapositiva é equivalente à implicação original. A veracidade de $\neg q \Rightarrow \neg p$ implica a veracidade de $p \Rightarrow q$, e vice-versa.
A técnica é útil quando é mais fácil demonstrar a contrapositiva que a implicação original.
Para demonstrarmos a contrapositiva de uma implicação, podemos utilizar qualquer técnica de demonstração.

Exemplo:
Provar que se $2 \mid 3m$, então $2 \mid m$.

Demonstração por Contradição

A *Demonstração por contradição* envolve supor absurdamente que a afirmação a ser demonstrada é falsa e obter, através de implicações válidas, uma conclusão contraditória.
A contradição obtida implica que a hipótese absurda é falsa e, portanto, a afirmação é de fato verdadeira.
No caso de uma implicação $p \Rightarrow q$, equivalente a $\neg p \vee q$, a negação é $p \wedge \neg q$.

Exemplo:
Dados os inteiros positivos n e $n + 1$, provar que o maior inteiro que divide ambos n e $n + 1$ é 1.

Demonstração por Casos

Na *Demonstração por Casos*, particionamos o universo de possibilidades em um conjunto finito de casos e demonstramos a veracidade da implicação para cada caso.
Para demonstrar cada caso individual, qualquer técnica de demonstração pode ser utilizada.

Exemplo:
Provar que a soma de dois inteiros x e y de mesma paridade é sempre par.

Demonstração por Indução

Na *Demonstração por Indução*, queremos demonstrar a validade de $P(n)$, uma propriedade P com um parâmetro natural n associado, para todo valor de n.
Há um número infinito de casos a serem considerados, um para cada valor de n. Demonstramos os infinitos casos de uma só vez:

- **Base da Indução:** Demonstramos $P(1)$.
- **Hipótese de Indução:** Supomos que $P(n)$ é verdadeiro.
- **Passo de Indução:** Provamos que $P(n+1)$ é verdadeiro, a partir da hipótese de indução.

Exemplo:

Provar que $\sum_{i=1}^{k} 2i - 1 = k^2$, agora por indução.

Indução Fraca $\times$ Indução Forte

A *indução forte* difere da *indução fraca* (ou *simples*) apenas na suposição da hipótese.
No caso da indução forte, devemos supor que a propriedade vale para todos os casos anteriores, não somente para o anterior, ou seja:

- **Base da Indução:** Demonstramos $P(1)$.
- **Hipótese de Indução Forte:** Supomos que $P(k)$ é verdadeiro, para todo $k \leq n$.
- **Passo de Indução:** Provamos que $P(n+1)$ é verdadeiro, a partir da hipótese de indução.

Exemplo:

Prove que todo inteiro n pode ser escrito como a soma de diferentes potências de 2.