# MO417 — Complexidade de Algoritmos I

Cid Carvalho de Souza    Cândida Nunes da Silva
Orlando Lee

12 de setembro de 2011

Revisado por Zanoni Dias



## Introdução

O que veremos nesta disciplina?

- Como provar a "corretude" de um algoritmo
- Estimar a quantidade de recursos (tempo, memória) de um algoritmo = análise de complexidade
- Técnicas e idéias gerais de projeto de algoritmos: divisão-e-conquista, programação dinâmica, algoritmos gulosos, etc
- Tema recorrente: natureza recursiva de vários problemas
- A dificuldade intrínseca de vários problemas: inexistência de soluções eficientes



## Algoritmos

O que é um algoritmo?

Informalmente, um algoritmo é um procedimento computacional bem definido que:

- recebe um conjunto de valores como entrada e
- produz um conjunto de valores como saída.

Equivalentemente, um algoritmo é uma ferramenta para resolver um problema computacional. Este problema define a relação precisa que deve existir entre a entrada e a saída do algoritmo.



## Exemplos de problemas: teste de primalidade

Problema: determinar se um dado número é primo.

Exemplo:

Entrada: 9411461

Saída: É primo.

Exemplo:

Entrada: 8411461

Saída: Não é primo.

## Exemplos de problemas: ordenação

Definição: um vetor $A[1 \dots n]$ é crescente se $A[1] \leq \dots \leq A[n]$.

Problema: rearranjar um vetor $A[1 \dots n]$ de modo que fique crescente.

Entrada:

| 1 | | | | | | | | | | $n$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 33 | 55 | 33 | 44 | 33 | 22 | 11 | 99 | 22 | 55 | 77 |

Saída:

| 1 | | | | | | | | | | $n$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 22 | 22 | 33 | 33 | 33 | 44 | 55 | 55 | 77 | 99 |



## Instância de um problema

Uma instância de um problema é um conjunto de valores que serve de entrada para esse.

Exemplo:
Os números 9411461 e 8411461 são instâncias do problema de primalidade.

Exemplo:
O vetor

| 1 | | | | | | | | | | $n$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 33 | 55 | 33 | 44 | 33 | 22 | 11 | 99 | 22 | 55 | 77 |

é uma instância do problema de ordenação.



## A importância dos algoritmos para a computação

- Onde se encontra aplicações para o uso/desenvolvimento de algoritmos "eficientes"?
  - projetos de genoma de seres vivos
  - rede mundial de computadores
  - comércio eletrônico
  - planejamento da produção de indústrias
  - logística de distribuição
  - *games* e filmes
  - ...



## Dificuldade intrínseca de problemas

- Infelizmente, existem certos problemas para os quais **não se conhece** algoritmos eficientes capazes de resolvê-los (e nem existe uma esperança real que tais algoritmos possam existir). Eles são chamados problemas $\mathcal{NP}$-completos.
- Esses problemas tem a característica notável de que se um deles admitir um algoritmo "eficiente" então todos admitem algoritmos "eficientes".
- Por que devo me preocupar com problemas $\mathcal{NP}$-completos?
  Problemas dessa classe surgem em inúmeras situações práticas!

## Dificuldade intrínseca de problemas

Exemplos:

- calcular as rotas dos caminhões de entrega de uma distribuidora de bebidas em São Paulo, minimizando a distância percorrida. (vehicle routing)
- calcular o número mínimo de *containers* para transportar um conjunto de caixas com produtos. (bin packing 3D)
- calcular a localização e o número mínimo de antenas de celulares para garantir a cobertura de uma certa região geográfica. (facility location)
- e muito mais...

É importante saber indentificar quando estamos lidando com um problema $\mathcal{NP}$-completo!



## Algoritmos e tecnologia

- O mundo ideal: os computadores têm velocidade de processamento e memória infinita. Neste caso, qualquer algoritmo é igualmente bom e esta disciplina é inútil! Porém...
- O mundo real: computadores têm velocidade de processamento e memória limitadas.

Neste caso faz **muita** diferença ter um **bom** algoritmo.



## Algoritmos e tecnologia

Exemplo: **ordenação de um vetor de $n$ elementos**

- Suponha que os computadores $A$ e $B$ executam $1G$ e $10M$ instruções por segundo, respectivamente. Ou seja, $A$ é **100 vezes mais rápido** que $B$.
- Algoritmo 1: implementado em $A$ por um excelente programador em linguagem de máquina (ultra-rápida). Executa $2n^2$ instruções.
- Algoritmo 2: implementado na máquina $B$ por um programador mediano em linguagem de alto nível dispondo de um compilador "meia-boca". Executa $50n \log n$ instruções.



## Algoritmos e tecnologia

- O que acontece quando ordenamos um vetor de um milhão de elementos? **Qual algoritmo é mais rápido?**
- Algoritmo 1 na máquina $A$:
  $\frac{2.(10^6)^2 \text{ instruções}}{10^9 \text{ instruções/segundo}} \approx 2000$ segundos
- Algoritmo 2 na máquina $B$:
  $\frac{50.(10^6 \log 10^6) \text{ instruções}}{10^7 \text{ instruções/segundo}} \approx 100$ segundos
- Ou seja, $B$ foi **VINTE VEZES** mais rápido do que $A$!
- Se o vetor tiver 10 milhões de elementos, esta razão será de 2.3 dias para 20 minutos!

## Algoritmos e tecnologia – Conclusões

- O uso de um algoritmo adequado pode levar a ganhos extraordinários de desempenho.
- Isso pode ser tão importante quanto o projeto de *hardware*.
- A melhora obtida pode ser tão significativa que não poderia ser obtida simplesmente com o avanço da tecnologia.
- As melhorias nos algoritmos produzem avanços em outras componentes básicas das aplicações (pense nos compiladores, buscadores na internet, etc).



## Descrição de algoritmos

Podemos descrever um algoritmo de várias maneiras:

- usando uma linguagem de programação de alto nível: `C`, `Pascal`, `Java` etc
- implementando-o em linguagem de máquina diretamente executável em *hardware*
- em português
- em um pseudo-código de alto nível, como no livro do CLRS

Usaremos essencialmente as duas últimas alternativas nesta disciplina.



## Exemplo de pseudo-código

Algoritmo ORDENA-POR-INSERÇÃO: rearranja um vetor $A[1 \ldots n]$ de modo que fique crescente.

```
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
```



## Corretude de algoritmos

- Um algoritmo (que resolve um determinado problema) está correto se, para toda instância do problema, ele pára e devolve uma resposta correta.
- Algoritmos incorretos também têm sua utilidade, se soubermos prever a sua probabilidade de erro.
- Neste curso vamos trabalhar apenas com algoritmos corretos.

## Complexidade de algoritmos

- Em geral, não basta saber que um dado algoritmo pára. Se ele for muito lento terá pouca utilidade.
- Queremos projetar/desenvolver algoritmos eficientes (rápidos).
- Mas o que seria uma boa medida de eficiência de um algoritmo?
- Não estamos interessados em quem programou, em que linguagem foi escrito e nem qual a máquina foi usada!
- Queremos um critério uniforme para comparar algoritmos.



## Modelo Computacional

- Uma possibilidade é definir um modelo computacional de um máquina.
- O modelo computacional estabelece quais os recursos disponíveis, as instruções básicas e quanto elas custam (= tempo).
- Dentre desse modelo, podemos estimar através de uma análise matemática o tempo que um algoritmo gasta em função do tamanho da entrada (= análise de complexidade).
- A análise de complexidade depende sempre do modelo computacional adotado.



## Máquinas RAM

Salvo mencionado o contrário, usaremos o Modelo Abstrato RAM (Random Access Machine):

- simula máquinas convencionais (de verdade),
- possui um único processador que executa instruções seqüencialmente,
- tipos básicos são números inteiros e reais,
- há um limite no tamanho de cada *palavra de memória*: se a entrada tem "tamanho" $n$, então cada inteiro/real é representado por $c \log n$ bits onde $c \geq 1$ é uma constante.



## Máquinas RAM

- executa operações aritméticas (soma, subtração, multiplicação, divisão, piso, teto), comparações, movimentação de dados de tipo básico e fluxo de controle (teste *if/else*, chamada e retorno de rotinas) em **tempo constante**,
- certas operações caem ficam em uma **zona cinza**, por exemplo, exponenciação,
- veja maiores detalhes do modelo RAM no CLRS.

## Tamanho da entrada

Problema: Primalidade

Entrada: inteiro $n$

Tamanho: número de bits de $n \approx \lg n = \log_2 n$

Problema: Ordenação

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

Tamanho: $n \lg U$ onde $U$ é o maior número em $A[1 \dots n]$



## Medida de complexidade e eficiência 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.
- Geralmente adota-se uma "atitude pessimista" e faz-se uma análise de pior caso. Determina-se 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 e eficiência 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: $n$, $3n - 7$, $4n^2$, $143n^2 - 4n + 2$, $n^5$.
- Mas por que polinômios?
  Polinômios são funções bem "comportadas".



## Vantagens do método de análise proposto

- O modelo RAM é robusto e permite prever o comportamento de um algoritmo para instâncias GRANDES.
- O modelo permite comparar algoritmos que resolvem um mesmo problema.
- A análise é mais robusta em relação às evoluções tecnológicas.

## Desvantagens do método de análise proposto

- Fornece um limite de complexidade pessimista sempre considerando o pior caso.
- Em uma aplicação real, nem todas as instâncias ocorrem com a mesma freqüência e é possível que as “instâncias ruins” ocorram raramente.
- Não fornece nenhuma informação sobre o comportamento do algoritmo no caso médio.
- A análise de complexidade de algoritmos no caso médio é bastante difícil, principalmente, porque muitas vezes não é claro o que é o “caso médio”.



Começando a trabalhar



## Ordenação

Problema: ordenar um vetor em ordem crescente

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

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

Vamos começar estudando o algoritmo de ordenação baseado no método de inserção.



## 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.

## Como fazer a inserção

*chave* = 38

| 1 | | | | | *i* | *j* | | | | *n* |
|---|---|---|---|---|---|---|---|---|---|---|
| 20 | 25 | 35 | 40 | 44 | 55 | 38 | 99 | 10 | 65 | 50 |

| 1 | | | | *i* | | *j* | | | | *n* |
|---|---|---|---|---|---|---|---|---|---|---|
| 20 | 25 | 35 | 40 | 44 | | 55 | 99 | 10 | 65 | 50 |

| 1 | | | *i* | | | *j* | | | | *n* |
|---|---|---|---|---|---|---|---|---|---|---|
| 20 | 25 | 35 | 40 | | 44 | 55 | 99 | 10 | 65 | 50 |

| 1 | | *i* | | | | *j* | | | | *n* |
|---|---|---|---|---|---|---|---|---|---|---|
| 20 | 25 | 35 | | 40 | 44 | 55 | 99 | 10 | 65 | 50 |

| 1 | | *i* | | | | *j* | | | | *n* |
|---|---|---|---|---|---|---|---|---|---|---|
| 20 | 25 | 35 | 38 | 40 | 44 | 55 | 99 | 10 | 65 | 50 |



## Ordenação por inserção

| *chave* | 1 | | | | | | | *j* | | | *n* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 99 | 20 | 25 | 35 | 38 | 40 | 44 | 55 | 99 | 10 | 65 | 50 |

| *chave* | 1 | | | | | | | *j* | | | *n* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 99 | 20 | 25 | 35 | 38 | 40 | 44 | 55 | 99 | 10 | 65 | 50 |

| *chave* | 1 | | | | | | | | *j* | | *n* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 10 | 20 | 25 | 35 | 38 | 40 | 44 | 55 | 99 | 10 | 65 | 50 |

| *chave* | 1 | | | | | | | | *j* | | *n* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 10 | 10 | 20 | 25 | 35 | 38 | 40 | 44 | 55 | 99 | 65 | 50 |



## Ordenação por inserção

| *chave* | 1 | | | | | | | | | *j* | *n* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 65 | 10 | 20 | 25 | 35 | 38 | 40 | 44 | 55 | 99 | 65 | 50 |

| *chave* | 1 | | | | | | | | | *j* | *n* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 65 | 10 | 20 | 25 | 35 | 38 | 40 | 44 | 55 | 65 | 99 | 50 |

| *chave* | 1 | | | | | | | | | | *j* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 50 | 10 | 20 | 25 | 35 | 38 | 40 | 44 | 55 | 65 | 99 | 50 |

| *chave* | 1 | | | | | | | | | | *j* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 50 | 10 | 20 | 25 | 35 | 38 | 40 | 44 | 50 | 55 | 65 | 99 |



## Ordena-Por-Inserção

**Pseudo-código**

ORDENA-POR-INSERÇÃO$(A, n)$
1 **para** $j \leftarrow 2$ **até** $n$ **faça**
2 $\quad$ *chave* $\leftarrow A[j]$
3 $\quad \triangleright$ 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*

## Análise do algoritmo

O que é importante analisar ?

- Finitude: o algoritmo pára?
- Corretude: o algoritmo faz o que promete?
- Complexidade de tempo: quantas instruções são necessárias no pior caso para ordenar os $n$ elementos?



## O algoritmo pára

```
ORDENA-POR-INSERÇÃO(A, n)
1   para j ← 2 até n faça
    ...
4       i ← j − 1
5       enquanto i ≥ 1 e A[i] > chave faça
6           ...
7           i ← 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**.



## Ordena-Por-Inserção

```
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
```

O que falta fazer?

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



## 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 fornecido a priori...

## 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
```

**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, mas...
- Não provamos que ordenou os elementos da entrada!



## Melhorando a argumentação

```
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
```

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

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

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

(i5) $A[i+2 \dots j] > chave$.

Invariantes (i2) a (i5)
+condição de parada na linha 5
+atribuição da linha 7
$\Longrightarrow$ 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 os 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 $chave \leftarrow A[j]$ | $c_2$ | ? |
| 3 ▷ Insere $A[j]$ em $A[1 \dots j-1]$ | 0 | ? |
| 4 $i \leftarrow j - 1$ | $c_4$ | ? |
| 5 **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça** | $c_5$ | ? |
| 6 $A[i+1] \leftarrow A[i]$ | $c_6$ | ? |
| 7 $i \leftarrow i - 1$ | $c_7$ | ? |
| 8 $A[i+1] \leftarrow chave$ | $c_8$ | ? |

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 | # execuções |
|---|---|---|
| 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* | $c_8$ | $n-1$ |

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 $j = 2, \dots, n$ temos $A[i] \leq$ *chave* na linha 5 quando $i = j-1$. Assim, $t_j = 1$ para $j = 2, \dots, n$.

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 $A[1 \dots j-1]$, temos que compará-la com todos os elementos neste subvetor. Assim, $t_j = j$ para $j = 2, \dots, n$.

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 desprezar os valores destas constantes, i.e., aquilo que não depende 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 constante positiva $d$.
- Mais adiante discutiremos em detalhes o uso da notação assintótica em análise de algoritmos.

## Ordenação por intercalação

O 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

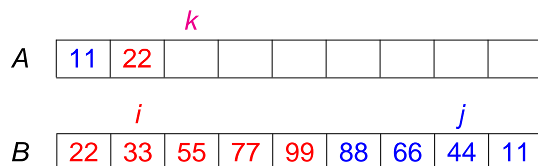
| | k | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| A | | | | | | | | | |
| | i | | | | | | | | j |
| B | 22 | 33 | 55 | 77 | 99 | 88 | 66 | 44 | 11 |



## Intercalação

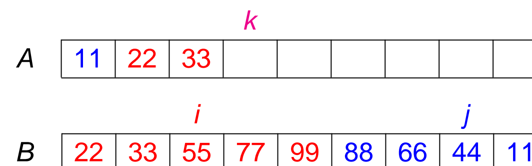
| | | k | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| A | 11 | | | | | | | | |
| | i | | | | | | | j | |
| B | 22 | 33 | 55 | 77 | 99 | 88 | 66 | 44 | 11 |

## Intercalação



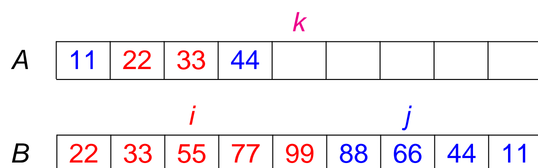



## Intercalação



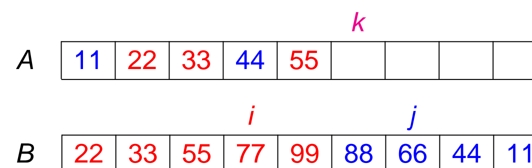



## Intercalação






## Intercalação



## 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)
 1  para i ← p até q faça
 2       B[i] ← A[i]
 3  para j ← q + 1 até r faça
 4       B[r + q + 1 − j] ← A[j]
 5  i ← p
 6  j ← r
 7  para k ← p até r faça
 8       se B[i] ≤ B[j]
 9          então A[k] ← B[i]
10                    i ← i + 1
11          senão A[k] ← B[j]
12                    j ← 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. $A[p \dots k-1]$ está ordenado,
2. $A[p \dots k-1]$ contém todos os elementos de $B[p \dots i-1]$ e de $B[j+1 \dots r]$,
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

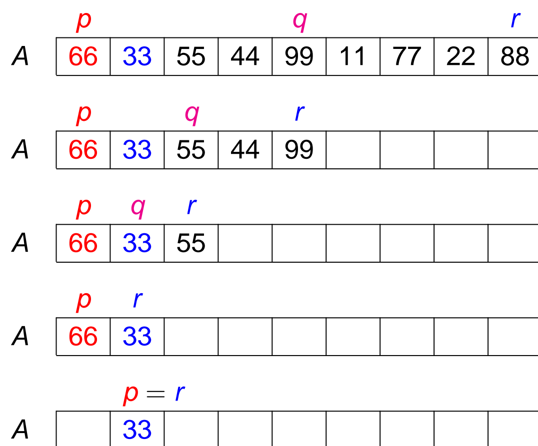
| | $p$ | | | | $q$ | | | | $r$ |
|---|---|---|---|---|---|---|---|---|---|
| $A$ | 66 | 33 | 55 | 44 | 99 | 11 | 77 | 22 | 88 |

| | $p$ | | $q$ | | $r$ | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $A$ | 66 | 33 | 55 | 44 | 99 | | | | |

| | $p$ | $q$ | $r$ | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $A$ | 66 | 33 | 55 | | | | | | |

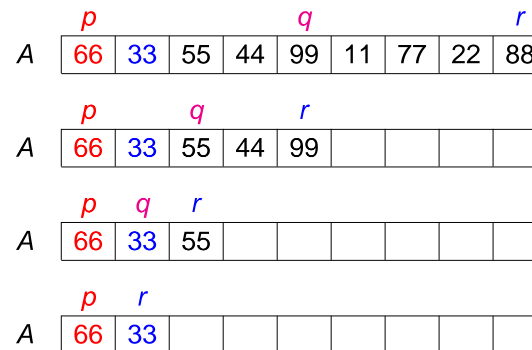
| | $p$ | $r$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $A$ | 66 | 33 | | | | | | | |

## Mergesort



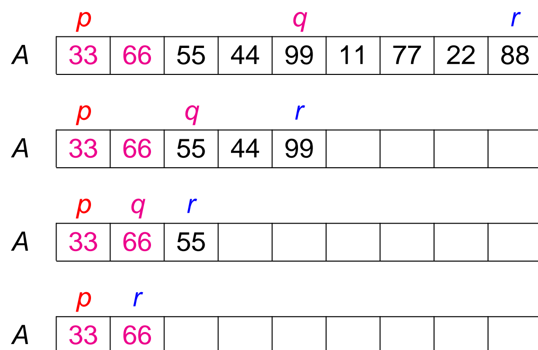



## Mergesort



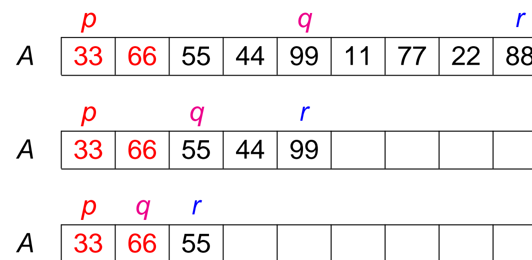



## Mergesort

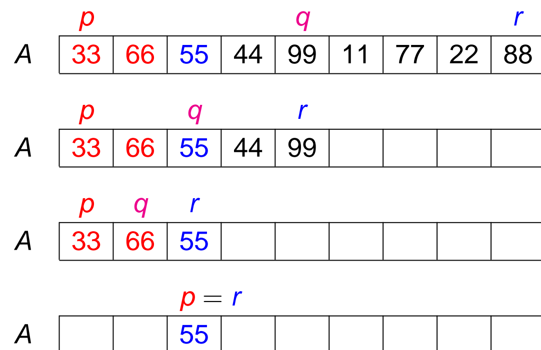





## Mergesort

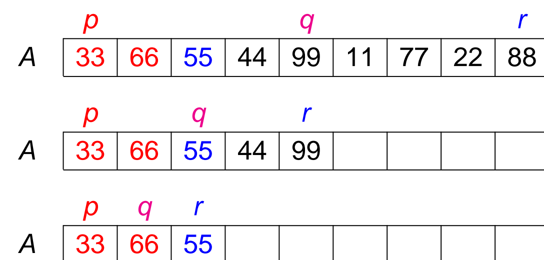


## Mergesort



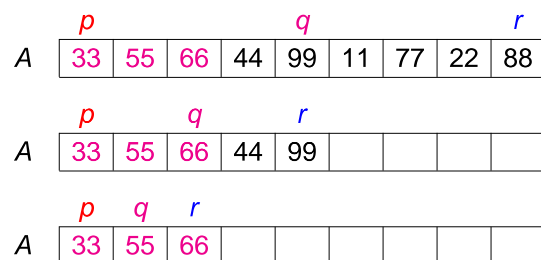



## Mergesort



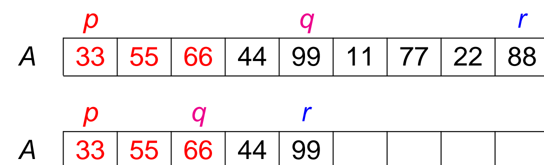



## Mergesort

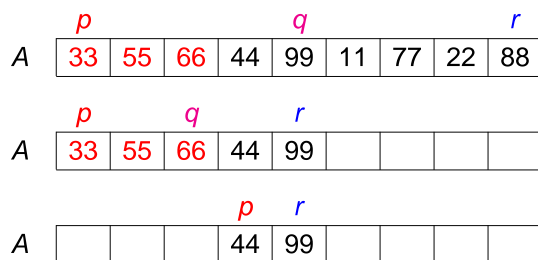





## Mergesort

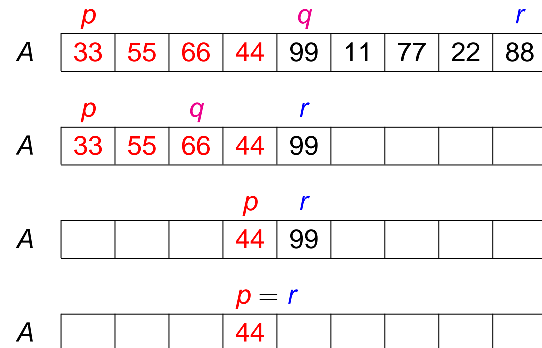


## Mergesort



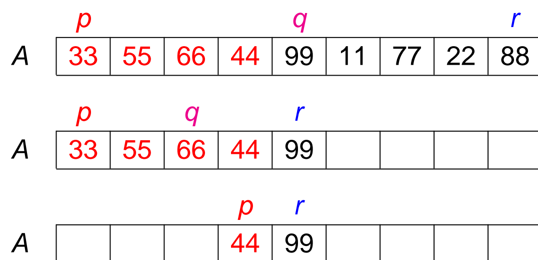



## Mergesort



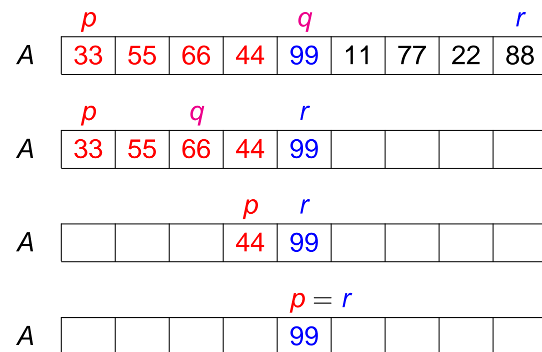



## Mergesort

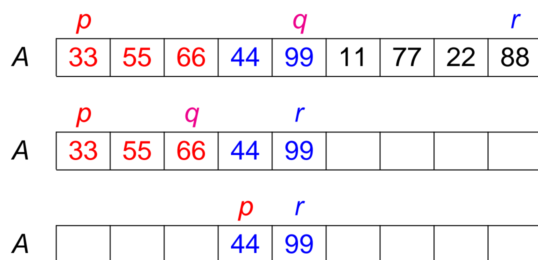





## Mergesort

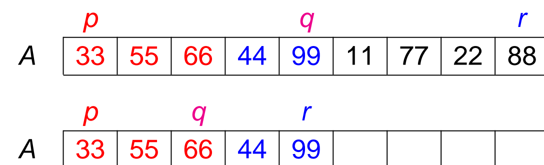


## Mergesort



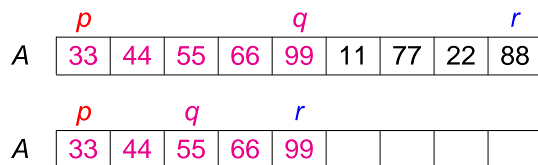



## Mergesort



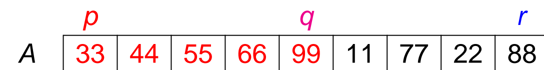



## Mergesort






## Mergesort



## 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 | | | |



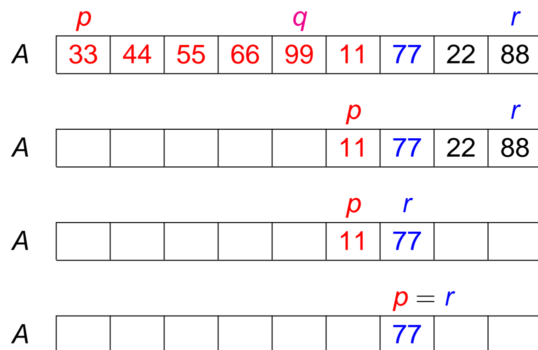
## Mergesort

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

| | | | | | | $p$ | | | $r$ |
|---|---|---|---|---|---|---|---|---|---|
| $A$ | | | | | | 11 | 77 | 22 | 88 |

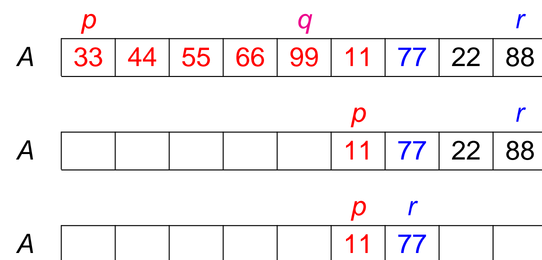
| | | | | | | $p$ | $r$ | | |
|---|---|---|---|---|---|---|---|---|---|
| $A$ | | | | | | 11 | 77 | | |

## Mergesort



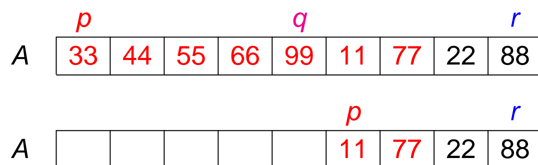



## Mergesort



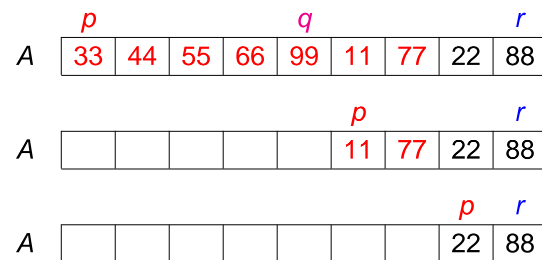



## Mergesort

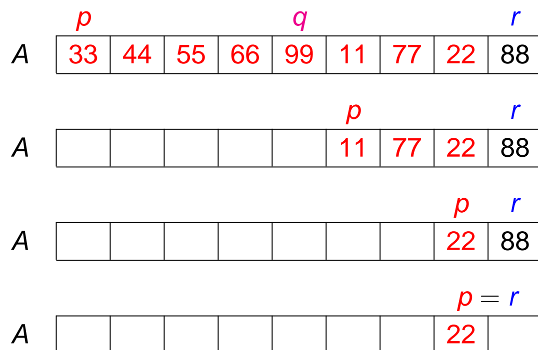





## Mergesort

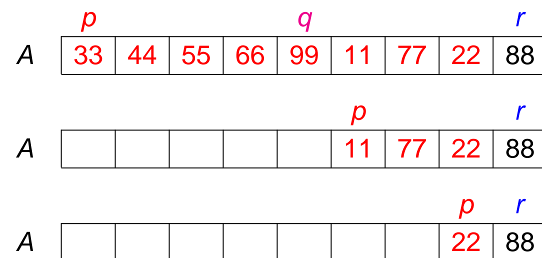


## Mergesort



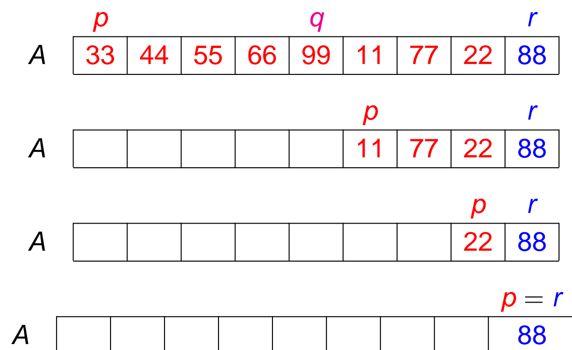



## Mergesort



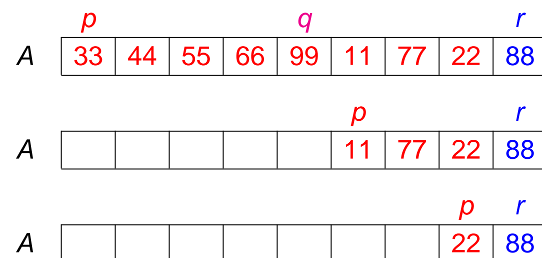



## Mergesort

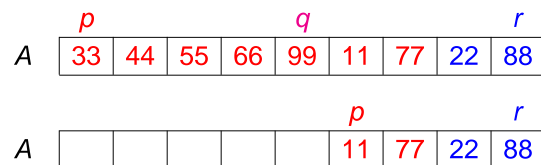





## Mergesort

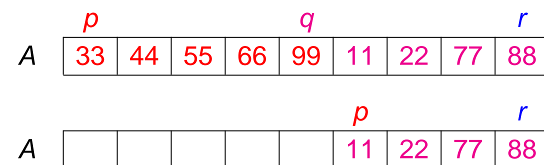


## Mergesort



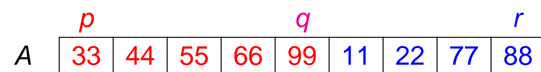



## Mergesort



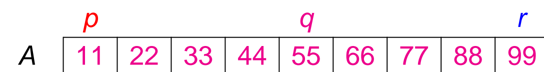



## Mergesort






## Mergesort



## Mergesort

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



## Mergesort

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

```
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)
```

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



## Mergesort

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

```
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)
```

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



## Mergesort

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

```
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)
```

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

## Mergesort

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

```
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)
```

| | 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).

$$T(1) = \Theta(1)$$
$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) \quad \text{para } n = 2, 3, 4, \ldots$$

- 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.