ESTRUTURAS DE DADOS

2024/2025

Aula 02

- O que são Estruturas de Dados
- Estruturas Lineares
- Estruturas Não-Lineares
- Análise dos Algoritmos

ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Ricardo Santos | rjs@estg.ipp.pt

O que são Estruturas de Dados?

- Um **jornal** é uma agregação, ou colecção de **páginas**
  - Cada **página** tem um colecção de **cabeçalhos**
    - Em cada um dos **cabeçalhos** está um conjunto de **colunas**
  - Cada **coluna** consiste num número de **parágrafos**
    - Cada **parágrafo** consiste em diversas **frases**
      - Cada **frase** consiste num conjunto **palavras** e **pontuação**
        - Cada **palavra** é um conjunto de **caracteres**
          - A **pontuação** é um **caracter**
            - Cada **caracter** é?

- Uma estrutura de dados é uma agregação de componentes de dados que, todos juntos, constituem um todo com significado
  - Os componentes por sua vez podem também constituir **estruturas de dados**
  - Só pára quando chegarmos a uma unidade **atómica**

- A definição de unidade atómica depende do observador
  - Como leitor, os caracteres que formam uma palavra são a unidade atómica
  - Para aquele que imprime o jornal, o que vai fazer com que os caracteres apareçam na página a unidade atómica é outra, mais relacionada com os componentes relacionados com todo o processo

Que Estruturas de Dados iremos estudar?

- Um array é uma agregação de entradas arranjadas de forma contínua com acesso sequencial a cada uma das entradas
- Existem muitas outras situações em que são necessárias estruturas de dados muito mais sofisticadas
- Categorias de estruturas de dados:
  - **lineares**
  - **não lineares**
- A categoria baseia-se em como os dados são organizados ou agregados

- Podemos também classificar as estruturas de dados de acordo com os sistemas físicos ou abstractos que estas modelam
  - **Árvores**
    - Modelam hierarquias
  - **Grafos**
    - Modelam relacionamentos simétricos

Estruturas Lineares

- **Listas**, **Queues** e **Stacks** são colecções lineares, cada uma serve como repositório em que as entradas são adicionadas e removidas à vontade
  - Diferem umas das outras na forma como as entradas podem ser acedidas depois de adicionadas

- **Lista**
  - A lista é uma colecção linear de entradas em que estas podem ser adicionadas, removidas e pesquisadas sem restrições
    - Existem dois tipos de listas
      - Listas ordenadas
      - Listas desordenadas

- ***Stack***
  - As entradas apenas podem ser removidas de acordo com a ordem contrárias com que foram adicionadas
    - Estruturas de dados *Last-in-first-out (LIFO)*
    - Não existe pesquisa por entradas na *Stack*
- ***Queue***
  - As entradas apenas podem ser removidos de acordo com a ordem com que foram adicionadas
    - Estruturas de dados *First-in-first-out (FIFO)*
    - Não existe pesquisa por entradas na *Queue*

Árvores

- Organização não linear
- Existem várias estruturas de árvore

- **Árvore Binária**
  - Consiste em entradas em que cada uma delas contribui para a árvore como um todo baseando-se na sua posição na árvore
    - Mover uma entrada de uma posição para outra altera o significado da árvore binária
- **Árvores Gerais**
  - Modelam uma hierarquia como a estrutura de uma organização, ou uma árvore genealógica
    - Organização não linear das entradas é uma generalização da estrutura em árvore binária

- **Árvore Binária de Pesquisa**
  - Possui a mesma forma estrutural de uma **Árvore Binária** mas cada entrada é independente: não contribui de maneira diferente se sua posição na árvore é alterada, nem a árvore como um todo possui um significado relacionado com a organização relativa das entradas.
  - Organização de forma ordenada: árvore análoga à lista ordenada

- **Tabela de *Hash***
  - Armazena as entradas com o único objectivo de permitir uma pesquisa eficiente
  - Requer um certo conhecimento matemático e das propriedades dos número, e das denominadas funções de *hash* que irão permitir manipular os números

Grafos

- Um grafo é um tipo especial de árvore geral já que a hierarquia é um sistema especial de relações entre as entidades
- Os grafos podem ser usados para modelar sistemas de ligações físicas tais como redes de computadores, linhas aéreas, etc. e ainda relacionamentos abstractos

- Depois de conseguirmos modelar um sistema através de um grafo podemos usar alguns algoritmos padrão para responder a algumas perguntas que podemos realizar aos sistema

- Existem dois tipos de grafos:
  - **Grafos dirigidos** - relacionamento assimétrico
  - **Grafos não dirigidos** - relacionamento simétrico

O que são Tipos de Dados Abstractos?

- Os programadores de *software* tentam escrever código que seja robusto, fácil de manter e reutilizável
  - Uma estrutura de dados é um tipo de dados abstracto
    - Para os tipos de dados primitivos de uma linguagem: ***integer***, ***real***, ***character*** e ***boolean***
      - Não nos preocupamos com a sua representação interna

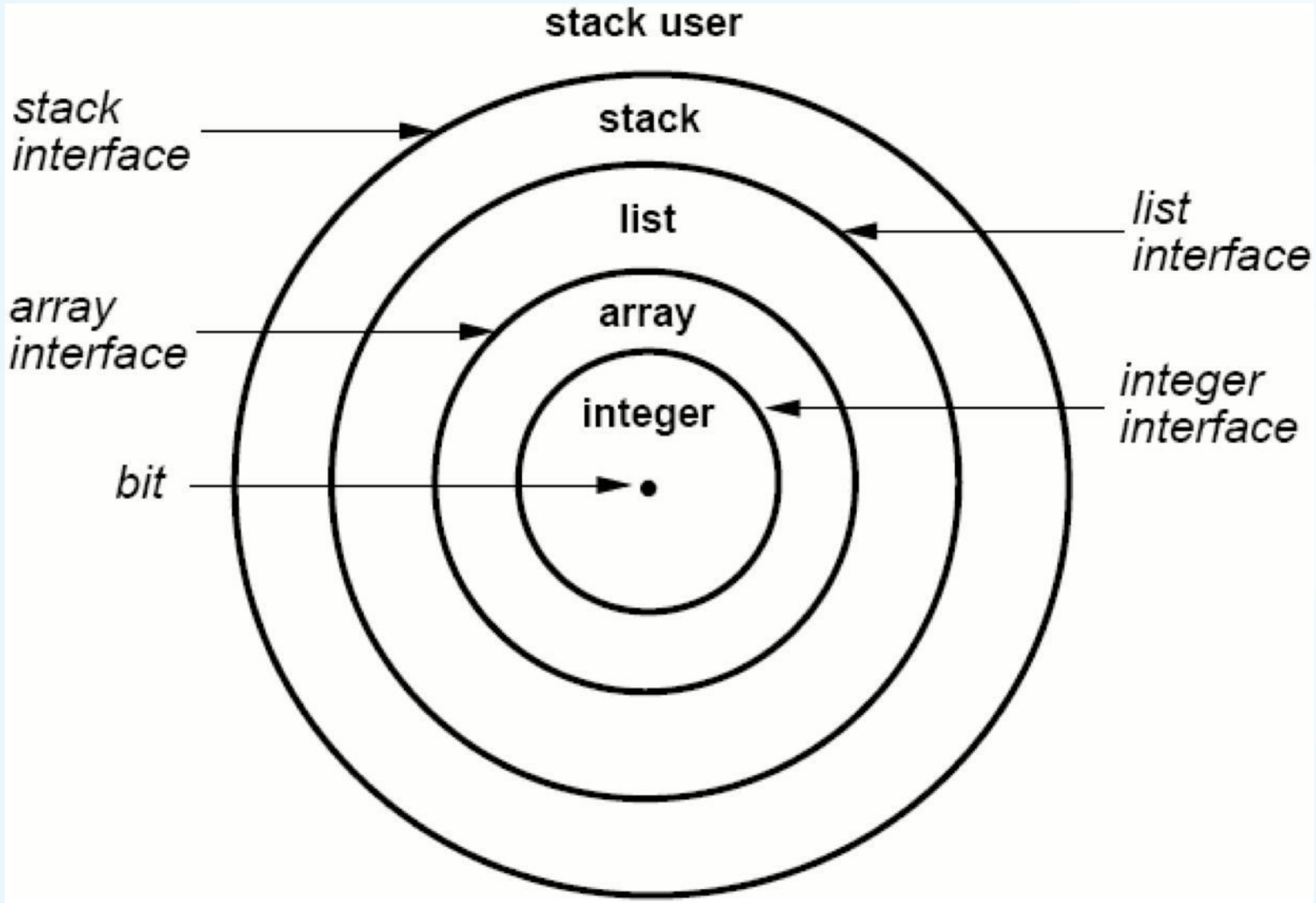
- Sabemos que podemos realizar algumas operações sob inteiros e que estes irão funcionar garantidamente de acordo com o esperado
  - Os criadores da linguagem e quem escreveu o compilador foram responsáveis por criar a interface (“**+**”, “**-**”, “*****” e “**/**”) e implementação para estes tipos de dados

- A forma como estas operações são implementadas pelo compilador no código máquina é indiferente
  - Numa máquina diferente o comportamento de um inteiro não altera, mesmo que a sua representação interna tenha alterado
  - Para os programadores o que interessa é o tipo de dados inteiro

- **Exemplo:** Os telespectadores vêm uma televisão em termos de volume, contraste, brilho, assim como outros controlos
  - Se descermos um nível, a televisão é também constituída por vários componentes
  - Um destes componentes pode ser os chips programáveis

- Por sua vez um *chip* consiste num conjunto de transístores
  - Um transístor é feito em silício que tem como constituintes átomos que representam a camada de mais baixo nível deste sistema de abstracções

- Uma estrutura de dados consiste num conjunto de camadas de abstracção
  - Uma ***Stack*** (de inteiros) pode ser criada tendo por base uma ***Lista*** (de inteiros), que por sua vez pode ser criada tendo por base um *array* (de inteiros)
- Os inteiros e o *array* são tipos definidos pela linguagem
- A ***Stack*** e a ***Lista*** são definidos pelo utilizador


stack user
stack interface
stack
list
list interface
array interface
array
integer
integer interface
bit

- Como um ADT faz uma clara separação entre interface e implementação, o utilizador vê apenas a interface e, portanto, não precisa mexer com a implementação.
  - A responsabilidade de manter a implementação é separada da responsabilidade de manutenção do código que usa um ADT.
  - Isto torna o código mais fácil de manter
  - Os ADTs podem ser usados muitas vezes em vários contextos
    - Uma ***List*** ADT pode ser usada directamente no código da aplicação ou pode ser usada por exemplo para criar um outro ADT como por exemplo a ***Stack***

- **Programação orientada a objectos é o paradigma que aborda exactamente todas estas questões!**

Estruturas de dados em POO

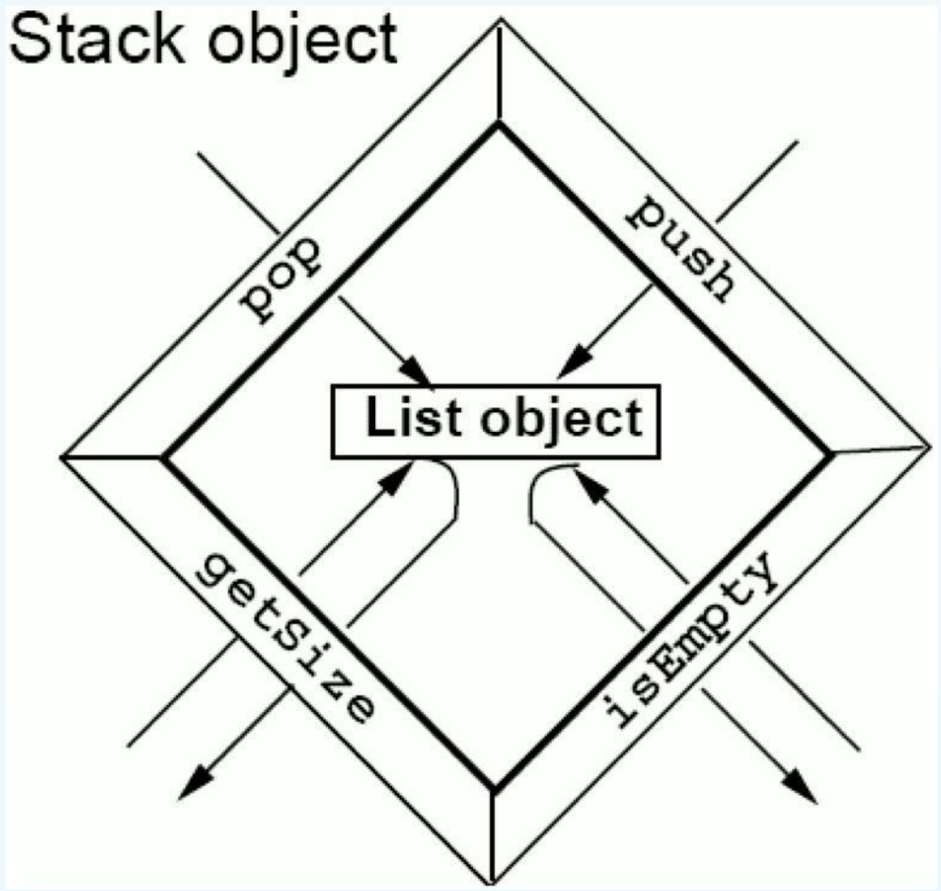
- O paradigma orientado a objectos vê um programa como um sistema de objectos que interagem entre si
  - Objectos num programa pode modelar objectos físicos, ou entidades abstractas

  - Um objecto encapsula estado e comportamento
    - Por exemplo, o estado de um inteiro, é o seu valor actual, o comportamento é o conjunto de operações aritméticas que podem ser aplicadas a inteiros

- Uma classe é análoga a um ADT, um objecto é semelhante a uma variável de um ADT
  - O utilizador de um objecto é chamado de seu cliente

- Uma ***Stack*** pode ser criada tendo por base uma ***List*** ADT
- A interface ***Stack*** define 4 operações
  - push
  - pop
  - getSize
  - isEmpty

- O objecto ***Stack*** contém um objecto ***List*** que implementa o seu estado, e o comportamento do objecto ***Stack*** é implementado com base no comportamento do objecto ***List***



- Reutilização por **herança**
  - As estruturas de dados pode ser criadas por herança de outras estruturas de dados;
  - se a estrutura de dados **A** herda da estrutura de dados **B**, então todo o objecto **A** é um objecto **B**...
  - Atenção que o contrário não é verdade!

- Herdar significa partilhar o código de implementação
- Uma linguagem que implementa o paradigma OO garante automaticamente:
  - Separação da interface e implementação pelo encapsulamento
  - Reutilização de código herdado quando permitido

Como escolher a ED mais adequada?

A interface de operações que é suportada pela estrutura de dados é um factor a considerar ao escolher entre diversas estruturas de dados disponíveis

- A eficiência das estruturas de dados:
  - Quanto espaço é necessário para a estrutura de dados?
  - Quais são os tempos de execução das operações na interface?

- Exemplo
  - Ao implementar uma fila (***Queue***) de impressão, requer uma estrutura de dados fila (***Queue***)
    - Mantém uma colecção de entradas em nenhuma ordem particular
    - Uma lista não-ordenada seria a estrutura de dados apropriada neste caso
  - Não é muito difícil atender aos requisitos da aplicação para as operações apoiadas por uma estrutura de dados
    - É mais difícil a escolha de um conjunto de estruturas de dados de candidatos que cumprem os requisitos operacionais

- Importante, e por vezes factor contraditório a considerar:
  - O tempo de execução de cada operação na interface
    - Uma estrutura de dados com a melhor interface com a melhor adequabilidade pode não ser necessariamente a melhor solução global, se por exemplo os tempos de execução das suas operações não estão à altura do pretendido
    - Quando temos mais do uma implementação da estrutura de dados cujas interfaces satisfazem as nossas necessidades, poderemos ter de seleccionar uma delas com base na comparação entre os tempos de execução das operações da interface

- O tempo é trocado pelo espaço,
  - ou seja, é consumido mais espaço para aumentar a velocidade ou seja,
  - uma redução na velocidade é trocada por uma redução do consumo de espaço

- Negociação Espaço-Tempo
  - Pretendemos "comprar" a melhor implementação de uma ***Stack***
    - ***Stack A*** - Não fornece a operação getSize
      - ou seja, não existe nenhum operação que o cliente possa usar para obter o número de entradas na ***Stack A***
    - ***Stack B*** - Fornece a operação **getSize**, e a implementação é da seguinte forma: cada vez que é invocada a operação **getSize** os elementos são removidos da ***Stack*** actual e adicionados a uma ***Stack*** temporária, durante a troca são contados os elementos

- ***Stack C*** - Fornece a operação **getSize**, e a implementação é da seguinte forma: a variável denominada ***size*** é incrementada cada vez que é introduzido um elemento, e decrementada cada vez que é removido

- Três situações:
  - Necessidade de manter um grande número de pilhas (***Stacks***), sem a necessidade de encontrar o número de entradas
  - Necessidade de manter apenas uma pilha(***Stacks***), com a frequente necessidade de encontrar o número de entradas
  - Necessidade de manter um grande número de pilhas (***Stacks***). Com raras necessidade de encontrar o número de entradas

- **Situação 1**, a ***Stack A*** cumpre os requisitos
  - No entanto podemos ser tentados a escolher a ***Stack C*** simplesmente porque no futuro poderemos querer usar a operação

- **Situação 2**, ***Stack B*** ou ***Stack C***
  - É necessário usar **getSize**
  - O **getSize** na ***Stack B*** consome mais tempo que o da ***Stack C***
  - Precisamos apenas de uma ***Stack***, a variável adicional *size* usada pela ***Stack C*** não é um problema para nós
  - Como pretendemos usar a operação **getSize** frequentemente é melhor optarmos pela ***Stack C***

- **Situação 3** apresenta uma escolha entre as ***Stacks B*** e ***C***
  - Se as chamadas a **getSize** não são frequentes podemos escolher a ***Stack B*** e sofrer uma perda de performance
  - O **getSize** mais rápido fornecido pela ***Stack C*** é obtido através do custo extra de mais uma variável por ***Stack***, que poderá consumir bastante mais espaço já que pretendemos manter um determinado número de ***Stacks***

- O **getSize** em ***Stack B*** consome mais tempo que o da ***Stack C***

- **<u>Como podemos quantificar o tempo que leva em cada um dos casos?</u>**

Análise de Algoritmos

- Um importante aspecto da qualidade de *software* é o uso eficiente dos recursos, incluindo o CPU
- A análise algorítmica é então um tópico nuclear de extrema importância
- Dá-nos uma base para a comparação da eficiência dos algoritmos
- **Exemplo:** qual o algoritmo de ordenação mais eficiente?

Funções de Crescimento

- A análise é definida à custa de termos gerais baseados:
  - no tamanho do problema (exemplo: número de itens a ordenar)
  - operações chave (exemplo: comparação de dois valores)
- Uma **função de crescimento** apresenta o relacionamento entre o tamanho do problema (**n**) e o tempo que leva a resolver o problema

$$t(n) = 15n^2 + 45n$$

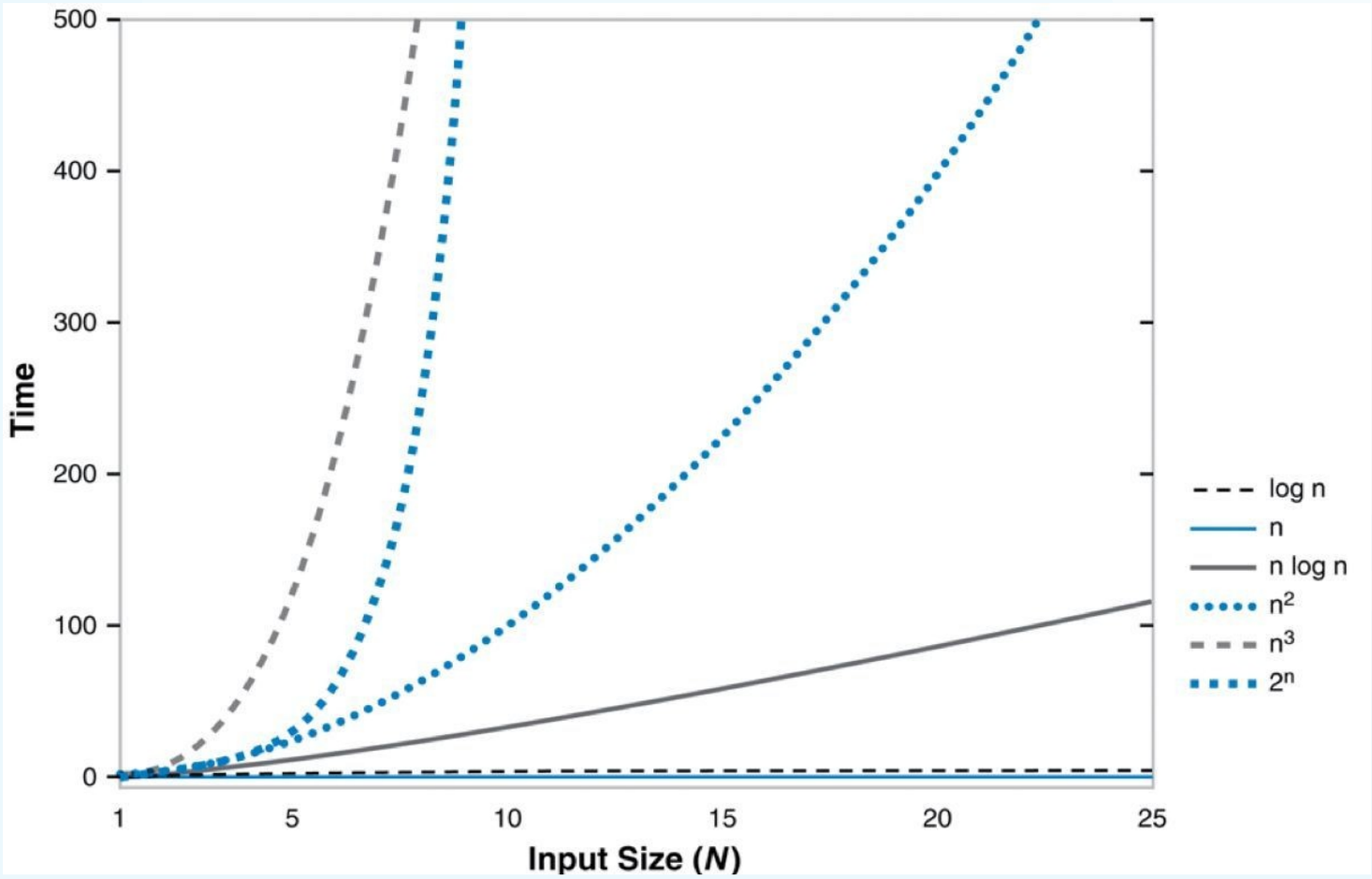
Number of dishes (n)	$15n^2$	$45n$	$15n^2 + 45n$
1	15	45	60
2	60	90	150
5	375	225	600
10	1,500	450	1,950
100	150,000	4,500	154,500
1,000	15,000,000	45,000	15,045,000
10,000	1,500,000,000	450,000	1,500,450,000
100,000	150,000,000,000	4,500,000	150,004,500,000
1,000,000	15,000,000,000,000	45,000,000	15,000,045,000,000
10,000,000	1,500,000,000,000,000	450,000,000	1,500,000,450,000,000

- Normalmente não é necessário saber exactamente qual é a função de crescimento
- O problema principal é a **complexidade assimptótica** da função - como cresce à medida que **n** aumenta
- Determinada pelo termo dominante na função de crescimento
- Referimo-nos a isto como ordem do algoritmo
- É normalmente usada a notação ***Big-Oh*** para especificar a ordem tal como **O(n^2)**

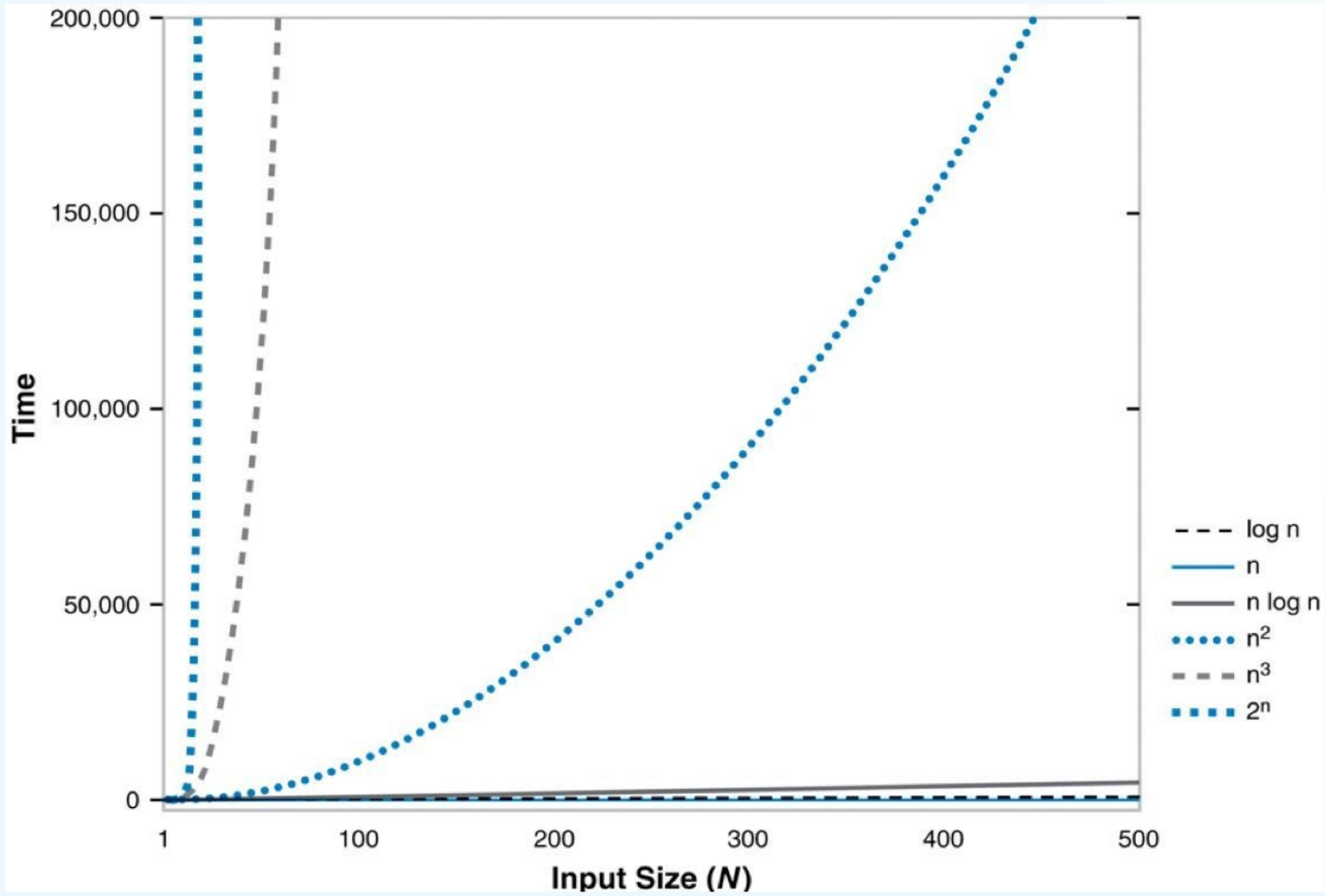
- Algumas funções de crescimento e a sua complexidade assimptótica

Função de Crescimento	Ordem	Descrição
f(n) = 17	O(1)	Constante
f(n) = 3log n	O(log n)	Logarítmica
f(n) = 20n - 4	O(n)	Linear
f(n) = 12n log n + 100n	O(n log n)	n log n
f(n) = $3n^2$ + 5n -2	O(n^2)	Quadrática
f(n) = $8n^3$ + $3n^2$	O(n^3)	Cúbica
f(n) = 2n + $18n^2$ + 3n	O(2^n)	Exponencial

- Comparação de funções de crescimento típicas para valores de **n** pequenos



- Comparação de funções de crescimento típicas para valores de **n** grandes



Análise da Execução de um Ciclo

- Um ciclo executa um certo número de vezes (digamos **n**)
- Assim, a complexidade de um ciclo é **n** vezes a complexidade do corpo do ciclo
- Quando os ciclos estão aninhados o corpo do ciclo exterior inclui a complexidade do ciclo aninhado (interior)

- O ciclo apresentado de seguida é **O(n)** devido ao ciclo executar **n** vezes o corpo do ciclo que é **O(1)**:

```
for (int i=0; i<n; i++){
  x = x + 1;
}
```

- O ciclo seguinte é **O(n²)**, porque o ciclo é executado **n** vezes e o corpo do ciclo, que inclui um ciclo aninhado, é **O(n)**:

```
for (int i=0; i<n; i++){
  x = x + 1;
  for (int j=0; j<n; j++){
    y = y - 1;
  }
}
```

Análise da Execução de um Ciclo

Para analisar as chamadas de métodos, nós simplesmente substituímos as chamada de métodos com a ordem do corpo do método

- Uma chamada para o método seguinte é **O(1)**

```
public void printsum(int count){
  sum = count*(count+1)/2;
  System.out.println(sum);
}
```