

SEMANAS:

- INICIO/APRESENTAÇÃO
- INTRODUÇÃO AOS ALGORITMOS E ESTRUTURAS DE DADOS
- ESTRUTURAS DE CONTROLO DE FLUXO
- CONCEITO DE CLASSE E OBJETO
- SEPARAÇÃO ENTRE INTERFACE PÚBLICA E IMPLEMENTAÇÃO PRIVADA
- ESTRUTURAS DE DADOS – INTRODUÇÃO
- ANÁLISE DE ALGORITMOS E COMPLEXIDADE
- ALGORITMOS DE ORDENAÇÃO
- ALGORITMOS DE ORDENAÇÃO (PARTE 2)
- PROVA DE AVALIAÇÃO
- LISTAS E FILAS
- TABELAS DE HASH
- TRATAMENTO DE COLISÕES
- ÁRVORES BINÁRIAS
- TIPOS GENÉRICOS E TRATAMENTO DE EXCEPÇÕES
- PROVA DE AVALIAÇÃO

Sumário:

- Introdução aos algoritmos e estruturas de dados: motivação.
- Modelo base de programação.
- Um primeiro exercício prático.
- Revisão/apresentação da sintaxe em C#
 - Tipos básicos em C#
 - Constantes
 - Leitura e escrita (em consola)

Algoritmos e Estruturas de Dados

CTeSP - Tecnologias e Programação de Sistemas de Informação 2020/21



Docente: [Ricardo Henriques](#).

Introdução

Desenvolver *software* para um computador pode ser um processo complicado. Ao longo das últimas décadas, os investigadores identificaram várias atividades distintas no âmbito do desenvolvimento de *software*:

- Definição do problema
- Identificação e especificação dos requisitos
- Planeamento
- Desenho alto nível; arquitetura do *software*
- **Desenho detalhado**
- **Codificação e depuração (*debug*)**
- **Testes unitários**
- Testes de integração
- Integração
- Testes funcionais
- Manutenção corretiva

A **negrito** podemos dizer que está o núcleo das atividades de construção, nas quais 70% ou mais do esforço de um projeto será investido.

O objetivo desta disciplina é estudar uma variedade de algoritmos úteis e bem conhecidos. *Um algoritmo não é mais do que um método (de cálculo) para resolver problemas, cujas soluções podem ser implementados num computador.*

Significa então que vamos, nesta disciplina, trabalhar essencialmente na área do desenho detalhado e codificação, sem esquecer as outras, nomeadamente, a compreensão do problema e dos seus requisitos.

O enorme número de áreas do conhecimento e algoritmos que existem não dará a oportunidade de os estudarmos a todos nesta cadeira. Contudo vamos investir o tempo necessário em cada algoritmo para compreender as características essenciais e respeitar as suas subtilezas.



Vamos também (re)ver algumas simples estruturas de dados, como os *arrays* (vetores), *queues* (filas) e *trees* (árvores).

Em algumas áreas de aplicação será necessário cálculo elementar. Não é necessário conhecimento prévio de álgebra linear, geometria e matemática discreta, embora o seu conhecimento prévio possa ajudar nalguns tópicos.

Algoritmos e estruturas de dados...

Os algoritmos "andam de mãos dadas" com as estruturas de dados - esquemas para organizar os dados de forma a deixá-los passíveis de processamento eficiente por um algoritmo.

Os algoritmos são métodos para resolver problemas:

- problemas têm dados;
- dados são tratados computacionalmente.

O processo de organização dos dados pode determinar a eficiência do algoritmo:

- algoritmos simples podem requerer estruturas de dados complexas;
- algoritmos complexos podem requerer estruturas de dados simples.

Para maximizar a eficiência de um algoritmo as estruturas de dados que são usadas têm de ser projetadas em simultâneo com o desenvolvimento do algoritmo. A definição das estruturas de dados (a forma como a informação se relaciona) impõe muitas vezes um padrão no algoritmo.

Porque se utilizam meios computacionais?... Com utilização de meios computacionais:

- deseja-se que a computação seja mais rápida;
- deseja-se ter a possibilidade de processar maior volume de dados;
- com o avanço tecnológico os computadores são mais rápidos e com maior capacidade, contudo:
 - a tecnologia apenas melhora o desempenho por um factor constante;
 - bons algoritmos podem fazer muito melhor;
 - um mau algoritmo num super-computador pode ter um pior desempenho que um bom algoritmo num numa calculadora de bolso ou telemóvel (não *smart*).

Utilização de algoritmos

- Para problemas de pequena complexidade/dimensão a escolha do algoritmo não se torna relevante:
 - desde que esteja correto, ou seja funcione de acordo com as especificações!
- Para problemas de grande complexidade/dimensão (volume de dados):
 - obviamente tem de estar correto;
 - é maior a motivação para otimizar o tempo de execução e o espaço (memória) utilizado;
 - o desafio é enorme:
 - em termos de melhoria de desempenho;
 - o poder fazer algo que seria impossível de outra forma.

Passos na resolução de um problema

1. Compreensão e definição do problema a resolver
 - completa e sem ambiguidades
 - que dados necessito de lidar
 - que informação necessito de emitir
 - que estruturas de dados podem ser úteis para lidar com os dados.



2. Gestão da complexidade

- decompor o problema em sub-problemas de menor dimensão/complexidade
- conhecer os algoritmos básicos e ver se podem ser usados na resolução do problema ou dos sub-problemas

Modelo base de programação

O nosso estudo de algoritmos tem por base a implementação dos mesmos em *programas* escritos em C#. As razões de o fazer prendem-se com:

- A necessidade dos programas serem concisos, elegantes e descreverem completamente os algoritmos;
- Poder correr (animar) o programa para estudar as propriedades do algoritmo;
- Poder usar o algoritmo em aplicações concretas.

Isto é uma vantagem importante em vez da alternativa de utilizar "*português estruturado*" para descrever algoritmos. Uma desvantagem é ter de trabalhar numa linguagem específica de programação, podendo dificultar a clarificação da ideia por detrás do algoritmo dos detalhes da sua implementação.

Por isso vamos tentar ao longo do curso:

- Utilizar construtores do C# que são comuns em muitas das modernas linguagens de programação;
- Adicionar comentários e descrições que traduzam o pensamento do algoritmo.

Exercício 1:

Suponha que deseja compor uma solução computacional capaz de ler o nome e a idade de uma série de pessoas e, no fim, deverá devolver a idade e o nome da pessoa mais velha.

Pense no papel! Implemente em C#.

Revisão/apresentação da sintaxe em C#

Passos necessários para instalação do Visual Studio, usando o *campus agreement*:

1. Aceda à página <https://azureforeducation.microsoft.com/devtools>
2. Autentique-se, utilizando as credenciais de estudante (a0xxxx@ipmaia.pt).
3. No menu, escolha: Products > Developer Tools > Visual Studio.
4. Siga as instruções e referências de informação da Microsoft.

Tipos básicos em C#

Toda a representação de dados em aparelhos de eletrónica digital (como os computadores) têm por base a sua representação em *bits* (0 ou 1), ao mais baixo nível. A mais pequena unidade de memória endereçável é normalmente um conjunto de *bits* designado por *byte* (um octeto de 8 *bits*). A unidade de processamento de uma instrução em código máquina é designado por *word*, tipicamente com 32 ou 64 *bits*.

Estes tipos de dados "máquina" necessitam de ser expostos ou disponibilizados nos sistemas e nas linguagens (mais baixo nível) de programação permitindo um nível mais refinado sobre o *hardware*. São desse modo expostas em tipos de dados ditos "primitivos" como em C.



Linguagens de programação de mais alto nível escondem ou abstraiem detalhes de programação como o número de *bits* para representar um inteiro; tal permite que o código seja mais portável. Em C# lista-se na tabela abaixo a representação dos tipos de dados "primitivos".

CTS Data Type	C# keyword	Descrição
System.Byte	byte	Define 8 bits, 0..255
System.Sbyte	sbyte	8 bits com sinal, -128..127
System.Int16	short	Inteiro com sinal de 16 bits, -32,768..32,767
System.Int32	int	Inteiro com sinal de 32 bits, -2147483648..2147483647
System.Int64	long	Inteiro com sinal de 64 bits, -9223372036854775808..9223372036854775807
System.UInt16	ushort	
System.UInt32	uint	
System.UInt64	ulong	
System.Single	float	Número real (vírgula flutuante) de 32 bits com 7 dígitos de precisão, $\pm 1.5 \times 10^{-45} \dots \pm 3.4 \times 10^{38}$
System.Double	double	Número em vírgula flutuante de 64 bits com 15-16 dígitos de precisão, $\pm 5 \times 10^{-324} \dots \pm 1.7 \times 10^{308}$ Nota: por omissão o valor atribuído a uma variável deste tipo é um <code>double</code> ; desejar que o número seja tratado como inteiro deverá acrescentar o sufixo D, como se denota: <code>double x = 3D</code>
System.Object	object	É a raiz da hierarquia de todas as classes
System.Char	char	Representa um carácter <i>unicode</i> de 16 bits <pre>char char1 = 'Z'; // Character literal char char2 = '\x0058'; // Hexadecimal char char3 = (char)88; // Cast from integral type char char4 = '\u0058'; // Unicode</pre>
System.String	string	Permite a representação de texto, o qual deve ser escrito entre aspas: <pre>"This is a string." "123" "The word ""hello"" is quoted."</pre>
System.Decimal	decimal	Denota um tipo de dados com 128 bits, cuja precisão pode ser de 28-29 dígitos, $\pm 1.0 \times 10^{-28} \dots \pm 7.9 \times 10^{28}$ Nota: se desejar que um numérico real seja tratado como <code>decimal</code> é necessário usar o sufixo M: <code>decimal myMoney = 300.5M;</code> ou caso contrário o compilador interpreta como <code>double</code> disparando um erro
System.Boolean	bool	Permite guardar os valores <code>true</code> e <code>false</code> . Nota: se necessitar que a variável possa também guardar o valor <code>null</code> deverá usar o tipo <code>bool?</code>

Constantes

Em C# há três tipos de constantes:

- **Literais** - um valor, ou constante literal. É constante porque é de facto impossível mudar-lhe o valor, por exemplo:

```
x = 32;
```

o valor 32 é um literal.

- **Constantes simbólicos** - são definidos com uma sintaxe própria e permitem atribuir um nome a um valor constante:

```
const type x = 3.1415;
```

- **Enumerações** - são uma forma muito útil de definir constantes com base num conjunto, ou lista de enumeração:

```
enum estadosCivis {
    sol = "solteiro/a",
    div = "divorciado/a",
    viu = "viuva/a",
    cas = "casado/a"
}
```

Leitura e escrita (em consola)

Por ser útil usar algumas primitivas de leitura de valores e escrita de valores em consola, para poder validar algoritmos e não só, deixa-se aqui uma referência ao modo de formatação e à tabela de significado do carácter de formatação.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace lerEscrever_1
{
    class Program
    {
        static void Main(string[] args)
        {
            double a = Convert.ToDouble(args[0]);
            double b = Convert.ToDouble(args[1]);

            Console.WriteLine("Soma real: {0} + {1} = {2}", a, b, a + b);
            Console.WriteLine("Soma aproximada: {0,5:f3} + {1,-20:f} = {2}", a, b, a + b);
        }
    }
}
```

Caracter de formato	Significado
C ou c	Utilizado para formatar unidade monetária
D ou d	Números decimais
E ou e	Notação exponencial
F ou f	Números de vírgula fixa
G ou g (general)	Pode ser usado para vírgula fixa ou exponencial
X ou x	Formatação hexadecimal

Main

Em C# não existem funções per si. Estas estão sempre associadas a uma classe (em sentido lato designadas como um tipo), ou a instâncias de um tipo, tal como iremos estudar. No caso do exemplo acima apenas se faz referência a *static members*, o que



significa que são usados os métodos (funcionalidade) associada ao tipo e não a uma sua instância.

O C# reconhece o método `Main` como sendo o ponto por omissão de início de execução do programa.

Exercício 2:

Estude a seguinte ligação <https://www.wikihow.com/Calculate-Horsepower> e implemente em C# (*console application*) um algoritmo parametrizado para calcular o *horse power* do próprio corpo humano.

Pense no papel! Implemente em C#.

MRH, 2020/21

