INTRODUÇÃO À ANÁLISE DE COMPLEXIDADE DE ALGORITMOS

PUC MINAS

ALGORITMOS E ESTRUTURAS DE DADOS II

ALGORITMO

- **Sequência de passos computacionais;**
  - que **transformam** uma **entrada em** uma **saída.**
- Sequência de **ações executáveis para** a **obtenção de** uma **solução;**
  - **para** um **determinado tipo de problema.**
- **Sequência** não-ambígua **de instruções;**
  - **executada até que** determinada **condição se verifique.**

COMPLEXIDADE DE ALGORITMOS

- Estudo da **complexidade de** um **algoritmo**:
  - **previsão dos recursos necessários:**
    - memória;
    - largura de banda de comunicação;
    - **tempo de execução.**

COMPLEXIDADE DE ALGORITMOS

- **Por que estudar complexidade de algoritmos?**
  - limitações de memória;
  - **limitações de processamento;**
  - **tempo de execução** dos algoritmos **tende a crescer;**
    - **à medida que** a **quantidade de dados de entrada cresce.**

COMPLEXIDADE DE ALGORITMOS

- **Como investigar o custo de um algoritmo?**

COMPLEXIDADE DE ALGORITMOS

- **Como investigar** o **custo de um algoritmo?**
  - **Execução do programa** em um computador real e **medição** de seu **tempo de execução.**
    - **Problemas**:
      - diferenças entre compiladores;
      - **dependência do *hardware***;
      - uso de memória virtual.

COMPLEXIDADE DE ALGORITMOS

- **Como investigar** o **custo de um algoritmo?**
  - Uso de um **modelo matemático**:
    - **Define-se** o **conjunto de operações** a ser executado;
      - e o **custo de cada uma** dessas operações.
    - Costuma-se levar em consideração **apenas** a(s) **operação(ções) mais significativas.**

FUNÇÃO DE COMPLEXIDADE

- Utilizaremos **complexidade de tempo.**
- Relaciona-se à **quantidade de vezes** que a **operação mais relevante é executada.**
- ***f*(*n*):**
  - **tempo** necessário para a **execução** de um **programa;**
    - **com entrada de dados de tamanho *n*.**

EXEMPLO 1

```
int max (int vetor[], int tamVetor){
    int i, temp;

    temp = vetor[0];
    for (i = 1; i < tamVetor; i++)
        if (temp < vetor[i])
            temp = vetor[i];
    return temp;
}
```

EXEMPLO 1

- Qual é a **operação mais relevante?**
- **Quanto tempo** se gasta **para executar a função;**
  - **em relação ao tamanho *n* do vetor?**

EXEMPLO 1

```
int max (int vetor[], int tamVetor){
    int i, temp;

    temp = vetor[0];
    for (i = 1; i < tamVetor; i++)
        if (temp < vetor[i])
            temp = vetor[i];
    return temp;
}
```

Uma comparação em cada iteração

EXEMPLO 1

```
int max (int vetor[], int tamVetor){
    int i, temp;

    temp = vetor[0];
    for (i = 1; i < tamVetor; i++)
        if (temp < vetor[i])
            temp = vetor[i];
    return temp;
}
```

Laço se repete (*n*-1) vezes

EXEMPLO 1

- Resposta:
  - $f(n) = n - 1$, para $n > 0$.

CASOS DE COMPLEXIDADE

- **Custo computacional depende** de:
  - **tamanho da entrada**;
  - **características** peculiares **da entrada**:
    - ordenação com dados já ordenados;
    - ordenação com dados em ordem decrescente;
    - ordenação com dados aleatórios.

CASOS DE COMPLEXIDADE

- Três **cenários:**
  - **melhor caso;**
    - **menor "tempo de execução"** para todas as entradas possíveis de tamanho *n*;
  - **pior caso;**
    - **maior "tempo de execução"** para todas as entradas possíveis de tamanho *n*;
  - **caso médio;**
    - **média dos tempos de execução** para todas as entradas possíveis.

CASOS DE COMPLEXIDADE

- Exemplo:
  - **pesquisa sequencial** em um **vetor** de **tamanho *n***:
    - melhor caso:
      - $f(n) = 1$;
    - pior caso:
      - $f(n) = n$;
    - caso médio:
      - $f(n) = (n + 1)/2$.

CASOS DE COMPLEXIDADE

- Interesse no **pior caso:**
  - estimativa do **tempo máximo** de **execução do algoritmo.**
  - A menos que dito o contrário.

EXEMPLO 2: MÁXIMO E MÍNIMO DE UM CONJUNTO

```
int* maxMin_1(int vetor[], int tamVetor){
    int i;
    int *temp = (int *) calloc(2, sizeof(int));

    temp[0] = vetor[0]; temp[1] = vetor[0];
    for(i = 1; i < tamVetor; i++){
        if(temp[0] < vetor[i]) temp[0] = vetor[i];
        if(temp[1] > vetor[i]) temp[1] = vetor[i];
    }
    return temp;
}
```

EXEMPLO 2: MÁXIMO E MÍNIMO DE UM CONJUNTO

- **Quanto tempo** se gasta para **executar** a **função;**
  - **em relação** ao **tamanho *n* do vetor?**

EXEMPLO 2: MÁXIMO E MÍNIMO DE UM CONJUNTO

- **Quanto tempo** se gasta para **executar** a **função;**
  - **em relação** ao **tamanho *n* do vetor?**
- Resposta:
  - $f(n) = 2(n - 1)$, para $n > 0$.

EXEMPLO 2: MELHORANDO...

```
int* maxMin_2(int vetor[], int tamVetor){
    int i;
    int *temp = (int *) calloc(2, sizeof(int));

    temp[0] = vetor[0]; temp[1] = vetor[0];
    for(i = 1; i < tamVetor; i++){
        if(temp[0] < vetor[i]) temp[0] = vetor[i];
        else if(temp[1] > vetor[i]) temp[1] = vetor[i];
    }
    return temp;
}
```

EXEMPLO 2: MELHORANDO...

- **Melhor caso:**
  - $\boldsymbol{f(n) = n - 1}$, para $n > 0$;
  - **vetor ordenado** em **ordem crescente.**
- **Pior caso:**
  - $\boldsymbol{f(n) = 2(n - 1)}$, para $n > 0$;
  - **vetor ordenado** em **ordem decrescente.**
- **Caso médio:**
  - $\boldsymbol{f(n)} = (n - 1) + (n - 1)/2 = \boldsymbol{(3n)/2 - 3/2}$, para $n > 0$;
  - temp[0] < vetor[i] **metade das vezes.**

EXEMPLO 2: OTIMIZANDO...

- **Alterando** a **abordagem:**
  - **Comparar** os **elementos do vetor aos pares;**
    - **separando-os em dois subconjuntos.**
  - **O máximo** é **obtido do subconjunto que contém os maiores elementos.**
  - **O mínimo** é **obtido do subconjunto que contém os menores elementos.**
- $f(n) = n/2 + n/2 - 1 + n/2 - 1$, para $n > 0$;
- $\boldsymbol{f(n) = (3n)/2 - 2}$, para $n > 0$.