Aula 0.1 - Notação Big-O

Estruturas de Dados 2018/1
Prof. Diego Furtado Silva

Complexidade de Algoritmos

Complexidade de Algoritmos

Um dos principais objetivos desta disciplina é aprendermos a utilizar **corretamente** as estruturas de dados estudadas em aplicações diversas

Corretamente, neste caso, significa **de modo eficiente**

Complexidade de Algoritmos

Complexidade de algoritmos é a ferramenta para estudar a eficiência de nossos algoritmos

Tempo e memória

Complexidade de Algoritmos

Complexidade de algoritmos é a ferramenta para estudar a eficiência de nossos algoritmos

Tempo e memória

Complexidade de Algoritmos

Esse tópico é abordado em detalhe a abrangência na disciplina de Projeto e Análise de Algoritmos

Por enquanto, vamos ficar com o básico

A notação *Big-O*

Notação *Big-O*

Diretamente ligada à noção de **escalabilidade**, ou seja, quanto a eficiência de seu algoritmo está associada a o tamanho de uma dada entrada

Big-O traz a noção de **proporcionalidade** entre o tempo de execução (ou uso de memória) e o tamanho da entrada

Notação *Big-O*

Diretamente ligada à noção de **escalabilidade**, ou seja, quanto a eficiência de seu algoritmo está associada a o tamanho de uma dada entrada

Big-O traz a noção de **proporcionalidade** entre o tempo de execução (ou uso de memória) e o tamanho da entrada

Pombo correio vs. internet

Imagine que você queira enviar dados a um colega

Então, você pensa em duas maneiras de fazer:

1) Mandar por email
2) Usar um pombo (provavelmente um bem fortão) com um HD amarrado na pata

Quem chega antes?

Pombo correio vs. internet

Transferência via pombo:

1 GB - 30 min

5 GB - 30 min

100 GB - 30 min

O tempo é **constante!**

Pombo correio vs. internet

Transferência via email:

1 GB - 30 min

5 GB - 150 min

100 GB - 3000 min = 30 * 100 min

O tempo é **linear em relação ao tamanho do arquivo** de dados,

ou seja, o tempo é **proporcional** ao número de GB

Pombo correio vs. internet

Ou seja, a partir de 1GB de dados, o pombo vence

Pombo correio vs. internet

Tempo (ou memória) constante

Se esse pombo existisse, poderíamos dizer que o tempo que ele leva um tempo **O(1)** para entregar os dados

O(1) - notação Big-O para tempo constante

Algorimicamente falando: casos em que o tamanho da entrada não afeta o tempo de execução

Tempo (ou memória) constante

Exemplo (pseudo-código)

```
Funcao calcula(n) {
    retorna n * Pi;
}
```

Crescimento linear

O tempo da entrega por email é proporcional ao tamanho do arquivo. Considerando *n* o tamanho do arquivo de dados

O(n) - notação Big-O para crescimento linear

Algorimicamente falando: quando o tempo de execução é linearmente proporcional ao tamanho da entrada

Crescimento linear

```
funcao encontra(x, vetor) {
    para cada i em vetor {
        se vetor[i] == x {
            retorna i
        }
    }
}
```

Crescimento linear

```
funcao encontra(x, vetor) {
    para cada i em vetor {
        se vetor[i] == x {
            retorna i
        }
    }
}
```

Neste caso, *n* é o tamanho do vetor

Pior caso

No exemplo anterior, o algoritmo poderia encontrar o elemento *x* logo na primeira posição do vetor

No entanto, dizemos que o algoritmo é O(n), pois seu tempo de execução é **proporcional a *n* no pior caso**, ou seja, quando não encontramos o elemento ou ele estiver na última posição do vetor

Crescimento polinomial

Crescimento polinomial é o nome dado a qualquer $O(n^k)$, com k constante

Ex:

$O(n^2)$ - quadrático
$O(n^3)$ - cúbico

...

Crescimento polinomial

```
funcao imprimePares(vetor) {
    para cada i em vetor {
        para cada j em vetor {
            imprime vetor[i] + ‘,’ + vetor[j]
        }
    }
}
```

Variável de entrada

Até agora usamos O(*n*) para linear e $O(n^2)$ para quadrático, mas *n* é só uma "convenção"

- No caso da busca no vetor, poderíamos ter utilizado O(|*vetor*|), por exemplo

Muitas vezes, o tempo é proporcional a variáveis específicas e muitas vezes a mais de uma variável

Variável de entrada

```
funcao imprimePares(vetor1, vetor2) {
    para cada i em vetor1 {
        para cada j em vetor2 {
            imprime vetor1[i] + ‘,’ + vetor2[j]
        }
    }
}
```

O(|vetor1|*|vetor2|) ou, simplificando, O(n*m)

Variável de entrada

```
funcao fazTudo(n) {
   funcao1(n) // O(a)
   funcao2(n) // O(b)
}
```

Variável de entrada

```
funcao fazTudo(n) {
    funcao1(n) // O(a)
    funcao2(n) // O(b)
}
```

O(a+b)

“Cortar” constantes

```
funcao minEMax(vetor) {
    meuMin = inf, meuMax = -inf
    para cada i em vetor {
        meuMin = min(meuMin, vetor[i])
    }
    para cada i em vetor {
        meuMax = max(meuMax, vetor[i])
    }
}
```

“Cortar” constantes

```
funcao minEMax(vetor) {
    meuMin = inf, meuMax = -inf
    para cada i em vetor {
        meuMin = min(meuMin, vetor[i])
    }
    para cada i em vetor {
        meuMax = max(meuMax, vetor[i])
    }
}
```

O(n)

O(n)

O(2*n)?

“Cortar” constantes

Nesse caso: O(|vetor|), pois

$O(2*n) = O(n)$

$O(n/2) = O(n)$

$O(n + 10) = O(n)$

“Cortar” constantes

```
funcao minEMax(vetor) {
    meuMin = inf, meuMax = -inf
    para cada i em vetor {
        meuMin = min(meuMin, vetor[i])
        meuMax = max(meuMax, vetor[i])
    }
}
```

O(n)

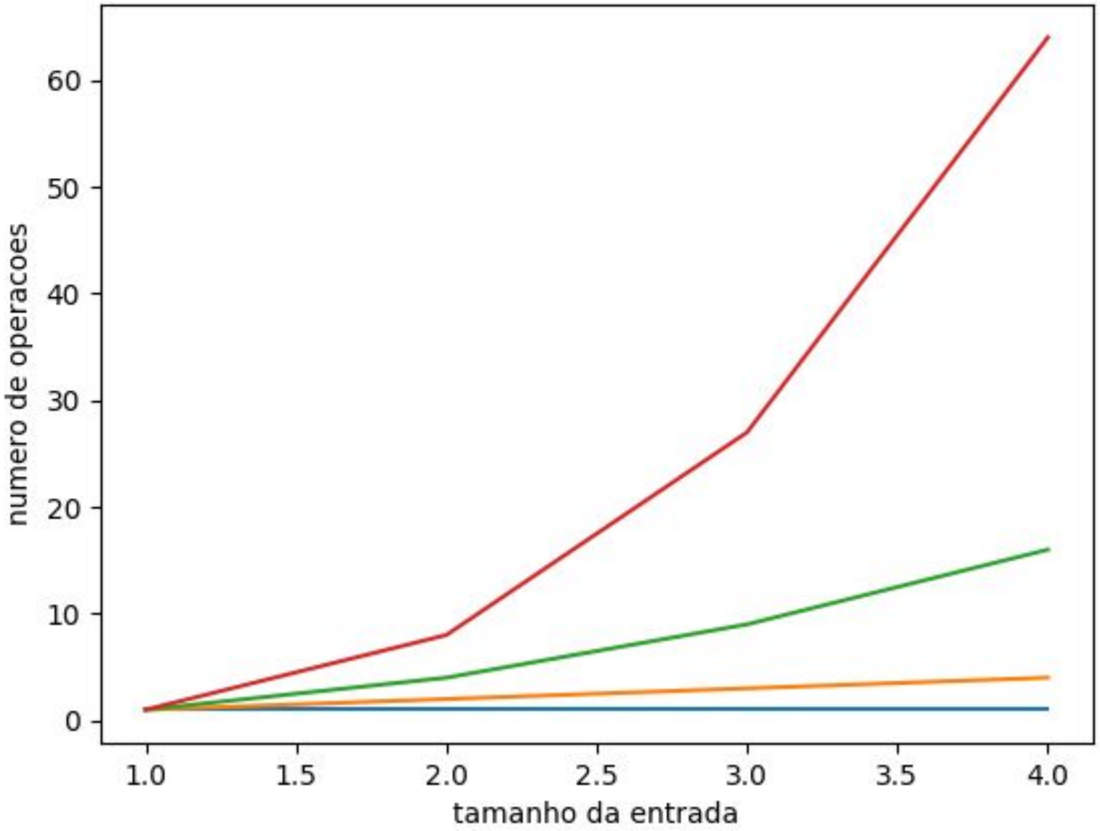
“Cortar” valores não-dominantes

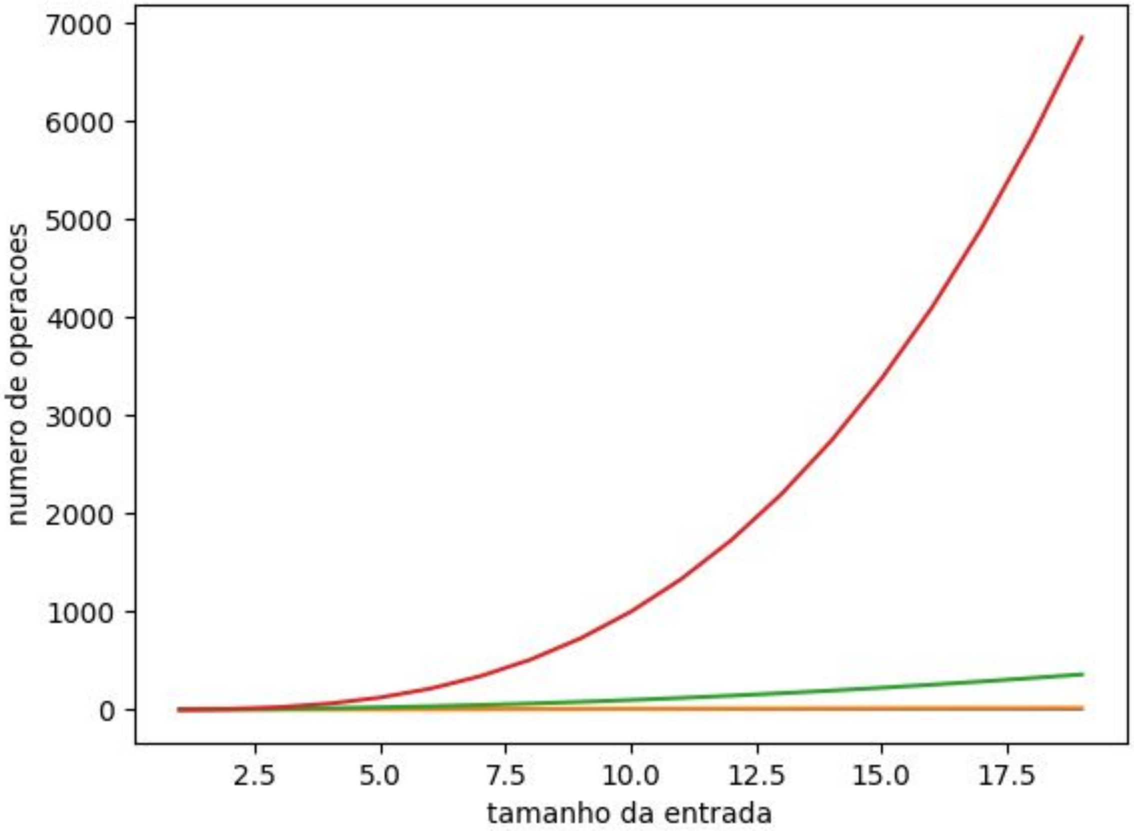
Exemplos

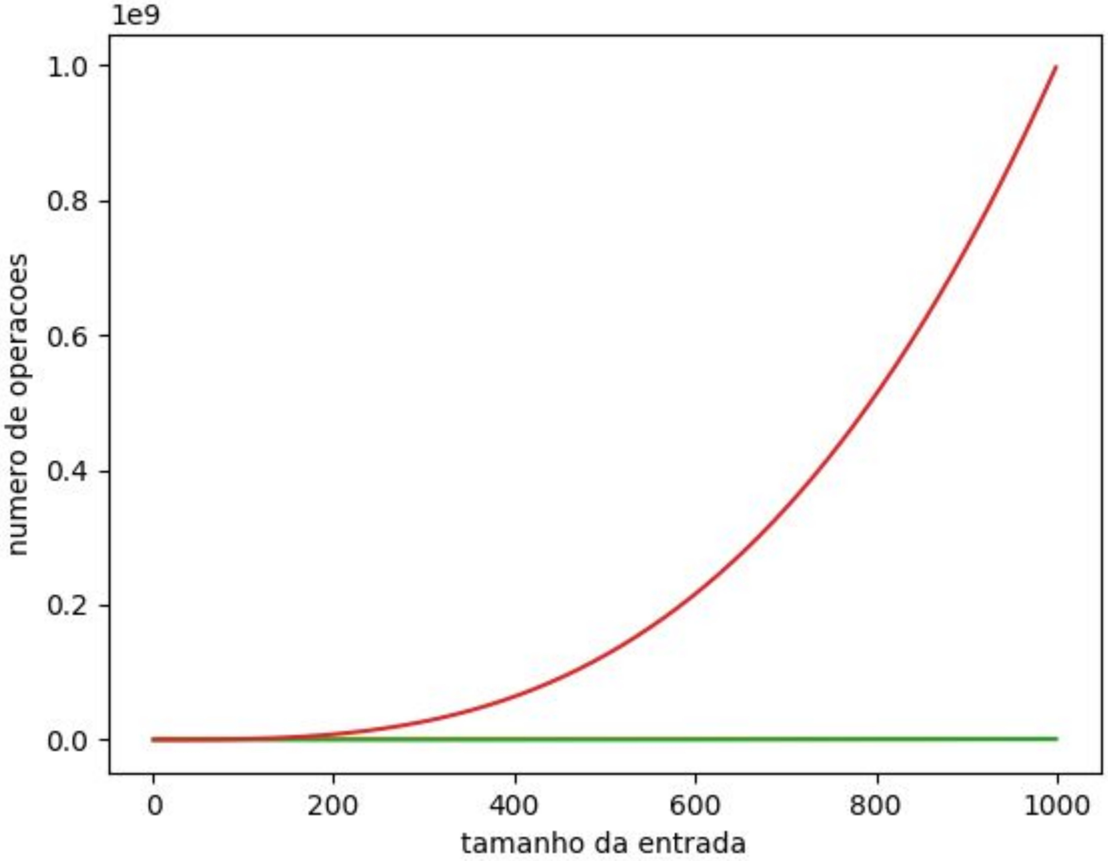
$O(n^2 + n) = O(n^2)$

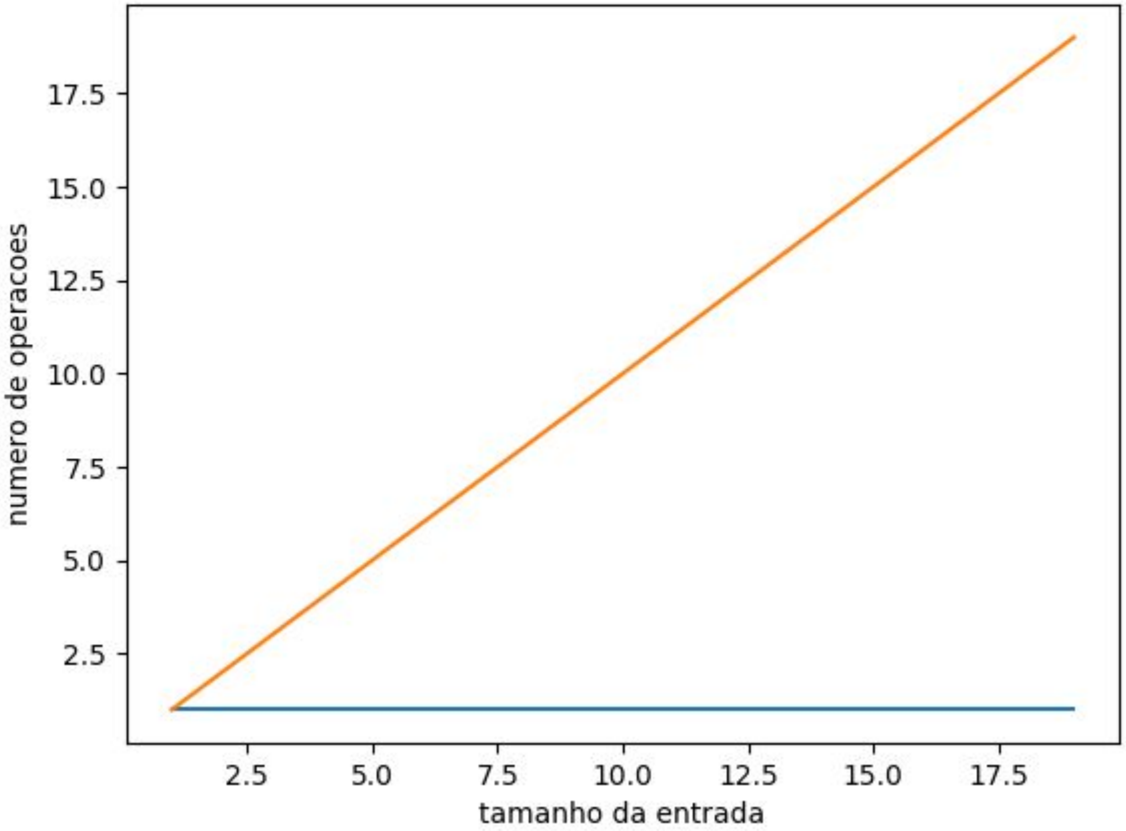
$O(n^3 + n^2) = O(n^3)$

$O(3*n^3 + n^2 + 5*n) = O(n^3)$


numero de operacoes
tamanho da entrada
0
10
20
30
40
50
60
1.0
1.5
2.0
2.5
3.0
3.5
4.0


7000
6000
5000
4000
3000
2000
1000
0
numero de operacoes
2.5
5.0
7.5
10.0
12.5
15.0
17.5
tamanho da entrada


1e9
1.0
0.8
0.6
0.4
0.2
0.0
numero de operacoes
0
200
400
600
800
1000
tamanho da entrada


17.5
15.0
12.5
10.0
7.5
5.0
2.5
numero de operacoes
2.5
5.0
7.5
10.0
12.5
15.0
17.5
tamanho da entrada

Tarefa para casa

Busca binária