Computabilidade e Complexidade Computacional

Teoria da computação

Prof. Allan Rodrigo Leite

Computabilidade e complexidade computacional

- Computabilidade
  - Verifica a existência de algoritmos que resolva uma classe de linguagens e trata a possibilidade da sua construção

- Complexidade
  - Trata sobre a eficiência da computação (algoritmos) em computadores existentes
    - Complexidade temporal: tempo de processamento requerido
    - Complexidade espacial: espaço de armazenamento requerido

Custo computacional

Computabilidade

- Objetivos
  - Quais problemas os computadores conseguem efetivamente resolver?
  - Como a resolução pode ser evidenciado?

- Concentra-se sobretudo em problemas com respostas binárias
  - Problemas de decisão (sim ou não, aceita ou rejeita, etc.)

Tipos de problemas

- Problemas computáveis
  - Problemas solucionáveis ou parcialmente solucionáveis
  - Existe um algoritmo descrito em uma Máquina Universal que solucione o problema aceitando ou rejeitando qualquer entrada

- Problemas não computáveis
  - Problemas completamente insolúveis
  - Não existe um algoritmo descrito em uma Máquina Universal capaz de solucionar o problema aceitando ou rejeitando qualquer entrada e sempre para
    - Determinar se dois programas são equivalentes
    - Determinar se uma gramática livre de contexto é ambígua
    - Determinar se duas gramáticas livres de contexto são equivalentes

Computabilidade

- Máquinas Universais são muito utilizadas em formalismos para verificação da computabilidade
  - Máquina Norma e Máquina de Turing
  - Tese de Church: “Qualquer computação que pode ser executada por meios mecânicos pode ser executada por uma Máquina de Turing”

- Qual a importância das Máquinas Universais no estudo da computabilidade?
  - Estas máquinas (que podem simular um computadores reais) não apresentam as limitações citadas anteriormente
  - Portanto, estas máquianas podem ser utilizadas para verificar o que um dispositivo de computação pode calcular em um determinado tempo
    - Considerando a execução em tempo finito (decidibilidade)

Problemas não solucionáveis

- Por que estuda-los?
  - Verificar as capacidades e limitações dos computadores
  - Evitar pesquisas por soluções inexistentes
  - Problemas que não podem ser resolvidos computacionalmente
  - Para verificar que outros problemas relacionados também são insolúveis

- Em outras palavras: o objetivo consiste em verificar o que um computador pode fazer

Problemas não solucionáveis

- Uma Máquina de Turing pode ser dividida em duas classes
  - Linguagens recursivas ou decidíveis
    - Para qualquer cadeia de entrada, a computação sempre termina
    - Sempre respondem se a cadeia de entrada pertence ou não à linguagem
  - Linguagens recursivamente enumeráveis
    - Para qualquer cadeia de entrada, a computação termina aceitando a cadeia, se for parte da linguagem
    - Mas também podem funcionar indefinidamente sobre entradas que elas não aceitam (loop infinito)
    - Não é possível determinar se a computação vai aceitar ou rejeitar a entrada

Problemas não solucionáveis

- Exemplos de problemas clássicos da computação
  - Detector universal de loops infinitos (problema da parada)
    - Definição: dado um programa qualquer e uma entrada qualquer, não existe um algoritmo genérico capaz de verificar se o programa vai parar ou não
  - Equivalência de compiladores
    - Definição: não existe algoritmo genérico que sempre para e que seja capaz de comparar quaisquer pares de compiladores de linguagens livre de contexto e indicar se são equivalentes

Problemas não solucionáveis

- Classifique o problema de acordo com a entrada (x)

```
Leia x.
Enquanto x != 10 faça
  x ← x + 1.
Imprime x.
Fim enquanto
```

Problemas não solucionáveis

- Classifique o problema de acordo com a entrada (x)

```
Leia x.
Enquanto x != 10 faça
  x ← x + 1.
Imprime x.
Fim enquanto
```

- Dependendo da entrada, o algoritmo:
  - Em algum momento terá a execução será finalizada ($x \leq 10$)
  - Ficará executando eternamente ($x > 10$)

Problemas parcialmente ou não solucionáveis

- Alguns problemas não solucionáveis são parcialmente solucionáveis
  - Existe um algoritmo capaz de aceitar a entrada
  - Eventualmente este algoritmo pode ficar em um loop infinito para uma resposta que deveria ser rejeita

  - A classe dos problemas parcialmente solucionáveis é equivalente à classe das linguagens recursivamente enumeráveis
    - Ou seja, os problemas parcialmente solucionáveis são computáveis

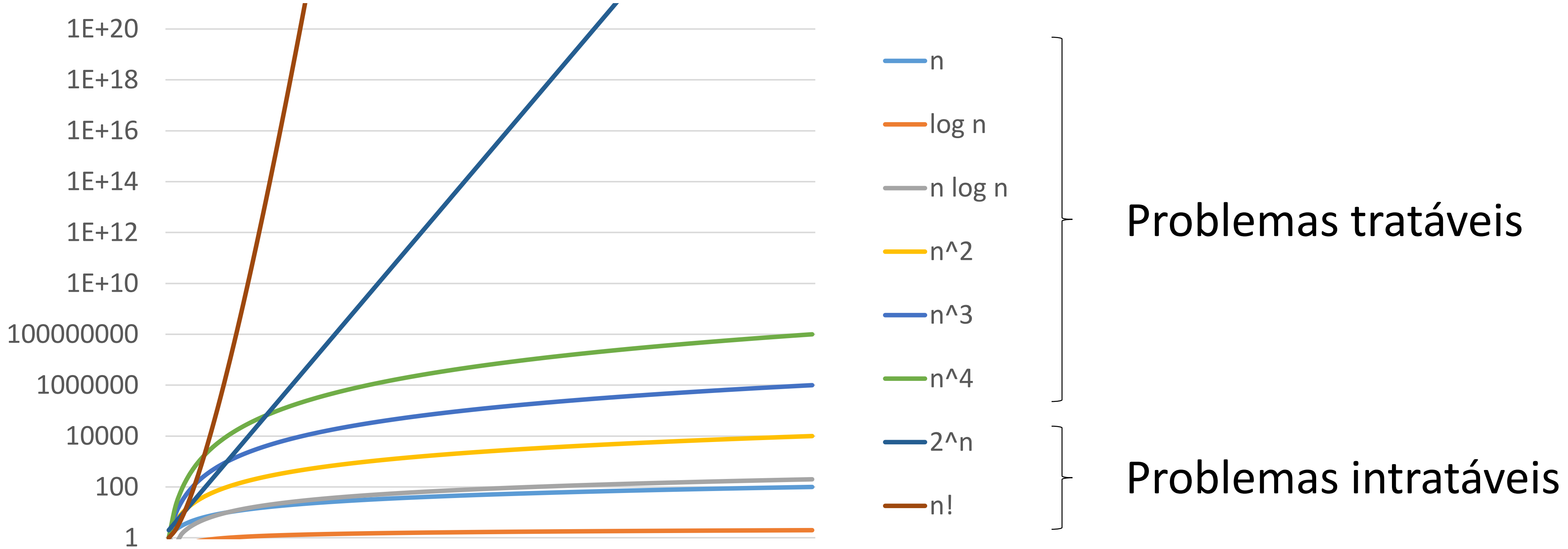
Complexidade computacional

- O conjunto das linguagens decidíveis pode ser dividida em classes de complexidade
  - Caracterizam os limites dos recursos computacionais usados para as decidir

- Uma classe de complexidade é especificada por um modelo de computação
  - Por exemplo, uma Máquina de Turing com computação determinística ou não determinística
  - As questões a serem consideradas estão relacionadas a recursos de tempo ou espaço

Universo de problemas

- Problemas tratáveis: apresentam solução polinomial
- Problemas intratáveis: nenhum algoritmo polinomial pode resolvê-lo



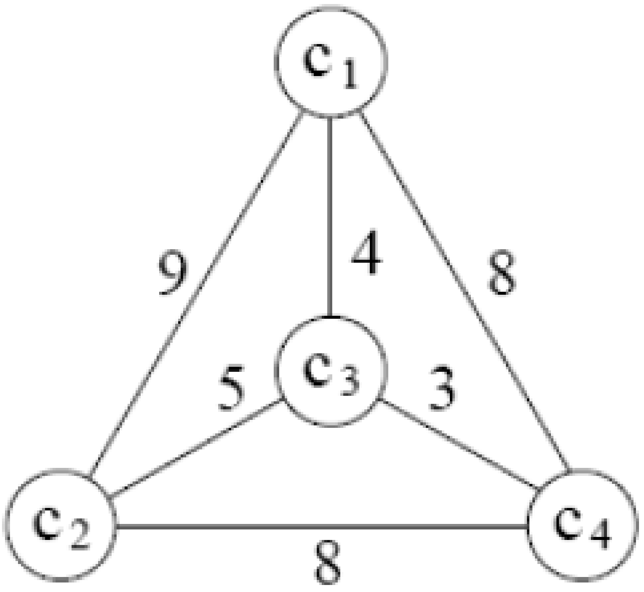
Complexidade computacional

- Por que analisar a complexidade de algoritmos?
  - Tempo de processamento e recursos são finitos

- Para resolver um problema, podem existir diferentes algoritmos capazes de encontrar a solução
  - Quais são mais eficientes em termos de
    - Tempo de processamento?
    - Espaço (memória requerida)?
  - Problemas complexos podem ser proibitivos se o algoritmo não for otimizado

Complexidade computacional

- Exemplo de problema: caixeiro viajante
  - O grafo ao lado representa um mapa com 4 cidades
  - A distância (km) entre as cidades é o peso das arestas



- Problema: encontrar um percurso que visite todas as cidades com a menor rota (km)
  - C1, C3, C4, C2, C1 → 24 km
- Um algoritmo simples utiliza força bruta para avaliar todas as rotas e escolher a menor delas
  - Neste exemplo, há (n – 1)! possibilidades
  - E se o número de cidades fossem 10?

Complexidade computacional

- Como definir a complexidade de um algoritmo?
  - Medir o tempo ou consumo de memória
    - Não é uma opção confiável
    - Depende do compilador em casos de otimização
    - Depende do hardware (CPU, SO, dispositivo, etc.)

  - Definição de complexidade a partir do estudo do número de vezes que operações são executadas

Complexidade computacional

- Exemplo: encontrar o maior valor de um vetor numérico

```
public int max(int v[]) {             // -
  int i;                              // -
  int max = v[0];                     // 1

  for (i = 1; i < v.length; i++) {    // n - 1
    if (v[i] > max) {                 // n - 1
      max = v[i];                     // A < n - 1
    }                                 // -
  }                                   // -

  return max;                         // -
}
```

Complexidade computacional

- Exemplo: encontrar o maior valor de um vetor numérico

```
public int max(int v[]) {             // -
  int i;                              // -
  int max = v[0];                     // 1

  for (i = 1; i < v.length; i++) {    // n - 1
    if (v[i] > max) {                 // n - 1
      max = v[i];                     // A < n - 1
    }                                 // -
  }                                   // -

  return max;                         // -
}
```

$\boxed{\text{Complexidade: } f(n) = n - 1}$

Análise de complexidade

- Complexidade é medida em função de *n*
  - *n* indica o tamanho da entrada para o algoritmo

- Exemplos:
  - Número de cidades de um mapa
  - Número de elementos de um vetor
  - Número de linhas e colunas de uma matriz

- Diferentes entradas podem ter esforços computacionais diferentes
  - Melhor caso, pior caso ou caso médio

Análise de complexidade

- Exemplo: encontrar o maior e menor valor de um vetor

```
public int[] minmax1(int v[]) {    // -
  int i;                           // -
  int min = v[0];                  // 1
  int max = v[0];                  // 1
  for (i = 1; i < v.length; i++) { // n - 1
    if (v[i] > max) {              // n - 1
      max = v[i];                  // A < n - 1
    }                              // -
    if (v[i] < min) {              // n - 1
      min = v[i];                  // B < n - 1
    }                              // -
  }                                // -
  return new int[]{ min, max };    // -
}                                  // -
```

Análise de complexidade

- Exemplo: encontrar o maior e menor valor de um vetor

```java
public int[] minmax1(int v[]) {    // -
  int i;                           // -
  int min = v[0];                  // 1
  int max = v[0];                  // 1
  for (i = 1; i < v.length; i++) { // n - 1
    if (v[i] > max) {              // n - 1
      max = v[i];                  // A < n - 1
    }                              // -
    if (v[i] < min) {              // n - 1
      min = v[i];                  // B < n - 1
    }                              // -
  }                                // -
  return new int[]{ min, max };    // -
}                                  // -
```

Pior caso (decrescente): $f(n) = 2(n – 1)$
Melhor caso (crescente): $f(n) = 2(n – 1)$
Caso médio : $f(n) = 2(n – 1)$

Análise de complexidade

- Exemplo: encontrar o maior e menor valor de um vetor

```
public int[] minmax2(int[] v) {     // -
  int i;                            // -
  int min = v[0];                   // 1
  int max = v[0];                   // 1
  for (i = 1; i < v.length; i++) {  // n - 1
    if (v[i] > max) {               // n - 1
      max = v[i];                   // A < n - 1
    } else {                        // -
      if (v[i] < min) {             // (n – 1) - A
        min = v[i];                 // B < (n – 1) - A
      }                             // -
    }                               // -
  }                                 // -
  return new int[] { min, max };    // -
}                                   // -
```

Análise de complexidade

- Exemplo: encontrar o maior e menor valor de um vetor

```
public int[] minmax2(int[] v) {     // -
  int i;                            // -
  int min = v[0];                   // 1
  int max = v[0];                   // 1
  for (i = 1; i < v.length; i++) {  // n - 1
    if (v[i] > max) {               // n - 1
      max = v[i];                   // A < n - 1
    } else {                        // -
      if (v[i] < min) {             // (n - 1) - A
        min = v[i];                 // B < (n - 1) - A
      }                             // -
    }                               // -
  }                                 // -
  return new int[] { min, max };    // -
}                                   // -
```

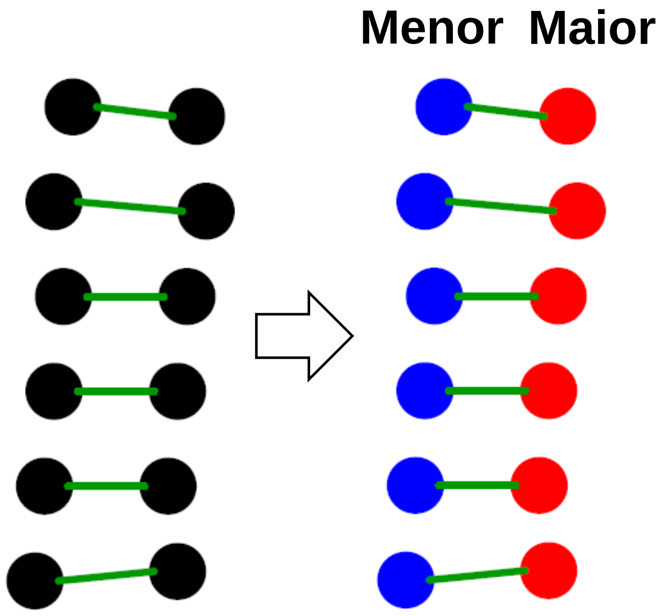
Pior caso (decrescente): $f(n) = 2(n - 1)$

Melhor caso (crescente): $f(n) = n - 1$

Caso médio: $f(n) = 3(n - 1) / 2$

Análise de complexidade

- E se comparássemos os elementos dois-a-dois?
  - Comparações: n / 2
  - Encontrar mínimo entre azuis: n / 2
  - Encontrar máximo entre vermelhos: n / 2



Análise de complexidade

- Exemplo: encontrar o maior e menor valor de um vetor

```
public int[] minmax3(int v[]) {          // -
  int i,a,b;                             // -
  int min = Integer.MAX_VALUE;           // 1
  int max = Integer.MIN_VALUE;           // 1
  for (i = 0; i < v.length; i += 2) {   // n / 2
    if (v[i] < v[i + 1]) {               // n / 2
      a = i; b = i + 1;                  // n / 4
    } else {                             // -
      a = i + 1; b = i;                  // n / 4
    }                                    // -
    if (v[a] < min)                      // n / 2
      min = v[a];                        // A < n / 2
    if (v[b] > max)                      // n / 2
      max = v[b];                        // A < n / 2
  }                                      // -
  return new int[] { min, max };         // -
}
```

Análise de complexidade

- Exemplo: encontrar o maior e menor valor de um vetor

```
public int[] minmax3(int v[]) {        // -
  int i,a,b;                           // -
  int min = Integer.MAX_VALUE;         // 1
  int max = Integer.MIN_VALUE;         // 1
  for (i = 0; i < v.length; i += 2) {  // n / 2
    if (v[i] < v[i + 1]) {             // n / 2
      a = i; b = i + 1;                // n / 4
    } else {                           // -
      a = i + 1; b = i;                // n / 4
    }                                  // -
    if (v[a] < min)                    // n / 2
      min = v[a];                      // A < n / 2
    if (v[b] > max)                    // n / 2
      max = v[b];                      // A < n / 2
  }                                    // -
  return new int[] { min, max };       // -
}
```

Pior caso (decrescente): $f(n) = 3(n / 2)$
Melhor caso (crescente): $f(n) = 3(n / 2)$
Caso médio: $f(n) = 3(n / 2)$

Análise de complexidade

- Resumindo

Algoritmo	Melhor caso	Pior caso	Caso médio
minmax1	2(n – 1)	2(n – 1)	2(n – 1)
minmax2	**n – 1**	2(n – 1)	3 (n – 1) / 2
minmax3	**3(n / 2)**	**3(n / 2)**	**3(n / 2)**

Comportamento assintótico

- Situações onde *n* é pequeno
  - Qualquer algoritmo, mesmo ineficiente, é factível

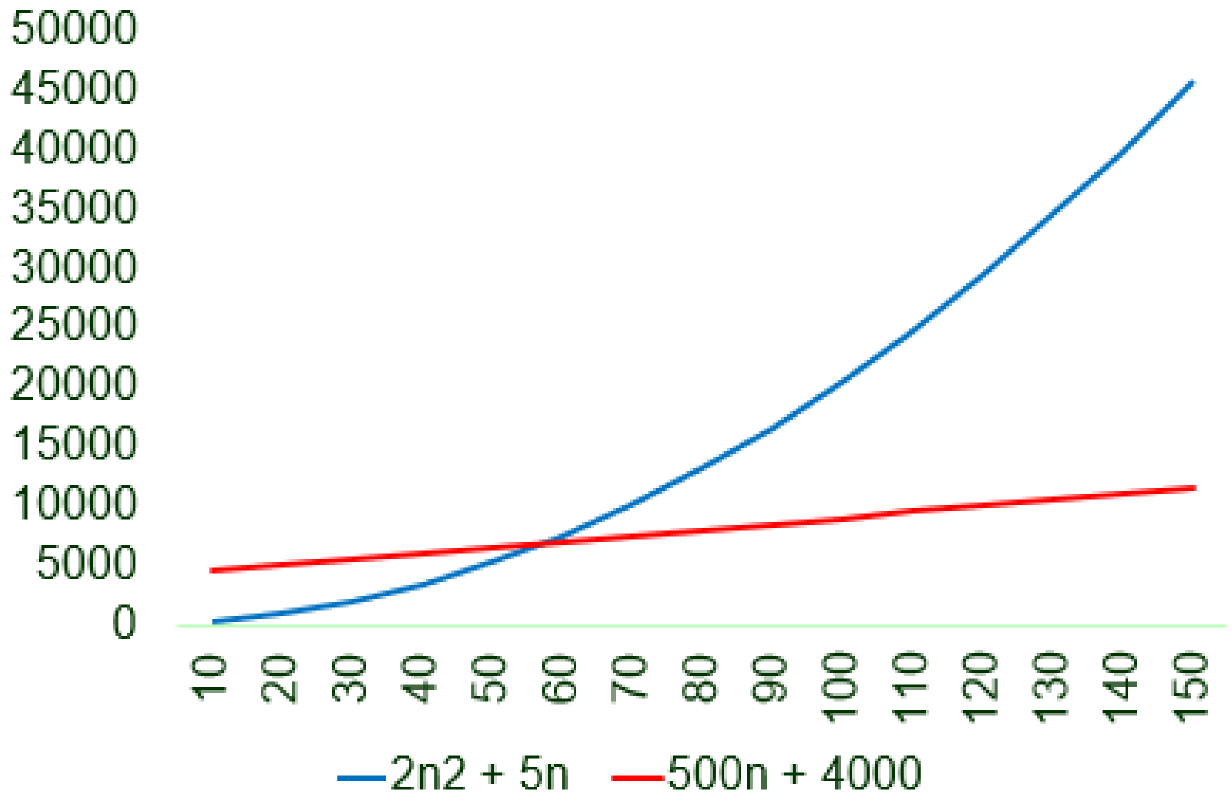
- Situações onde *n* é grande
  - Custo elevado ou computacionalmente inviável
  - Estudo do comportamento assintótico quando (n $\sim$ ∞)
  - Esforço cresce em grandes escalas
    - Quadrática, exponencial ou fatorial

Comportamento assintótico

- Um exemplo: considere o número de operações de cada um dos dois algoritmos abaixo que resolvem o mesmo problema:
  - Algoritmo 1: $f1(n) = 2n^2 + 5n$
  - Algoritmo 2: $f2(n) = 50n + 4000$

- Dependendo do valor de *n*, o algoritmo 1 pode requerer mais ou menos operações que o algoritmo 2.
  - Compare as duas funções para $n = 10$ e $n = 100$

Comportamento assintótico



Comportamento assintótico

- Um exemplo: considere o número de operações de cada um dos dois algoritmos abaixo que resolvem o mesmo problema:
  - Algoritmo 1: $f1(n) = 2n^2 + 5n$
  - Algoritmo 2: $f2(n) = 50n + 4000$

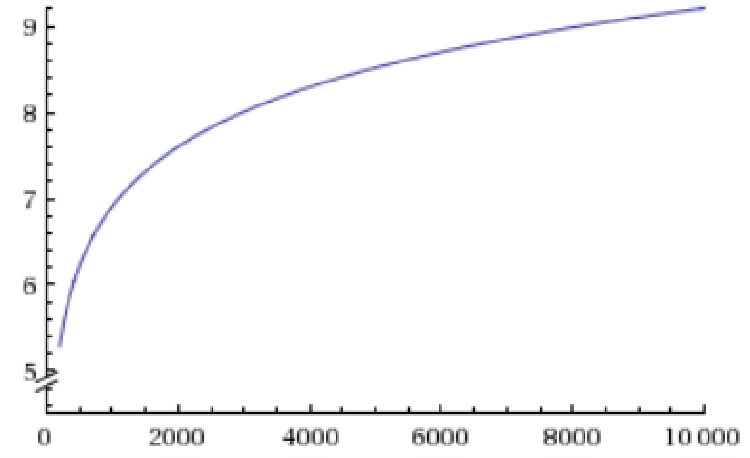
- Se desprezarmos os elementos de baixa ordem, ignoramos o coeficiente constante, teremos:
  - Algoritmo 1: $O(n^2)$
  - Algoritmo 2: $O(n)$

Comparação da complexidade

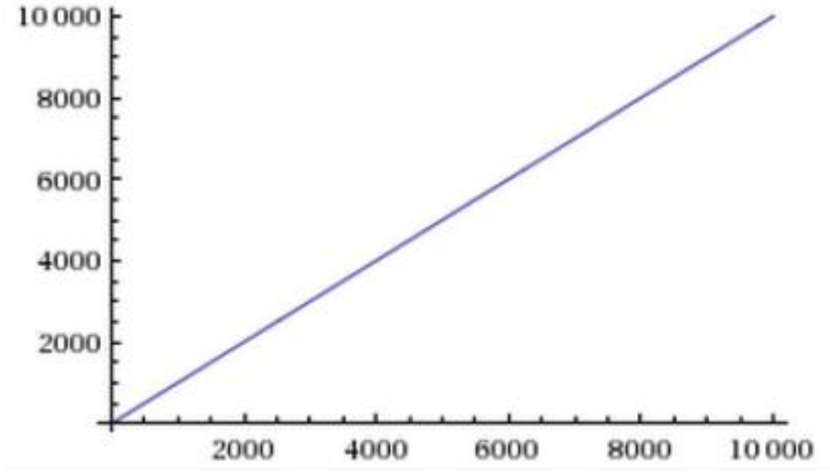
- O(1): não existe algoritmo mais rápido
  - Constante, independente de *n*
- O(log log *n*) ou O(log *n*): muito rápido
  - Custo não dobra até que n tenha sido aumentado para 2 x *n*
- O(*n*): rápido
  - Crescimento linear em função de *n*
- O(*n* log *n*): razoável, limiar de muitos problemas práticos
  - Quando *n* cresce próximo ao dobro, irá dobrar o custo também
- $O(n^2)$: quadrático, ruim
  - Quando *n* dobra, o custo aumenta 4 vezes, se for n^3, será 8 vezes
- $O(k^n)$, $O(n^n)$ ou $O(n!)$: exponencial
  - Pouco aplicável para muitos problemas práticos

Comparação da complexidade

O(log *n*)

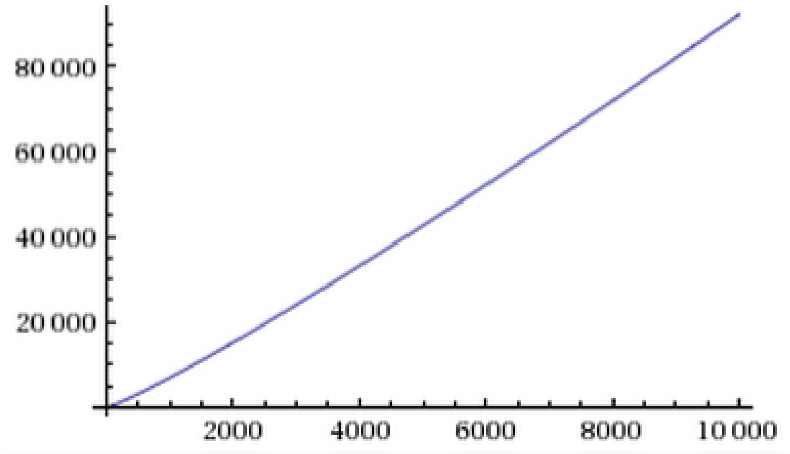


O(*n*)

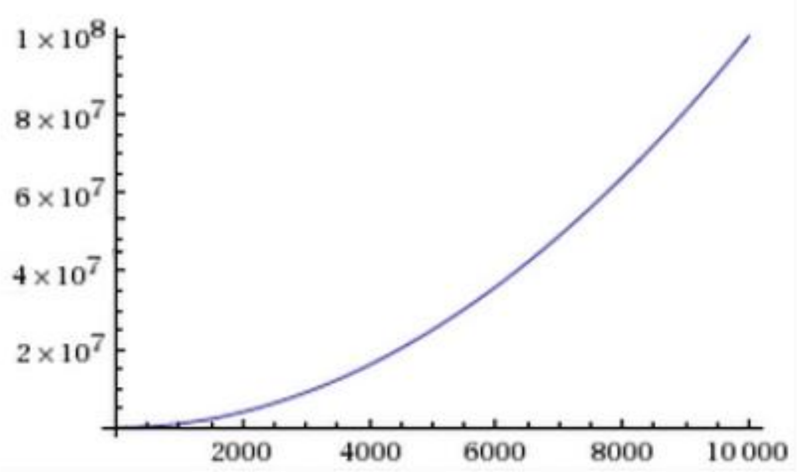


Comparação da complexidade

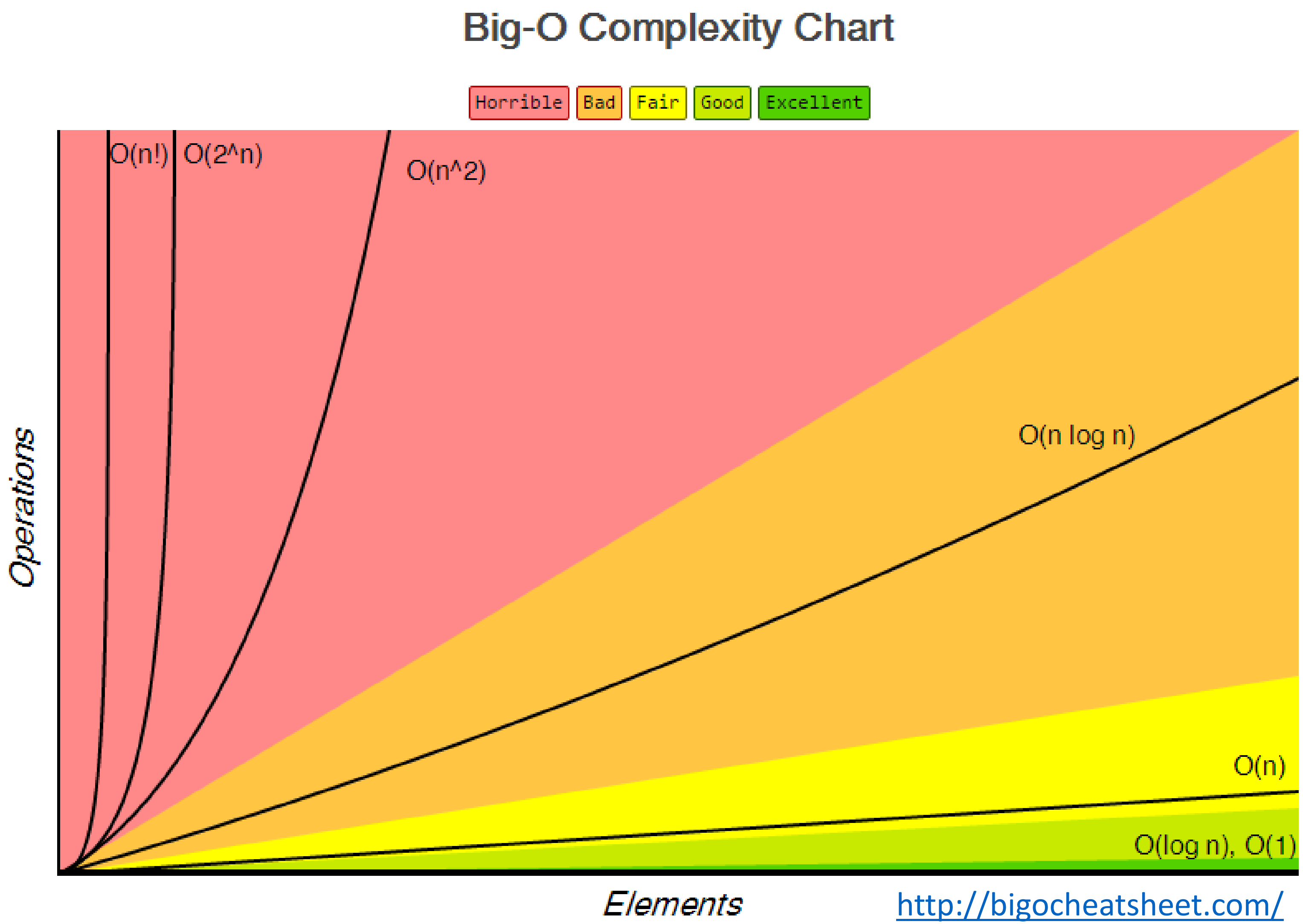
O($n \log n$)



O(n^2)



Comparação da complexidade



http://bigocheatsheet.com/