MAC122 - PRINCÍPIOS DE DESENVOLVIMENTO DE ALGORITMOS

Análise de algoritmos

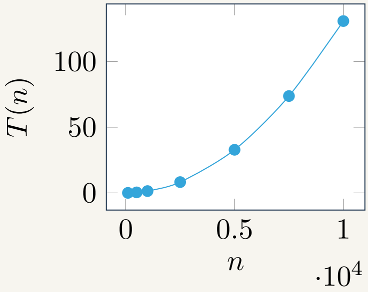
```
int main(int argc, char** argv)
{
  char c = 0;
  char* commands = "ads pq"; // key commands: "left,right,rotate,confirm,pause,quit"
  int speed = 2; // sets max moves per row
  int moves_to_go = 2;
  int full = 0; // whether board is full
  init(); // initialize board an tetrominoes


  cur =
  state[
  state[

  system
  printf

  while
    {
      dr

      // process user action
      c = getchar();   // get new action
      if (c == commands[0] && !intersect(cur, state[0]-1, state[1])) state[0]--; // move left
      if (c == commands[1] && !intersect(cur, state[0]+1, state[1])) state[0]++; // move right
      if (c == commands[2] && !intersect(cur->rotated, state[0], state[1])) cur = cur->rotated;
      if (c == commands[3]) moves_to_go=0;

      // scroll down
      if (!moves_to_go--)
        {
          if (intersect(cur,state[0],state[1]+1)) // if tetromino intersected with sth
            {
              cramp_tetromino();
              remove_complete_lines();
              cur = &tetrominoes[rand() % NUM_POSES];
              state[0] = (WIDTH - cur->width)/2;
```

- Como medir a eficiência de algoritmos?
  - independente de especificidades do sistema (SO, hardware, uso etc)
- Como comparar a eficiência de algoritmos?
  - independente de especificidades do sistema (SO, hardware, uso etc)

```c
/* Rearranja vetor v[0..n-1] para que seja crescente */
void Selecao (int v[], int n) {
  int i, j, k, x;
  for (i = 0; i < n; i++) {
    k = i; /* k = argmax v[i..n-1] */
     for (j = i + 1; j < n; j++)
        if (v[j] < v[k]) k = j;
    x = v[i]; v[i] = v[k]; v[k] = x;
  }
}
```

Geramos $N = 30$ vetores de inteiros arbitrários de tamanho n e medimos tempo de execução médio[1]

n	Tempo (ms)
100	0,018
500	0,395
1000	1,383
2500	8,223
5000	32,809
7500	73,718
10000	130,802



[1] Usando um iMac 1.6GHz Dual-Core i5

```
void Selecao (int v[], int n) {
  int i, j, k, x; /* k1 tempo */
  for (i = 0; i < n; i++) {
    k = i;
    for (j = i + 1; j < n; j++) /* k3 tempo por iteração */
      if (v[j] < v[k]) k = j;
    /* k2 tempo por iteração */
    x = v[i]; v[i] = v[k]; v[k] = x;
  }
}
```

k_1 alocação estática de inteiros

k_2 atribuição de inteiro

k_3 comparação de inteiros

$$k_1 + k_2 n + (k_2 + k_3)(n - 1 + n - 2 + \cdots + 2 + 1) = a + bn + cn^2$$

n	Tempo (ms)
1000	1,383
2500	8,223
5000	32,809

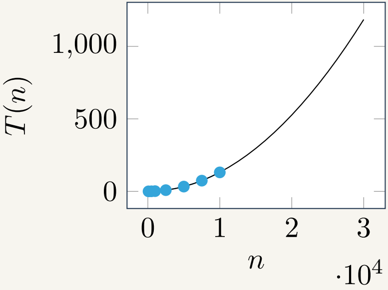
$$P(n) = a + bn + cn^2$$

$$\begin{aligned}
a+ \quad & 1000b+ \quad & 1000^2c = \quad & 1,383 \\
a+ \quad & 2500b+ \quad & 2500^2c = \quad & 8,223 \\
a+ \quad & 5000b+ \quad & 5000^2c = \quad & 32,809
\end{aligned}$$

$$\Rightarrow a = 0.195, b = -0.000055, c = 0.0000013$$

[2] Melhor seria estimar coeficientes do polinômio por mínimos quadrados

n	T(n)	P(n)
100	0,018	0,12
500	0,395	0,42
1000	1,383	1,383
2500	8,223	8,223
5000	32,809	32,809
7500	73,718	73,877
10000	130,802	131,42



- Modelo nos permite estimar tempo para instâncias de tamanhos não medidas

- Análise detalhada é difícil e tediosa (especialmente em algoritmos mais complexos)

- Análise detalhada é difícil e tediosa (especialmente em algoritmos mais complexos)
- Constantes são específicas de cada arquitetura (SO, hardware, uso etc)

- Análise detalhada é difícil e tediosa (especialmente em algoritmos mais complexos)
- Constantes são específicas de cada arquitetura (SO, hardware, uso etc)
- Resultado de medição não nos informa sobre fonte de ineficiência do algoritmo

- Análise detalhada é difícil e tediosa (especialmente em algoritmos mais complexos)
- Constantes são específicas de cada arquitetura (SO, hardware, uso etc)
- Resultado de medição não nos informa sobre fonte de ineficiência do algoritmo
- Ordenação por seleção é impraticável para n grande devido a sua complexidade de tempo quadrática (independente das constantes involvidas)

- Análise detalhada é difícil e tediosa (especialmente em algoritmos mais complexos)
- Constantes são específicas de cada arquitetura (SO, hardware, uso etc)
- Resultado de medição não nos informa sobre fonte de ineficiência do algoritmo
- Ordenação por seleção é impraticável para n grande devido a sua complexidade de tempo quadrática (independente das constantes involvidas)
  - Ao dobrar o tamanho da entrada o tempo de execução quadriplica: $T(n) \approx C \cdot n^2$

- Análise detalhada é difícil e tediosa (especialmente em algoritmos mais complexos)
- Constantes são específicas de cada arquitetura (SO, hardware, uso etc)
- Resultado de medição não nos informa sobre fonte de ineficiência do algoritmo
- Ordenação por seleção é impraticável para n grande devido a sua complexidade de tempo quadrática (independente das constantes involvidas)
  - Ao dobrar o tamanho da entrada o tempo de execução quadriplica: $T(n) \approx C \cdot n^2$
- Comportamento em instâncias pequenas não é muito relevante; devemos focar em instâncias grandes $n \to \infty$

PROBLEMA COMPUTACIONAL

Conjunto de instâncias de pares entrada-saída

- Problema da Ordenação: $x_1, \ldots, x_n \mapsto x_{\sigma(1)}, \ldots, x_{\sigma(n)}$ t.q. $x_{\sigma(1)} \leq \cdots \leq x_{\sigma(n)}$
- Problema do Máximo: $x_1, \ldots, x_n \mapsto \arg\max_i x_i$
- Problema do Teto do Logaritmo: $x \mapsto \lceil \lg x \rceil$

TAMANHO DA ENTRADA

Complexidade de algoritmo aumenta com a complexidade de entrada; uma propriedade simples relacionada à complexidade é seu tamanho grosseiramente definido como a quantidade de dados necessários para representar uma instância

- Problema da Ordenação: $x_1, \ldots, x_n \mapsto n$
- Problema do Teto do Logaritmo: $x \mapsto x$

COMPLEXIDADE DE TEMPO

Tempo de execução é comumente medido em relação ao tamanho da instância: $T(n)$

- tempo é médido em unidades básicas de operação (aritmética, atribuições, alocações de memória, etc)

COMPLEXIDADE DE ESPAÇO

Também é possível analizar um algoritmo pela quantidade de memória que ele usa em função do tamanho da entrada

- espaço é médido em unidades básicas de memória (número de números inteiros, doubles etc)

- Objetivo: Comparar eficiência de algoritmos em instâncias grandes, ignorando detalhes de implementação (hardware, SO etc)

- Objetivo: Comparar eficiência de algoritmos em instâncias grandes, ignorando detalhes de implementação (hardware, SO etc)

NOTAÇÃO O

Ordenações de funções por crescimento assintótico

- Avalia velocidade de crescimento em função do tamanho da entrada
- Ignora termos cuja contribuição seja insignificante para instâncias muito grandes $\mapsto$ ajuda a focar na fonte de ineficiência
- Ignora fatores constantes $\mapsto$ ignora implementação concreta (hardware, linguagem de programação etc)
- Representada pela notação $O(\cdot)$

$$O\left(a + bn + cn^2\right)$$

n	n^2	$n^2 + n$	$n^2/2$
10	100	110	50
100	10 000	10 100	5 000
1000	1 000 000	1 001 000	500 000
10000	100 000 000	100 010 000	50 000 000
100000	10 000 000 000	10 000 100 000	5 000 000 000

$$O\left(a + bn + cn^2\right)$$

n	n^2	$n^2 + n$	$n^2/2$
10	100	110	50
100	10 000	10 100	5 000
1000	1 000 000	1 001 000	500 000
10000	100 000 000	100 010 000	50 000 000
100000	10 000 000 000	10 000 100 000	5 000 000 000

- ignore fatores assintoticamente insignificantes

$$O\left(a + bn + cn^2\right) = O\left(cn^2\right)$$

n	n^2	$n^2 + n$	$n^2/2$
10	100	110	50
100	10 000	10 100	5 000
1000	1 000 000	1 001 000	500 000
10000	100 000 000	100 010 000	50 000 000
100000	10 000 000 000	10 000 100 000	5 000 000 000

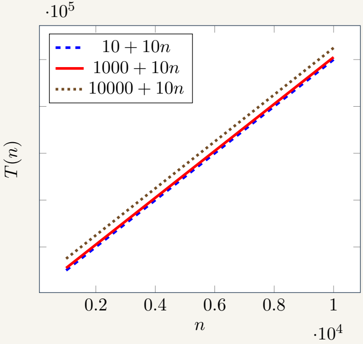
$$O\big(a + bn + cn^2\big) = O\big(cn^2\big)$$

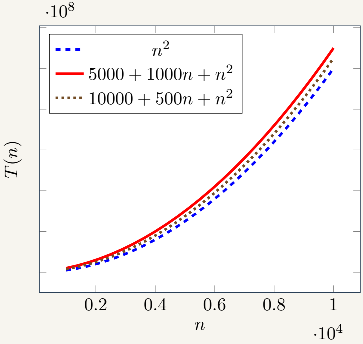
n	n^2	$n^2 + n$	$n^2/2$
10	100	110	50
100	10 000	10 100	5 000
1000	1 000 000	1 001 000	500 000
10000	100 000 000	100 010 000	50 000 000
100000	10 000 000 000	10 000 100 000	5 000 000 000

- ignore fatores constantes

$$O\left(a + bn + cn^2\right) = O\left(n^2\right)$$

n	n^2	$n^2 + n$	$n^2/2$
10	100	110	50
100	10 000	10 100	5 000
1000	1 000 000	1 001 000	500 000
10000	100 000 000	100 010 000	50 000 000
100000	10 000 000 000	10 000 100 000	5 000 000 000


$\cdot 10^5$
$10 + 10n$
$1000 + 10n$
$10000 + 10n$
$T(n)$
0.2
0.4
0.6
0.8
1
n
$\cdot 10^4$


$\cdot 10^8$
n^2
$5000 + 1000n + n^2$
$10000 + 500n + n^2$
$T(n)$
0.2
0.4
0.6
0.8
1
n
$\cdot 10^4$

Análise assintótica

```
void Selecao (int v[], int n) {
  int i, j, k, x;
  for (i = 0; i < n; i++) {
    for (k = i, j = i + 1; j < n; j++)
      ▶if (v[j] < v[k]) k = j;◀
    x = v[i]; v[i] = v[k]; v[k] = x;
  }
}
```

$$O\big(T(n)\big) = O\big(n^2\big)$$

- Captura essência da ineficiência do algoritmo
- Complexidade de tempo quadrática
- Mede número de comparações (▶ ◀)

```c
/* Retorna o maximo de v[0..n-1] */
int Max (int v[], int n) {
   int i, m;
  m = v[0];
   for (i = 1; i < n; i++)
       if (v[i] > m) m = v[i];
   return m;
}
```

$$O\big(T(n)\big) = ?$$

ANÁLISE ASSINTÓTICA

```c
/* Retorna o maximo de v[0..n-1] */
int Max (int v[], int n) {
   int i, m;
  m = v[0];
   for (i = 1; i < n; i++)
    ▶ if (v[i] > m) m = v[i]; ◀
   return m;
}
```

$$O\big(T(n)\big) = O\big(n\big)$$

- Complexidade de tempo linear
- Mede número de comparações (▶ ◀)

Complexidade assintótica

```
/* Calcula o qtd. de algarismos significativos do inteiro x
     */
int NumAlgarismos (int x) {
  int i;
  for (i = 0; x > 0; i++)
    x = x / 10;
  return i;
}
```

$$O\big(T(n)\big) = ?$$

```c
/* Calcula o qtd. de algarismos significativos do inteiro x
     */
int NumAlgarismos (int x) {
  int i;
  for (i = 0; x > 0; i++)
   ► x = x / 10; ◄
  return i;
}
```

$$O\big(T(n)\big) = O\big(\lceil \log_{10} x \rceil\big) = O\big(1 + \log_b x / \log_b 10\big) = O\big(\log x\big)$$

- Complexidade de tempo logarítmica
- Base do logaritmo não importa
- Mede número de divisões (► ◄)

```
/* Calcula no. de dígitos da representação binária de x */
int NumBits (int x) {
  int i;
  for (i = 0; x > 0; i++)
   ► x = x / 2; ◄
  return i;
}
```

$$O\big(T(n)\big) = ?$$

Complexidade assintótica

```
/* Calcula no. de dígitos da representação binária de x */
int NumBits (int x) {
  int i;
  for (i = 0; x > 0; i++)
   ► x = x / 2; ◄
  return i;
}
```

$$O\big(T(n)\big) = O\big(\log x\big)$$

- Complexidade de tempo logarítmica
- Mede número de divisões (► ◄)

```c
/* Retorna a area de um circula de raio r */
double AreaCirculo (double r) {
  return PI * r * r;
}
```

$$O\big(T(n)\big) = ?$$

```
/* Retorna a area de um circula de raio r */
double AreaCirculo (double r) {
  return PI * r * r;
}
```

$$O\big(T(n)\big) = O\big(1\big)$$

- Operações com aritmética de ponto flutuante levam tempo constante
- Complexidade de tempo constante
- Mede número de multiplicações

Complexidade assintótica

```c
/* Retorna posição i tal que v[i]=x ou -1 */
int BuscaLinear (int x, int v[], int n) {
  for (int i = 0; i < n; i++)
   ▶ if (v[i] == x) return i; ◀
  return -1;
}
```

$$O\big(T(n)\big) = ?$$

- Complexidade de tempo linear no pior caso
- Mede número de comparações (▶ ◀)

```
/* Retorna posição i tal que v[i]=x ou -1 */
int BuscaLinear (int x, int v[], int n) {
  for (int i = 0; i < n; i++)
   ▶ if (v[i] == x) return i; ◀
  return -1;
}
```

$$O\big(T(n)\big) = O\big(n\big)$$

- Complexidade de tempo linear no pior caso
- Mede número de comparações (▶ ◀)

Análise de pior caso

$$T_{\text{max}}(n) = \max_{\text{instância } I \text{ de tamanho } n} T(I)$$

Análise de melhor caso

$$T_{\text{min}}(n) = \min_{\text{instância } I \text{ de tamanho } n} T(I)$$

Análise de caso médio

$$T_{\text{média}}(n) = \sum_{\text{instância } I \text{ de tamanho } n} T(I) \Pr(I)$$

$\Pr(I)$ é a probabilidade de ocorrência da instância I

Complexidade de caso médio

```
int BuscaLinear (int x, int v[], int n) {
  for (int i = 0; i < n; i++)
    if (v[i] == x) return i;
  return -1;
}
```

Hipótese: Vetor contém única ocorrência de $\mathtt{x}$ e ocorre com probabilidade uniforme ($\Pr(I) = $ cte)

- $\therefore$ probabilidade de i-ésima posição ser igual a $\mathtt{x}$ é $1/n$

$$O\big(T_{\text{média}}(n)\big) =$$

```
int BuscaLinear (int x, int v[], int n) {
  for (int i = 0; i < n; i++)
    ▶ if (v[i] == x) return i; ◀
  return -1;
}
```

Hipótese: Vetor contém única ocorrência de $\mathtt{x}$ e ocorre com probabilidade uniforme ($\Pr(I) =$ cte)

- $\therefore$ probabilidade de i-ésima posição ser igual a $\mathtt{x}$ é $1/n$

$$O\big(T_{\text{média}}(n)\big) = O\left(\frac{1}{n}(1+2+\cdots+n)\right) =$$

Complexidade de caso médio

```
int BuscaLinear (int x, int v[], int n) {
  for (int i = 0; i < n; i++)
    ▶ if (v[i] == x) return i; ◀
  return -1;
}
```

Hipótese: Vetor contém única ocorrência de $\mathtt{x}$ e ocorre com probabilidade uniforme ($\Pr(I) =$ cte)

- ∴ probabilidade de i-ésima posição ser igual a $\mathtt{x}$ é $1/n$

$$O\big(T_{\text{média}}(n)\big) = O\left(\frac{1}{n}(1+2+\cdots+n)\right) = O\left(\frac{1}{n}\,\frac{n(n+1)}{2}\right) =$$

Complexidade de caso médio

```
int BuscaLinear (int x, int v[], int n) {
  for (int i = 0; i < n; i++)
    if (v[i] == x) return i;
  return -1;
}
```

Hipótese: Vetor contém única ocorrência de $\mathtt{x}$ e ocorre com probabilidade uniforme ($\Pr(I) =$ cte)

- $\therefore$ probabilidade de i-ésima posição ser igual a $\mathtt{x}$ é $1/n$

$$O(T_{\text{média}}(n)) = O\left(\frac{1}{n}(1+2+\cdots+n)\right) = O\left(\frac{1}{n}\frac{n(n+1)}{2}\right) = O(n)$$

Ordenação por inserção

```
void Insercao (int n, int v[]) {
  int i, j, x;
  for (j = 1; j < n; j++) {
    for (x = v[j], i = j-1; i >= 0 && v[i] > x; i--)
      v[i+1] = v[i];
    v[i+1] = x;
  }
}
```

Complexidade de pior caso?

```
void Insercao (int n, int v[]) {
  int i, j, x;
  for (j = 1; j < n; j++) {
    for (x = v[j], i = j-1; i >= 0 && ▶v[i] > x◀; i--)
      v[i+1] = v[i];
    v[i+1] = x;
  }
}
```

Complexidade de pior caso? $O\big(1 + 2 + \cdots + n - 1\big) = O\big(n^2\big)$

Ordenação por inserção

```c
void Insercao (int n, int v[]) {
  int i, j, x;
  for (j = 1; j < n; j++) {
    for (x = v[j], i = j-1; i >= 0 && v[i] > x; i--)
      v[i+1] = v[i];
    v[i+1] = x;
  }
}
```

Complexidade de caso médio?

```c
void Insercao (int n, int v[]) {
  int i, j, x;
  for (j = 1; j < n; j++) {
    for (x = v[j], i = j-1; i >= 0 && v[i] > x; i--)
      v[i+1] = v[i];
    v[i+1] = x;
  }
}
```

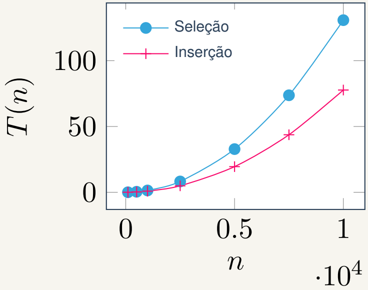
Complexidade de caso médio?

- Assuma que no laço interno todas sub-instâncias ocorrem com probabilidade uniforme

$$O(T(n)) = O\left(\sum_{i=1}^{n} \frac{1}{i}(1+2+\cdots+i)\right) = O\left(\frac{n^2}{4} + \frac{3}{4}n-\right) = O(n^2)$$

Geramos $N = 30$ vetores de inteiros arbitrários de tamanho n e medimos tempo de execução médio[3]

n	T_{Sel} (ms)	T_{Ins} (ms)
100	0,018	0,014
500	0,395	0,277
1000	1,383	0,847
2500	8,223	4,910
5000	32,809	19.49
7500	73,718	43,72
10000	130,802	77,70



[3] Usando um iMac 1.6GHz Dual-Core i5

Notação O

Dizemos que $g(n)$ é ordem $f(n)$ se existem constantes c e n_0 tais que

$$g(n) < c_0 f(n)\,, \qquad \text{para todo } n > n_0$$

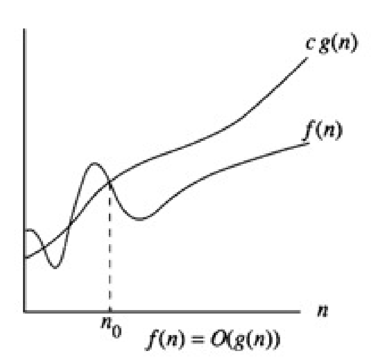
Dizemos que $g(n)$ é ordem $f(n)$ se existem constantes c e n_0 tais que

$$g(n) < c_0 f(n)\,, \qquad \text{para todo } n > n_0$$

Escrevemos isso como

$$g(n) \text{ é } O(f(n))$$

(lê-se $g(n)$ é ordem $f(n)$)



Alternativamente, podemos definir a ordem de uma função como uma classe de funções:

$$O(f(n)) = \left\{g(n) : \exists c_0, n_0 \text{ tais que } g(n) < c_0 f(n) \, \forall n > n_0\right\}$$

Nesse caso, escrevemos:

$$g(n) \in O(f(n))$$

Alternativamente, podemos definir a ordem de uma função como uma classe de funções:

$$O(f(n)) = \left\{g(n) : \exists c_0, n_0 \text{ tais que } g(n) < c_0 f(n)\, \forall n > n_0\right\}$$

Nesse caso, escrevemos:

$$g(n) \in O(f(n))$$

A ordem O estabelece apenas um limite superior, que pode ser bastante pessimista

- *Exemplo:* A função n^2 é $O(2^n)$ pois $n^2 \leq 2^n$ para todo $n \geq 4$ (ela também é $O(n^2)$)

1. $O(f(n)) + O(g(n)) = O(\max\{f(n), g(n)\})$

$$\begin{aligned} \subseteq : \quad & c_1 f(n) + c_2 g(n) < 2\max\{c_1, c_2\}\max\{f(n), g(n)\} \\ \supseteq : \quad & c_0 \max\{f(n), g(n)\} < c_0 f(n) + c_0 g(n) \end{aligned}$$

1. $O(f(n)) + O(g(n)) = O(\max\{f(n), g(n)\})$

$$\begin{aligned} \subseteq: &\quad c_1 f(n) + c_2 g(n) < 2\max\{c_1, c_2\}\max\{f(n), g(n)\} \\ \supseteq: &\quad c_0 \max\{f(n), g(n)\} < c_0 f(n) + c_0 g(n) \end{aligned}$$

2. $O(1 + n + n^2 + \cdots + n^d) = O(n^d)$

1. $O(f(n)) + O(g(n)) = O(\max\{f(n), g(n)\})$

$$\begin{aligned} \subseteq : \quad & c_1 f(n) + c_2 g(n) < 2\max\{c_1, c_2\}\max\{f(n), g(n)\} \\ \supseteq : \quad & c_0 \max\{f(n), g(n)\} < c_0 f(n) + c_0 g(n) \end{aligned}$$

2. $O(1 + n + n^2 + \cdots + n^d) = O(n^d)$

3. $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$

$$\begin{aligned} \subseteq : \quad & c_1 f(n) \cdot c_2 g(n) < c_1 c_2 f(n) \cdot g(n) \\ \supseteq : \quad & c_0 f(n) \cdot g(n) < \sqrt{c_0} f(n) \cdot \sqrt{c_0} g(n) \end{aligned}$$

Propriedades

4. $O(n) \cdot O(m) = O(n \cdot m)$

4. $O(n) \cdot O(m) = O(n \cdot m)$

5. $O(k \cdot f(n)) = O(f(n))$, $k \neq 0$

$$\begin{aligned} \subseteq : \quad & k \cdot f(n) < c_0 f(n)\,, c_0 > k \\ \supseteq : \quad & c_0 f(n) < c_0 \cdot k \cdot f(n) \end{aligned}$$

4. $O(n) \cdot O(m) = O(n \cdot m)$

5. $O(k \cdot f(n)) = O(f(n))$, $k \neq 0$

$$\begin{aligned} \subseteq: \quad & k \cdot f(n) < c_0 f(n)\,, c_0 > k \\ \supseteq: \quad & c_0 f(n) < c_0 \cdot k \cdot f(n) \end{aligned}$$

6. $O(1) = O(2) = O(c)$

4. $O(n) \cdot O(m) = O(n \cdot m)$

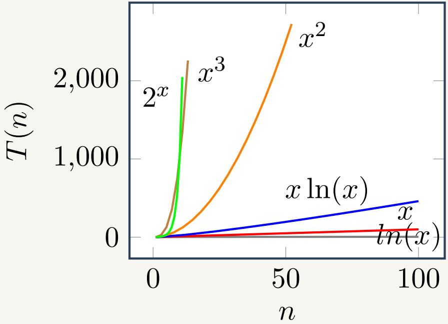
5. $O(k \cdot f(n)) = O(f(n))$, $k \neq 0$

$$\begin{aligned} \subseteq : \quad & k \cdot f(n) < c_0 f(n)\,, c_0 > k \\ \supseteq : \quad & c_0 f(n) < c_0 \cdot k \cdot f(n) \end{aligned}$$

6. $O(1) = O(2) = O(c)$

7. $O(a_0 + a_1 \cdot n + a_2 \cdot n^2 + \cdots + a_d n^d) = O(n^d)$

Classe	Ordem
constante	$O(1)$
logarítmica	$O(\log n)$
linear	$O(n)$
linearítmico	$O(n \log n)$
quadrática	$O(n^2)$
cúbico	$O(n^3)$
polinomial	$O(n^k)$
exponencial	$O(c^n)$



Resumo

- Eficiência é melhor analisada em função do tamanho da instância (N) e seu comportamento assintótico (N grande)
- Complexidade de pior caso e caso médio podem diferir (ou não)
- Notação O nos permite focar em partes cruciais de ineficiência e analisar qualitativamente eficiência algorítimica
- Análise assintótica é útil para prever tempo de execução em instâncias grandes