# Análise de Complexidade de Algoritmos

**Algoritmos e Estruturas de Dados**

2020/2021

FEUP

# Introdução

- Algoritmo: conjunto claramente especificado de instruções a seguir para resolver um problema

- Análise de algoritmos:
  - provar que um algoritmo está correto
  - determinar recursos exigidos por um algoritmo (tempo, espaço)
    - comparar os recursos exigidos por diferentes algoritmos que resolvem o mesmo problema (um algoritmo mais eficiente exige menos recursos para resolver o mesmo problema)
    - prever o crescimento dos recursos exigidos por um algoritmo à medida que o tamanho dos dados de entrada cresce

# Complexidade espacial e temporal

- Complexidade espacial de um programa ou algoritmo:

  espaço de memória que necessita para executar até ao fim

  *S(n)* - espaço de memória exigido em função do tamanho (n) da entrada

- Complexidade temporal de um programa ou algoritmo:

  tempo que demora a executar (tempo de execução)

  *T(n)* - tempo de execução em função do tamanho (n) da entrada

- Complexidade ↑ versus Eficiência ↓

- Por vezes estima-se a complexidade para o *"melhor caso"* (pouco útil), o *"pior caso"* (mais útil) e o *"caso médio"* (igualmente útil)

# Crescimento de funções

- Na prática, é difícil (senão impossível) prever com rigor o tempo de execução de um algoritmo ou programa
  - Obter o tempo a menos de:
    - constantes multiplicativas (normalmente estas constantes são tempos de execução de operações atómicas)
    - parcelas menos significativas para valores elevados de *n*

- Comparar crescimento
  - Comparação de funções em pontos particulares: muito dependente dos coeficientes
  - Comparação relevante: <u>taxas de crescimento</u>

- Avaliar taxa de crescimento
  - Em função com vários termos, crescimento é determinado pelo termo de crescimento mais rápido
  - Coeficientes constantes influenciam o andamento inicial

# Notação $O(\bullet)$

- Definição:

  $T(n) = O(f(n))$ ( ler: $T(n)$ é de ordem $f(n)$ )

  se e só se existem constantes positivas $c$ e $n_0$ tal que $T(n) \leq cf(n)$ para todo o $n > n_0$

- Exemplos:
  - $c_k n^k + c_{k-1} n^{k-1} + \ldots + c_0 = O(n^k)$ ($c_i$ - constantes)
  - $\log_2 n = O(\log n)$

    (não se indica a base porque mudar de base é multiplicar por constante)
  - $4 = O(1)$ (usa-se 1 para ordem constante)

# Notação $O(\bullet)$

- Notação para o crescimento relativo de funções
  - $T(n) = O(\ f(n)\ )$

    se existem constantes c e $n_0$ tais que $T(n) \leq c\ f(n)$ para $n \geq n_0$
  - $T(n) = \Omega(\ f(n)\ )$

    se existem constantes c e $n_0$ tais que $T(n) \geq c\ f(n)$ para $n \geq n_0$
  - $T(n) = \Theta(\ f(n)\ )$

    se e só se $T(n) = O(\ f(n)\ )$ e $T(n) = \Omega(\ f(n)\ )$
  - $T(n) = o(\ f(n)\ )$

    se existem constantes c e $n_0$ tais que $T(n) < c\ f(n)$ para $n \geq n_0$

# Ordens mais comuns

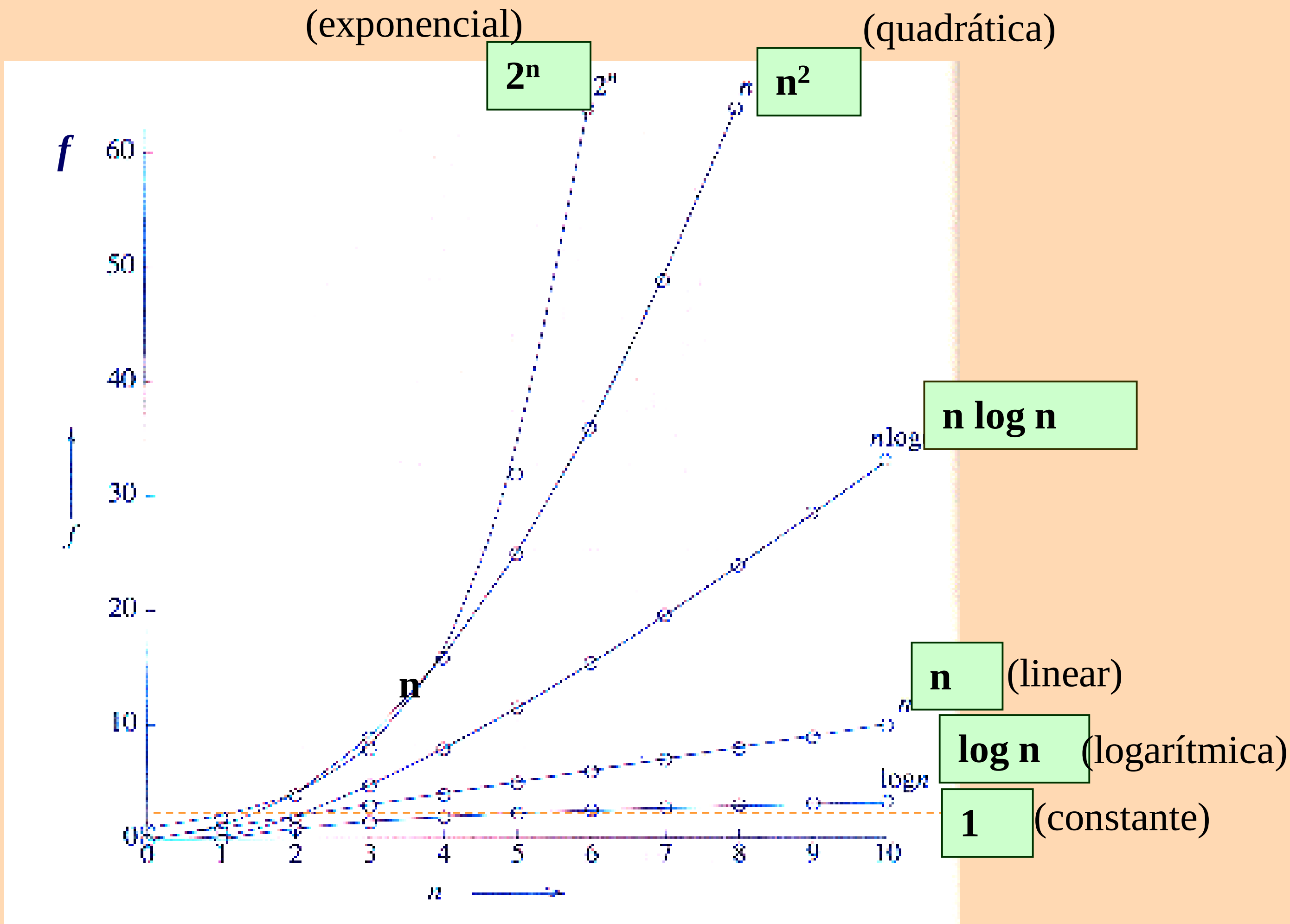


Figure 2.24 Plot of various functions

Fonte: Sahni, "Data Structures, Algorithms and Applications in C++"

# Termo Dominante

- Suponha que se usa $N^3$ para estimar $N^3+350N^2 + N$
- Para $N = 10\ 000$
  - valor real = 1 0003 5000 010 000
  - valor estimado = 1 000 000 000 000
  - erro = 0.35% (não é significativo)
- Para valores elevados de $N$
  - o termo dominante é indicativo do comportamento do algoritmo
- Para valores pequenos de $N$
  - o termo dominante não é necessariamente indicativo do comportamento, mas geralmente programas executam tão rapidamente que não importa

# Estudo de um caso : subsequência máxima

- Problema:
  – Dado um conjunto de valores (positivos e/ou negativos) $A_1, A_2, \ldots, A_n$, determinar a subsequência de maior soma
- A subsequência de maior soma é zero se todos os valores são negativos

- Exemplos:

  -2, <u>11, -4, 13</u>, -4, 2

  1, -3, <u>4, -2, -1, 6</u>

# Subsequência máxima - cúbico

```
template <class Comparable>
Comparable maxSubSum1(const vector<Comparable> &a)
{
    Comparable maxSum = 0;
    for (int i = 0 ; i < a.size() ; i++)
       for (int j = i; j < a.size(); j++)
       {
          Comparable thisSum = 0;
          for (int k = i; k <= j; k++)
             thisSum += a[k];
          if (thisSum > maxSum)
             maxSum = thisSum;
       }
     return maxSum;
}
```

# Subsequência máxima - cúbico

- Análise
  - Ciclo de *N* iterações no interior de um outro ciclo de *N* iterações no interior de um outro ciclo de *N* iterações $\Rightarrow$ $\boldsymbol{O(N^3)}$ , algoritmo cúbico
  - Valor estimado por excesso, pois alguns ciclos possuem menos de *N* iterações

- Como melhorar
  - Remover um ciclo
  - Ciclo mais interior não é necessário
  - *thisSum* para próximo *j* pode ser calculado facilmente a partir do antigo valor de *thisSum*

# Subsequência máxima - quadrático

```
template <class Comparable>
Comparable maxSubSum2(const vector<Comparable> &a)
{
    Comparable maxSum = 0;
    for (int i = 0 ; i < a.size() ; i++)
     {
       Comparable thisSum = 0;
        for (int j = i; j < a.size(); j++)
        {
          thisSum += a[j];
           if (thisSum > maxSum)
              maxSum = thisSum;
        }
     }
     return maxSum;
}
```

# Subsequência máxima - quadrático

- Análise
  - Ciclo de *N* iterações no interior de um outro ciclo de *N* iterações $\Rightarrow$ ***O(N²)*** , algoritmo quadrático

- É possível melhorar?
  - Algoritmo linear é melhor : tempo de execução é proporcional a tamanho de entrada (difícil fazer melhor)
    - Se $A_{ij}$ é uma subsequência com custo negativo, $A_{iq}$ com q>j não é a subsequência máxima

# Subsequência máxima - linear

```
template <class Comparable>
Comparable maxSubSum3(const vector<Comparable> &a)
{
    Comparable thisSum = 0; Comparable maxSum = 0;
    for (int j=0 ; j < a.size() ; j++)
     {
       thisSum += a[j];
       if (thisSum > maxSum)
            maxSum = thisSum;
       else if (thisSum < 0)
            thisSum = 0;
     return maxSum;
}
```

# Subsequência máxima - recursivo

- Método “divisão e conquista”
  - Divide a sequência a meio
  - A subsequência máxima está:
    a) na primeira metade
    b) na segunda metade
    c) começa na 1ª metade, vai até ao último elemento da 1ª metade, continua no primeiro elemento da 2ª metade, e termina em um elemento da 2ª metade.
  - Calcula as três hipóteses e determina o máximo
  - a) e b) calculados recursivamente
  - c) realizado em dois ciclos:
    - percorrer a 1ª metade da direita para a esquerda, começando no último elemento
    - percorrer a 2ª metade da esquerda para a direita, começando no primeiro elemento

# Subsequência máxima - recursivo

```
template <class Comparable>
Comparable maxSubSum(const vector<Comparable> &a, int left, int
   right)
{
   Comparable maxLeftBorderSum = 0, maxRightBorderSum = 0
   Comparable leftBorderSum = 0, rightBorderSum = 0;
   int center = (left + right ) / 2;

   if (left == right)
        return ( a[left] > 0 ? a[left] : 0 )

   Comparable maxLeftSum = maxSubSum (a, left, center);
   Comparable maxRightSum = maxSubSum (a, center + 1, right);
```

# Subsequência máxima - recursivo

```
   for (int i = center ; i >= left ; i--)
   {
      leftBorderSum += a[i];
      if (leftBorderSum > maxLeftBorderSum)
         maxLeftBorderSum = leftBorderSum;
  }
   for (int j = center +1 ; j <= right ; j++)
   {
      rightBorderSum += a[j];
      if (rightBorderSum > maxRightBorderSum)
         maxRightBorderSum = rightBorderSum;
  }

   return max3( maxleftSum, maxRightSum,
                maxLeftBorderSum + maxRightBorderSum);

}
```

# Subsequência máxima - recursivo

- Análise complexidade temporal
  - Seja $T(N)$ = tempo execução para problema tamanho N
  - $T(1) = 1$ (recorda-se que constantes não interessam)
  - $T(N) = 2 * T(N/2) + N$
    - duas chamadas recursivas, cada uma de tamanho $N/2$. O tempo de execução de cada chamada recursiva é $T(N/2)$
    - tempo de execução de caso c) é $N$

- *E análise espacial ?*

  S(n) = O(log N)

# Subsequência máxima - recursivo

- Análise complexidade temporal

$$\begin{cases} T(N) = 2 * T(N/2) + N \\ T(1) = 1 \end{cases}$$

$T(N/2) = 2 * T(N/4) + N/2$

$T(N/4) = 2 * T(N/8) + N/4$

…

$T(N) = 2*2*T(N/4) + 2*N/2 + N$

$T(N) = 2*2*2*T(N/8) + 2*2*N/4 + 2*N/2 + N$

$T(N) = 2^k * T(N/2^k) + kN$

$T(1) = 1 : \; N/2^k = 1 \;\Rightarrow k = \log_2 N$

$T(N) = N*1 + N* \log_2 N = O(N*logN)$

# O problema da Torre de Hanói

Torre de Hanói é um "quebra-cabeça"

- Uma base contém três pinos, num dos quais estão dispostos alguns discos uns sobre os outros, por ordem crescente de diâmetro

- O problema consiste em passar todos os discos de um pino para outro qualquer, de maneira que um disco maior nunca fique em cima de outro menor.

Complexidade temporal ?

Complexidade espacial?