Introdução à Análise de Complexidade

Sumário:
Introdução à análise de complexidade. Notações O, Omega e Theta. Comportamentos típicos. Análise de algoritmos não recursivos.

18-Março-2021

Análise de Algoritmos

- Motivação
  - Nem todos programas requerem o mesmo esforço computacional, seja em tempo de execução seja no uso que fazem da memória
  - Quais as medidas dessa exigência?
  - Dados dois programas que realizem a mesma tarefa como escolher o melhor?

Análise de Algoritmos

- Considere-se o problema da selecção: Dado um array com elementos comparáveis determinar o k-ésimo maior

  - Exemplo: Dado o array

0	1	2	3	4	5	6	7	8	9
4	3	30	27	7	18	13	19	33	8

  - Qual o 5º maior elemento?

Análise de Algoritmos

- Algoritmo 1
  - Tome-se o array

0	1	2	3	4	5	6	7	8	9
4	3	30	27	7	18	13	19	33	8

  - Ordenem-se os seus elementos por ordem decrescente

0	1	2	3	4	5	6	7	8	9
33	30	27	19	18	13	8	7	4	3

  - Seleccione-se o 5º elemento

Análise de Algoritmos

- Algoritmo 1
  - Que tipo de problemas apresenta?
    - A ordenação do array
      - Que algoritmo escolher para ordenar?
        - Há tantos!
        - Diferem em quê?

Análise de Algoritmos

- Algoritmo 2

- Tomar os primeiros k números para um array e ordenar (usando o mesmo método)

0	1	2	3	4	5	6	7	8	9
4	3	30	27	7	18	13	19	33	8

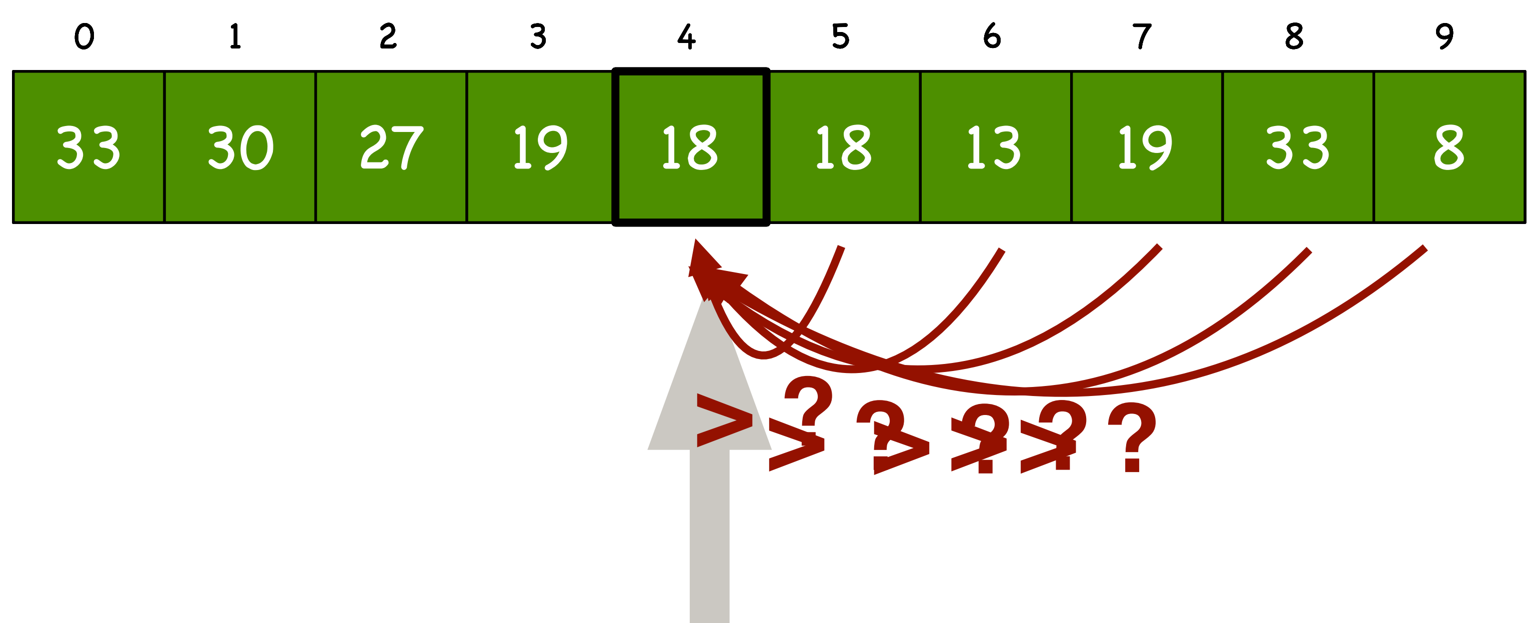
0	1	2	3	4
4	3	30	27	7

- **ordenar**

0	1	2	3	4
30	27	7	4	3

Análise de Algoritmos

- Para os restantes N-k números:
    - Se x>k-ésimo, introduzi-lo na posição correcta eliminando o último elemento
    - Senão ignora-se



Análise de Algoritmos

- Qual dos dois algoritmos, é melhor ?
- É possível encontrar outros ainda melhores ?
- Em que medida ? I.e. qual o significado de melhor?

Análise de Algoritmos

- Critérios para medição de programas:
  - Número de instruções
  - Número de transferências da memória secundária para a principal
  - O tempo de execução (**C. Temporal**)
  - O espaço em memória (**C. Espacial**)
    - Quando omissa refere-se à **complexidade temporal**

Análise de Algoritmos

- Como avaliar as exigências de um programa?
  - Por observação
  - Método experimental
  - Analiticamente

Análise de Algoritmos

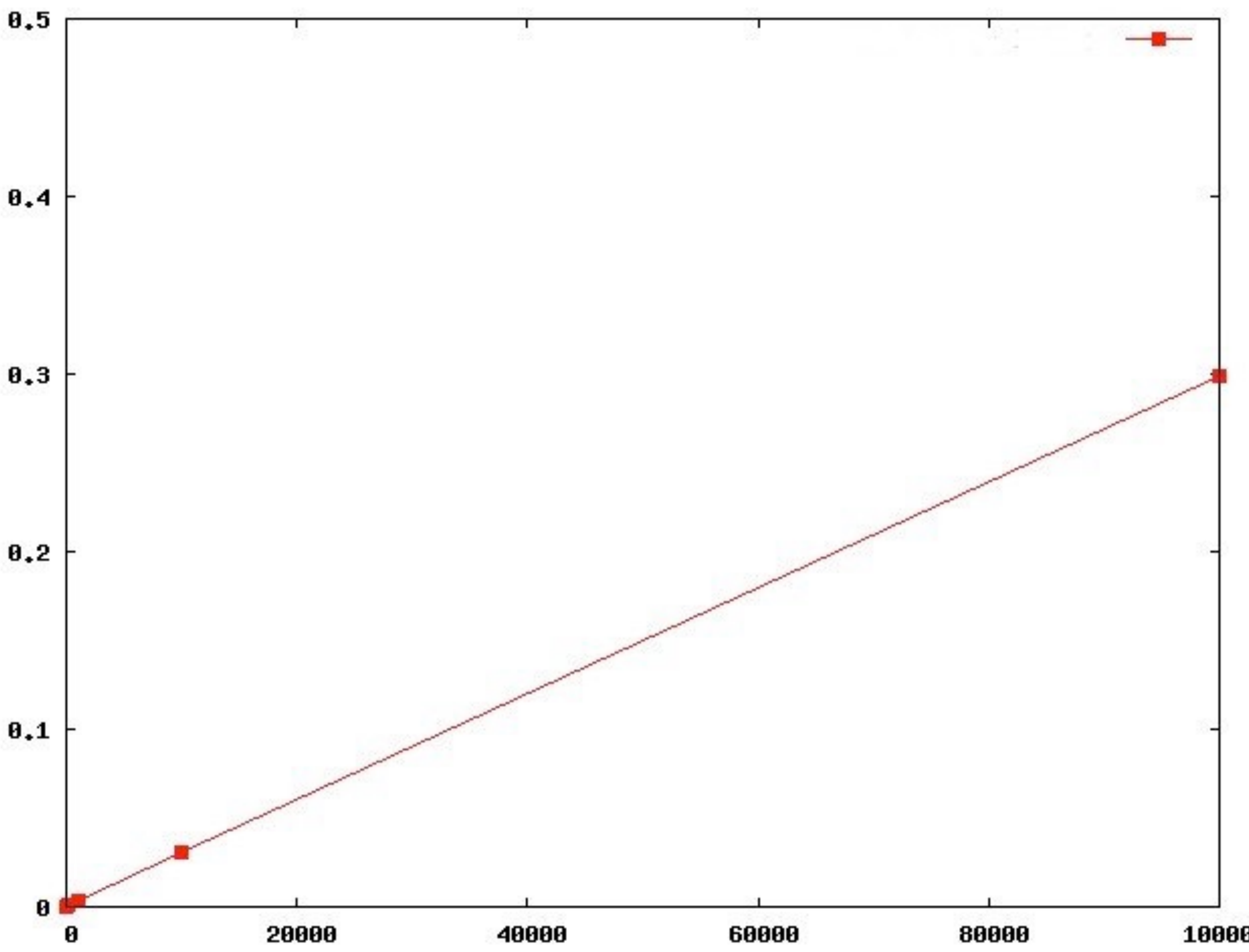
- Seja P um problema de dimensão N e p1 um programa que o resolva. Foram realizadas as seguintes observações na execução de p1:

N	Seg.
10	0,00034
100	0,00063
1000	0,00333
10000	0,03042
100000	0,29832

Análise de Algoritmos

- O gráfico da figura mostra nos quadradinhos vermelhos as observações efectuadas; a tendência está marcada em linha.

- Que concluir?



Que o tempo de execução gasto por p1, cresce linearmente com N

Análise de Algoritmos

- O tempo de execução dum programa depende do tipo de recursos usados:
  - arquitectura, velocidade relógio, velocidade de acesso às memórias, sistema operativo, linguagem usada, do compilador, etc.
  - Analisar o comportamento dum algoritmo só por observação dos tempos de execução pode ser inconclusivo (metros , polegadas?)

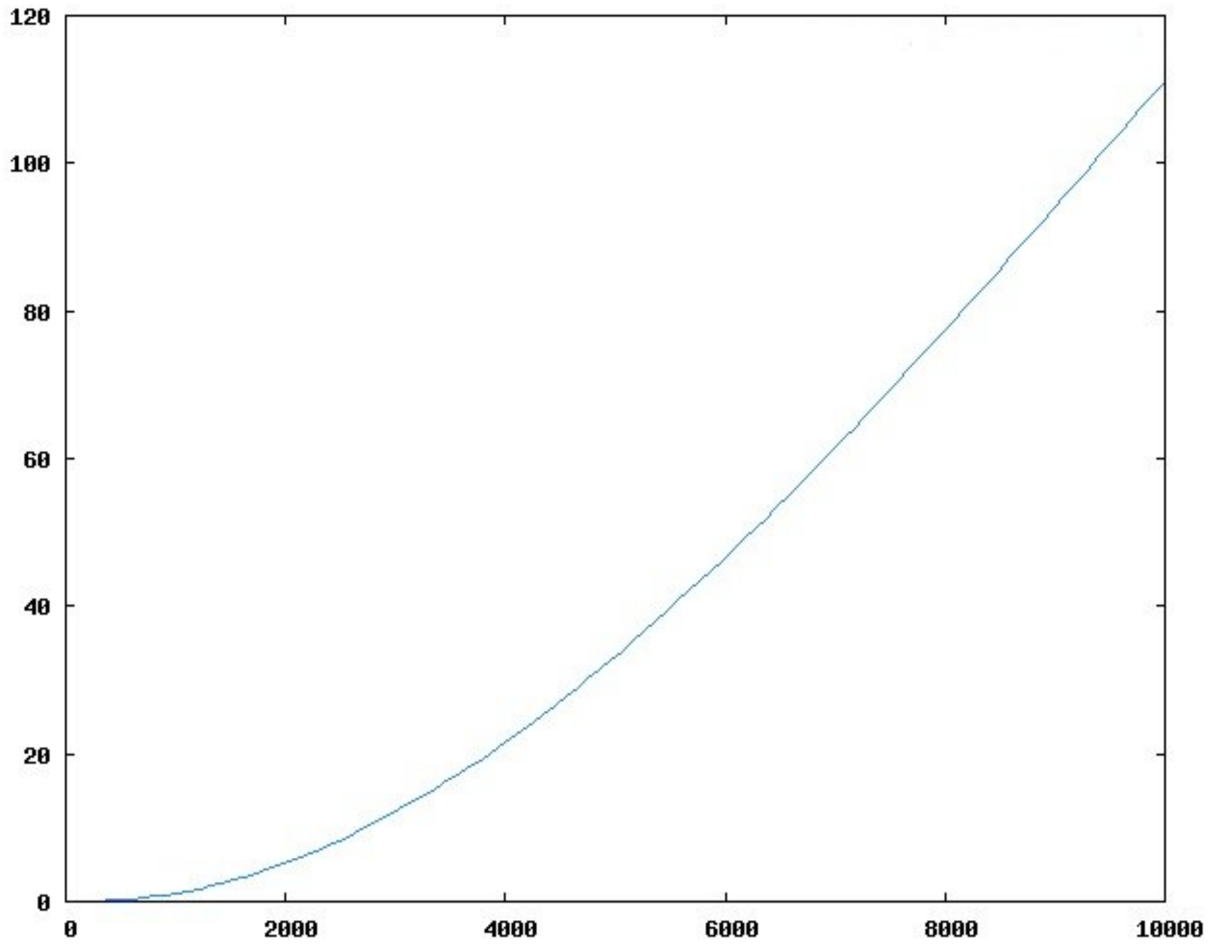
Análise de Algoritmos

- Complexidade Assimptótica:
  - A complexidade assimptótica é um modo analítico para estudar/estimar o tempo de execução
  - A mesma experiência realizada noutro computador pode baixar/subir o declive da recta mas continuará uma recta e o comportamento linear

ANÁLISE DE ALGORITMOS

- Vejamos agora que conclusões retirar dum algoritmo cujas observações são as da tabela abaixo:

N	Seg
10	0,00045
100	0,01112
1000	1,1233
10000	111,13



- Que concluir?

Não é linear! A taxa de crescimento do tempo não é proporcional ao crescimento descrito pelo declive duma recta

Análise Algoritmos

- Ao analisar o tempo de execução de determinado algoritmo, três casos são possíveis de considerar
  - Pior caso T_{worst} (N)
  - Caso médio T_{avg} (N)
  - Melhor caso T_{Best} (N)
  - Obviamente $T_{Best}(N) \leq T_{avg}(N) \leq T_{worst}(N)$

Análise Algoritmos

- O caso médio pode ser difícil de calcular
- O pior caso estabelece um limite superior para o tempo de execução
  - É o pior que poderá acontecer

Análise de Algoritmos

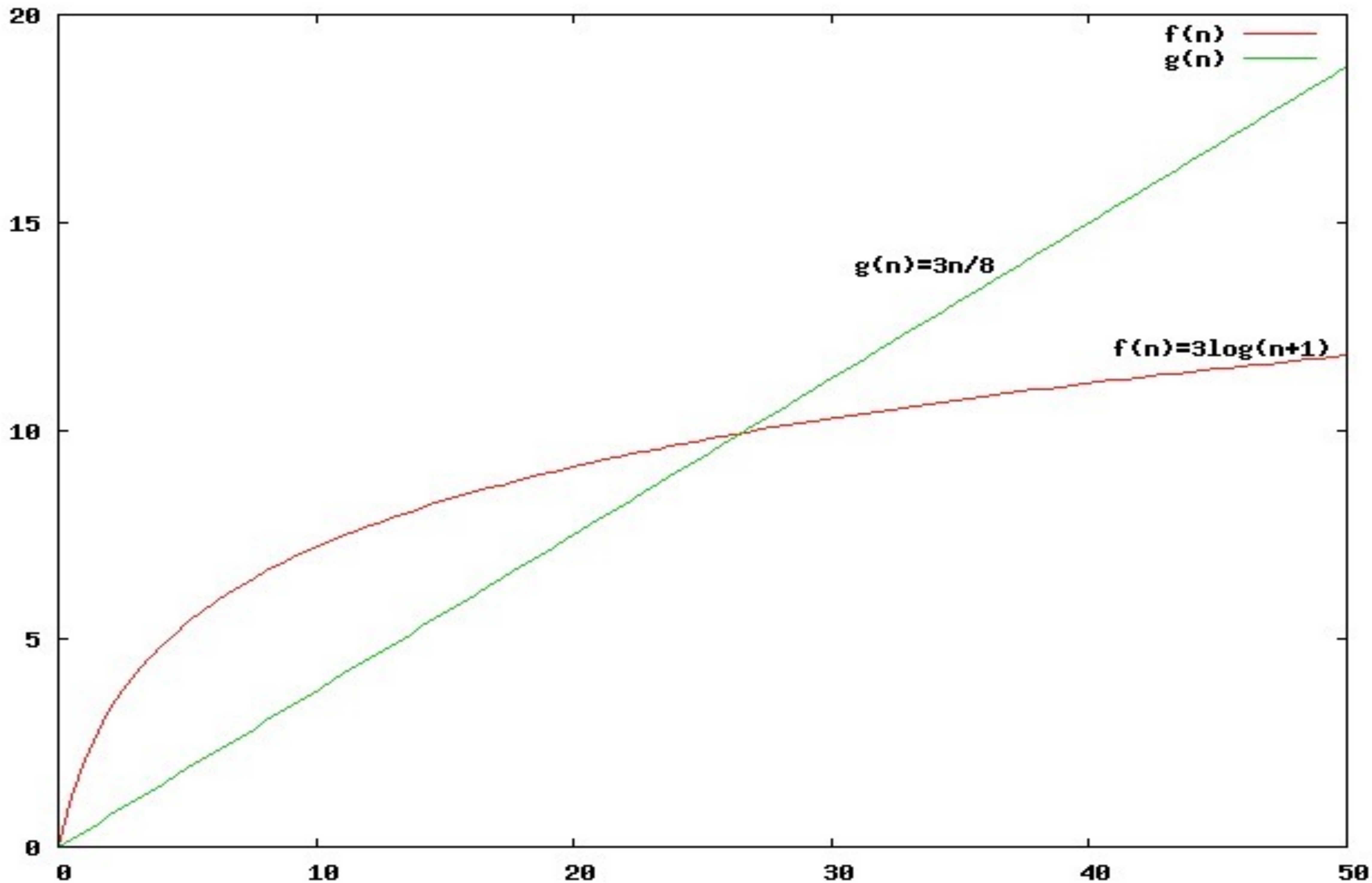
- As ferramentas da complexidade analítica

    - Def: Sejam f e g duas funções de domínio inteiro não negativo e contradomínio real. Diz-se que g domina assimptoticamente f e escreve-se f é $O(g)$ ou f é da ordem de g se:

        - $\exists k>0 \exists c>0 \forall N>k \Rightarrow |f(N)| \leq c.|g(N)|$

    - $O(g)$ representa o conjunto de todas as funções limitadas superiormente por g, logo é também possível escrever f ∈ $O(g)$ ou $O(f) \subseteq O(g)$

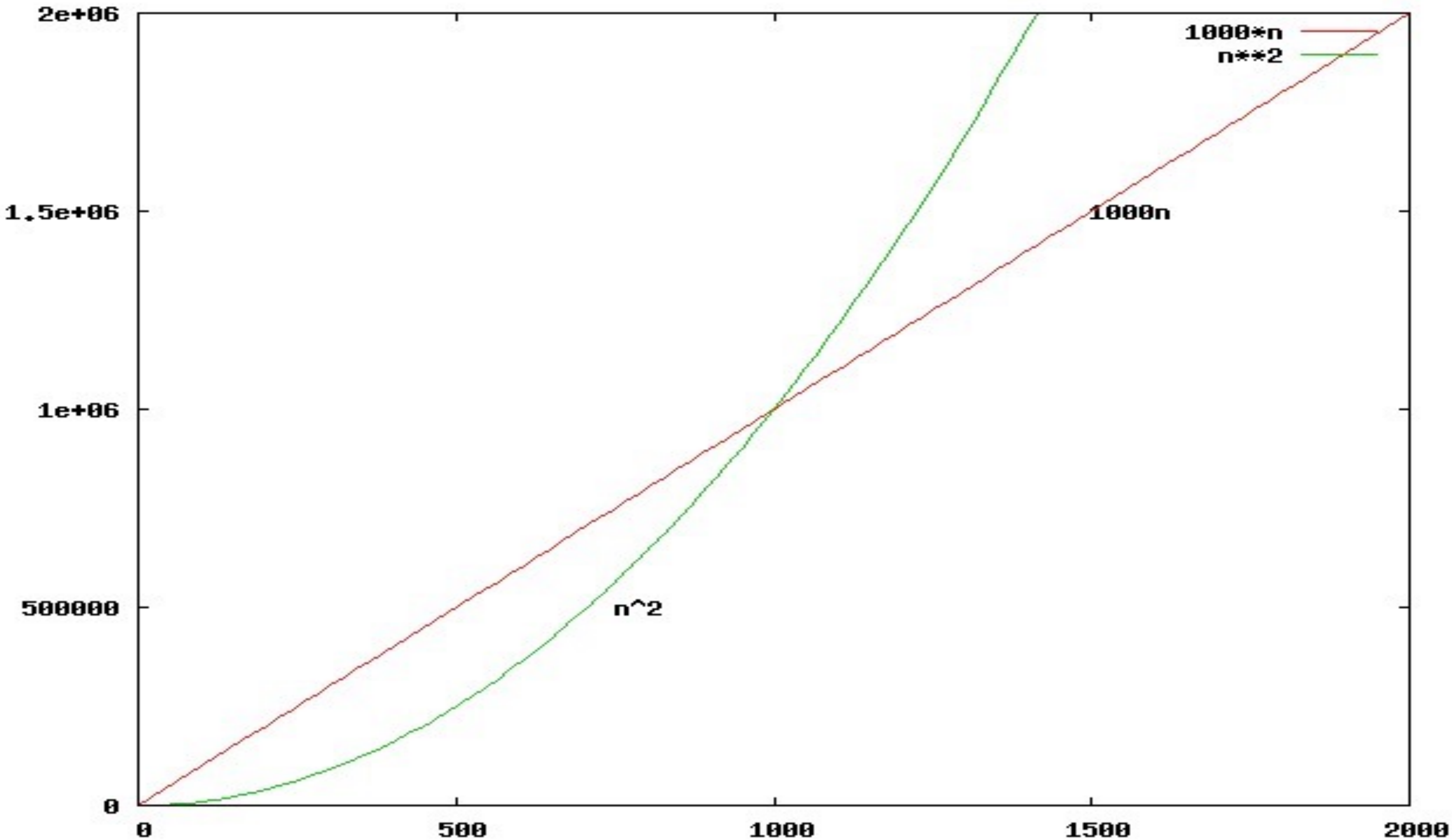
Análise de Algoritmos



Análise de Algoritmos

- Suponhamos que dadas $f(N)=1000N$ e $g(N)=N^2$ queremos relacioná-las ($f \leq g$?g domina f?)
  - Para pequenos valores de N, 1000N é maior que N^2
  - Mas sabemos também que existe um ponto de viragem i.e. uma ordem a partir da qual 1000N será sempre inferior a N^2.
  - É precisamente este ponto de viragem que na definição é referenciado por k

Análise de Algoritmos



Análise de Algoritmos

- Regra geral a análise de complexidade é realizada não por aplicação das definições formais, mas usando propriedades e resultados conhecidos.

Propriedades

- Definições:
    - A expressão f é $O(g)$ representa um limite superior da função f. O limite inferior resulta da definição
        - $\exists k>0 \exists c>0 \forall N>k \Rightarrow |f(N)| \geq c.|g(N)|$
    - Diz-se neste caso que f é $\Omega(g)$
    - Se f ∈ $O(g)$ então g é um limite superior para f, e portanto f é um limite inferior para g logo g é $\Omega(f)$

Propriedades

- Se $T_1(n)$ é $O(g(n))$ e $T_2(n)$ é $O(h(n))$ então
  - $T_1(n) + T_2(n)$ é $max(O(g(n), O(h(n)))$
  - $T_1(n) \times T_2(n)$ é $O(g(n)\,.\,h(n))$
- Se $T(x)$ é um polinómio de grau n, então $T(x)$ é $\Theta(x^n)$ (ver definição)
- $log^k(n)$ é $O(n)$ qualquer que seja o k.
- definição: $f(n)$ é $\Theta(g(n))$ sse $f(n)$ é $O(g(n))$ e $f(n)$ é $\Omega(g(n))$

Propriedades

- Não se incluem constantes ou termos de ordem inferior nas notações O
- Não se diz que f é $O(n^2 + n)$ ou f é $O(100n^2)$
  - em ambos os casos dizemos que f é $O(n^2)$

Taxa de crescimento de duas funções

- Podemos sempre calcular a razão de crescimento relativo de duas funções calculando $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ (regra de L´Hôpital, se for preciso) e avaliando o resultado: seja $k = \lim_{n\to\infty} \frac{f(n)}{g(n)}$

  ➡ se $k = 0$, $f(n)$ é $o(g(n))$ $(f(n) < g(n))$

  ➡ se $k = c, c \neq 0$ $f(n)$ é $\Theta(g(n))$

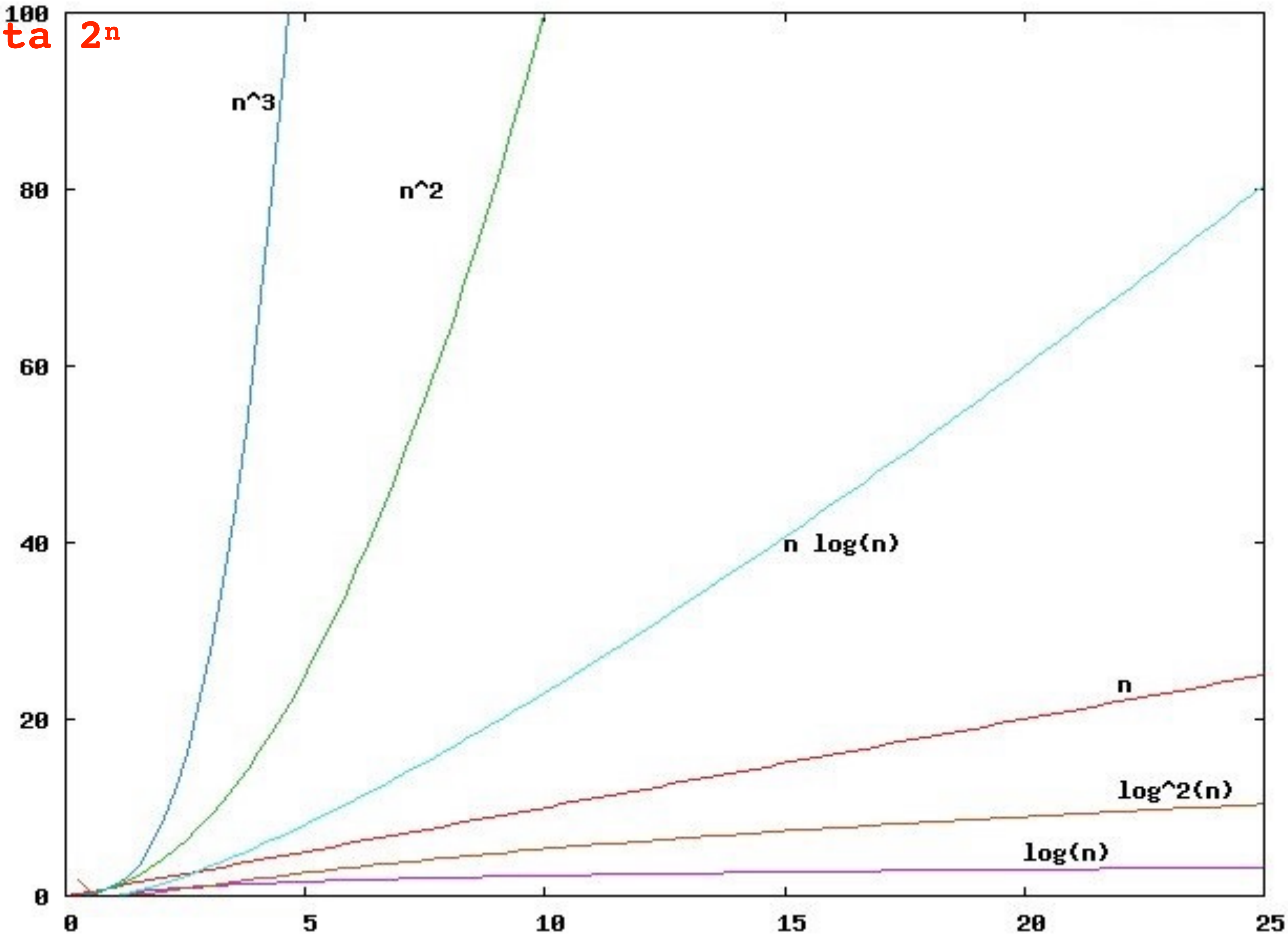
  ➡ se $k = \infty$, $g(n)$ é $o(f(n))$ $(g(n) < f(n))$

Crescimentos Típicos

Função	Crescimento
c	constante
$log(n)$	logarítmico
$log^2(n)$	logaritmo quadrado
n	linear
$nlog(n)$	n log n
n^2	quadrático
n^3	cúbico
2^n	exponencial

Crescimentos Típicos

Falta 2^n



Análise de programas

- A complexidade temporal dum programa pode ser estudada analisando a sua estrutura. Como as linguagens de programação possuem sintaxe e semânticas perfeitamente definidas a aplicação de algumas regras facilita a tarefa.

- Considera-se, regra geral, o pior caso. Porque...

ANÁLISE DE PROGRAMAS

- Modelo?
  - num computador “normal”, em que as instruções são executadas sequencialmente
    - Há instruções básicas(da linguagem)(assignements, declarações, operações…)
    - Assuma-se que essas instruções têm um custo fixo(exactamente 1a unidade de tempo)
  - Outras estruturas?

Regra 1 : Ciclos For

- O tempo de execução de um ciclo For é no máximo o tempo de execução das instruções dentro do ciclo (incluindo testes) vezes o número de iterações

```
for  (i=0;i<n;i++ )
          a[i]=0;
```

n vezes

$O(1)$

$O(n)$

Regra 2 : Ciclos For aninhados

- Para ciclos aninhados, calcula-se primeiro o tempo de execução dos ciclos mais interiores(de dentro para fora)

```
for( i = 0; i < n; i++)
    for( j = 0 ; j < n; j++)
            k++;
```

n vezes

n vezes

$O(1)$

$O(n)$

$O(n^2)$

Regra 3 : Declarações consecutivas

- O tempo de execução de um conjunto de duas declarações consecutivas(A;B), pela propriedade da soma significa que só contabilizamos o maior valor

```
for (i = 0; i < n; i++)
       a[i]=0;

for(i = 0; i < n; i++)
    for(j = 0; j < n; j++){
       a[i]+=a[j] + i + j;
    }
```

$O(n)$

$O(n^2)$

$O(n^2)$

Regra 3 : If/Else

- O tempo de execução do fragmento de código:

```
if (Condição)
    S1
else
    S2
```

- nunca é maior que o tempo do teste mais o maior dos tempos de execução de S1, S2.

Análise de programas

- Declarações consecutivas: A;B
  - max(O(A),O(B))
- if´s: if A then B else C
  - O(fA +max(O(fB),O(fC))
- Ciclos For: For (i=k;cond;step) A
  - O(nº iterações x (fA + testes e atribuições))
- Ciclos While: While (c) A
  - O(nº iterações X (fc +(fA))

Tempo de execução na recursão?

- Considere-se o exemplo da função recursiva:

```
unsigned long fact(unsigned int n){
    if (n==0)
        return 1;
    else
        return n*fact(n-1);

}
```

- Será fácil substituir a recursão por um único ciclo For

```
…
int v=1;
 for (int i=1;i<=n;i++){
     v*=i;
 }
 return v;
```

Tempo de execução na recursão?

- Quando a recursão é uma dissimulação dum ciclo(exemplo pobre de recursão..) análise da função recursiva é fácil… e claramente é O(n)

- Quando a recursão é usada em todo o seu esplendor pode até ser difícil de analisar(a complexidade).

- Exemplo duma função recursiva (muito ineficiente…) mas que pode até à primeira vista parecer "brilhante"

```
uunsigned int fib1(unsigned int n){
    if(n<=1)
        return 1;
    return fib1(n-1)+fib1(n-2);

}
```

Um pouco de matemática

```
uunsigned int fib1(unsigned int n){
    if(n<=1)
        return 1;
    return fib1(n-1)+fib1(n-2);

}
```

- Tentemos analisar o tempo de execução de fib1:
- Seja $T(n)$ o tempo de execução para $fib1(n)$
    - Se $n = 0 \vee n = 1$, tempo de execução constante $T(0) = T(1) = 1$
    - Se $n \geq 2$, temos o tempo de execução do teste(1), mais as duas chamadas $fib1(n-1)$ e $fib1(n-2)$.

Um pouco de matemática

```
uunsigned int fib1(unsigned int n){
    if(n<=1)
        return 1;
    return fib1(n-1)+fib1(n-2);

}
```

- Por definição $T(n) = T(n-1) + T(n-2) + 2$ (2 pelo teste e pela adição)

- Dado que $fib(n) = fib(n-1) + fib(n-2)$ é fácil demonstrar (por indução) que

    - $T(n) \geq fib(n)$ e como (tb por indução se prova)

    - $fib(n) < (5/3)^n$ e $fib(n) > (3/2)^n$, então o tempo de execução cresce exponencialmente…

Don't compute anything more than once

- Analisando a expressão (return fib1(n-1)+ fib2(n-1) é fácil vermos que temos muita computação desnecessária e redundante
  - para calcular fib(n-1) calculo fib(n-2) e depois volto a calculá-lo… para somar com fib(n-1) e retornar…
  - uma regra básica de eficiência é nunca repetir trabalho
  - Usar a recursão de forma eficiente, é também um dos objectivos de EDA1
    - to be done…

Análise de programas

- Ex1

```
int fact=1;

for (int i=1; i<n; i++)
    fact=i*fact;
```

$O(1)$

$O(1)$

$O(n)$

$O(n)$

Análise de programas

- Ex2:

```
for (int i=0;i<n;i++)
     a[i]=0;
```

O(1) O(n)

O(n²)

```
for (int i=0; i<n; i++)
     for(int j=0; j<n ; j++)
          a[i] += a[j] + i + j;
```

O(1) O(n) O(n²)

Análise de Programas

- Qual a complexidade dos algoritmos 1 e 2 para o problema da selecção apresentados no início da aula?

- Diferentes!

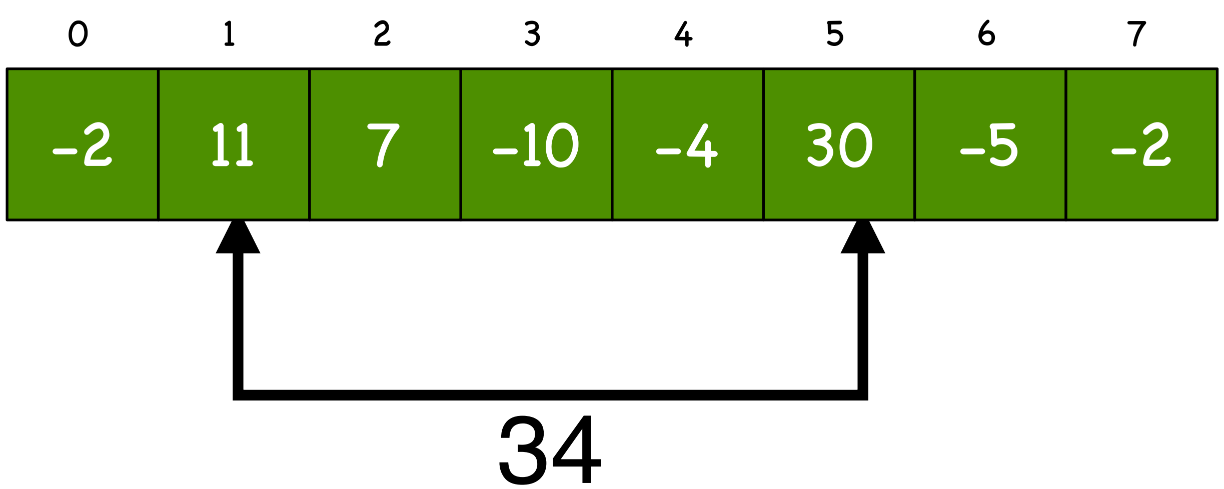
- Algum é claramente melhor que o outro?

- Têm a mesma complexidade?

- Justifiquem!

Análise de Algoritmos

- Dada uma sequência de N números (eventualmente negativos) encontrar a subsequência cuja soma seja máxima:



```
int max_subsequence_sum( int a[], unsigned int n ) {
    int this_sum, max_sum, best_i, best_j, i, j, k;
    max_sum=a[0];
    best_i=best_j=0;

    for(i=0;i<n;i++){
        for(j=i;j<n;j++){
            this_sum=0;
            for (k=i;k<=j;k++)
                this_sum+=a[k];
            if (this_sum>max_sum){
                max_sum=this_sum;
                best_i=i;
                best_j=j;
            }

        }

    }
    printf("de %d a %d, soma maxima % d\n",best_i,best_j,max_sum);
    return max_sum;
}
```

- **Ordem de complexidade?**

```
Fib value is 267914296 time 0.0000000000
de 1 a 5, soma maxima  34
Program ended with exit code: 0
```

$$T(N) = \sum_{i=1}^{N} \sum_{j=i}^{N} \sum_{k=i}^{j} 1$$

$$T(N) = \sum_{i=1}^{N} \sum_{j=i}^{N} j - i + 1$$

$$T(N) = \sum_{i=1}^{N} \frac{(N-i+2)(N-i+1)}{2}$$

$$T(N) = \frac{N^3 + 3N^2 + 2N}{6}$$

$$S_n = \frac{U_1 + U_n}{2} \times n$$

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n-1)(2n+1)}{6}$$

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

$$= \frac{1}{2}\frac{n(n+1)(2n+1)}{6} - \left(n + \frac{3}{2}\right)\frac{n(n+1)}{2} + \frac{n^2 + 3n + 2}{2}n$$

$$= \frac{n^3 + 3n^2 + 2n}{6}$$