Questão

Considere dois algoritmos que executam n operações:

- Algoritmo 1: $f_1(n) = 2n^2 + 5n$ operações
- Algoritmo 2: $f_2(n) = 500n + 4000$ operações

Qual algoritmo é mais eficiente? Em quais casos?

Questão

Prove que f(n) é O(n2), sendo:

$$f(n) = 2n^2 + 3n + 4$$

Questão

Prove que:

$$n^2 - 200n - 300 = O(n^2)$$

Ordem O

Dadas funções assintoticamente não negativas f e g, dizemos que f **está na ordem** O de g e escrevemos $f = O(g)$ se

$$f(n) \leq c.g(n)$$

para algum c positivo e para todo n suficientemente grande.

Questão

Prove que f(n) é O(n2), sendo:

$$f(n) = 2n^2 + 3n + 4$$

Questão

Prove que:

$$n^2 - 200n - 300 = O(n^2)$$

Motivação e objetivos

- Motivação
  - Não sempre a análise empírica é uma boa alternativa.
  - As vezes, é necessário se ter alguma indicação de eficiência antes de qualquer investimento em desenvolvimento.
  - ... e todos os demais incovenientes da análise empírica.

- Objetivos
  - Mostrar que é possível avaliar a eficiência de algoritmos sem necessariamente ter que implementá-los.
  - Apresentar como representar e avaliar matematicamente a

Eficiência de algoritmos

Eficiência de algoritmos

- Medida quantitativa inversa da quantidade de recursos (tempo de processamento, memória, etc) requeridos para a execução do algoritmo
- Quanto maior a eficiência menos recursos são gastos

Eficiência de algoritmos

- Medida quantitativa inversa da quantidade de recursos (tempo de processamento, memória, etc) requeridos para a execução do algoritmo
- Quanto maior a eficiência menos recursos são gastos

- Como *medir* a eficiência de um algoritmo?

Eficiência de algoritmos

- Medida quantitativa inversa da quantidade de recursos (tempo de processamento, memória, etc) requeridos para a execução do algoritmo
- Quanto maior a eficiência menos recursos são gastos
- Como *medir* a eficiência de um algoritmo?
- Método experimental
  - Implementar diversos algoritmos
  - Executar um grande número de vezes
  - Analisar os resultados

Eficiência de algoritmos

- Medida quantitativa inversa da quantidade de recursos (tempo de processamento, memória, etc) requeridos para a execução do algoritmo
- Quanto maior a eficiência menos recursos são gastos

- Como *medir* a eficiência de um algoritmo?

- Método experimental
  - Implementar diversos algoritmos
  - Executar um grande número de vezes
  - Analisar os resultados

- Método analítico
  - A ideia é encontrar funções matemáticas que descrevam o crescimento do tempo de execução dos algoritmos em relação ao tamanho da entrada
  - Comparar as funções

- Método experimental

```c
#include <stdio.h>
#include <time.h>

int main(int argc, char *argv[])
{
    float tempo;
    time_t t_ini, t_fim;
    int i, j;

    t_ini = time(NULL);
    for(i = 0; i < 50000; i++)
    {
        for(j = 0; j < 50000; j++);
    }
    t_fim = time(NULL);

    tempo = difftime(t_fim, t_ini);

    printf("Tempo: %f\n", tempo);
    return 0;
}
```

Eficiência de algoritmos

- Método analítico
  - Como *expressar* a eficiência de um algoritmo?
  - Através da ordem de crescimento do tempo de execução
    - Apenas o termo de mais alta ordem é considerado
    - Caracterização simples da eficiência que permite comparar o desempenho relativo entre algoritmos alternativos
  - Quando observamos tamanhos de entradas grandes o suficiente, de forma que apenas a ordem de crescimento do tempo de execução seja relevante, estamos estudando a eficiência **assintótica**
  - **Analisar** um algoritmo significa prever os recursos (tempo) de que o algoritmo necessitará

Eficiência de algoritmos

- Vamos analisar o seguinte algoritmo

```
def soma_min_max(xs):
    '''
    Soma os valores mínimos e máximos de uma lista.
    >>> soma_min_max([4, 2, 3, 5, 3])
    7
    '''

    min = xs[0]
    for x in xs:
        if x < min:
            min = x
    max = xs[0]
    for x in xs:
        if x > xs:
            max = x
    return min + max
```

Eficiência de algoritmos

- Vamos chamar o tamanho da entrada (`len(xs)`) de n
- Vamos contabilizar o custo de cada linha
  - Cada operação primitiva tem custo 1 (demora uma unidade de tempo)
  - Contamos quantas vezes (no máximo) cada linha é executada
  - Somamos o custo total de cada linha

Eficiência de algoritmos

- Vamos analisar o seguinte algoritmo

```
def soma_min_max(xs):
    '''
    Soma os valores mínimos e máximos de uma lista.
    >>> soma_min_max([4, 2, 3, 5, 3])
    7
    '''

    min = xs[0]
    for x in xs:
        if x < min:
            min = x
    max = xs[0]
    for x in xs:
        if x > xs:
            max = x
    return min + max
```

Eficiência de algoritmos

- Vamos analisar o seguinte algoritmo

```python
def soma_min_max(xs):
    '''
    Soma os valores mínimos e máximos de uma lista.
    >>> soma_min_max([4, 2, 3, 5, 3])
    7
    '''
                                # Custo     Vezes          Total
    min = xs[0]                 #   1         1              1
    for x in xs:                #   1       n + 1          n + 1
        if x < min:             #   1         n              n
            min = x             #   1     no máximo n        n
    max = xs[0]                 #   1         1              1
    for x in xs:                #   1       n + 1          n + 1
        if x > xs:              #   1         n              n
            max = x             #   1     no máximo n        n
    return min + max            #   1         1              1
```

- O `for` é executado $n + 1$ vezes, n vezes, um para cada elemento, 1 vez para concluir que não tem mais elementos

Eficiência de algoritmos

- Somando o custo total de todas as linhas obtemos
  - $1 + n + 1 + n + n + 1 + n + 1 + n + n + 1 = 6n + 5$
  - Ficamos com o termo de mais alta ordem
  - Portanto, o tempo de execução do algoritmo é $O(n)$

Eficiência de algoritmos

- Vamos analisar o seguinte algoritmo

```python
def esta_na_lista(xs, x):
    '''
    Devolve True se x está na lista xs. False caso contrário
    >>> esta_na_lista([4, 6, 2, 1], 4)
    True
    >>> esta_na_lista([4, 6, 2, 1], 6)
    True
    >>> esta_na_lista([4, 6, 2, 1], 10)
    False
    '''

    i = 0
    while i < len(xs):
        if xs[i] == x:
            return True
        i = i + 1
    return False
```

Eficiência de algoritmos

- Vamos analisar o seguinte algoritmo

```
def esta_na_lista(xs, x):
    '''
    Devolve True se x está na lista xs. False caso contrário
    >>> esta_na_lista([4, 6, 2, 1], 4)
    True
    >>> esta_na_lista([4, 6, 2, 1], 6)
    True
    >>> esta_na_lista([4, 6, 2, 1], 10)
    False
    '''
                            # Custo      Vezes            Total
    i = 0                   #    1          1               1
    while i < len(xs):      #    1    no máximo n + 1     n + 1
        if xs[i] == x:      #    1      no máximo n         n
            return True     #    1      no máximo n         n
        i = i + 1           #    1      no máximo n         n
    return False            #    1          1               1
```

Eficiência de algoritmos

- Somando o custo total de todas as linhas obtemos
  - $1 + n + 1 + n + n + n + 1 = 4n + 3$
  - Ficamos com o termo de mais alta ordem
  - O tempo de execução do algoritmo é $O(n)$

Eficiência de algoritmos

- Vamos analisar o seguinte algoritmo

```
def dobro_primeiro(xs):
    '''
    Devolve o dobro do primeiro elemento da lista xs.
    >>> dobro_primeiro([5, 4, 5, 1])
    10
    '''
                          # Custo    Vezes    Total
    return xs[0] + 1      #   1          1        1
```

- Somando o custo de todas as linhas obtemos 1
- Neste caso, o tempo de execução do algoritmo é $O(1)$
- Ou seja, o tempo de execução é constante e não depende do tamanho da entrada
- Observe que usamos $O(1)$ para qualquer algoritmo que tenha tempo de execução constante, não importa se a soma total dos custos seja 1, 10 ou 50, o importante neste caso é não depender do tamanho da entrada

Eficiência de algoritmos

- Vamos analisar o seguinte algoritmo

```python
def ordena_insercao(xs):
    '''
    Ordena xs usando o algoritmo de ordenação por inserção.
    >>> xs = [5, 3, 4, 1, 9]
    >>> ordena_insercao(xs)
    >>> xs
    [1, 3, 4, 5, 9]
    '''

    for j in range(1, len(xs)):
        chave = xs[j]
        i = j - 1
        while i >= 0 and xs[i] > chave:
            xs[i + 1] = xs[i]
            i = i - 1
        xs[i + 1] = chave
```

Conceitos Básicos: melhor caso, pior caso e caso médio

Três cenários dependentes da entrada:

- **Melhor caso**: menor tempo de execução;

Conceitos Básicos: melhor caso, pior caso e caso médio

Três cenários dependentes da entrada:

- **Melhor caso**: menor tempo de execução;
- **Pior caso**: maior tempo de execução. Geralmente, priorizamos determinar o pior caso;

Conceitos Básicos: melhor caso, pior caso e caso médio

Três cenários dependentes da entrada:

- **Melhor caso**: menor tempo de execução;
- **Pior caso**: maior tempo de execução. Geralmente, priorizamos determinar o pior caso;
- **Caso médio**: média dos tempos de execução. Mais difícil de obter;

Conceitos Básicos: melhor caso, pior caso e caso médio

Exemplo: busca sequencial.

Operação relevante: comparação de x com elementos de V;

buscaSequencial(x, V)

$i \leftarrow 1$;

enquanto $(i \leq n)$ **e** $(V[i] \neq x)$ **faça** // executa n vezes no máximo

    $i \leftarrow i + 1$;

se $i > n$ **então** "Busca sem sucesso"

    senão "Busca com sucesso"

Conceitos Básicos: melhor caso

Melhor caso da busca sequencial: x está em $V[1]$!

buscaSequencial(x, V)
1: $i \leftarrow 1$;
2: **enquanto** $(i \leq n)$ **e** $(V[i] \neq x)$ **faça** // executa n vezes no máximo
3: $i \leftarrow i + 1$;
4: **se** $i > n$ **então** "Busca sem sucesso"
5: **senão** "Busca com sucesso"

Complexidade de tempo: 1

Conceitos Básicos: pior caso

Pior caso da busca sequencial: x está em $V[n]$ ou não está em V!

buscaSequencial(x, V)
1: $i \leftarrow 1$;
2: **enquanto** $(i \leq n)$ **e** $(V[i] \neq x)$ **faça** // executa n vezes no máximo
3: $i \leftarrow i + 1$;
4: **se** $i > n$ **então** "Busca sem sucesso"
5: **senão** "Busca com sucesso"

Complexidade de tempo: n

Conceitos Básicos: caso médio

- **Caso médio da busca sequencial:** assumindo que x está em V, $f(n) = 1 \times p_1 + 2 \times p_2 + \ldots + n \times p_n$ onde p_i é probabilidade de x estar na posição i;

Conceitos Básicos: caso médio

- **Caso médio da busca sequencial:** assumindo que x está em V, $f(n) = 1 \times p_1 + 2 \times p_2 + \ldots + n \times p_n$ onde p_i é probabilidade de x estar na posição i;
- **probabilidades são iguais**: $p_i = \frac{1}{n}$;

Conceitos Básicos: caso médio

- **Caso médio da busca sequencial:** assumindo que x está em V, $f(n) = 1 \times p_1 + 2 \times p_2 + \ldots + n \times p_n$ onde p_i é probabilidade de x estar na posição i;
- **probabilidades são iguais**: $p_i = \frac{1}{n}$;
- $f(n) = \frac{1}{n}(1$

Conceitos Básicos: caso médio

- **Caso médio da busca sequencial:** assumindo que x está em V, $f(n) = 1 \times p_1 + 2 \times p_2 + \ldots + n \times p_n$ onde p_i é probabilidade de x estar na posição i;
- **probabilidades são iguais**: $p_i = \frac{1}{n}$;
- $f(n) = \frac{1}{n}(1 + 2$

Conceitos Básicos: caso médio

- **Caso médio da busca sequencial:** assumindo que x está em V, $f(n) = 1 \times p_1 + 2 \times p_2 + \ldots + n \times p_n$ onde p_i é probabilidade de x estar na posição i;
- **probabilidades são iguais**: $p_i = \frac{1}{n}$;
- $f(n) = \frac{1}{n}(1 + 2 + 3$

Conceitos Básicos: caso médio

- **Caso médio da busca sequencial:** assumindo que x está em V, $f(n) = 1 \times p_1 + 2 \times p_2 + \ldots + n \times p_n$ onde p_i é probabilidade de x estar na posição i;
- **probabilidades são iguais**: $p_i = \frac{1}{n}$;
- $f(n) = \frac{1}{n}(1 + 2 + 3 + \ldots + n)$

Conceitos Básicos: caso médio

- **Caso médio da busca sequencial:** assumindo que x está em V, $f(n) = 1 \times p_1 + 2 \times p_2 + \ldots + n \times p_n$ onde p_i é probabilidade de x estar na posição i;
- **probabilidades são iguais**: $p_i = \frac{1}{n}$;
- $f(n) = \frac{1}{n}(1 + 2 + 3 + \ldots + n) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right)$

Conceitos Básicos: caso médio

- **Caso médio da busca sequencial:** assumindo que x está em V, $f(n) = 1 \times p_1 + 2 \times p_2 + \ldots + n \times p_n$ onde p_i é probabilidade de x estar na posição i;
- **probabilidades são iguais**: $p_i = \frac{1}{n}$;
- $f(n) = \frac{1}{n}(1 + 2 + 3 + \ldots + n) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}$

Conceitos Básicos: caso médio

- **Caso médio da busca sequencial:** assumindo que x está em V, $f(n) = 1 \times p_1 + 2 \times p_2 + \ldots + n \times p_n$ onde p_i é probabilidade de x estar na posição i;
- **probabilidades são iguais**: $p_i = \frac{1}{n}$;
- $f(n) = \frac{1}{n}(1 + 2 + 3 + \ldots + n) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}$
- **Complexidade de tempo:** $\frac{n+1}{2}$, ou seja, uma pesquisa bem-sucedida examina aproximadamente metade dos registros.

Eficiência de algoritmos

- Vamos analisar o seguinte algoritmo

```python
def ordena_insercao(xs):
    '''
    Ordena xs usando o algoritmo de ordenação por inserção.
    >>> xs = [5, 3, 4, 1, 9]
    >>> ordena_insercao(xs)
    >>> xs
    [1, 3, 4, 5, 9]
    '''

    for j in range(1, len(xs)):
        chave = xs[j]
        i = j - 1
        while i >= 0 and xs[i] > chave:
            xs[i + 1] = xs[i]
            i = i - 1
        xs[i + 1] = chave
```

Eficiência de algoritmos

- Considere a linha número 4
  - O número de vezes que a linha é executada dependo do valor de j e da condição `xs[i]` $>$ `chave`
  - Vamos considerar o pior caso (o arranjo em ordem invertida) em que `xs[i]` $>$ `chave` é sempre verdadeiro

Eficiência de algoritmos

- Vamos analisar o seguinte algoritmo

```python
def ordena_insercao(xs):
    '''
    Ordena xs usando o algoritmo de ordenação por inserção.
    >>> xs = [5, 3, 4, 1, 9]
    >>> ordena_insercao(xs)
    >>> xs
    [1, 3, 4, 5, 9]
    '''

    for j in range(1, len(xs)):
        chave = xs[j]
        i = j - 1
        while i >= 0 and xs[i] > chave:
            xs[i + 1] = xs[i]
            i = i - 1
        xs[i + 1] = chave
```

# Num	Custo	Vezes
#1	1	n
#2	1	n - 1
#3	1	n - 1
#4	2	1+2+3+...+n
#5	1	1+2+3+...+n-1
#6	1	1+2+3+...+n-1
#7	1	n - 1

Eficiência de algoritmos

- Considere a linha número 4

  - Portanto, a quantidade de vezes que a linha é executa é

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

  - Portanto, o custo total da linha é

$$T4 = 2 \left(\frac{n(n+1)}{2} \right) = n(n+1) = n^2 + n$$

Eficiência de algoritmos

- O custo total das linhas 5 e 6 podem ser calculados de forma semelhante a da linha 4

$$T5 = T6 = \frac{n^2 + n}{2}$$

- Somando o custo total de todas as linhas obtemos
  - Ficamos com o termo de mais alta ordem
  - O tempo de execução do algoritmo é $O(n^2)$

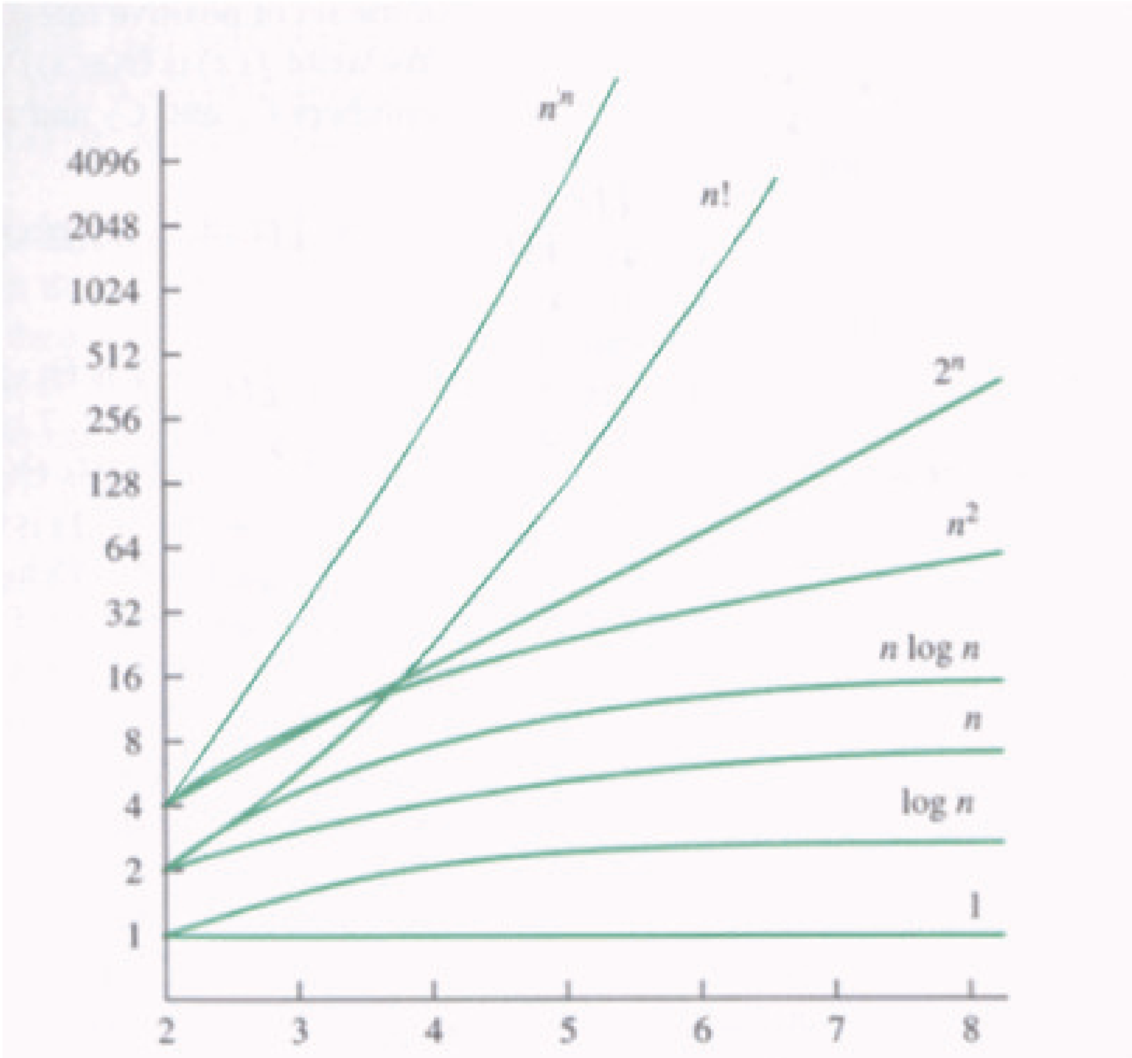
Eficiência de algoritmos

Notação	Nome	Exemplos
$O(1)$	constante	Determinar se um número é par ou ímpar, encontrar o valor máximo em um arranjo ordenado
$O(\log n)$	logarítmico	Encontrar um valor em um arranjo ordenado usando busca binária
$O(n)$	linear	Encontrar um valor em um arranjo não ordenado usando busca linear
$O(n \log n)$	loglinear	quicksort
$O(n^2)$	quadrático	bubblesort
$O(c^n)$, $c > 1$	exponencial	Encontrar a solução exata para o problema do caixeiro viajante usando programação dinâmica
$O(n!)$	fatorial	Encontrar a solução exata para o problema do caixeiro viajante usando força bruta

Tabela 1: Funções comuns encontradas quando analisamos o tempo de execução de algoritmos

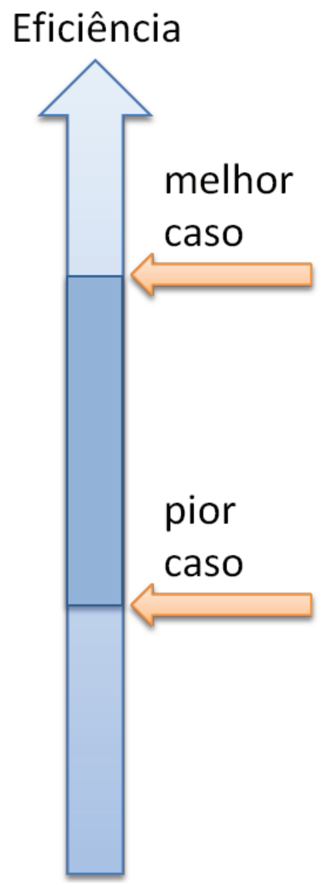
Eficiência de algoritmos

Funções comuns de crescimento de tempo. O eixo Y (tempo de execução) está em escala logarítmica. O eixo X representa o tamanho da entrada.



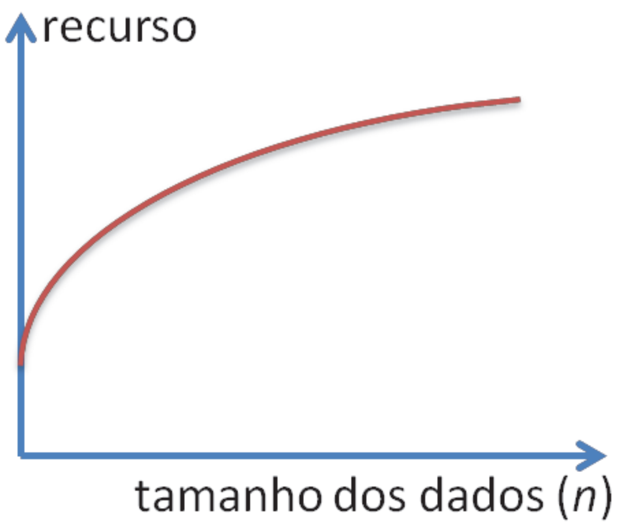
Análise matemática de algoritmos

- É possível comparar algoritmos independente de suporte computacional.
- Permite visualizar uma tendência de comportamento independente de ambiente.
- Ideia de "faixa de trabalho" em função dos dados (melhor e pior caso).



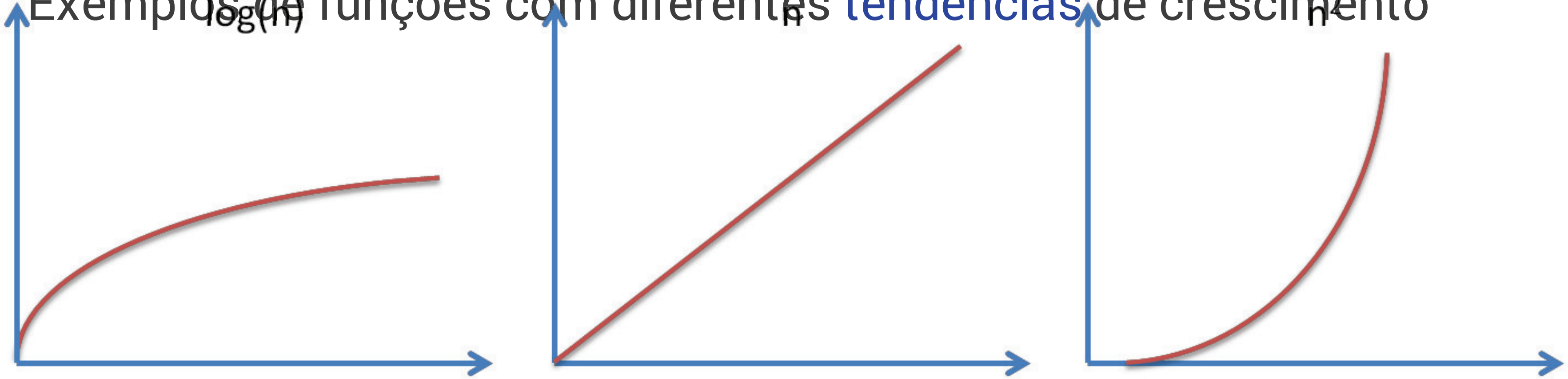
Princípios de análise de algoritmo

- A maioria dos algoritmos possui um parâmetro primário n, que afeta significativamente o tempo de execução
- Normalmente n é diretamente proporcional ao tamanho dos dados a serem processados
- Objetivos
  - Expressar a necessidade de recursos em termos de n (função)
  - Usar expressões algébricas simples, mas que expressem uma tendência
  - Oferecer uma análise independente



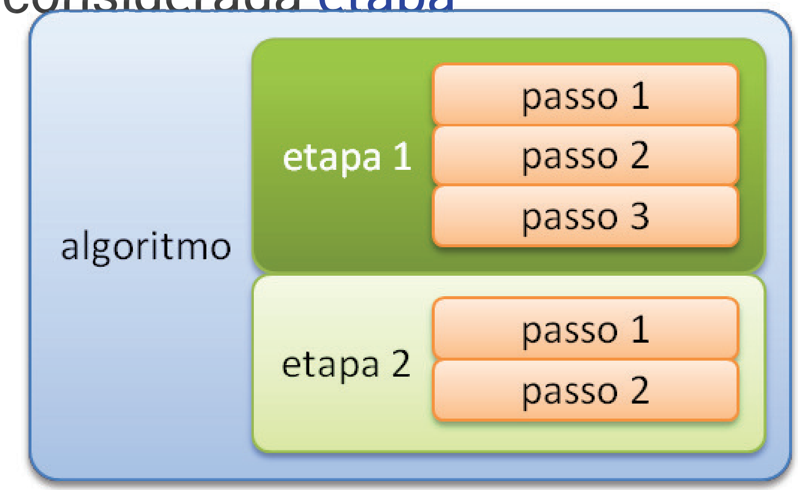
Expressando tendências

- Simplificações das expressões
  - Não são consideradas constantes aditivas ou multiplicativas
  - Apenas o termos de maior grandeza é considerado
  - Exemplo: $2n^3 + 5n/2 - 29 \Rightarrow n^3$
- Exemplos de funções com diferentes tendências de crescimento



Procedimento

- Um algoritmo pode ser dividido em etapas
- Cada etapa possui uma ou mais operações básicas (ou passos)
- Cada passo envolve um número fixo de operações básicas cujos tempos de execução são considerados constantes
- A etapa com maior número de execuções é considerada etapa donimante
- A expressão do algoritmo será a que representa a etapa dominante



Principais expressões

- 1 (constante): quando a execução independe dos dados de entrada
- n: quando é necessário processar todos elementos, cada um em tempo constante
- n^2: quando, para cada elemento, é necessário verificar todos os demais
- n^3: quando é necessário verificar combinações triplas dos dados
- $\log_2 n$: quando o problema é reduzido em 2 subproblemas com a metade do tamanho original e apenas 1 deles é processado.
- $n \log_2 n$: quando, para cada elemento, o problema é subdividido
- 2^n: normalmente, quando verifica-se todas as possíveis alternativas

Atividade de reforço (1)

- O que faz e como representar o número de passos do algoritmo:

```c
int mystery(int v[], int n) {
  int count = 0;
  for (int i = 0; i < n; i++)
    for (int j = 0, j < n; j++)
      if (v[i] == v[j])
        count++;
  printf("%i\n", count/2);
}
```

Atividade de reforço [1]

- O que faz e como representar o número de passos do algoritmo:

```c
int mystery(int v[], int n) {
  int count = 0;
  for (int i = 0; i < n; i++)
    for (int j = 0, j < n; j++)
      if (v[i] == v[j])
        count++;
  printf("%i\n", count/2);
}
```

- O 1o laço executa n vezes, o 2o também executa n e, dentro do segundo é realizado no máximo 2 operações.
- Então, a função executa $n.\, n.2 = 2.n^2 \Rightarrow n^2$ passos.

Alguns padrões (para identificar)

- Uma sequência sem laço ou recursão conta passo constante (1)

```
/* bloco com número de passos constante */
```

- Um único laço com n passos internos constante: linear (n)

```
for(i=0; i < n; i++)
    /* bloco com número de passos constante */
```

- Dois laços de tamanho n aninhados: quadrático (n^2)

```
for(i=0; i < n; i++)
    for(j=0; j < n; j++)
        /* bloco com número de passos constante */
```

Alguns padrões (para identificar)

- Um laço interno dependente de um externo: quadrático (n^2)

  (requer uso de somatório duplo)

```
for(i=0; i < n; i++)
    for(j=0; j < i ; j++)
        /* bloco com número de passos constante */
```

- Quando divide o problema pela metade: logarítmico ($log_2 n$)

```
if (test)
    subprob(0, n/2);    /* do início à metade */
else
    subprob(n/2+1, n-1) /* da metade ao fim */
```