Algoritmos e estrutura de dados

Conceitos básicos

Marco A L Barbosa

Conteúdo

Como avaliar um algoritmo?

Eficiência de algoritmos

Estrutura de dados

Tipo abstrato de dados

Como avaliar um algoritmo?

Como avaliar um algoritmo?

Como avaliar um algoritmo?

- Simplicidade
- Corretude
- Eficiência

Simplicidade

- Um algoritmo é **simples** se puder ser facilmente entendido, implementado e mantido

Simplicidade

- Um algoritmo é **simples** se puder ser facilmente entendido, implementado e mantido
- Como escrever um algoritmo simples?

Simplicidade

- Um algoritmo é **simples** se puder ser facilmente entendido, implementado e mantido
- Como escrever um algoritmo simples?
- Não conhecemos técnicas para isto

Corretude

- Um algoritmo é dito **correto** se para toda entrada específica a saída correta é produzida. Dizemos que um algoritmo correto **resolve** o problema dado

Corretude

- Um algoritmo é dito **correto** se para toda entrada específica a saída correta é produzida. Dizemos que um algoritmo correto **resolve** o problema dado

- Como saber se uma algoritmo está correto?

Corretude

- Um algoritmo é dito **correto** se para toda entrada específica a saída correta é produzida. Dizemos que um algoritmo correto **resolve** o problema dado

- Como saber se uma algoritmo está correto?

- Usando técnicas matemáticas de prova (não estudaremos este item formalmente)
- Vamos "verificar" a corretude dos nossos programas (implementação de algoritmos) usando testes automatizados
  - Testes servem apenas para provar que um algoritmo tem erros, nunca para provar que o algoritmo está correto (Dijkstra)

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

Eficiência de algoritmos

- 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
    '''
                                # 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

```
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

```
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

- Vamos analisar o seguinte algoritmo

```
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]
    '''
                                                # Num Custo   Vezes
    for j in range(1, len(xs)):                 #  1    1       n
        chave = xs[j]                           #  2    1     n - 1
        i = j - 1                               #  3    1     n - 1
        while i >= 0 and xs[i] > chave:         #  4    2     2+3+...+n
            xs[i + 1] = xs[i]                   #  5    2     1+2+3+...+n-1
            i = i - 1                           #  6    2     1+2+3+...+n-1
        xs[i + 1] = chave                       #  7    2     n - 1
```

Eficiência de algoritmos

- Considere a linha número 4
  - O número de vezes que a linha é executada depende 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

- Considere a linha número 4
  - Quando $j = 1$, a linha é executada 2 vez
  - Quando $j = 2$, a linha é executada 3 vezes,
  - ...
  - Quando $j = n - 1$, a linha é executada n vezes
  - Portanto, a quantidade de vezes que a linha é executa é

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

  - Portanto, o custo total da linha é

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

Eficiência de algoritmos

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

$$T5 = T6 = n^2 - n$$

- Somando o custo total de todas as linhas obtemos
  - $n + n - 1 + n - 1 + n^2 + n - 2 + n^2 - n + n^2 - n + 2(n - 1)$
  - $= 3n^2 + 4n - 6$
  - Ficamos com o termo de mais alta ordem
  - O tempo de execução do algoritmo é $O(n^2)$

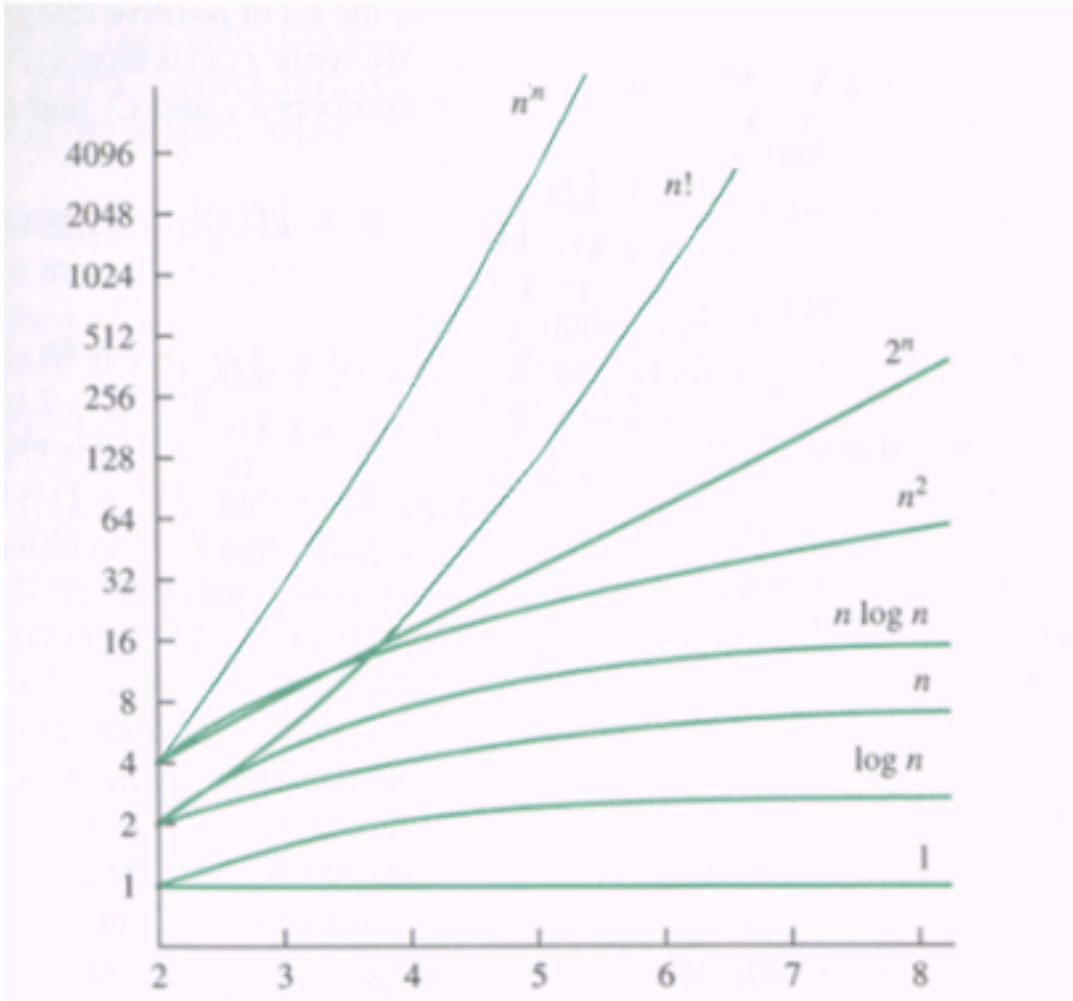
Eficiência de algoritmos

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.



Estrutura de dados

Estrutura de dados

- É um modo particular de armazenamento e organização dos dados em um programa de modo que estes dados possam ser usados eficientemente
  - Tempo de execução
  - Memória
- Diferentes tipos de estruturas são adequadas a diferentes aplicações
- Para manter uma estrutura de dados são necessários algoritmos
- Operações comuns em uma estrutura de dados
  - Inserir um valor
  - Remover um valor
  - Pesquisar um valor

Estrutura de dados

- Exemplos
  - Arranjos dinâmicos
  - Listas encadeadas
  - Árvores
  - Tabelas

Tipo abstrato de dados

Tipo abstrato de dados

- É uma tipo de dado (estrutura de dados) definida de forma abstrata pelas operações que podem ser realizadas sobre elas
- Os dados e as operações sobre os dados são definidas na mesma unidade sintática
- A representação interna do tipo de dado é oculta do cliente do tipo

Tipo abstrato de dados / Vantagens

- Encapsulamento
  - Os detalhes ficam ocultos para quem usa o tipo
- Mudanças localizadas
  - Quando a implementação do tipo é alterada, os clientes do tipo não precisam ser alterados
- Flexibilidade
  - A implementação do tipo pode ser substituída

Formas de alocação

- Alocação estática
  - A quantidade de memória para execução do programa é determinada antes do programa começar a sua execução e permanece inalterada durante a execução do programa
  - Vantagens: desempenho e facilidade de programação
  - Desvantagens: não é flexível, possível desperdício de memória
- Alocação dinâmica
  - A quantidade de memória para execução do programa pode variar durante a execução
  - Vantagens: programas mais genéricos e flexíveis
  - Desvantagens: o programador deve fazer a gerência da memória - dificuldade de programação (que é amenizada em linguagens com gerência automática de memória, como o Python)
- O Python suporta apenas alocação dinâmica