

Performance de algoritmos, Dicionários e Conjuntos

Carlos Tavares

25 de Novembro de 2024

A complexidade é a área da ciência da computação que estuda os recursos necessários para fazer computações: memória, tempo, acessos a disco, energia ou outros.

Tipicamente o tempo é medido em termos do número de passos necessários para realizar uma tarefa.

Exemplo:

Encontrar um elemento numa lista. No pior caso, é necessário percorrer todos os elementos da lista: são necessários N passos, onde N é o tamanho da lista.

Desempenho de Programas: Complexidade Básica ii

Geralmente, usamos as assíntotas principais (as de crescimento mais rápido) das funções de desempenho, que se denotam por $O(\text{função})$ (Notação O).

Funções de interesse neste contexto:

- *Tempo constante*, $O(1)$: o tempo não depende do tamanho da entrada/dados a serem explorados;
- *Tempo logarítmico*, $O(\log N)$: o tempo é exponencialmente menor que o tamanho da entrada;
- *Tempo linear*, $O(N)$: o tempo necessário é diretamente proporcional ao tamanho da entrada;
- *Tempo linearítmico*, $O(N \log N)$: o tempo é linear multiplicado pelo logarítmico em termos do tamanho da entrada;

Desempenho de Programas: Complexidade Básica iii

- *Tempo quadrático*, $O(N^2)$: o tempo é quadrático em termos da entrada;
- *Tempo cúbico*, $O(N^3)$: o tempo é cúbico em termos da entrada.

As funções formam uma hierarquia:

$$O(1) < O(\log N) < O(N) < O(N \log N) < O(N^2) < O(N^2 \log N) < O(N^3)$$

Informalmente, as funções que têm desempenho polinomial são consideradas eficientes (classe P), em oposição às que têm desempenho exponencial. Na prática: depois de N^2 , já são algo ineficientes.

Alguma álgebra:

A complexidade de uma sequência de instruções:

instrução_1

instrução_2

é dada por:

$$O(\text{instrução}_1) + O(\text{instrução}_2)$$

Porém, apenas a maior delas deve ser considerada. Exemplo:

$$O(N \log N) + O(1) \sim O(N \log N)$$

Complexidade das Operações das Listas i

Uma lista no Python é, na verdade, um array, ou seja, um bloco de posições na memória.

Element 0	Element 1	Element 3	...	Element N
Element 0+N	Element 1+N	Element 2+N	...	Element N+N
...
Element 0+(N.(N-1))	Element 1+(N.(N-1))	Element 2+(N.(N-1))	...	Element N.N

Qual será a complexidade de operações como:

- **append** ou **pop** no final da lista ?
- obter um item (**elem = l [index]**), definir um item (**l [index] = elem**) ?
- obter o comprimento da lista (**len**)?

Complexidade das Operações das Listas ii

Operação	Caso Médio	Pior Caso Amortizado
Append	$O(1)$	$O(1)$
Pop no final	$O(1)$	$O(1)$
Obter item	$O(1)$	$O(1)$
Definir item	$O(1)$	$O(1)$
Obter comprimento	$O(1)$	$O(1)$

Complexidade das Operações das Listas iii

E quanto a inserir no meio de uma lista ou apagar um item do meio?

Operação	Caso Médio	Pior Caso Amortizado
x in s	$O(n)$	
Inserir	$O(n)$	$O(n)$
Deletar item	$O(n)$	$O(n)$
Pop intermediário	$O(n)$	$O(n)$
Iterar	$O(n)$	$O(n)$

Como conclusão final, listas são rápidas para algumas operações, mas muito lentas para várias outras.

Eliminar elementos repetidos (solução elegante)

```
l = [9, 5, 4, 6, 7, 8, 8, 9, 7, 7, 1, 2, 3]
```

```
# Solução elegante
```

```
lr = []
```

```
for i in l:
```

```
    if i not in lr:
```

```
        lr.append(i)
```

Complexidade de Algoritmos Usando Listas ii

E quanto a usar a ordenação para agrupar os elementos repetidos?

```
l.sort() # Custo de  $O(N\log N)$   
i = 0  
j = 0  
while j < len(l):  
    i = j  
    while i + 1 < len(l) and l[i] == l[i+1]:  
        l.pop(i+1)  
    j = j + 1
```

Podemos fazer melhor, digamos, $O(N)$?

Dicionários

Um dicionário é uma estrutura de dados fundamental no Python que associa chaves a valores, ou seja, valores são indexados por chaves.

Listas vs Dicionários

Imagine que precisamos armazenar elementos do tipo (chave, valor).

Com uma lista:

Para armazenar um elemento

```
» l = []  
» chave_valor = (chave, valor)  
» l.append(chave_valor)
```

Para recuperar um elemento por chave

```
» for i in l:  
»     if i[0] == chave:  
»         return i
```

Com um dicionário:

Para armazenar um elemento

- » chave_valor = (chave, valor)
- » d = dict ()
- » d[chave] = (chave, valor)

Para recuperar um elemento

- » ...
- » return d[chave]

Além de ser mais simples programaticamente, dicionários têm um desempenho muito melhor: **tempo constante** para ambas as operações **armazenar** e **recuperar**, enquanto que listas têm **armazenar** em **tempo constante**, mas **recuperar** em **tempo linear (N)**.

Como é que isto é possível?

Se houver uma função matemática que mapeie um objecto qualquer do python para uma posição da lista

$$h : \text{Objecto} \rightarrow \{1 \dots N\}$$

onde N é o numero de posições disponíveis na lista.

Este tipo de funções chamam-se funções de hash

N = 20

dict = list (range (N))

def **hash** (nome):

 hash = 0

 for i in nome:

 hash += ord (i)

 return hash % N

def **salvar** (chave, valor):

 dict[hash(chave)] = valor

def **recuperar** (chave):

 return dict(hash(chave))

Na verdade, esta implementação tem o problema admitir colisões, chaves diferentes que originam o mesmo **hash**. Existem funções de hash melhores.

Um pequeno exercício: Estude o comportamento de um dicionário de acordo com os diferentes valores envolvidos, nomeadamente, o tamanho da lista base e a função hash.

Dicionários em Python

Criar um dicionário vazio

```
» dict = {}
```

Alterar ou criar uma chave

```
» dict[chave] = valor
```

As chaves podem ser qualquer objeto imutável no Python: tuplas, strings ou números.

Exemplo:

```
» dict = {}
```

```
» tuplo = ("Exemplo", 1, 2, 3)
```

```
» dict[tuplo] = "Uma afirmação qualquer"# Isto funciona
```

```
» lista = ["Exemplo", 1, 2, 3]
```

```
» dict[lista] = "Outra afirmação"# Isso falha
```

Outras operações possíveis com dicionários

`dict.keys()` - Aceder a todas as chaves de um dicionário

`dict.values()` - Aceder a todos os valores de um dicionário

Tanto as chaves quanto os valores não podem ser indexados, mas podem ser iterados.

Criar dicionários em compreensão (de maneira semelhante às listas)

`i: i**2 for i in range (0, 10)`

Complexidade das Operações com Dicionários

Operação	Caso Médio	Pior Caso Amortizado
$k \in d$	$O(1)$	$O(n)$
Cópia[3]	$O(n)$	$O(n)$
Obter Item	$O(1)$	$O(n)$
Definir Item[1]	$O(1)$	$O(n)$
Excluir Item	$O(1)$	$O(n)$
Iteração[3]	$O(n)$	$O(n)$

Exercício. Implemente um programa para eliminar repetidos utilizando dicionários.

Conjuntos (Sets)

Definição em matemática: uma coleção de objetos distintos e bem definidos formando um grupo (teoria dos conjuntos de Zermelo-Fraenkel (**ZFC**)).

Definição em Python: uma coleção não ordenada de elementos únicos.

Projetado para ser especialmente eficaz em operações de conjuntos:

- Existência;
- Interseção;
- União;
- Diferença;
- Diferença simétrica.

Conjunto vazio

```
» s = set()
```

```
» s
```

```
{}
```

```
» s = {}
```

```
» s
```

```
{}
```

Conjunto com elementos

```
» s = {12, 13, 14, 15}
```

```
» s
```

```
{12, 13, 14, 15}
```

Criação de Conjuntos ii

Um conjunto pode conter qualquer tipo de elementos **hashable**

» `s = {"R", "C", "A"}`

» `s = {"R", 13, "D", 12}`

Não pode conter elementos **não-hashable**:

» `s = {{1,2,3}, {4,5,6}}`

(...)

`TypeError: unhashable type: 'set'`

Criação de Conjuntos iii

Também é possível construir um conjunto a partir de uma lista ou tuplo:

```
» s = set(("A", 12, 13))
```

```
» s
```

```
{"A", 12, 13}
```

```
» s = set([12, 13, 14, 14])
```

```
» s
```

```
{12, 13, 14}
```

Um conjunto não possui elementos repetidos! Elementos repetidos são automaticamente ignorados.

Manipulação Básica de Conjuntos i

Também é possível criar um conjunto em compreensão:

```
» {i for i in range (0,20)}
```

Obter o tamanho do conjunto: função **len**

```
» len({12, 13, 14})
```

3

Verificar se um elemento faz parte de um conjunto: operador **in**

```
» 12 in {12, 13, 14}
```

True

Adicionar elementos a um conjunto

» **s.add("exemplo")**

Remover um elemento de um conjunto

» **s.remove("exemplo")**

Manipulação Básica de Conjuntos iii

Um conjunto não possui ordem: não é uma sequência. Portanto, não é indexável e não pode ser iterado com uma instrução while:

```
» s
```

```
{12, 13, 14, 15, 16}
```

```
» i = 0
```

```
» while i < 4:
```

```
»     print(s[i])
```

```
....
```

```
TypeError: 'set' object is not subscriptable
```

No entanto, isso pode ser feito com uma instrução for:

```
» for i in s:  
»     print(i)
```

16

12

13

14

15

Interseção

» **s.intersection(d)**

{13}

» **s & d**

{13}

União

» **s.union(d)**

{10, 11, 12, 13, 14, 15, 16}

» **s | d**

{10, 11, 12, 13, 14, 15, 16}

Diferença

» **s.difference(d)**

{10, 11, 12}

» **s - d**

{10, 11, 12}

Este operador não é simétrico! $s - d \neq d - s$

Diferença simétrica ($\equiv s - d \mid d - s$)

» **s.symmetric_difference(d)**

{10, 11, 12, 14, 15, 16}

Operações com Conjuntos iii

Operação	Caso Médio	Pior Caso Amortizado
$x \in s$	$O(1)$	$O(n)$
União ($s \cup t$)	$O(\text{len}(s) + \text{len}(t))$	
Interseção ($s \cap t$)	$O(\min(\text{len}(s), \text{len}(t)))$	$O(\text{len}(s) * \text{len}(t))$
Diferença ($s - t$)	$O(\text{len}(s))$	$O(\text{len}(t))$
Diferença simétrica ($s \oplus t$)	$O(\text{len}(s))$	$O(\text{len}(s) * \text{len}(t))$

Exercícios

Exercício 1. Faça um programa para eliminar elementos repetidos de uma lista, usando conjuntos. Qual é a complexidade da solução?

Exercício 2. Faça um programa para detectar "zonas cercadas" num tabuleiro de jogo de Go.

Perguntas?