Algorithmic performance, Dictionaries and Sets

Carlos Tavares

November 24, 2024

Performance of programs: basic complexity i

Complexity is the area of computer science that studies the resources employed in computations: memory, time, disk accesses, energy or others

Typically time is measured in terms of the number of steps that have to be taken to do a task.

Example:

Find an element in a list. Worst case it is necessary to go through all the elements in a list: N steps are necessary, where N is the list size.

Performance of programs: basic complexity ii

Generally, we usually use the major assymptotes (the fastest growing) of the performance functions, that we call $O(function)$.

Functions of interest for this context:

- *Constant time*, $O(1)$ the time does not depend in the size of input/data to be explored;
- *Logarithmic time*, $O(logN)$, the time is exponentially smaller than the size of the input;
- *Linear time*, $O(N)$ the time necessary is exactly dependent
- *Linearithmic time*, $O(NLogN)$. The time is linear times logarithmic in terms of the size of the input.
- *Quadratic time*, $O(N^2)$. The time is quadratic in terms of the input;

Performance of programs: basic complexity iii

- *Cubic time*, $O(N^3$, the time is cubic in terms of the input.

The functions form a hierarchy:
$O(1) < O(logN) < O(N) < O(NlogN) < O(N^2) < O(N^2logN) < O(N^3)$

Informally, the functions that have polynomial performance are considered efficient (class P), as opposed to those who have expoenential performance. In practice: from N^2 we are already entering the realm of inefficiency.

Performance of programs: basic complexity iv

Some algebra:

The complexity of a sequence of instructions is given by:

$O(instruction_1) + O(instruction_2)$

However only the largest one is to be considered. Example:

$O(NlogN) + O(1) = O(NlogN)$

Complexity of list operations i

A list in python is actually an array, i.e. a batch of positions in memory

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

What is the complexity of operations such as append, pop last, get item (elem = l [index]), set item (l [index] = elem), get length?

Operation	Average Case	Amortized Worst Case
Append	O(1)	O(1)
Pop last	O(1)	O(1)
Get Item	O(1)	O(1)
Set Item	O(1)	O(1)
Get Length	O(1)	O(1)

Complexity of list operations ii

What about insert in the middle of a list, delete an item from the middle?

Operation	Average Case	Amortized Worst Case
x in s	O(n)	
Insert	O(n)	O(n)
Delete Item	O(n)	O(n)
Pop intermediate	O(n)	O(n)
Iterate	O(n)	O(n)

As a final conclusion lists are fast in some operations, but very slow in several other ones.

Eliminate repeated elements (elegant solution)

```
l = [9, 5, 4, 6, 7, 8, 8, 9, 7, 7, 1, 2, 3]
# Elegant solution
lr = []
for i in l:
    if i not in lr:
        lr.append (i)
```

Complexity of algorithms using lists ii

What about when using the sort to put the repeated elements together?

```
l.sort ()# This has a cost of O(NlogN)
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
```

Can we do better, say *N* for instance?

Dictionaries

A dictionary is a fundamental data structure in python that is able to associate keys and values, i.e. values are indexed by keys.

Lists vs Dictionaries

Imagine that we need to store elements of the type (key, value).

With a list:

```
# To store an element
» l = []
» key_value = (key, value)
» l.append (key_value)

# To retrieve an element by key
» for i in l:
»       if i [0] == key:
»           return i
```

With a dictionary:

To store an element
» key_value = (key, value)
» d = dict ()
» d [key] = (key, value)

To retrieve an lement
» ...
» return d [key]

Besides more simple programmatically, dictionaries have much better performance.: **constant** time for both **store** and **retrieval**, in dictionaries while **store** is **constant** for lists, but **linear (N)** for **retrieval**.

How is this possible?

… by the use of an hash function to index elements

```
N = 20
dict = list (range (N))

def hash (name):
    hash = 0
    for i in name:
        hash += ord (i)
    return hash % N

def save (key, value):
    dict [hash (key)] = value

def retrieve (key):
    return dict (hash (key))
```

Actually this implementation has the problem of not admiting more than one element whose key originates the same hash. A better implementation is supplied.

A small exercise: study the behavior of a dictionary according to the different values involved, namely, size of base list and hash function.

Python dictionaries

Create an empy dictionary
» dict = {}

Change or create a key
» dict [key] = value

Keys can be any immutable object in python: tuples or strings, numbers.

Example:
» dict = {}
» tuple = ("Wonder", 1, 2, 3)
» dict [tuple] = "Some statement" # This will work
» l = ["Wonder", 1, 2,3]
» dict [l] = "Another statement"# This will fail

Other things that can be done with dictionaries

dict.keys() - Access all keys in a dictionary

dict.values () - Access all values in a dictionary

Both the keys and values cannot be indexed, but can be iterated.

Complexity of dictionary actions

Operation	Average Case	Amortized Worst Case
k in d	O(1)	O(n)
Copy[3]	O(n)	O(n)
Get Item	O(1)	O(n)
Set Item[1]	O(1)	O(n)
Delete Item	O(1)	O(n)
Iteration[3]	O(n)	O(n)

Sets

Definition from maths: a collection of distinct, well-defined objects forming a group (Zermelo-Frankel set theory (**(ZFC)**)

Definition from python: an ordered collection of non-repeated elements.

Designed to be spetially effective in set operations:

- Existence;
- Intersection;
- Union;
- Difference;
- Symmetric difference.

Set creation i

Empty set

» s = set()

» s

{}

» s = {}

» s

{}

Set with elements

» s = {12, 13, 14, 15}

» s

{12, 13, 14, 15}

Set creation ii

A set can contain any type of **hashable** elements

» s = {"R", "C", "A"}

» s = {"R", 13, "D", 12}

Cannot have **non-hashable** elements:

» s = {{1,2,3}, {4,5,6}}
(...)
TypeError: unhashable type: ’set’

Set creation iii

One can also build a set from a list or tuple:

» s = **set (("A", 12, 13))**
» s
$\{''A'', 12, 13\}$

»s = **set ([12, 13, 14, 14])**
» s
$\{12, 13, 14\}$

A set does not have repeated elements! Repeated elements are automatically ignored.

Basic manipulation of sets i

Know the size of the set: function **len**

» len ($\{12, 13, 14\}$)
3

See an element is part of a set: the operator **in**

» 12 in $\{12, 13, 14\}$
True

Basic manipulation of sets ii

Add elements to a set

» **s.add ("razin")**

Remove an element from a set

» **s.remove ("razin")**

Basic manipulation of sets iii

A set does not have an order: it is not a sequence. So it cannot be iterated by a while statement:

```
» s
{12, 13, 14, 15, 16}
» i = 0
» while i < 4:
»       print (s [i])
....
TypeError: 'set' object is not subscriptable
```

Basic manipulation of sets iv

However this can be done with a for instruction

```
» for i in s:
»       print (i)
16
12
13
14
15
```

Set operations i

Instersection

» s.intersection (d)
{13}
» s & d
{13}

Union

» s.union (d)
{10, 11, 12, 13, 14, 15, 16}
» s | d
{10, 11, 12, 13, 14, 15, 16}

Set operations ii

Difference

» **s.difference (d)**
$\{10, 11, 12\}$
» **s - d**
$\{10, 11, 12\}$

This operator is not symmetric! s - d != d - s

Symmetric difference ($\equiv$ s - d | d - s)

» **s.symmetric_difference (d)**
$\{10, 11, 12, 14, 15, 16\}$

Set operations iii

Operation	Average Case	Amortized Worst Case
x $\in$ s	O(1)	O(n)
Union (s|t)	O(len(s)+len(t))	
Intersection ($s \cap t$)	O(min(len(s), len(t)))	O(len(s) * len(t))
Difference (s-t)	O(len(s))	O(len(t))
Symmetric difference (s - t)	O(len(s))	O(len(s) * len(t))

Exercises

Exercise 1. Make a program to eliminate repeated elements from a list, using sets. What's the complexity of the solution?

Exercise 2. Given a list of entities and friendships, identify communities.

Questions?