Programming Fundamentals

List Algorithms

João Correia Lopes

INESC TEC, FEUP

20 November 2018

GOALS

By the end of this class, the student should be able to:

- Be able to explain and implement linear search and binary search
- Describe other algorithms that work with lists
- Compare the performance of those algorithms

BIBLIOGRAPHY

- Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers, How to Think Like a Computer Scientist — Learning with Python 3 (RLE), 2012 (Chapter14) [HTML]
- Brad Miller and David Ranum, Learning with Python: Interactive Edition. Based on material by Jeffrey Elkner, Allen B. Downey, and Chris Meyers (Section 5.3, Section 5.4) [HTML]

TIPS

- There's no slides: we use a script, illustrations and code in the class. Note that this PDF is NOT a replacement for **studying the bibliography** listed in the *class plan*
- "Students are responsible for anything that transpires during a class—therefore **if you're not in a class**, you should get notes from someone else (not the instructor)"—David Mayer
- The best thing to do is to **read carefully** and **understand** the documentation published in the Content wiki (or else **ask** in the recitation class)
- We will be using **Moodle** as the primary means of communication

CONTENTS

LIST ALGORITHMS

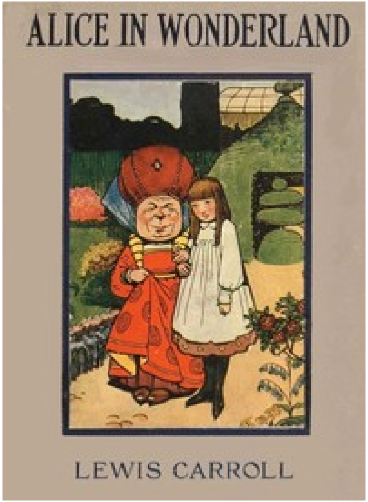
MPFC:

And now for something completely different!

- Rather than introduce more programming constructs, or new Python syntax and features
- . . . we focus on some algorithms that work with lists

ALICE IN WONDERLAND

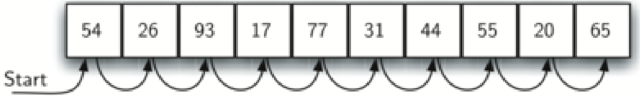
- Examples work with the book "Alice in Wonderland" and a "vocabulary file"



⇒ Alice in Wonderland
⇒ Vocabulary

THE LINEAR SEARCH ALGORITHM

- We'd like to know the index where a specific item occurs within in a list of items
- Specifically, we'll return the index of the item if it is found, or we'll return -1 if the item doesn't occur in the list



⇒ InteractivePython
⇒ https://github.com/fpro-admin/lectures/blob/master/15/linear.py

LINEAR SEARCH

```
def search_linear(xs, target):
    """ Find and return the index of target in sequence xs """
    for (i, v) in enumerate(xs):
        if v == target:
            return i
    return -1
```

- Searching all items of a sequence from first to last is called a **linear search**
- Each time we check whether `v == target` we'll call it a **probe**
  - We like to count probes as a measure of how efficient our algorithm is
  - this will be a good enough indication of how long our algorithm will take to execute
- Linear searching is characterized by the fact that the number of probes needed to find some target depends directly on the length of the list
- On average, when the target is present, we're going to need to go about halfway through the list, or N/2 probes

LINEAR SEARCH

```
def search_linear(xs, target):
    """ Find and return the index of target in sequence xs """
    for (i, v) in enumerate(xs):
        if v == target:
            return i
    return -1
```

- Searching all items of a sequence from first to last is called a **linear search**
- Each time we check whether `v == target` we'll call it a **probe**
  - We like to count probes as a measure of how efficient our algorithm is
  - this will be a good enough indication of how long our algorithm will take to execute
- Linear searching is characterized by the fact that the number of probes needed to find some target depends directly on the length of the list
- On average, when the target is present, we're going to need to go about halfway through the list, or N/2 probes

LINEAR PERFORMANCE

- We say that linear search has linear performance (linear meaning *straight line*)
- Analysis like this is pretty meaningless for small lists
  - The computer is quick enough not to bother if the list only has a handful of items
- So generally, we're interested in the **scalability** of our algorithms
  - How do they perform if we throw bigger problems at them
  - What happens for really large datasets, e.g. how does Google search so brilliantly well?

A MORE REALISTIC PROBLEM

- As children learn to read, there are expectations that their vocabulary will grow
- So a child of age 14 is expected to know more words than a child of age 8
- When prescribing reading books for a grade, an important question might be **"which words in this book are not in the expected vocabulary at this level?"**
  - Let us assume we can read a vocabulary of words into our program
  - Then read the text of a book, and split it into words

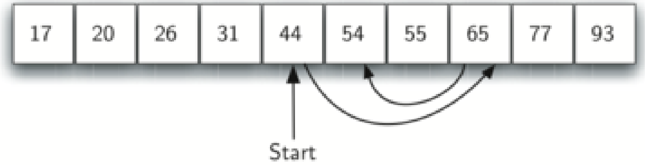
⇒ https://github.com/fpro-admin/lectures/blob/master/15/alice.py

BINARY SEARCH

- If you were given a vocabulary and asked to tell if some word was present, you'd probably start in the middle
- You can do this because the vocabulary is ordered — so you can probe some word in the middle, and immediately realize that your target was before (or perhaps after) the one you had probed
- Applying this principle repeatedly leads us to a very much better algorithm for searching in a list of items that are already ordered
- This algorithm is called **binary search**
- It is a good example of *divide and conquer*

REGION OF INTEREST (ROI)

- Our algorithm will start with the ROI set to all the items in the list
- On the first probe in the middle of the ROI, there are three possible outcomes:
  - either we find the target
  - or we learn that we can discard the top half of the ROI
  - or we learn that we can discard the bottom half of the ROI
- Trying with 54. . .



⇒ InteractivePython
⇒ https://github.com/fpro-admin/lectures/blob/master/15/binary.py

BACK TO "A MORE REALISTIC PROBLEM"

- What a spectacular difference! More than 200 times faster!
- If we uncomment the print statement on lines 15 and 16, we'll get a trace of the probes done during a search

⇒ https://github.com/fpro-admin/lectures/blob/master/15/alice.py

REMOVING ADJACENT DUPLICATES FROM A LIST

- We often want to get the unique elements in a list, i.e. produce a new list in which each different element occurs just once
- Consider our case of looking for words in Alice in Wonderland that are not in our vocabulary
- We had a report that there are 3398 such words, but there are duplicates in that list
- In fact, the word “alice” occurs 398 times in the book, and it is not in our vocabulary!
- How should we remove these duplicates?
- A good approach is to sort the list, then **remove all adjacent duplicates**

⇒ https://github.com/fpro-admin/lectures/blob/master/15/adjacent.py

REMOVING ADJACENT DUPLICATES FROM A LIST

- We often want to get the unique elements in a list, i.e. produce a new list in which each different element occurs just once
- Consider our case of looking for words in Alice in Wonderland that are not in our vocabulary
- We had a report that there are 3398 such words, but there are duplicates in that list
- In fact, the word “alice” occurs 398 times in the book, and it is not in our vocabulary!
- How should we remove these duplicates?
- A good approach is to sort the list, then **remove all adjacent duplicates**

⇒ `https://github.com/fpro-admin/lectures/blob/master/15/adjacent.py`

BACK TO "A MORE REALISTIC PROBLEM"

- Let us go back now to our analysis of *Alice in Wonderland*
- Before checking the words in the book against the vocabulary, we'll sort those words into order, and eliminate duplicates
- Lewis Carroll was able to write a classic piece of literature using only 2570 different words!

⇒ `https://github.com/fpro-admin/lectures/blob/master/15/alice2.py`

MERGING SORTED LISTS

- Suppose we have two sorted lists (`xs` and `yx`)
- Devise an algorithm to merge them together into a single sorted list
- A **simple but inefficient** algorithm could be to simply append the two lists together, and sort the result
- But this doesn't take advantage of the fact that the two lists are already sorted
  - It is going to have poor scalability and performance for very large lists

```
    newlist = (xs + ys)
    newlist.sort()
```

⇒ https://github.com/fpro-admin/lectures/blob/master/15/merge.py

MERGING SORTED LISTS

- Suppose we have two sorted lists (`xs` and `yx`)
- Devise an algorithm to merge them together into a single sorted list
- A **simple but inefficient** algorithm could be to simply append the two lists together, and sort the result
- But this doesn't take advantage of the fact that the two lists are already sorted
  - It is going to have poor scalability and performance for very large lists

```
newlist = (xs + ys)
newlist.sort()
```

⇒ https://github.com/fpro-admin/lectures/blob/master/15/merge.py

BACK TO "A MORE REALISTIC PROBLEM"

- Let us go back now to our analysis of *Alice in Wonderland*
- Previously we sorted the words from the book, and eliminated duplicates
- Our vocabulary is also sorted
- So, find all items in the second list that are not in the first list, would be another way to implement `find_unknown_words`

⇒ `https://github.com/fpro-admin/lectures/blob/master/15/alice3.py`

SUMMARY

- Let's review what we've done.
  - We started with a word-by-word **linear lookup** in the vocabulary that ran in about 50 seconds
  - We implemented a clever **binary search**, and got that down to 0.22 seconds, more than 200 times faster
  - But then we did something even better: we sorted the words from the book, eliminated duplicates, and used a **merging pattern** to find words from the book that were not in the dictionary; this was about five times faster than even the binary lookup algorithm
  - At the end, our algorithm is more than a 1000 times faster than our first attempt!

EXERCISES

- Moodle activity at: LE15: List Algorithms