Lab 09: Sorting and Searching

Contents

1 **Sorting** 1

2 `datetime` 2

3 **Binary Search** 3

4 **Putting It All Together** 4

5 **Handing In** 5

Objectives

By the end of this lab you will understand:

- How different ways of operating affect efficiency
- The importance of searching and sorting in data processing

By the end of this lab you will be able to:

- Sort a complex list based on its underlying components
- Use the `datetime` module
- Search through an ordered list in a more efficient manner

1 Sorting

While it may seem fairly mundane, *sorting* a list (i.e., putting its elements into some defined order) turns out to be a very well researched problem in computing,[1] and the algorithms for it are elegant examples of the divide-and-conquer strategy we saw in Lab 06. These algorithms are also the first examples we've seen that run in time $n \cdot \log(n)$, where n is the number of elements in the list.

This module won't go into any detail in terms of how sorting works, because of course the easiest way to perform sorting is to just use the built-in Python function. Indeed, we already saw all the way back in Lab 02 how to sort a list `mylist` using `mylist.sort()`. As a reminder, this is a mutation (meaning it alters the list directly, and its output should not be assigned to any variable); we can also obtain a sorted copy of a list using `sorted(mylist)`. It is worth mentioning, however, that custom sorting algorithms become essentially when dealing with truly large datasets, such as in digital forensics, DNA analysis, and social network analysis.

The simple sorting function works well for lists with simple atomic types, like integers and strings. To sort a list whose entries are other containers (e.g., lists or dictionaries), however, it is often necessary to tell the sorting function explicitly which part of the entry we want to use to sort. For example, consider the following list of pairs:

```
>>> mylist = [('a', 100), ('b', 240), ('c', 96), ('d', 54), ('e', 72)]
```

This list is already sorted according to the first value in the pair (the letter), so running `mylist.sort()` will not change it. We may, however, wish to sort the list according to the second value in the pair (the integer) instead. To do this, we can provide a *key* that points to the value we want to use to sort. This would mean running:

[1] `https://en.wikipedia.org/wiki/Sorting_algorithm`

```
>>> mylist.sort(key = lambda x : x[1])
```

This tells `sort` that for each value `x` in the list, we want to sort according to `x[1]`, which is the integer. If we run this code, we'll then get

```
>>> mylist.sort(key = lambda x : x[1])
>>> mylist
[('d', 54), ('e', 72), ('c', 96), ('a', 100), ('b', 240)]
```

We could similarly use keys to indicate specific entries in a dictionary that we want to use to sort lists of dictionaries. Finally, these keys also work in other built-in functions in Python, such as `min` and `max`. For example, to get the key-value pair in a dictionary `mydict` such that the value is the maximum value in the dictionary, we could run

```
(max_key,max_val) = max(mydict.items(), key = lambda x : x[1])
```

2 datetime

In terms of orderings, there are two types we could probably name immediately: numerical ordering over integers, and alphabetical ordering over strings. Another type of ordering that is particularly important in data science is *chronological* ordering.

Thus far, we have not seen any nice way to represent dates in Python. To do this, and in particular to provide a good way to compare dates and times to determine a chronological ordering, we can use the `datetime` module. Beyond providing chronological orderings, this module is used very often in computer science, such as for seeing how long a program takes or when it started running. The best way to import this module (or really to import one of its most useful submodules) is to run `from datetime import datetime as dt`. Within this module, there are several functions that could be useful; these are:

- `dt.now()` returns a `datetime` object representing the exact date and time at which the function was called.
- `dt.strptime(date_str, date_format)` takes a string representing a date and a string indicating the format in which the date is represented, and returns a `datetime` object for that date and time. For example, `mydate = dt.strptime('23/04/2015', '%d/%m/%Y')` can be used to represent the date 23 April 2015.
- `dt.fromtimestamp(time_int)` takes a Unix timestamp and returns a `datetime` object for the date and time it represents. For example, `dt.fromtimestamp(1512132154)` can be used to represent the date 1 December 2017 and time 12:42, since this is what the timestamp represents.
- `date.strftime(date_format)` can be called by a `datetime` object `date`, and returns the value represented by that object in the format given by the string. For example, for `mydate` defined as above, `mydate.strftime('%d/%m/%Y')` returns the string '23/04/2015' and in general `dt.strptime(date_str, date_format).strftime(date_format)` always returns the string `date_str`.
- `date.day` can be called by a `datetime` object, and returns the day it represents. For example, `mydate.day` returns 23.
- `date.month` can be called by a `datetime` object, and returns the month it represents (as a number). For example, `mydate.month` returns 4.
- `date.year` can be called by a `datetime` object, and returns the year it represents. For example, `mydate.year` returns 2015.
- `date1 < date2` will return a boolean indicating whether or not `date1` is earlier than `date2`; we can similarly perform all other boolean operations (e.g., `==` and `!=`). For example, `mydate < dt.strptime('25/04/2015', '%d/%m/%Y')` returns `True`.

3 Binary Search

The problem of finding an element in a list, while again seemingly a mundane task, is arguably even more well studied than the problem of sorting, and is again essential when dealing with truly large datasets. The algorithm we look at today is called *binary* search,[2] thus named because it divides the list in half at every step. This is another elegant example of a divide-and-conquer algorithm, and it runs in time log(n) (where again n is the number of elements in the list).

Before we get to binary search, let's consider and implement a simple *linear* search algorithm (thus named because it runs in time n) that just iterates through the list in order until it finds the element it wants.

Task 1. In a file `utilities.py`, write a function `lin_find` that, given an integer `e` and a list of integers `data`, uses a simple `for` loop to find and return that entry in the list, if it is there. Add to this function some code for counting the number of times it goes through the loop, so that in addition to returning the matching entry it also prints out "Number of times with linear search is 5" (where 5 is just an example).

Solution 1. The version in `utilities.py` is the version updated for Task 4. The uncommented version that works just for lists of integers is:

```
def lin_find(e,data):
    count = 0
    for entry in data:
        if entry == e:
            print ‘number of times with linear search is\t’, count
            return entry
        count += 1
    print ‘number of times with linear search is\t’, count
    return None
```

In contrast to linear search, binary search works only if the list is ordered. To find an element, it begins by picking some *pivot* element; typically, this is the element in the middle of the list (so at the index `len(mylist) / 2`). It then examines this element, and decides — according to the ordering — if the element it's looking for comes before or after this element. If it comes before, then we know we do not need to look at all at the later half of the list, since the element definitely won't be there. Similarly, if it comes after, then we know we do not need to look at all at the earlier half of the list. We have thus reduced the problem of finding the element in the entire list to the problem of finding it in the relevant half of the list.

In every step of the algorithm, we thus pick a pivot element at the midpoint of the current list, and then examine that element to decide which half of the list to keep: the earlier half or the later half. We then update the current list with the relevant half and continue. If we ever find the matching element, we return it immediately. If we ever get to the point where there is no list left (i.e., our current list is the empty list), then we know the element must not be in the list, so return an appropriate value (e.g., we could return `None`).

For example, consider

```
mylist = [1, 2, 3, 4, 5, 7, 9, 10, 13, 15, 16, 19, 20]
```

Let's say we are trying to find the number 13 in this list. We go through the following steps:

1. In the first step, we want to consider the whole list. We can think of this as initializing two indices, `first` and `last`, where `first` is the value 0 (indicating we're starting at the beginning of the list) and `last` is `len(mylist)-1` (indicating we're ending at the end of the list). We pick as a pivot the element at the midpoint of the list; i.e., the element at index `(first + last) / 2`, which for these values of `first` and `last` is 6. This means we look at `mylist[6]`, which is 9. Because $13 > 9$, we now know that the only possible place for 13 in `mylist` — crucially, because the list is ordered — is the later half. We can thus update `first` to be one more than the index of the pivot element, which is $6 + 1 = 7$.

[2] https://en.wikipedia.org/wiki/Binary_search_algorithm

2. In the second step, we now have that `first = 7` and `last = 12`. We then identify the midpoint between these, which is 9, and examine the value `mylist[9]`, which is 15. Since $13 < 15$, we know that we need to look in the earlier half of this part of the list. This means that, rather than move `first` up, we move `last` down to be one less than the index of the pivot element, which is $9 - 1 = 8$.
3. In the third step, we now have that `first = 7` and `last = 8`. We again identify the midpoint between these, which is 7, and examine the value `mylist[7]`, which is 10. Since $13 > 10$, we move `first` up, as we did in the first step, to be one more than the index of the pivot, which is $7 + 1 = 8$.
4. In the fourth and final step, we now have that `first = 8` and `last = 8`. We again identify the midpoint between these, which is 8, and examine the value `mylist[8]`, which is 13. Since $13 = 13$, we return this element.

By using binary search, we were thus able to find the element in four steps. In contrast, linear search would require nine steps, as 13 is the ninth element in the list.

Task 2. In `utilities.py`, write a function `bin_find` that takes in the same inputs and produces the same output as `lin_find`, but looks for the matching entry using binary search. Again, add to this function code for counting and printing out the number of times it goes through a loop.

Solution 2. Again, the version in `utilities.py` is the version updated for Task 4. The uncommented version that works just for lists of integers is:

```
def bin_find(date,data):
    count = 0
    first = 0
    last = len(data) - 1
    while first <= last:
        middle = (first + last) / 2
        entry = data[middle]
        if entry == e:
            print ‘number of times with binary search is\t’, count
            return entry
        else:
            if e < entry:
                last = middle - 1
            else:
                first = middle + 1
        count += 1
    print ‘number of times with binary search is\t’, count
    return None
```

4 Putting It All Together

Remember the ShapeShift scraper you wrote in Lab 07? Let's imagine that you actually wrote it in October, and that you'd left it running continuously since then. In that case, you'd have scraped over 600,000 transactions by now. Luckily, we don't have to imagine this, since my research group has been doing exactly that and have this data available. For the purposes of this lab, we have made 1,000 random transactions available in a file `data/shapeshift.csv`. Please see Lab 07 for a reminder of the structure of this data.

Task 3. In a file `shapeshift.py`, open the CSV file and parse it as a list of dictionaries. These transactions are associated with a Unix timestamp, but the order they appear in the file is not in order of this timestamp. So, begin by creating a version of the data ordered by timestamp, where the timestamp is a `datetime` object. (You can either order the data first and then convert the timestamp, or convert it first and then order the data; you may find the second option slightly easier).

Solution 3. The full solution is in `shapeshift.py`. The relevant (uncommented) lines of code, assuming `prefix` points to the directory where the CSV file is, are:

```
f = open(prefix + ‘shapeshift.csv’,‘r’)
r = csv.DictReader(f,delimiter=‘|’)
spreadsheet = []
for row in r:
    date = dt.fromtimestamp(int(row[‘timestamp’]))
    row[‘date’] = date
    spreadsheet.append(row)

spreadsheet.sort(key = lambda x : x[‘date’])
```

Now that we have the data in order, we'd like to search for transactions that occurred on specific days. The search code we wrote before, however, won't work here, as it looks for an integer in a list of integers, but we want to find a dictionary such that a specific value in that dictionary is equal to a date.

Task 4. Update the code for both `lin_find` and `bin_find` so that they take in a value `date` (as a `datetime` object) and a value `data` (as a list of dictionaries), and return an entry that happened on that date (or `None` if no such entry exists). Now that the functions are compatible with the dataset, import `utilities` in `shapeshift.py` and test them on three dates: (1) 5 October 2017, (2), 12 November 2017, and (3) 2 December 2017. Make sure that for each case your code prints out the number of steps run by both linear search and binary search. Is the number of steps taken by linear search ever lower than for binary search? If so, explain in a short comment why this is the case.

Solution 4. The solution is in `shapeshift.py`.

The nice thing about writing utility functions like binary search is that, once they're done, we can reuse them across many different datasets. To show how easy it is to do this, let's turn our attention to our favorite metal theft dataset. As a reminder, these thefts took place between April 2009 and June 2013.

Task 5. Create a file `church_theft.py` that parses the church metal thefts into a list of dictionaries, or copy your or my solution from Lab 07. As test cases, use both search functions to look for thefts that occurred on four dates: (1) 26 February 2011, (2) 6 July 2009, (3) 22 April 2013, and (4) 15 November 2010. Continue to make sure that your code prints out the number of steps taken by each function. Again, is the number of steps taken by linear search ever lower than for binary search? If so, explain in a short comment why this is the case.

Solution 5. The solution is in `church_theft.py`.

5 Handing In

As always, feel free to continue trying out the various concepts we've covered in the lab. Once you are done, add the files into a `lab09` directory; this should mean adding:

- `utilities.py`
- `shapeshift.py`
- `church_theft.py`

Push these files, along with the usual `partner.txt`, to the remote repository, and you're done! Please check with one of us on your way out.