

# Algorithms and Data Structures

## Self-Review Questions

---

**Self-review 10.1** Why are data structures other than sequence and dictionary useful?

Because they have properties that are different and many algorithms need data structures that have properties different to those of sequence and dictionary.

**Self-review 10.2** Why is Binary Search better than Linear Search?

Because the performance of Binary Search is  $O(\log_2 n)$  whereas the performance of Linear Search is  $O(n)$ .

**Self-review 10.3** Why is it not always possible to use Binary Search for all searches in sequences?

Because the Binary Search algorithm requires the data in the sequence to be sorted.

**Self-review 10.4** What alternatives to sequence are there for storing data to be searched?

The data could be in a dictionary, in which case the search is  $O(1)$ , or in a search tree in which case the performance is  $O(\log_2 n)$ . Dictionaries are clearly the best solution. The exception is where the data has to be held in a sorted order.

**Self-review 10.5** What is a comparator?

It is a predicate (a Boolean function) that compares values and imposes a total order on the set of values.

**Self-review 10.6** Why is Big O notation useful?

It gives us a tool for measure performance of algorithms and software and so gives us a tool for comparing.

**Self-review 10.7** Why might we want to sort a list?

Two possible reasons:

1. The data is required to be sorted for some external reason, i.e. data has to be printed out in sorted order.
2. An algorithm requires use of a sequence rather than a dictionary, and the data has to be sorted.

**Self-review 10.8** Why are 'divide and conquer' algorithms important?

The time complexity of 'divide and conquer' algorithms is invariably better than any linear algorithm. Binary Search ( $O(\log_2 n)$ ) is better than Linear Search ( $O(n)$ ). Quicksort and Merge Sort ( $O(n \log_2 n)$ ) are better (at least in general) than Bubble Sort ( $O(n^2)$ ).

Of course, 'divide and conquer' should not be used if 'direct access' ( $O(1)$ ) methods are available. So don't use sequences if dictionaries can do the job.

**Self-review 10.9** Why investigate tree data structures at all when we have sequences and dictionaries?

Dictionaries are implemented using open hash tables which give fine performance but the data is not stored in a sorted form. There are map-related algorithms that can happily suffer decreased performance for access and update because the need is for the data to be sorted. For these algorithms maps implemented using trees are very important.

**Self-review 10.10** What is output when the following code is executed?

```
a = [ 1 , 2 , 3 , 4 ]
b = a
b[0] = 100
print a
```

What is output when the following code is executed?

```
a = [ 1 , 2 , 3 , 4 ]
b = a
b = a + [ 5 , 6 , 7 ]
b[0] = 100
print a
```

Explain why in the first code sequence there is a change to the list referred to by a when there is a change to b, but in the second code sequence there is not.

In the first code sequence a and b refer to the same list, i.e. the list object is shared by two variables. In the second code sequence b refers to a different list object.

**Self-review 10.11** Is an algorithm that has time complexity  $O(n)$  faster or slower than one that has time complexity  $O(\log_2 n)$ ? How does a time complexity of  $O(n \log_2 n)$  compare to them?

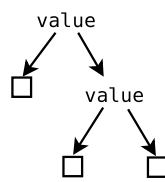
**Self-review 10.12** In the recursive version of Quicksort, what are the base cases and recursive cases of the algorithm?

**Self-review 10.13** When is it better to use Bubble Sort rather than Quicksort?

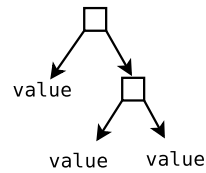
**Self-review 10.14** The following class represents a tree structure in Python.

```
class MyTree :
    def __init__( self , value , left , right ) :
        self.value = value
        self.left = left
        self.right = right
```

Which of the figures below correctly represents trees constructed with this class?



Tree 1



Tree 2

**Self-review 10.15** What is the *root node* of a tree data structure?

## Programming Exercises

---

**Exercise 10.1** Extend the unit test program for testing the search functions so that its coverage is far better.

**Exercise 10.2** Extend the unit test program for testing the sort functions so that its coverage is far better.

**Exercise 10.3** Create a quicker Quicksort by extending the implementation in this chapter so that Bubble Sort is used instead of a recursive call of Quicksort when the parameter is a sequence with less than 10 items.

**Exercise 10.4** Create an improved Quicksort implementation by changing the algorithm for selecting the pivot so that the two sub-sequences created by the partition are approximately the same length.

**Exercise 10.5** This exercise is about using the ideas from the binary search algorithm to play a simple “20 Questions” type game. The (human!) player thinks of a number between 0 and 100. The program asks the player a series of yes/no questions. The first question will be “Is the number between 0 and 50?”, the rest follows the binary chop approach.

What is the maximum number of questions the program will have to ask the player in order to obtain the correct answer?

**Exercise 10.6** Different comparators can be used to sort different kinds of data. Write comparators implementing the following comparisons and test them by using them to sort some appropriate data:

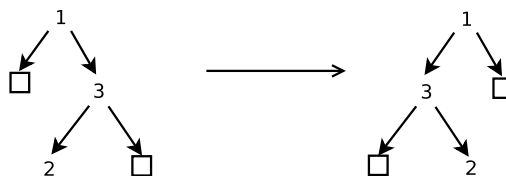
1. A lexicographical comparator – i.e. one that compares two strings based on their alphabetic ordering.
2. An ASCII comparator that can be used to sort characters based on their ASCII value. *Hint: remember the `ord` function!*
3. A comparator that can be used to sort names based on their surname. This comparator should use strings of the form ‘Sarah Mount’, ‘James Shuttleworth’, ‘Russel Winder’, and ‘James T Kirk’ and compare them alphabetically on the *last* word in the string.

**Exercise 10.7** When deciding what sorting algorithm to use on a list, it is useful to find out whether a list is nearly sorted. There are several ways to do this. Write functions which implement the following measures – if any of the functions return 0 the list is already sorted. Make sure that you compare the different functions on a range of sorted and unsorted data:

1. Return 1 if the list is unsorted.
2. Consider every adjacent pair of list elements. Add one to the ‘unsortedness’ score for every pair which is unsorted.
3. Find the longest sorted sub-sequence of elements in the list. Return the length of the list minus this number. For example, if the list is [2, 9, 3, 5, 6, 7, 1, 4], the longest sorted sub-sequence is 3, 5, 6, 7, which is of length 4. So your function should return 8-4 which is 4.

**Exercise 10.8** In Section ?? we introduced the built-in class set which efficiently implements sets (using hash tables). This class contains many useful set operations such as intersection, union and difference, but it does not have a Cartesian product (aka cross product) operation. The Cartesian product of two sets is the set of all pairs created by taking an item from the first set and an item from the second set. For example if we have the set { 1, 2 } and the set { 'a', 'b' } then the Cartesian product is the set { ( 1, 'a' ), ( 1, 'b' ), ( 2, 'a' ), ( 2, 'b' ) }. For this exercise write a subclass of set that provides a method for calculating Cartesian products.

**Exercise 10.9** Subclass the OrderedBinaryTree class from Section ?? to create a class that has a method reflect which exchanges the left and right subtrees of each branch, like this:



**Exercise 10.10** The timeit module can be used to time the execution of code. For example, the following interpreter session shows that the built-in sort method for Python lists takes 1 s to sort the list [ 1, 2, 3, 4, 5 ] one million times on the machine we tried it on:

```

>>> import timeit
>>> t = timeit.Timer ( stmt = '[ 1, 2, 3, 4, 5 ].sort ( )' )
>>> t.timeit ( 1000000 ) # Run the statement 1000000 times.
0.99292302131652832
>>>
  
```

Use timeit to compare how long the various sorting algorithms covered in this chapter take to sort a given list.

## Challenges

---

**Challenge 10.1** Create a type that has the same methods as a dictionary but which uses an ordered binary tree to store the data.

**Challenge 10.2** Other well-known sorting algorithms are:

- Insertion Sort
- Shell Sort
- Heapsort
- Radix Sort

Research these algorithms, extend the unit test program to test implementations of these algorithms, then add implementations of these algorithms.