**Sorting, searching and algorithm analysis**

**Overview**

|  |
| --- |
| In this chapter we will analyze four algorithms, two for each of the following tasks. Ordering a list of values, and finding the position of a value within a sorted list. Algorithm analysis should begin with a clear statement of the task to be performed. This allows us to both check that the algorithm is correct and ensure that the algorithms we are comparing perform the same task. |

We have learned that in order to write a computer program that performs some task, we must construct a suitable algorithm (set of instructions to solve a specific problem). However, some algorithms are in any sense better than others. Knowing how your input looks is critical. It can determine which algorithm you might have to use.

Although in general, there are many ways that algorithms might be compared, we will focus our attention on the two that are primary importance to many data processing algorithms:

* **Time complexity** (how the number of steps required depends on the length of the input)
* **Space complexity** (how the amount of extra memory or storage required depends on the length of the input)

**Sorting algorithms**

Sorting a list of values is a common computation task that has been well studied. The classic description of the task is as follows:

* Given a list of values and a function that compares two values, order the values in the list from smallest to largest. For example, you want to find out who is the highly paid employee or intern in the company, sorting the results and taking the max will be the ideal solution.

The values might be integers, or strings or even other kinds of objects. We examine two algorithms for now, there are others which you can explor later on:

* **Selection sort** (which relies on repeatedly selecting the next smallest item), and
* **Merge sor**t (which relies on repeatedly merging sections of the list that are already sorted)

Other well-known algorithms for sorting lists are Insertion sort, Bubble sort, Heap sort, Quicksort and Shell sort. You can check wikipedia for more information. See this [link](https://en.wikipedia.org/wiki/Sorting_algorithm).

There are also a variety of algorithms which perform the sorting task for restricted kinds of values, for example:

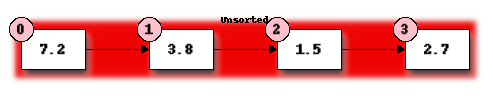
* Counting sort (relies on values all belonging to a small set of items)
* Bucket sort (relies on being able to map each value to one of a small set of items)
* Radix sort (relies on values being sequences of digits)

|  |
| --- |
| **Note**: The common sorting and searching algorithms are widely implemented and already available for most programming languages. You will seldom have to implement them yourself outside of the exercises in these notes. Nevertheless, understanding these algorithms is still important since you will likely be making use of them within your own programs and their space and time complexity will affect that of your program. You may also be required to select which sorting or searching algorithm to use which will require a good understanding of the characteristics of the algorithms available. |

**Selection sort**

Selection sort orders a given list by repeatedly selecting the smallest remaining element and moving it to the end of a growing sorted list.

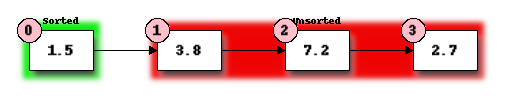
To illustrate selection sort, let us examine how it operates on a small list of four elements:



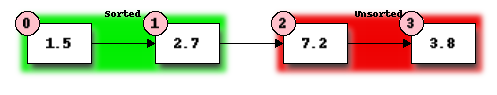
Initially the entire list is unsorted. We will use the front of the list to hold the sorted items in order to avoid using extra storage space but at the start this sorted list is empty.

First we must find the smallest element in the portion of the list. We take the first element of the unsorted list as a candidate and compare it to each of the following elements in turn, replacing our candidate with any element found to be smaller. This requires 3 comparisons and we find that element 1.5 at position 2 is smallest.

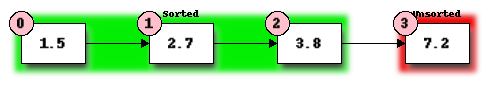
Now we will swap the first element of our unordered list with the smallest element to start our ordered list:

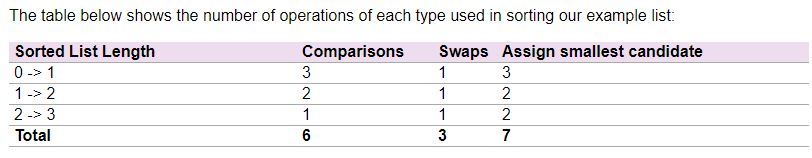


We now repeat our previous steps, determining that 2.7 is the smallest remaining element and swap it with 3.8, the first element of the current unordered section, to get:



Finally, we determine that 3.8 is the smallest of the remaining unordered elements and swap it with 7.2:





Note that the number of comparisons and the number of swaps are independent of the contents of the list (this is true for selection sort but not necessarily for other sorting algorithms) while the number of times we have to assign a new value to the smallest candidate depends on the contents of the list.

More generally, the algorithm for selection sort is as follows:

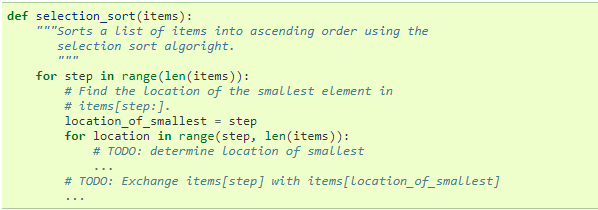
1. Divide the list to be sorted into a sorted portion at the front (initially empty) and an unsorted portion at the end (initially the whole list).
2. Find the smallest element in the unsorted list:
3. Select the first element of the unsorted list as the initial candidate.
4. Compare the candidate to each element of the unsorted list in turn, replacing the candidate with the current element if the current element is smaller.
5. Once the end of the unsorted list is reached, the candidate is the smallest element.
6. Swap the smallest element found in the previous step with the first element in the unsorted list, thus extending the sorted list by one element.
7. Repeat the steps 2 and 3 above until only one element remains in the unsorted list.

The selection sort has an O(n^2) time complexity, which makes it inefficient on a large list. (**hint**: having nested two for loops usually (not always) results in O(n^2)).

|  |
| --- |
| **Note**: The selection sort algorithm as described above has two properties which are often desirable in sorting algorithm.  The first is that the algorithm is **inplace**. This means that it uses essentially no extra storage beyond what is required for the input.  The second is that the sorting algorithm is **stable**. The means that two elements which are equal, retain their initial relative ordering. |

**Task 1**

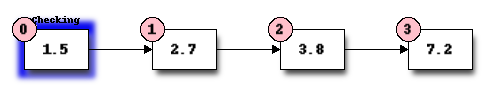
Complete the following that will perform a selection sort in Python. “...” denotes missing code that should be filled in:



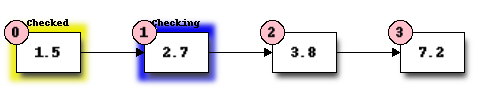
**Searching algorithms**

**Linear search**

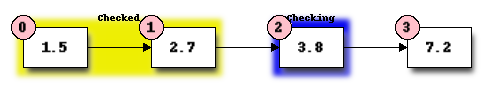
A linear search is the most basic kind of search method. It involves checking each element of the list in turn, until the desired element is found.

For example, suppose that we want to find the number 3.8 in the following list:

We start with the first element, and perform a comparison to see if its value is the value that we want. In this case, 1.5 is not equal to 3.8, so we move onto the next element:



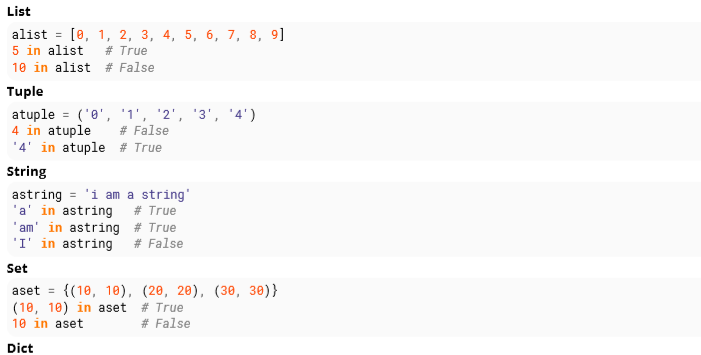
We perform another comparison, and see that 2.7 is also not equal to 3.8, we move onto the next element:



We perform another comparison and determine that we have found the correct element. Now we can end the search and return the position of the element (index 2).

Linear search runs in at most linear time and makes at most n comparison, where n is the length of the list.

Python has a built-in (**in**) which allows you to search if an item is in a collection. See examples on the next page. For an in depth usage, see Section 71.1 on the module notes.



**Task 2**

Write a function which implements linear search. It should take a list and an element as a parameter, and return the position of the element in the list. If the element is not in the list, the function should return **None.** If the element is in the list multiple times, the function should return the first position.