Sorting, searching and algorithm analysis

Introduction

We have learned that in order to write a computer program which performs some task we must construct a suitable algorithm. However, whatever algorithm we construct is unlikely to be unique – there are likely to be many possible algorithms which can perform the same task. Are some of these algorithms in some sense better than others? Algorithm analysis is the study of this question.

In this chapter we will analyse four algorithms; two for each of the following common tasks:

- *sorting*: ordering a list of values
- *searching*: finding the position of a value within a list

Algorithm analysis should begin with a clear statement of the task to be performed. This allows us both to check that the algorithm is correct and to ensure that the algorithms we are comparing perform the same task.

Although there are many ways that algorithms can be compared, we will focus on two that are of primary importance to many data processing algorithms:

- *time complexity*: how the number of steps required depends on the size of the input
- *space complexity*: how the amount of extra memory or storage required depends on the size of the input

Note

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. It is nevertheless important for you to understand these basic algorithms, because you are likely to use them within your own

programs – their space and time complexity will thus affect that of your own algorithms. Should you need to select a specific sorting or searching algorithm to fit a particular task, you will require a good understanding of the available options.

Sorting algorithms

The sorting of a list of values is a common computational task which has been studied extensively. 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.

The values might be integers, or strings or even other kinds of objects. We will examine two algorithms:

- *Selection sort*, which relies on repeated *selection* of the next smallest item
- *Merge sort*, which relies on repeated *merging* of 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*.

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

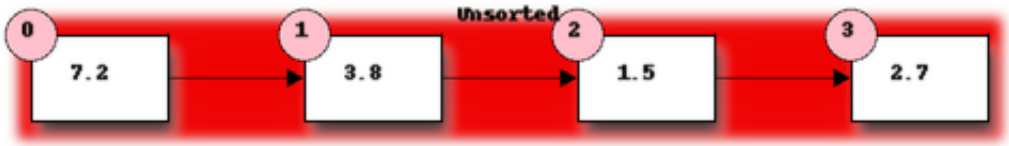
- *Counting sort*, which relies on the values belonging to a small set of items
- *Bucket sort*, which relies on the ability to map each value to one of a small set of items
- *Radix sort*, which relies on the values being sequences of digits

If we restrict the task, we can enlarge the set of algorithms that can perform it. Among these new algorithms may be ones that have desirable properties. For example, *Radix sort* uses fewer steps than any generic sorting algorithm.

Selection sort

To order a given list using selection sort, we repeatedly select the smallest remaining element and move it to the end of a growing sorted list.

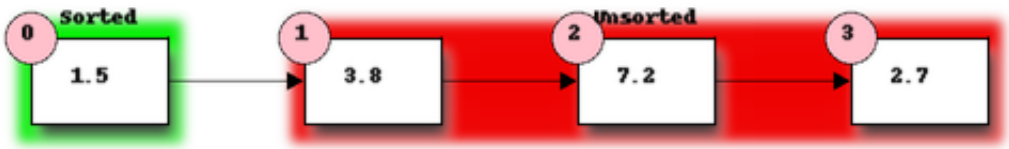
To illustrate selection sort, let us examine how it operates on a small list of four elements:



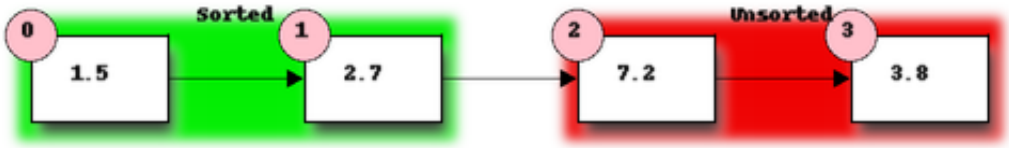
Initially the entire list is unsorted. We will use the front of the list to hold the sorted items – to avoid using extra storage space – but at the start this sorted list is empty.

First we must find the smallest element in the unsorted 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.

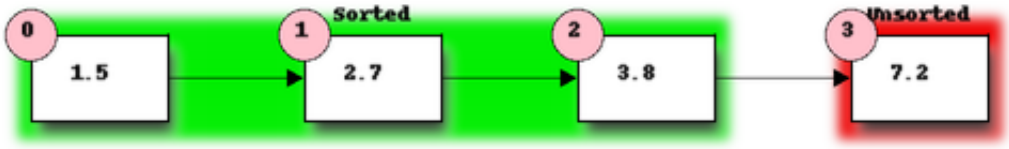
Now we will swap the first element of our unordered list with the smallest element. This becomes the start of our ordered list:



We now repeat our previous steps, determining that 2.7 is the smallest remaining element and swapping 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:



The table below shows the number of operations of each type used in sorting our example list:

Sorted List Length	Comparisons	Swaps	Assign smallest candidate
0 -> 1	3	1	3
1 -> 2	2	1	2
2 -> 3	1	1	2
Total	**6**	**3**	**7**

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:

    1. Select the first element of the unsorted list as the initial candidate.
    2. 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.
    3. Once the end of the unsorted list is reached, the candidate is the smallest element.

3. 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.
4. Repeat the steps 2 and 3 above until only one element remains in the unsorted list.

Note

The *Selection sort* algorithm as described here has two properties which are often desirable in sorting algorithms.

The first is that the algorithm is *in-place*. This means that it uses essentially no extra storage beyond that required for the input (the unsorted list in this case). A little extra storage may be used (for example, a temporary variable to hold the candidate for the smallest element). The important property is that the extra storage required should not increase as the size of the input increases.

The second is that the sorting algorithm is *stable*. This means that two elements which are equal retain their initial relative ordering. This becomes important if there is additional information attached to the values being sorted (for example, if we are sorting a list of people using a comparison function that compares their dates of birth). Stable sorting algorithms ensure that sorting an already sorted list leaves the order of the list unchanged, even in the presence of elements that are treated as equal by the comparison.

Exercise 1

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

```
def selection_sort(items):
    """Sorts a list of items into ascending order using the
       selection sort algoright.
       """
    for step in range(len(items)):
        # Find the location of the smallest element in
        # items[step:].
        location_of_smallest = step
        for location in range(step, len(items)):
            # TODO: determine location of smallest
            ...
        # TODO: Exchange items[step] with items[location_of_smallest]
        ...
```

Exercise 2

Earlier in this section we counted the number of *comparisons*, *swaps* and *assignments* used in our example.

1. How many swaps are performed when we apply selection sort to a list of N items?
2. How many comparisons are performed when we apply selection sort to a list of N items?
    1. How many comparisons are performed to find the smallest element when the unsorted portion of the list has M items?
    2. Sum over all the values of M encountered when sorting the list of length N to find the total number of comparisons.

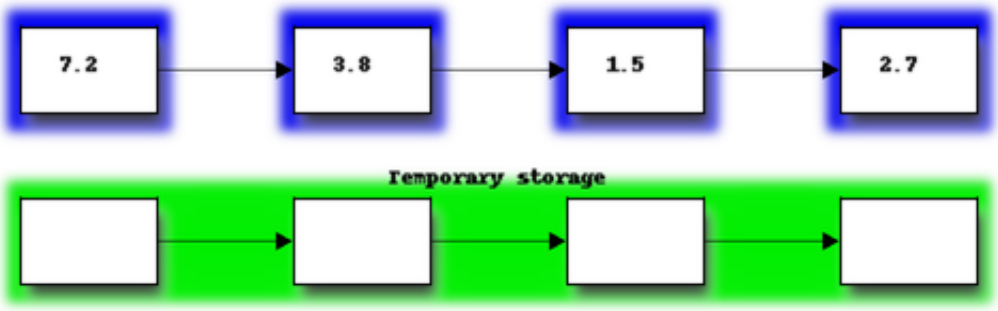
3. The number of assignments (to the candidate smallest number) performed during the search for a smallest element is at most one more than the number of comparisons. Use this to find an upper limit on the total number of assignments performed while sorting a list of length N.
4. Use the results of the previous question to find an upper bound on the total number of operations (swaps, comparisons and assignments) performed. Which term in the number of operations will dominate for large lists?

Merge sort

When we use merge sort to order a list, we repeatedly merge sorted sub-sections of the list – starting from sub-sections consisting of a single item each.

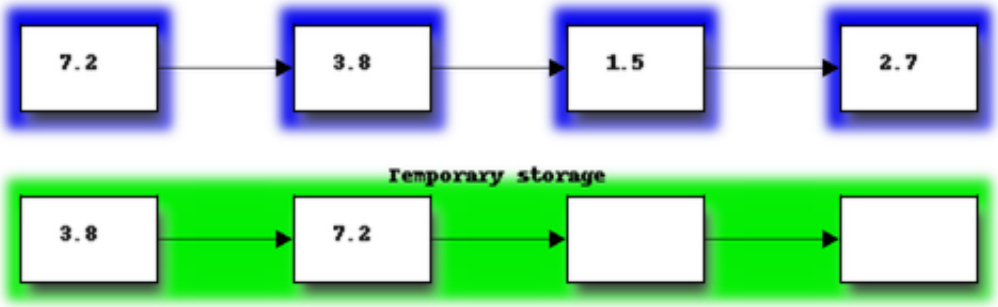
We will see shortly that merge sort requires significantly fewer operations than selection sort.

Let us start once more with our small list of four elements:

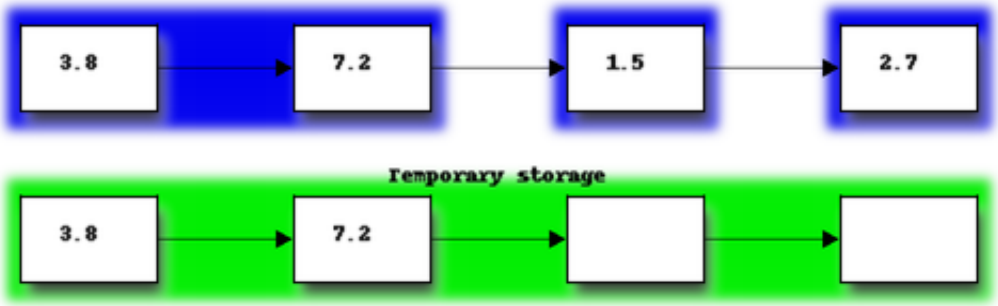


First we will merge the two sections on the left into the temporary storage. Imagine the two sections as two sorted piles of cards – we will merge the two piles by repeatedly taking the smaller of the top two cards and placing it at the end of the merged list in the

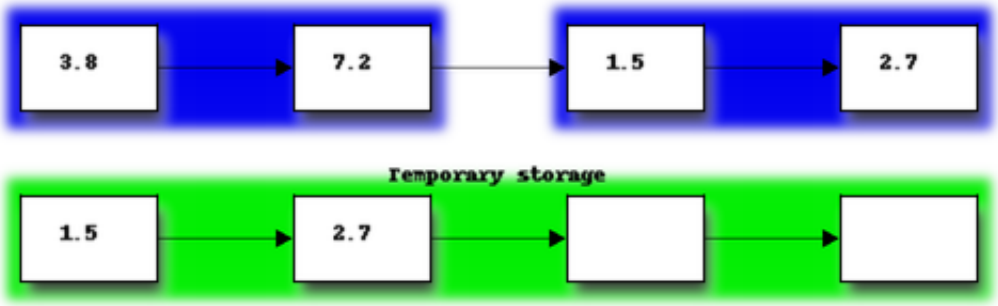
temporary storage. Once one of the two piles is empty, the remaining items in the other pile can just be placed on the end of the merged list:



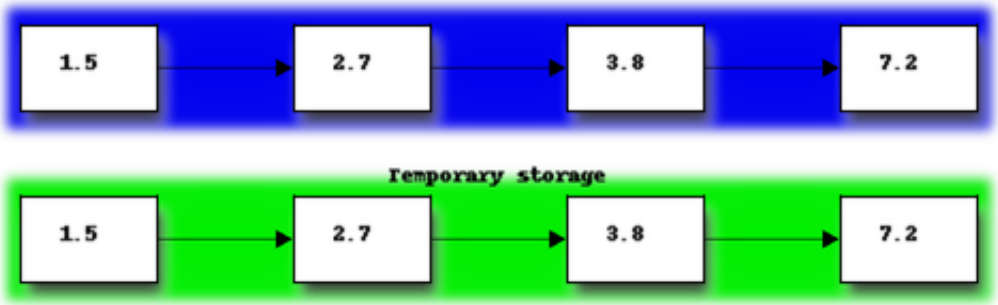
Next we copy the merged list from the temporary storage back into the portion of the list originally occupied by the merged subsections:



We repeat the procedure to merge the second pair of sorted sub-sections:



Having reached the end of the original list, we now return to the start of the list and begin to merge sorted sub-sections again. We repeat this until the entire list is a single sorted sub-section. In our example, this requires just one more merge:



Notice how the size of the sorted sections of the list doubles after every iteration of merges. After M steps the size of the sorted sections is 2^M. Once 2^M is greater than N, the entire list is sorted. Thus, for a list of size N, we need M equals $\log_2 N$ interations to sort the list.

Each iteration of merges requires a complete pass through the list and each element is copied twice – once into the temporary storage and once back into the original list. As long as there are items left in both sub-sections in each pair, each copy into the temporary list also requires a comparison to pick which item to copy. Once one of the lists runs out, no comparisons are needed. Thus each pass requires 2N copies and roughly N comparisons (and certainly no more than N).

The total number of operations required for our merge sort algorithm is the product of the number of operations in each pass and the number of passes – i.e. $2N\log_2 N$ copies and roughly $N\log_2 N$ comparisons.

The algorithm for merge sort may be written as this list of steps:

1. Create a temporary storage list which is the same size as the list to be sorted.
2. Start by treating each element of the list as a sorted one-element sub-section of the original list.
3. Move through all the sorted sub-sections, merging adjacent pairs as follows:
    1. Use two variables to point to the indices of the smallest uncopied items in the two sorted sub-sections, and a third variable to point to the index of the start of the temporary storage.
    2. Copy the smaller of the two indexed items into the indicated position in the temporary storage. Increment the index of the sub-section from which the item was copied, and the index into temporary storage.
    3. If all the items in one sub-section have been copied, copy the items remaining in the other sub-section to the back of the list in temporary storage. Otherwise return to

step 3 ii.
4. Copy the sorted list in temporary storage back over the section of the original list which was occupied by the two sub-sections that have just been merged.

4. If only a single sorted sub-section remains, the entire list is sorted and we are done. Otherwise return to the start of step 3.

Exercise 3

Write a Python function that implements merge sort. It may help to write a separate function which performs merges and call it from within your merge sort implementation.

Python's sorting algorithm

Python's default sorting algorithm, which is used by the built-in `sorted` function as well as the `sort` method of list objects, is called *Timsort*. It's an algorithm developed by Tim Peters in 2002 for use in Python. Timsort is a modified version of merge sort which uses insertion sort to arrange the list of items into conveniently mergeable sections.

> **Note**
>
> Tim Peters is also credited as the author of *The Zen of Python* – an attempt to summarise the early Python community's ethos in a short series of koans. You can read it by typing `import this` into the Python console.

Searching algorithms

Searching is also a common and well-studied task. This task can be described formally as follows:

> Given a *list of values*, a function that *compares two values* and a *desired value*, find the position of the desired value in the list.

We will look at two algorithms that perform this task:

- *linear search*, which simply checks the values in sequence until the desired value is found
- *binary search*, which requires a sorted input list, and checks for the value in the middle of the list, repeatedly discarding the half of the list which contains values which are

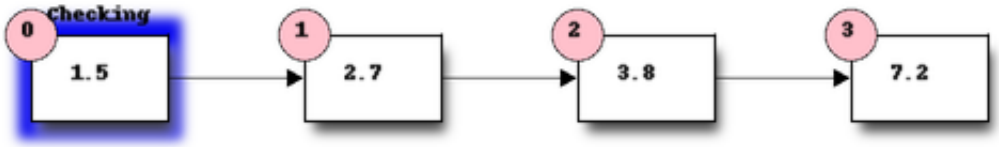
definitely either all larger or all smaller than the desired value

There are numerous other searching techniques. Often they rely on the construction of more complex data structures to facilitate repeated searching. Examples of such structures are *hash tables* (such as Python's dictionaries) and *prefix trees*. Inexact searches that find elements similar to the one being searched for are also an important topic.

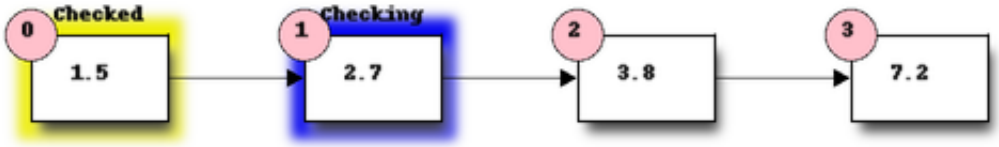
Linear search

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.

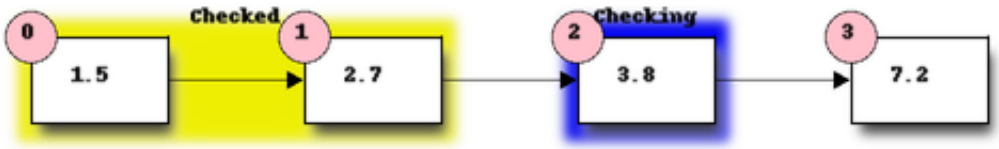
For example, suppose that we want to find the number 3.8 in the following list:



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, so 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).

We had to use a total of 3 comparisons when searching through this list of 4 elements. How many comparisons we need to perform depends on the total length of the list, but also whether the element we are looking for is near the beginning or near the end of the list. In the worst-case scenario, if our element is the last element of the list, we will have to search through the entire list to find it.

If we search the same list many times, assuming that all elements are equally likely to be searched for, we will on average have to search through half of the list each time. The cost (in comparisons) of performing linear search thus scales linearly with the length of the list.

Exercise 4

1. 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 raise an exception. If the element is in the list multiple times, the function should return the first position.

Binary search

Binary search is a more efficient search algorithm which relies on the elements in the list being sorted. We apply the same search process to progressively smaller sub-lists of the original list, starting with the whole list and approximately halving the search area every time.

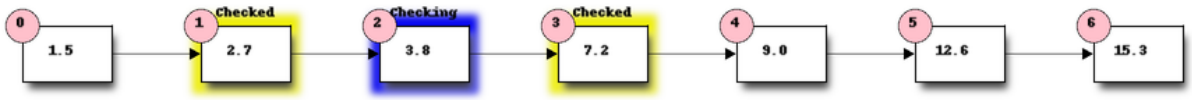
We first check the *middle* element in the list.

- If it is the value we want, we can stop.
- If it is *higher* than the value we want, we repeat the search process with the portion of the list *before* the middle element.
- If it is *lower* than the value we want, we repeat the search process with the portion of the list *after* the middle element.

For example, suppose that we want to find the value 3.8 in the following list of 7 elements:

First we compare the element in the middle of the list to our value. 7.2 is *bigger* than 3.8, so we need to check the first half of the list next.

Now the first half of the list is our new list to search. We compare the element in the middle of this list to our value. 2.7 is *smaller* than 3.8, so we need to search the *second half* of this sublist next.



The second half of the last sub-list is just a single element, which is also the middle element. We compare this element to our value, and it is the element that we want.

We have performed 3 comparisons in total when searching this list of 7 items. The number of comparisons we need to perform scales with the size of the list, but much more slowly than for linear search – if we are searching a list of length N, the maximum number of comparisons that we will have to perform is $\log_2 N$.

Exercise 5

1. Write a function which implements binary search. You may assume that the input list will be sorted. Hint: this function is often written recursively.

Algorithm complexity and Big O notation

We commonly express the cost of an algorithm as a function of the number N of elements that the algorithm acts on. The function gives us an estimate of the number of operations we have to perform in order to use the algorithm on N elements – it thus allows us to

predict how the number of required operations will increase as N increases. We use a function which is an *approximation* of the exact function – we simplify it as much as possible, so that only the most important information is preserved.

For example, we know that when we use linear search on a list of N elements, on average we will have to search through half of the list before we find our item – so the number of operations we will have to perform is N/2. However, the most important thing is that the algorithm scales *linearly* – as N increases, the cost of the algorithm increases in proportion to N, not N^2 or N^3. The constant factor of 1/2 is insignificant compared to the very large differences in cost between – for example – N and N^2, so we leave it out when we describe the cost of the algorithm.

We thus write the cost of the linear search algorithm as O(N) – we say that the cost is *on the order of N*, or just *order N*. We call this notation *big O notation*, because it uses the capital O symbol (for *order*).

We have dropped the constant factor 1/2. We would also drop any lower-order terms from an expression with multiple terms – for example, $O(N^3 + N^2)$ would be simplified to $O(N^3)$.

In the example above we calculated the *average* cost of the algorithm, which is also known as the *expected* cost, but it can also be useful to calculate the *best case* and *worst case* costs. Here are the best case, expected and worst case costs for the sorting and searching algorithms we have discussed so far:

Algorithm	**Best case**	**Expected**	**Worst case**
Selection sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
Merge sort	O(N log N)	O(N log N)	O(N log N)
Linear search	O(1)	O(N)	O(N)
Binary search	O(1)	O(log N)	O(log N)

What does O(1) mean? It means that the cost of an algorithm is *constant*, no matter what the value of N is. For both these search algorithms, the best case scenario happens when the first element to be tested is the correct element – then we only have to perform a single operation to find it.

In the previous table, big O notation has been used to describe the *time complexity* of algorithms. It can also be used to describe their *space complexity* – in which case the cost function represents the number of units of space required for storage rather than the required number of operations. Here are the space complexities of the algorithms above (for the worst case, and excluding the space required to store the input):

Algorithm	Space complexity
Selection sort	O(1)
Merge sort	O(N)
Linear search	O(1)
Binary search	O(1)

None of these algorithms require a significant amount of storage space in addition to that used by the input list, except for the merge sort – which, as we saw in a previous section, requires temporary storage which is the same size as the input (and thus scales linearly with the input size).

Note

The Python wiki has a summary of the time complexities of common operations on collections. You may also wish to investigate the `collections` module, which provides additional collection classes which are optimised for particular tasks.

Note

Computational complexity theory studies the inherent complexity of *tasks* themselves. Sometimes it is possible to prove that *any* algorithm that can perform a given task will require some minimum number of steps or amount of extra storage. For example, it can be shown that, given a list of arbitrary objects and only a comparison function with which to compare them, no sorting algorithm can use fewer than O(N log N) comparisons.

Exercise 6

1. We can see from the comparison tables above that binary search is more efficient than linear search. Why would we ever use linear search? Hint: what property must a list have for us to be able to use a binary search on it?

2. Suppose that each of the following functions shows the average number of operations required to perform some algorithm on a list of length N. Give the big O notation for the time complexity of each algorithm:
    1. $4N^2 + 2N + 2$
    2. N + log N
    3. N log N
    4. 3

Answers to exercises

Answer to exercise 1

Completed selection sort implementation:

```
def selection_sort(items):
    """Sorts a list of items into ascending order using the
       selection sort algoright.
       """
    for step in range(len(items)):
        # Find the location of the smallest element in
        # items[step:].
        location_of_smallest = step
        for location in range(step, len(items)):
            # determine location of smallest
            if items[location] < items[location_of_smallest]:
                location_of_smallest = location
        # Exchange items[step] with items[location_of_smallest]
        temporary_item = items[step]
        items[step] = items[location_of_smallest]
        items[location_of_smallest] = temporary_item
```

Answer to exercise 2

1. `N - 1` swaps are performed.
2. `(N - 1) * N / 2` comparisons are performed.
    1. `M - 1` comparisons are performed finding the smallest element.
    2. Summing `M - 1` from `2` to `N` gives:

```
1 + 2 + 3 + ... + (N - 1)

= (N - 1) * N / 2
```

3. At most `(N - 1) * N / 2 + (N - 1)` assignements are performed.
4. At most `N**2 + N - 2` operations are performed. For long lists the number of operations grows as `N**2`.

Answer to exercise 3

1. Here is an example program:

```
def merge(items, sections, temporary_storage):
    (start_1, end_1), (start_2, end_2) = sections
    i_1 = start_1
    i_2 = start_2
    i_t = 0

    while i_1 < end_1 or i_2 < end_2:
        if i_1 < end_1 and i_2 < end_2:
            if items[i_1] < items[i_2]:
                temporary_storage[i_t] = items[i_1]
                i_1 += 1
            else:  # the_list[i_2] >= items[i_1]
                temporary_storage[i_t] = items[i_2]
                i_2 += 1
            i_t += 1

        elif i_1 < end_1:
            for i in range(i_1, end_1):
                temporary_storage[i_t] = items[i_1]
                i_1 += 1
                i_t += 1

        else:  # i_2 < end_2
            for i in range(i_2, end_2):
                temporary_storage[i_t] = items[i_2]
                i_2 += 1
                i_t += 1

    for i in range(i_t):
        items[start_1 + i] = temporary_storage[i]


def merge_sort(items):
    n = len(items)
    temporary_storage = [None] * n
    size_of_subsections = 1

    while size_of_subsections < n:
        for i in range(0, n, size_of_subsections * 2):
            i1_start, i1_end = i, min(i + size_of_subsections, n)
            i2_start, i2_end = i1_end, min(i1_end + size_of_subsections, n)
            sections = (i1_start, i1_end), (i2_start, i2_end)
            merge(items, sections, temporary_storage)
        size_of_subsections *= 2

    return items
```

Answer to exercise 4

1. Here is an example program:

```
def linear_search(items, desired_item):
    for position, item in enumerate(items):
        if item == desired_item:
            return position

    raise ValueError("%s was not found in the list." % desired_item)
```

Answer to exercise 5

1. Here is an example program:

```
def binary_search(items, desired_item, start=0, end=None):
    if end == None:
        end = len(items)

    if start == end:
        raise ValueError("%s was not found in the list." % desired_item)

    pos = (end - start) // 2 + start

    if desired_item == items[pos]:
        return pos
    elif desired_item > items[pos]:
        return binary_search(items, desired_item, start=(pos + 1), end=end)
    else: # desired_item < items[pos]:
        return binary_search(items, desired_item, start=start, end=pos)
```

Answer to exercise 6

1. The advantage of linear search is that it can be performed on an *unsorted* list – if we are going to examine all the values in turn, their order doesn't matter. It can be more efficient to perform a linear search than a binary search if we need to find a value *once* in a large unsorted list, because just sorting the list in preparation for performing a binary search could be more expensive. If, however, we need to find values in the same large list multiple times, sorting the list and using binary search becomes more worthwhile.
2. We drop all constant factors and less significant terms:
    1. $O(N^2)$
    2. O(N)
    3. O(N log N)

4. O(1)