Divide and conquer algorithms

Author(s)
Silvio Peroni – silvio.peroni@unibo.it – https://orcid.org/0000-0003-0530-4305
Digital Humanities Advanced Research Centre (DHARC), Department of Classical Philology and Italian Studies, University of Bologna, Bologna, Italy

Keywords
Divide and conquer; John von Neumann; Merge sort; Mutable and immutable values

Abstract

This chapter introduces the notion of *divide and conquer* algorithms with the implementation of one algorithm of this kind: *merge sort*. The historic hero introduced in these notes is John von Neumann. He proposed a set of guidelines for the designing of the EDVAC named after him. These guidelines have been fundamental design principles for building the first electronic computers.

Historic hero: John von Neumann

John von Neumann (depicted in Figure 1) was a computer scientist, mathematician, and physicist. He was very active in all these disciplines. He made an incredibly huge number of contributions in several fields, such as quantum mechanics, game theory, and self-replicating machines.

One of his most important and famous contributions in the Computer Science domain was the digital computer architecture named after him [von Neumann, 1945]. He wrote about it for the very first time in an incomplete document for defining the main design principles of the *Electronic Discrete Variable Automatic Computer (EDVAC)*, the binary-based successor of the ENIAC. The von Neumann's architecture has been used as main guidelines for building several electronic computers in the following years, and still represent an excellent approximate model for describing several of the main components of today's digital computers.

Figure 1. A picture of John von Neumann at Los Alamos. Source: https://commons.wikimedia.org/wiki/File:JohnvonNeumann-LosAlamos.gif.

He also made other crucial contributions in Computer Science, such as the *merge sort* algorithm we introduce in this chapter. Also, he was one of the people involved in the top-secret Trinity project and its related parent project, i.e. the Manhattan Project, during World War II.

Clarification: immutable and mutable values

We have introduced the mutability and immutability of Python objects when talking about lists and tuples. A mutable object (e.g. a list) is an object that can change in time – we can create an empty list, we can populate it with new values, we can remove some of them, etc. On the other hand, an immutable object, like a tuple, is that entity that, once it is created, cannot be further modified. In particular, we can group Python basic types in the following way:

- strings, numbers, booleans, *None*, and tuples are immutable;
- lists, sets, and dictionaries are mutable.

```
def add_one(n):
    n = n + 1
    return n


my_num = 41
print(my_num)  # 41

result = add_one(my_num)
print(my_num)  # 41
print(result)  # 42
```

Listing 1. Showing the behaviour of immutable values in Python. The source code of this listing is available as part of the material of the course.

```
def append_one(lst):
    lst.append(1)
    return lst


my_list = list()
my_list.append(2)
print(my_list)  # list([2])

result = append_one(my_list)
print(my_list)  # list([2, 1])
print(result)  # list([2, 1])
```

Listing 2. Showing the behaviour of mutable values in Python. The source code of this listing is available as part of the material of the course.

This distinction is crucial when we use these kinds of objects as an input of functions or methods. They are handled in different ways, depending on dealing with mutable or immutable types. For instance, in the snippet of code in Listing 1, there is a simple function that sums *1* to the number passed as input and then returns it. However, we are always using the same variable `n` for storing the result of the operation before returning it. Since numbers are immutable, the actual value associated with the original `my_num`, used as the input of the execution of the function `def add_one(n)`, is not modified as a consequence of the execution of the function. This behaviour is the way (i.e. *by value*) Python uses to handle immutable values when passed as input of functions. It means that Python **copies** the value associated with the variable `my_num` to the variable which defines the input parameter of the function, i.e. `n`, before executing the code of the function itself.

Contrarily, mutable objects work in a slightly different way. In the snippet of code in Listing 2, Python does not copy the list specified in the variable `my_list` (a mutable object), used as the input of the function `def append_one(lst)`, into the variable defining the input parameter of the function, i.e. `lst`. Instead, Python **references** to it by such input parameter – i.e. both `my_list` and `lst` are referring to the very same list. This behaviour is the way (i.e. *by reference*) Python uses to handle mutable values when passed as input of functions.

We have a similar behaviour when we assign immutable and mutable values to a particular variable, as shown in Listing 3. One can observe how these executions and assignments affect the related objects by running all the codes in this section usingPython Tutor.

```
# Immutable objects
my_num_1 = 41
my_num_2 = my_num_1
my_num_1 = my_num_1 + 1
print(my_num_1)  # 42
print(my_num_2)  # 41, since it is a copy of the original value

# Mutable objects
my_list_1 = list()
my_list_2 = my_list_1
my_list_1.append(1)
print(my_list_1)  # [1]
print(my_list_2)  # [1], since it points to the same list
```

Listing 3. Showing the behaviour of immutable and mutable values when assigned to variables in Python. The source code of this listing is available as part of the material of the course.

Ordering billions of books

In one of the previous chapters, we have introduced an algorithm for ordering the items in a list called *insertion sort*. It is quite simple – even natural, we could say. This algorithm that works exceptionally well when the size of the list is small, but it is not very efficient when we have to deal with an incredibly significant amount of data. This behaviour is due to the brute-force approach it is implementing. In fact, while rather simple, the mechanism followed by the insertion sort obliges a computer to scan the list several times for constructing an ordered list from an unordered one.

However, the insertion sort is not the only algorithm proposed for ordering the items in a list. Other, rather efficient, approaches have been developed in the past, to perform a better (and quicker) sorting of list items in a more reasonable time – even for an electronic computer. Some of these algorithms works if particular pre-conditions hold – such as to specify the number of buckets for the *bucket sort* algorithm.

Divide and conquer sorting algorithms are among those that behave efficiently well on large input lists. Divide and conquer is an algorithmic technique. It splits the original computational problem to solve in two or more smaller problems of the same type until they became solvable directly by executing a simple set of operations. Potentially, such smaller problems can be addressed in parallel by different computers (e.g. humans). Then, the solutions to these sub-problems are recombined to provide the answer to the original problem. Any divide and conquer algorithm is based on the recursive call of the very same algorithm, according to the following (informal) steps:

1. **[base case]** address the problem directly on the input material if it is depicting an easy-to-solve problem; otherwise
2. **[divide]** split the input material into two or more balanced parts, each representing a sub-problem of the original one;
3. **[conquer]** run the same algorithm recursively for every balanced part obtained in the previous step;
4. **[combine]** reconstruct the final solution of the problem using the partial solutions obtained from running the algorithms on the smaller parts of the input material.

There are several divide and conquer sorting algorithms. In this chapter, we introduce just one of these: the *merge sort*.

Merge sort

John von Neumann proposed the *merge sort* (or *mergesort*) algorithm in 1945. It implements a divide a conquer approach for addressing the following computational problem (that we have already seen when we have introduced the *insertion sort*):

Computational problem: sort all the items in a given list.

Unlike the insertion sort, the sorting approach defined by the *merge sort* is less intuitive. Still, it is more efficient, even considering an extensive list as input. In particular, the *merge sort* is a recursion-based algorithm – like any other divide and conquer approach – and uses another ancillary algorithm in its body, called *merge*. This latter algorithm is responsible for combining two ordered input lists to return a new list which contains all the elements in the input lists ordered. The following steps illustrate the loop suggested by this algorithm to create such a new list:

1. consider the first items of both the input lists;

2. remove the lesser item from the related list and append it into the result list
3. if the input list, from where we removed the item, is not empty repeat from 1, otherwise append all the items of the other input list to the result list.

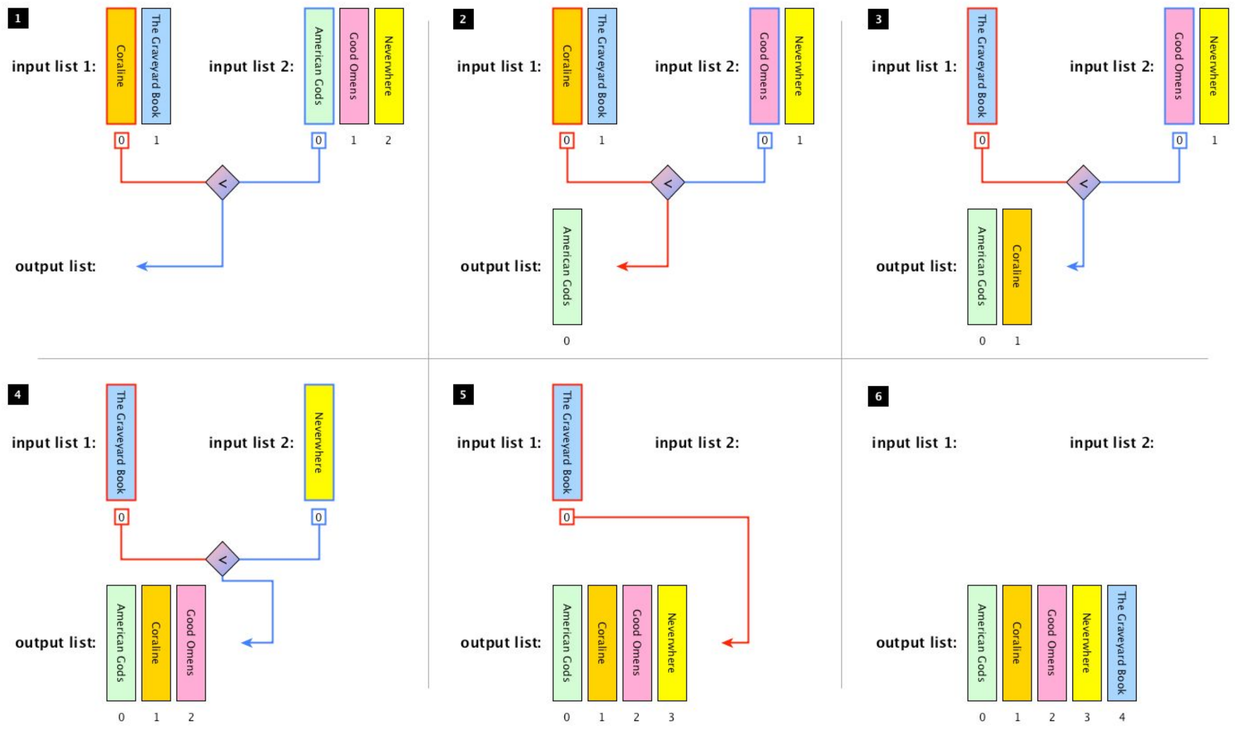

Figure 2. The process of merging two ordered lists of books together in a new list having all books ordered.

In Figure 2, we illustrate the execution of the algorithm *merge* graphically by using two lists of books as inputs, i.e. `list(["Coraline", "The Graveyard Book"])` and `list(["American Gods", "Good Omens", "Neverwhere"])` respectively. In particular, the first four steps of the execution start the creation of the output list by comparing the first items of the input lists at each iteration of the loop. Then, in the 5th step, all the items of the only non-empty input list are then appended to the output list in the order they appear in the input list. Finally (step 6), the output list is completed and is returned. Listing 4 shows the implementation in Python of the merge algorithm.

The *merge* algorithm is used in the *merge sort* to reconstruct a solution from two partial ones. In particular, the steps composing the algorithm[1] can be summarised as follows:

1. **[base case]** if the input list has only one item, return the list as it is; otherwise,

[1] Fekete and Morr released a nonverbal definition of the algorithm [Fekete, Morr, 2018b] which is useful to understand the rationale of its steps. Similarly, Kátai Zoltán and Tóth László directed a Transylvanian folk ballet which explains the execution of the merge sort algorithm visually.

2. **[divide]** split the input list into two balanced halves, i.e. containing almost the same number of items each;
3. **[conquer]** run recursively the *merge sort* algorithm on each of the halves obtained in the previous step;
4. **[combine]** merge the two ordered lists returned by the previous step by using the algorithm *merge* and return the result.

```
def merge(left_list, right_list):
    result = list()

    # Repeat until both lists have at least one item
    while len(left_list) > 0 and len(right_list) > 0:
        left_item = left_list[0]
        right_item = right_list[0]

        # If the item in left_list is less than the one in
right_list,
        # add the formed to the result and remove it from left_list
        if left_item < right_item:
            result.append(left_item)
            left_list.remove(left_item)
        # Otherwise, the item in right_list is added to the result
and
        # removed from the right_list
        else:
            result.append(right_item)
            right_list.remove(right_item)

    # Add to the result the remaining items from the lists
    result.extend(left_list)
    result.extend(right_list)

    return result
```

Listing 4. The ancillary function for merging two ordered lists together. The source code of this listing is available as part of the material of the course and also includes the related test cases.

In Figure 3, we illustrate the execution of the *merge sort* algorithm graphically by using one list of books as inputs, i.e. `my_list = list(["The Graveyard Book", "Coraline", "Good Omens", "Neverwhere", "American Gods"])`. Before introducing the implementation in Python of the algorithm, it is necessary to clarify some operations that we will use to create the balanced halves from the input list.

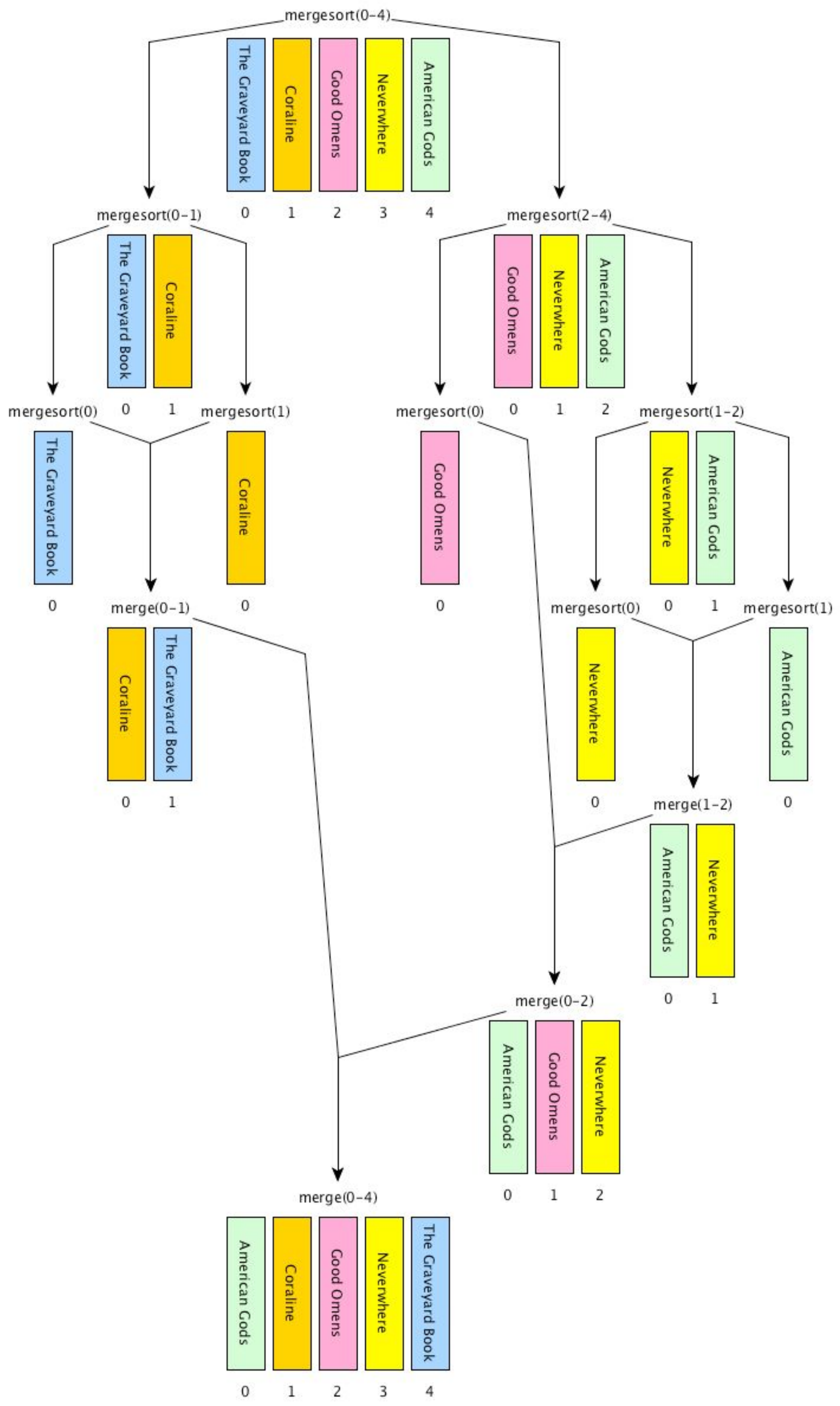

Figure 3. A graphical execution of the *merge sort* algorithm, which reuses the *merge* algorithm implemented by the function `def merge(left_list, right_list)` introduced in Listing 4.

```
# Import the function 'merge'
# from the module 'merge' (file 'merge.py')
from merge import merge

# Code of the algorithm
def merge_sort(input_list):
    input_list_len = len(input_list)

    # base case: the list is returned if it is empty or
    # contain just one element
    if len(input_list) <= 1:
        return input_list
    # recursive case
    else:
        # the floor division of the length, returning the quotient
        # in which the digits after the decimal point are removed
        # (e.g. 7 // 2 = 3)
        mid = input_list_len // 2

        # iterative steps, running the merge_sort on the
        # sublists split by mid
        left = merge_sort(input_list[0:mid])
        right = merge_sort(input_list[mid:input_list_len])

        # merge the two (ordered) lists and return the result
        return merge(left, right)
```

Listing 5. The *merge sort* algorithm implemented in Python. The source code of this listing is available as part of the material of the course and also includes the related test cases.

The first operation is the *floor division* between two numbers, i.e. `<number_1> // </number_2>`. It works like any common division, except that it returns only the integer part of the result number discarding the fractional part. For instance, `3 // 2` will be *1* (i.e. *1.5* without its fractional part), `6 // 2` will be *3* (since its fractional part is *0*), and `1 // 4` will be *0* (i.e. *0.25* without its fractional part).

The second operation allows us to create on-the-fly a new list from a selection of the elements in an input list: `<list>[<start_position>:<end_position>]`. Basically speaking, this operation creates a new list containing all the elements in `<list>` that range from `<start_position>` to `<end_position> - 1`. For instance, considering the aforementioned list in `my_list`, `my_list[0:2]` returns `list(["Coraline", "The Graveyard Book"])` while `my_list[2:5]` returns `list(["American Gods", "Good`

`Omens", "Neverwhere"])`. It is worth mentioning that the `<list>` won't be modified by such an operation.

Thus, now we have all the ingredients for introducing the Python implementation of the *merge sort* algorithm. Listing 5 illustrates the function `def merge_sort(input_list)`.

Exercises

1. Implement in Python the *binary search* algorithm – i.e. the recursive function `def binary_search(item, ordered_list, start, end)`. It takes an item to search (i.e. `item`), an ordered list and a starting and ending positions in the list as input. It returns the position of `item` in the list if it is in it, and `None` otherwise. The binary search first checks if the middle item of the list between `start` and `end` (included) is equal to `item`, and returns its position in this case. Otherwise, if the middle item is less than `item`, the algorithm continues the search in the part of the list that follows the middle item. Instead, in case the middle item is greater than `item`, the algorithms continues the search in the part of the list that precedes the middle item. Accompany the implementation of the function with the appropriate test cases. As supporting material, Fekete and Morr released a nonverbal definition of the algorithm [Fekete, Morr, 2018a] which is useful to understand the rationale of the binary search steps.
2. Implement in Python the *partition* algorithm – i.e. the non-recursive function `def partition(input_list, start, end, pivot_position)`. It takes a list and the positions of the first and last elements in the list to consider as inputs. It redistributes all the elements of a list having position included between `start` and `end` on the right of the pivot value `input_list[pivot_position]` if they are greater than it, and on its left otherwise – note: `pivot_position` must be a value between `start` and `end` (included). Also, the algorithm returns the new position where the pivot value is now stored. For instance, considering `my_list = list(["The Graveyard Book", "Coraline", "Neverwhere", "Good Omens", "American Gods"])`, the execution of `partition(my_list, 1, 4, 1)` changes `my_list` as follows: `list(["The Graveyard Book", "American Gods", "Coraline", "Neverwhere", "Good Omens"])` and *2* will be returned (i.e. the new position of `"Coraline"`). Note that `"The Graveyard Book"` has not changed its position in the previous execution since it was not included between the specified `start` and `end` positions (i.e. *1* and *4* respectively). Accompany the implementation of the function with the appropriate test cases. As supporting material, Ang recorded a video which is useful to understand the rationale of the partition steps.
3. Implement in Python the divide and conquer *quicksort* algorithm – i.e. the recursive `def quicksort(input_list, start, end)`. It takes a list and the positions of the first and last elements in the list to consider as inputs. Then, it calls `partition(input_list, start, end, start)` (defined in the previous exercise) to partition the input list into two slices. Finally, it executes itself recursively on

the two partitions (neither of which includes the pivot value since it has been already correctly positioned through the execution of partition). In addition, define the base case of the algorithm appropriately to stop if needed before to run the partition algorithm. Accompany the implementation of the function with the appropriate test cases. As supporting material, Fekete and Morr released a nonverbal definition of the algorithm [Fekete, Morr, 2018c] which is useful to understand the rationale of the binary search steps.

Acknowledgements

The author wants to thank some of the students of the Digital Humanities and Digital Knowledge second-cycle degree of the University of Bologna – Bruno Santini, Severin Josef Burg, Chantal Lengua, Mattia Spadoni and Francesco Fernicola – for having suggested corrections and improvements to the text of this chapter.

References

Fekete, S. P., Morr, S. (2018a). Binary search. IDEA. https://idea-instructions.com/binary-search/ (last visited 16 November 2019).

Fekete, S. P., Morr, S. (2018b). Merge sort. IDEA. https://idea-instructions.com/merge-sort/ (last visited 16 November 2019).

Fekete, S. P., Morr, S. (2018c). Quick sort. IDEA. https://idea-instructions.com/quick-sort/ (last visited 16 November 2019).

von Neumann, J. (1945). First Draft of a Report on the EDVAC. Available at https://web.archive.org/web/20130314123032/http://qss.stanford.edu/~godfrey/vonNeumann/vnedvac.pdf (last visited 16 November 2019).