Divide and conquer algorithms

Author(s)
Silvio Peroni – silvio.peroni@unibo.it
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

These lecture notes introduce 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, who proposed a set of guidelines for the designing of the EDVAC named after him, that have been used as 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, mathematicians, and physicists. He was very active in all these disciplines, and 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]. It has been described for the very first time in an incomplete document written by John von Neumann 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 physical electronic computers in the following years, and still represent a good 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 lecture. In addition, he was one of the people involved in the top-secret Trinity project and its related parent project, i.e. the Manhattan project, during the World War II.

Clarification: immutable and mutable values

We have already talked about the mutability and immutability of certain kinds of objects when we have introduced the difference between lists and tuples. In particular, a mutable object, like a list, is an object that can change in time – a list can be created empty, can be populated with new values, some of them can be removed, 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, Python basic types are grouped 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(l):
    l.append(1)
    return l


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 very important when we use these kinds of objects as input of functions or methods, since the way they are handled can change if we have to deal with mutable or immutable types. For instance, in the snippet of code in Listing 1, there is a simple function that sum *1* to the number passed as input and then returns it. However, we are using always the same variable `n` for storing the result of the operation before returning it. However, since numbers are immutable, the actual value associated to the original `my_num`, i.e. the variable used as the input of the execution of the function `def add_one(n)`, is not modified as consequence of the execution of the function itself. This is the behaviour of immutable values, since they are passed *by value* as input of functions. It means that the value associated with the variable `my_num` is actually copied 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. For instance, in the snippet of code in Listing 2, the list (which is a mutable object) passed as input to the function `def append_one(l)` by means of the variable `my_list` is not copied into the variable defining the input parameter of the function, i.e. `l`, but rather it is only referenced by such parameter – i.e. both `my_list` and `l` are actually referring to the very same list. This is the behaviour of mutable values, since they are passed *by reference* as inputs of functions.

These behaviours of immutable and mutable objects can be observed also in the assignment of values to variables. This is briefly described in Listing 3. One can observe how these executions and assignments affect the related objects by running all the codes in this section by means of Python 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 lectures, we have introduced an algorithm for ordering the items in a list called *insertion sort*. It is a 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 big 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, in order 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 ones that behave efficiently well on large input lists. Divide and conquer is an algorithmic technique that, in contrast to brute-force, splits the original computational problem to solve in two or more smaller problems of the same type – that, potentially, can be addressed in parallel by different computers (e.g. humans) – until they became solvable directly by executing a simple set of operations. Then, the solutions to these sub-problems are recombined so as to provide the solution for 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 actually depicting an easy-to-solve problem; otherwise
2. **[divide]** split the input material into two or more balanced parts, each depicting a sub-problem of the original one;
3. **[conquer]** run the same algorithm recursively for every balanced parts obtained in the previous step;
4. **[combine]** reconstruct the final solution of the problem by means of the partial solutions obtained from running the algorithms on the smaller parts of the input material.

Several divide and conquer sorting algorithms have been proposed in the past. In this lecture, we introduce just one of these: the *merge sort*.

Merge sort

The *merge sort* (or *mergesort*) algorithm has been proposed by John von Neumann 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.

Unlikely the insertion sort, the sorting approach defined by the *merge sort* is lesser intuitive, but it is more efficient even considering a large list as input – such as all the books included in the Library of Babel that we have illustrated a few lectures ago. 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 to combine two *ordered* input lists together so as to return a new list which contains all the elements in the input lists ordered. The loop suggested by this algorithm to create such a new list is illustrated as follows:

1. consider the first elements of both the input lists;
2. remove the lesser element from the related list and append it into the result list
3. if the input list from which the element has been removed is not empty repeat from 1, otherwise append all the elements of the other input list to the result list.

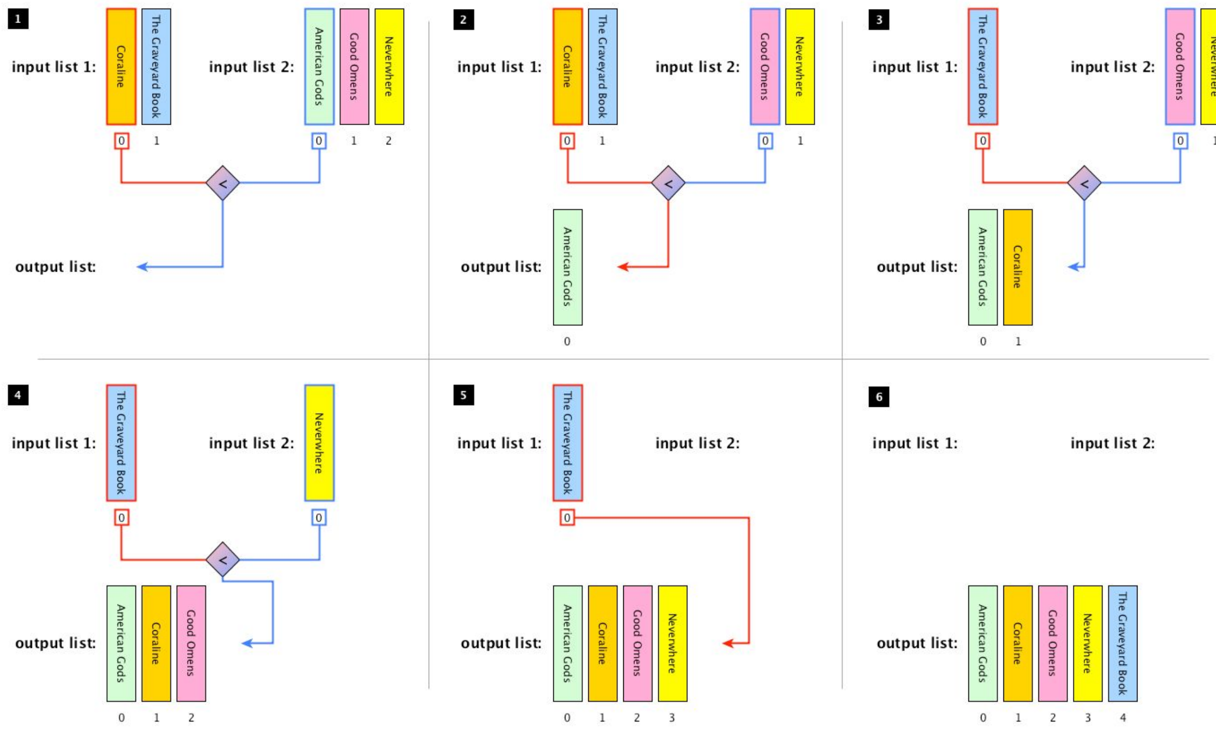

Figure 2. The process of merging two ordered lists of books together in a new list having all books ordered.

In Figure 2, we illustrate graphically the execution of the algorithm *merge* 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 elements of the input lists at each iteration of the loop. Then, in the 5th step, all the elements 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. The Python implementation in Python of the *merge* algorithm is illustrated in Listing 4.

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

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

2. **[divide]** split the input list into two balanced halves, i.e. containing almost the same number of elements 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 hate 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 includes also the related test cases.

In Figure 3, we illustrate graphically the execution of the *merge sort* algorithm 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.

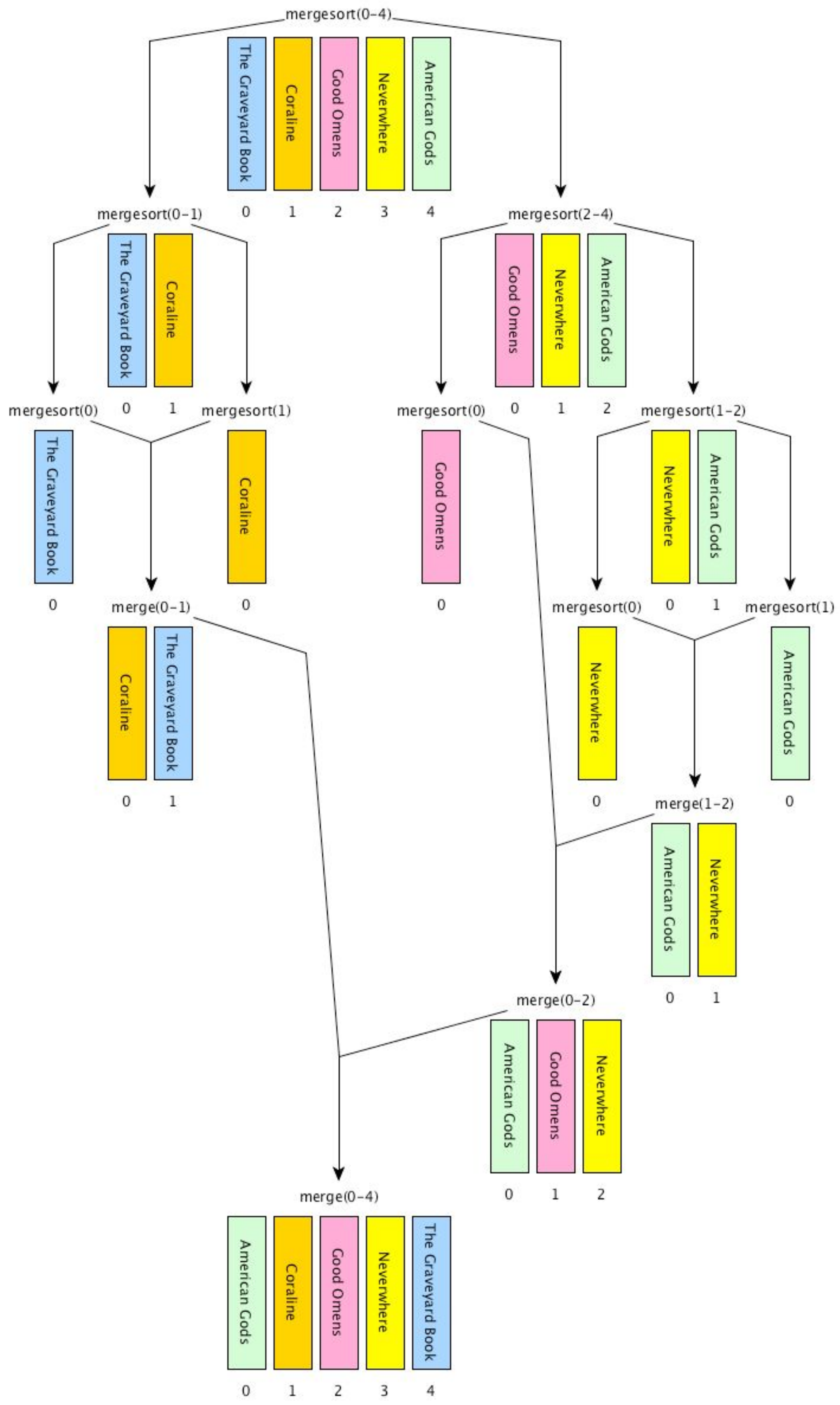
mergesort(0–4)
The Graveyard Book
Coraline
Good Omens
Neverwhere
American Gods
0
1
2
3
4
mergesort(0–1)
The Graveyard Book
Coraline
0
1
mergesort(2–4)
Good Omens
Neverwhere
American Gods
0
1
2
mergesort(0)
The Graveyard Book
0
mergesort(1)
Coraline
0
mergesort(0)
Good Omens
0
mergesort(1–2)
Neverwhere
American Gods
0
1
merge(0–1)
Coraline
The Graveyard Book
0
1
mergesort(0)
Neverwhere
0
mergesort(1)
American Gods
0
merge(1–2)
American Gods
Neverwhere
0
1
merge(0–2)
American Gods
Good Omens
Neverwhere
0
1
2
merge(0–4)
American Gods
Coraline
Good Omens
Neverwhere
The Graveyard Book
0
1
2
3
4

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 includes also 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 operation.

Thus, now we have all the ingredients for introducing the Python implementation of the *merge sort* algorithm, i.e. def merge_sort(input_list). It is illustrated in Listing 5.

Exercises

1. Implement in Python the *partition* algorithm – i.e. def partition(input_list, start, end, pivot_position) – that takes a list and the positions of the first and last elements in the list to consider as inputs, and 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. In addition, the new position where the pivot value is now stored will be returned by the algorithm. 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).
2. Develop the divide and conquer algorithm def quicksort(input_list, start, end) that takes a list and the positions of the first and last elements in the list to consider as inputs, and that calls partition(input_list, start, end, start) (defined in the previous exercise) to partition the input list into two slices, and then applies itself recursively on the two partitions (neither of which includes the pivot value since it has been already correctly positioned by means of the execution of partition). In addition, define appropriately the base case of the algorithm so as to stop if needed before to run the partition algorithm.

References

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 26 November 2017).