Greedy algorithms

Author(s)
Silvio Peroni – silvio.peroni@unibo.it
Department of Classical Philology and Italian Studies, University of Bologna, Bologna, Italy

Keywords
Evelyn Berezin; Line wrap; Word processor

Abstract

These lecture notes introduce the last kind of algorithms presented in this course, i.e. the *greedy algorithms*. The historic hero introduced in these notes is Evelyn Berezin, one of the most important business women of the past century, who have created the first word processor.

Historic hero: Evelyn Berezin

Evelyn Berezin (depicted in Figure 1) was a physicist who started to work in a company that produced digital computers, and where she started to work on particular the development of the logic designs of computers – e.g. [Auerbach et al., 1962]. After a bunch of years passed in changing job, and several contributions related to the development of large computer systems such as the computerised reservation system for United Airlines, in 1969, she founded her own company: Redactron Corporation.

In this new company, she started to work on computer systems to simplify the work of secretaries. The main product of the company was called *Data Secretary*, the very first word processor in history, which was a stand-alone device developed for addressing that specific task, so as to replace the more common typewriter. It has been the precursor of all the series of word processors that have been developed since that date, initially as stand alone devices, and then as independent software applications to be installed in personal computers – starting from Electric Pencil (1976) and WordStar (1978), to Microsoft Word (1983) and OpenOffice Writer (1999).

Figure 1. A picture of Evelyn Berezin taken in 2015. Picture from the Computer History Museum, source: https://images.computerhistory.org/blog-media/2015-fellow-awards-evelyn-berezin.jpg.

Greedy algorithms

A *greedy algorithm* is an approach that, at every stage of execution of a particular algorithm where we are seeking for possible candidates for constructing the solution to a computational problem, makes always the choice that is optimal (i.e. the best one) in that particular moment. For certain kinds of problems, this behaviour allows us to reach the best possible solution to the computational problem in consideration. For instance, if you have to determine the minimum number of euro coins needed for making a change, then a greedy algorithm will return an optimal solution overall:

1. consider the coins to choose for the change as ordered in a decrescent way, from the highest value (i.e. 2 euros) to the lowest one (i.e. 1 cent);
2. for each kind of value, add in the candidate set of the solution as much coins of that value as possible until their sum is lesser than the remaining of the change to give;
3. if the change value is reached, return it.

However, sometimes it is possible that the solution found, while it provides a correct solution to the problem, is just a suboptimal solution. For instance, driving from Florence to Bologna, we

can encounter a crossroad with two signs indicating two different routes to get to Bologna. The left route allows us to get to Bologna by travelling for 42 kilometres. On the other hand, the right route allows us to get to Bologna by travelling for 56 kilometres. A pure greedy approach would select the left route since at the moment seems the most convenient scenario. However, the approach does not predict the existence of possible traffic in the left road and, consequently, it would be possible to arrive in Bologna even after a car that takes the right route.

There are two main characteristics that a computational problem should show so as to be sure that the application of a greedy approach will bring to an optimal solution to the problem. The first one is that the *greedy choice property* should be guaranteed. This property means that, at a certain step, we can choose the best candidate for improving the set of candidates bringing to a solution.

The other characteristic is the problem as an *optimal substructure*. This means that the optimal solution to a computational problem can be built by considering the optimal solutions to its subproblems. For instance, the previous example of the travel from Florence to Bologna does not have an optimal substructure because there can be accidents that are encountered as a consequence of a previously-chosen optimal subsolution.

Line wrap

Understanding where to break a line in a page, i.e. line wrap, is one of the simplest and most relevant problems one has to tackle when dealing with documents, either in print or digital forms. For instance, when a person is using a typewriter for writing a document, at a certain point, after she has written a bunch of characters, there is a mandatory action to perform which is the carriage and return operation, that is performed mechanically on the typewriter itself. Basically speaking, when the writer notices that the page has no more space for imprinting a new word on that line, the configuration of the typewriter is initialised again in order to start from the very beginning of the left border but in the following line.

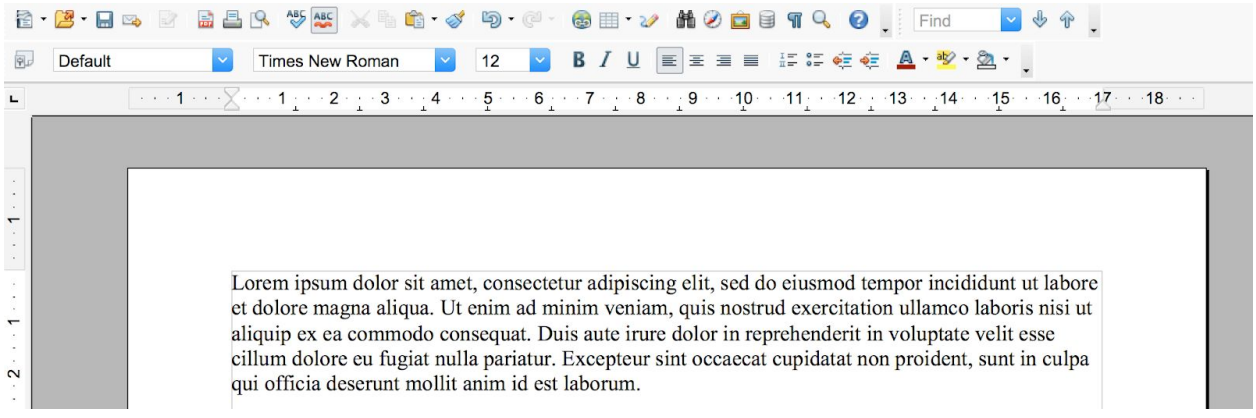

Figure 2. A screenshot depicting how OpenOffice Writer deals with line wrap.

```
# Test case for the algorithm
def test_line_wrap(text, line_width, expected):
      result = line_wrap(text, line_width)
      if expected == result:
            return True
      Else:
            return False


# Code of the algorithm
def line_wrap(text, line_width):
      # the list of all the lines of a document
      result = []

      # the maximum available space per a specific line
      space_left = line_width
      # the current line that is built
      line = []

      for word in text.split(" "):
      word_len = len(word)

      # we consider the length of the word plus one space character
      if word_len + 1 > space_left:
            result.append(" ".join(line))
            line = [word]
            space_left = line_width - word_len
      else:
            line.append(word)
            space_left = space_left - word_len + 1

      # we add the remaining line to the document
      result.append(" ".join(line))

      return "\n".join(result)


print(test_line_wrap("Just a word.", 15, "Just a word."))
print(test_line_wrap("Just a word.", 1, "\nJust\na\nword."))
print(test_line_wrap("This is a simple example.", 10,
                     "This is a\nsimple\nexample."))
```

Listing 1. The implementation of the algorithm for calculating the line-wrap problem in Python. The source code of this listing is available as part of the material of the course.

In modern tools, such as word processors (shown in Figure 2), the line wrap is totally handled by an algorithm that takes care of choosing when there is enough space to put that word in the current line. Generally speaking, we can describe the problem in the following manner:

Computational problem: break a text into lines such that it will fit in the available width of a page.

A greedy approach is very efficient and effective for addressing the aforementioned computational problem. It will proceed as follows:

1. For each word in the input text, see if there is enough space in the line for adding that word;
2. If there is space, add the word to the line; otherwise,
3. Declare finished the current line, and add the word as the first token of the following line.

In order to implement this algorithm, we need to introduce two ancillary methods for strings, in particular for tokenizing and recomposing strings. The first of these methods is `<string>.split(<string_separator>)`. This method allows us to separate the string according to a specific set of characters the string may contain, specified by the parameter `<string_separator>`. For instance, if we have the variable `my_string` assigned to `"a b c"`, the execution of the aforementioned method, i.e. `my_string.split(" ")`, returns the following list: `["a", "b", "c"]`.

The other method we need, i.e. `<string_separator>.join(<list_of_strings>)`, implements the opposite operation, i.e. it is able to concatenate the strings in a list again, according to a particular sequence of characters. For instance, if we have the list `my_list = ["a", "b", "c"]`, the execution of the aforementioned method, i.e. `" ".join(my_list)`, returns the following string: `"a b c"`.

We now have all the ingredients for implementing our algorithm for the line-wrap. In particular, it is introduced in Listing 1.

Exercises

1. Implement the informal algorithm introduced in Section "Greedy algorithms" for returning the minimum amount of coins for a change.

Acknowledgements

I would like to thank some of the students of the course, Tanise Pagnan Ceron and Eleonora Peruch, for having suggested corrections and improvements to the text of these lecture notes.

References

Auerbach, A. A., Evelyn, B., Samuel, L., Shaw, R. F. (1962). Electronic data file processor. U.S. Patent No. 3,017,610. Washington, DC: U.S. Patent and Trademark Office. https://patentimages.storage.googleapis.com/e5/ca/5b/9b2d6591f0cb14/US3017610.pdf (last visited 18 December 2018)