Problem Generalization for Designing Recursive Algorithms

Diana Borrego(✉), Irene Barba, Miguel Toro, and Carmelo Del Valle

Departamento de Lenguajes y Sistemas Informáticos, Universidad de Sevilla, Sevilla, Spain
{dianabn,irenebr,migueltoro,carmelo}@us.es

Abstract. This paper focuses on the difficulty for university students to acquire, within computational thinking, the skills to solve certain problems through recursion. The acquisition of this type of reasoning is essential to understand the different problem solving techniques that are based on recursive algorithms, such as divide and conquer or dynamic programming. Therefore, first, the generalization of problems is proposed as a strategy for designing recursive algorithms. As a second step, that generalization is formalized through a specification sheet that contains different fields that correspond to the characteristics that are relevant to solve a problem recursively.

Keywords: Recursive algorithms · Computational thinking · Problem generalization

1 Introduction

Computer science university students have the need to develop computational thinking, which is the mental process used to formulate problems whose solutions can be carried out by a computer. Computational thinking includes skills such as modeling and breaking down problems, processing data, creating algorithms and generalizing them.

There are several definitions related to computational thinking in the literature. Reviewing publications about this topic, Wing et al. [6], who made the term computational thinking famous, present the following definition: "Computational thinking involves solving problems, designing systems, and human behavior understanding, by drawing on the concepts fundamental to computer science. Computational thinking includes a range of mental tools that reflect the breadth of the field of computer science."

Therefore, students must acquire computational thinking to be able to carry out problem solving through modularization, top-down analysis, bottom-up analysis, recursion, and so on. Particularly, the understanding and application of this last concept of recursion [3,4] requires the students to make an extra mental effort. They usually understand and learn more easily the iterative formulations of problems. However, to understand the recursive formulation, a special type of thinking is needed, and therefore to be able to use recursive thinking [7]. This presents a difficulty that makes a mental predisposition necessary that is not always frequent or easy to reach depending on each person.

In detail, the recursive resolution of a problem is raised when the specification of its solution is made based on the solution of other problems of the same nature but a

smaller size. Therefore, every recursive definition has (i) base case(s), encompassing and resolving cases where the resolution of the problem is direct and does not require the use of the function being defined; (ii) recursive case(s), which are those whose resolution does require the use of the function being defined; and (iii) size of the problem, which is a cardinal indicating the dimension of the problem to be solved, and that decreases in each subproblem. Thus, a recursive algorithm is one that expresses the solution of a problem in terms of invoking itself.

As an example, the definition and recursive algorithm for the resolution of the factorial of a number n are shown below.

$$n! = \begin{cases} 1 & n = 0 \\ n * (n-1)! & \text{n} > 0 \end{cases}$$

```
N f(Integer n){
    N s;
    if{n == 0}{
        s = 1;
    }else {
        s = n * f(n-1);
    }
    return s;
}
```

Algorithm 1: Recursive algorithm for calculating the Factorial number (in Java)

Not all recursive definitions are suitable for building an algorithm. To be adequate, it must have: (1) at least one base case; (2) in each recursive step the size should be reduced (which implies that the execution of the algorithm will end).

Taking all this into account, and to facilitate the acquisition and application of recursive thinking, this paper proposes to carry out two steps prior to the design of a recursive algorithm to solve a given problem, as shown in Fig. 1. Specifically, these steps are:

1. Generalization of the problem through a set of problems [1]. More precisely, generalizing consists of adding properties to the original problem to consider it a particular case of a wider set of problems.

Fig. 1. Proposed approach for designing recursive algorithms

2. Specification of the solution to the problem through a predefined specification sheet based on the previous generalization made. This sheet contains different fields that correspond to the characteristics that are relevant to solve this problem recursively.

Note that the goal of the proposed approach is not to provide a formal method but to facilitate the acquisition of skills related to the design of recursive algorithms. For this, we use our experience in the design of types in the context of object-oriented programming to conceptualize the population of problems that are generated when designing recursive algorithms, i.e., when the generalization is carried out. In such context, the correct definition of the size of the problems (that is a derived property of such problems) is key when analyzing the correctness of the generalization performed. To be more precise, when generalizing a problem for designing a recursive algorithm, it is necessary to verify that there is an initial problem, and that all the problems are ordered by size.

The remainder of the paper is structured as follows: Sect. 2 presents the generalization of problems through sets of problems, Sect. 3 includes the specification sheet of the problem solution by recursion along with some illustrative examples, and Sect. 4 presents a critical discussion of the proposed approach. Finally conclusions are drawn and future work is proposed in Sect. 5.

2 Problem Generalization

This section presents the proposed approach: Sect. 2.1 explains the concept of set of problems in the context of problem generalization, Sect. 2.2 details how such set of problems can be used for problem generalization, and, lastly, Sect. 2.3 includes the general schema of recursion that is related to the proposed problem generalization.

2.1 Set of Problems

In a recursive definition it is always necessary to start from a set of problems $P = \{p, p_0, p_1, ...\}$. In such set each problem is characterized by a set of properties $x = \{x_0, x_1, ...\}$. Each property of a set of problems can be categorized as:

- Shared: shared properties are defined as properties of the set of problems that take the same value for all problems of such set.
- Individual: individual properties are defined as properties of the set of problems that take different values for each problem of such set.

Both shared and individual properties can be categorized as *basic* or *derived properties*. The inclusion of the latter, unlike the former, is optional since their values can be calculated from the values of the basic properties. Derived properties are typically included for the sake of clarity. It is important to clarify that the basic individual properties are the ones that unequivocally identify each problem within the set of problems, i.e., there are not two problems with the same values for all basic individual properties.

Moreover, each problem of a set of problems has a specific size, that is a derived individual property. To be more precise, the size of a problem is defined as an integer

greater than or equal to zero that is related to the complexity of the problem (i.e., how far is the problem from a case base). There are different ways of defining the size of a problem. However, it is necessary to check the correctness of such definition by considering that the size of a problem should (1) be a function over some of the properties of the problem, (2) be reduced as a base case is approached.

The problems of a set of problems can be categorized as:

- Base case of a set of problems P: a problem of a set of problems is a base case when it has a direct solution (i.e., without performing a recursive call). A problem that is a base case has a small size. Moreover, there can exist several base cases within the same set of problems.
- Recursive case of a set of problems P: a problem of a set of problems is a recursive case when it does not have a direct solution (i.e., it is necessary to perform a recursive call to solve it). The solution of a recursive problem is defined by combining the solutions related to problems of smaller sizes (i.e., subproblems).
- All problems, in the set of problems of interest, verify a predicate $d(x)$. This predicate establishes the domain of valid problems.

2.2 Problem Generalization

The key idea to solve a problem in a recursive way is based on expressing its solution in terms of operations over solutions to a set of problems of smaller size. For this, in many cases it becomes necessary to imagine that set of problems from the original problem (i.e., *problem generalization*).

To be more precise, the problem generalization consists of the following steps:

1. Adding, if necessary, a set of additional properties to those that already had the problem and classify them into basic, derived, individual, shared, etc.
2. Defining the domain of problems of interest through a predicate.
3. Defining the size of a problem.
4. Stating the base cases and the recursive cases.
5. Specifying how each recursive case is reduced to subproblems with a smaller size.
6. Specifying how each recursive case's solution is related to that of its subproblems.
7. Specifying the solution of base case.
8. Indicating which of the problems in the problem set corresponds to the original problem.

2.3 General Structure of a Recursive Algorithm

In order to design a recursive algorithm for solving a problem with parameters x (of type X) it is typically advised to perform 2 functions. The first one (i.e., function f, cf. Algorithm 2) receives as input parameters of the initial problem to solve x, and performs an initial call to a second function g (cf. Algorithm 3). The latter is a recursive function that receive as input parameter both the parameters of problem to solve x and the additional properties that are considered after problem generalization (i.e., y of type Y). From now on the properties of the generalized problem will be both the parameters

of the original problem and the new properties added. Function f is optional. There are cases where the function g is sufficient and f does not appear. This situation is found when it is possible to consider a broad set of problems by varying within a domain the values of the parameters. One function calls the other as depicted in Algorithm 2.

```
S f(X x) {
    return g(x,y_0);
}
S g(X x, Y y){
    ...
}
```

Algorithm 2: Functions in a recursive algorithm (in Java)

The general structure of a recursive algorithm is shown in Algorithm 3.

```
S f(X x) {
    return g(x,y_0);
}
S g(X x, Y y){
   S s;
   if(b(x,y){
        s = sb(x,y);
   } else {
        Y y_0 = sp_0(x,y);
        S s_0 = g(x,y_0);
        Y y_1 = sp_1(x,y);
        S s_1 = g(x,y_1);
        ...;
        Y y_{m-1} = sp_{m-1}(x,y);
        S s_{m-1} = g(x,y_{m-1});
        s = c(s_0,s_1,...,s_{m-1},x,y);
   }
   return s;
}
```

Algorithm 3: General structure of a recursive algorithm (in Java)

As follows, concepts related to Algorithm 3 are detailed:

- x: parameters of problem of type X.
- S: type of the solution.
- y: additional properties (of type Y) for the set of problems that are included after problem generalization.
- $size(x,y)$: function that returns the size of the problem (x,y).
- $d(x,y)$: predicate establishing the domain of (x,y).
- $b(x,y)$: predicate that shows if a problem is a base case

- $sb(x,y)$: function that returns the solution to a base case.
- $np(x,y)$: function that returns the number of subproblems related to (x,y).
- $m = np(x,y)$: the number of subproblems
- $y_0, y_1, ..., y_{m-1}$: variables of type Y that store the subproblems related to (x,y). To be more precise, y_i refers to the i-th problem related to (x,y).
- $sp_0(x,y), sp_1(x,y), ..., sp_{m-1}(x,y)$: functions that return the subproblems related to (x,y). To be more precise, $sp_i(x,y)$ returns the the i-th problem related to (x,y).
- $s_0, s_1, ..., s_{m-1}$: variables of type S storing the solutions to the subproblems related to (x,y). Thus, s_i refers to the solution to the i-th problem related to (x,y).
- $c(s_0, s_1, \ldots, x, y)$: function that returns the solution to x by combining the solutions to its subproblems.

3 Specification Sheet for the Problem Generalization

This section details the sheet that is proposed for specifying the problem generalization previously explained (cf. Sect. 2.2). The parts that are included in such specification sheet are explained as follows:

- Types: information related to the types that are relevant to solve the problem, e.g., the type of the solution.
- Shared properties: shared properties of the set of problems that is obtained after the problem generalization.
- Individual properties: individual properties of the set of problems that is obtained after the problem generalization.
- Size: size of the problem, which is a function over the properties of such problem.
- Initial instantiation: initial value that is given to the individual properties.
- Base cases: it gives information about whether a problem is a base case. In case there is more than one condition that causes a problem to be considered a base case, one per line is specified.
- Solution for base cases: solutions for the problems that are base cases. If there is more than one base case condition, the solution is provided to each of them on different lines, using the operator $\rightarrow$.
- Number of subproblems: number of subproblems that need to be solved to obtain the solution to the original problem.
- Subproblems: subproblems that need to be solved to obtain the solution to the original problem. When determining them, it is only necessary to specify the changes that occur in the individual properties with respect to the original problem, since the shared properties remain the same for all problems. Different subproblems are also specified in different lines, with the operator $\rightarrow$. In case that only one subproblem needs to be solved, its is determined through the condition to select which one, with the format $c_i \rightarrow s_i$. On the other hand, when there are more than one subproblem, a superscript is added to distinguish the referred one ($\overset{0}{\rightarrow}, \overset{1}{\rightarrow}, \ldots$).
- Combine: function that obtains the solution to the original problem by combining the solutions that are obtained for the subproblems.

As can be observed, there are many similarities between the elements of the specification sheet and the elements of the general structure of a recursive algorithm (for details cf. Table 1).

Table 1. Relation between the specification sheet and the recursive algorithm.

Specification sheet	General structure of a recursive algorithm
Types	S, X, Y
Shared properties	
Individual properties	
Domain	$d(x,y)$ predicate
Size	$size(x,y)$ function
Initial instantiation	y_0
Base cases	$b(x)$ predicate
Solution for base cases	$sb(x,y)$ function
Number of subproblems	$np(x,y)$ function
Subproblems	$sp_0, sp_1, \ldots$. functions
Combine	$c(s_0, s_1, \ldots, x, y)$ function

3.1 Running Examples

This section includes how the proposed approach is applied to two illustrative examples. For each example, (1) the specification sheet related to the problem generalization and (2) the related recursive algorithm are detailed.

Example 1. Calculating the Fibonacci numbers [5]**.** The Fibonacci number is recursively defined as follows:

$$f(n) = \begin{cases} n & n \leq 1 \\ f(n-1) + f(n-2) & \text{otherwise} \end{cases}$$

Regarding such definition, a recursive algorithm can be designed (cf. Algorithm 4) according to the problem generalization that is specified in Table 2.

```
N fib(Integer n){
    N s;
    if(n<=1){
        r = n;
    } else {
        Integer y_0 = n-1;
        s_0 = fib(y_0);
        Integer y_1 = n-2;
        s_1 = fib(y_1);
        s = s_0+s_1;
    }
    return s;
}
```

Algorithm 4: fib - Recursive algorithm for calculating Fibonacci numbers (in Java)

Table 2. Specification sheet for solving the Fibonacci problem.

Fibonacci	
Types	Solution: Integer
Shared properties	
Individual properties	n: Integer
Domain	$n \geq 0$
Size	n
Initial instantiation	
Base cases	$n \leq 1$
Solution for base cases	n
Number of subproblems	2
Subproblems	$(n) \xrightarrow{0} (n-1)$ $(n) \xrightarrow{1} (n-2)$
Combine	$s_0 + s_1$

Example 2. Binary search in a sorted list [2]**.** The binary search algorithm is an efficient search method, which finds the position of a target value within a sorted list. To do this, the algorithm divides the list by its middle element into two smaller sublists, and compares the target value with the middle element. If they are equal, the search ends. If the target value is lower, it must be (if present) in the first half of the list, in the second half otherwise. The process continues until the target is found, or until it reaches an empty sublist due to the target value is not in the list (returning -1), which are two base cases conditions.

For this problem, the following shared properties are considered: (1) list of elements ls, (2) element that is searched m, and (3) size of the list of elements, that is a derived property n. To generalize, the following individual properties are established: (1) index i that indicates the starting position of the sublist considered for the current subproblem, (2) index j that indicates the ending position of the sublist considered for the current subproblem, and (3) the middle position k, a derived property, between i, j. As detailed, the problem only invokes one of the two possible subproblems.

More details are given in Table 3, and the related algorithms are depicted (cf. Algorithm 5). As can be observed, the main algorithm (cf. bs in Algorithm 5) performs the initial call to the recursive algorithm (cf. bs_g in Algorithm 5) according to the initial instantiation of the individual properties.

4 Discussion

The proposal presented in this paper presents several advantages. To be more precise, on the one hand, the fact of analysing the solution to a problem before carrying out the implementation itself facilitates error isolation as well as error detection. On the other hand, the generalization of problems and the related specification sheet allow for establishing a systematic way of designing a recursive algorithm for solving a specific problem.

Based on our experience as teachers of programming subjects in the first university-level courses, students are reluctant at first to specify the generalization of problems through the specification sheet. On the one hand, this is something new for them, and therefore, a period of learning and training is required. In this period, students acquire the necessary knowledge regarding the meaning and usefulness of each of the properties that are included in the specification sheet. On the other hand, specifying the problem generalization is more time-consuming than directly implementing the solution. Moreover, the application of the proposed approach makes sense for problems that presents certain characteristics. To be more precise, the proposed approach makes more sense in the case that designing an efficient recursive algorithm for solving a problem is not trivial for the students, i.e., previous analysis is required (e.g., cf. Example 2). However, there are some problems, i.e., those problems whose definition itself is recursive, that

Table 3. Specification sheet for solving the Binary search problem.

Binary search	
Types	Solution: Integer
Shared properties	ls: $List < E >$ e: E n: Integer (derived - size of ls)
Individual properties	i: Integer j: Integer k: Integer (derived), $k = (j+i)/2$
Domain	$i \in [0,n) \&\& j \in [i,n]$
Size	$j - i$
Initial instantiation	$(0,n)$
Base cases	$e = ls[k]$ $j - i < 1$
Solution for base cases	$e == ls[k] \rightarrow k$ $j - i < 1 \rightarrow -1$
Number of subproblems	1
Subproblems	$e > ls[k] \rightarrow (k, j)$ $e \leq ls[k] \rightarrow (i,k)$
Combine	s_0

presents a recursive algorithm that can be directly implemented in an easy way without requiring a previous analysis (e.g., cf. Example 1).

Therefore, in summary, through the use of specification sheets, students are able to address complex problems in less time, since they are guided and assisted in the reasoning necessary to reach the solution.

```
Integer bs(ls,e){
   Integer n = ls.size();
   return bs_g(ls,e,n,i,j,(j+i)/2);
}
Integer bs_g(ls,e,i,j,k){
   Integer s;
   if(j-i < 1) {
       s = -1;
   } else if(e == ls[k]) {
       s = k;
   } else {
       Integer n1, nj, nk;
       if(e < ls[k]) {
           ni = i;
           nj = k;
           nk = (ni+nj)/2;
       } else if(e > ls[k]){
           ni = k;
           nj = j;
           nk = (ni+nj)/2;
       }
       s = bs_g(ls,e,ni,nj,nk);
   }
   return s;
}
```

Algorithm 5: Recursive algorithm for the binary search (in Java)

5 Conclusions and Future Work

In this paper, a methodology for solving recursive problems is proposed, so that the acquisition of this skill within computational thinking is easier for computer science university students. For this, the generalization of problems is proposed as a strategy for the design of recursive algorithms. After this generalization, and based on it, a formalization is made by filling in a defined specification sheet that contains different fields encompassing the characteristics that are relevant to solve a problem recursively. Through this step-by-step methodology, students can understand and learn recursive reasoning more quickly and easily.

As future work, we intend to make proposals regarding the specification sheets of the solution using techniques such as dynamic programming, or backtracking. It is also proposed to conduct a study about the acceptance of this type of techniques by the students, and the impact that their use has on the grades obtained.

References

1. Algoritmos iterativos y recursivos. https://drive.google.com/file/d/1qF7eDq0a7MvrFTbfWVJF7qKb9LZTrezb/view. Accessed 21 Jan 2020
2. Knuth, D.E.: The Art of Computer Programming, Volume 3, Searching and Sorting. Addison-Wisley, Reading, MA (1973)
3. Ljung, L., Söderström, T.: Theory and Practice of Recursive Identification. MIT press, Cambridge (1983)
4. Shih-hua, H.: Theory of recursive algorithms I. Sci. Sinica **9**, 843–875 (1960)
5. Vorobiev, N.N.: Fibonacci numbers. Birkhäuser (2012)
6. Wing, J.M.: Computational thinking. Commun. ACM **49**(3), 33–35 (2006). https://doi.org/10.1145/1118178.1118215
7. Zapata-Ros, M.: Pensamiento computacional: Una nueva alfabetización digital. Revista de Educación a Distancia (RED) (September 2015). https://doi.org/10.6018/red/46/4