The case of base cases: Why are they so difficult to recognize? Student difficulties with recursion

Conference Paper *in* ACM SIGCSE Bulletin · September 2002

DOI: 10.1145/637610.544441 · Source: DBLP

2 authors, including:

Bruria Haberman
Holon Institute of Technology
79 PUBLICATIONS **581** CITATIONS

SEE PROFILE

The Case of Base Cases: Why are They so Difficult to Recognize? Student Difficulties with Recursion

Bruria Haberman
Department of Computer Science
Holon Academic Institute of Technology,
and Department of Science Teaching
Weizmann Institute of Science, Rehovot, Israel
bruria.haberman@weizmann.ac.il

Haim Averbuch
Department of Computer Science
The Open University
Tel Aviv, Israel
averbuch_haim@hotmail.com

ABSTRACT
Recursion is a central concept in computer science, yet it is a very difficult concept for beginners to learn. In this paper we focus on a specific aspect of comprehending recursion - the conception of the base case as an integral component of a recursive algorithm. We found that students have difficulties in identifying base cases: they handle redundant base cases; ignore boundary values and degenerated cases; avoid out-of-range values; and may even not define any base cases when formulating recursive algorithms. We also found that students have difficulties in evaluating recursive algorithms that deal with imperceptible base cases. We suggest that teachers should make a special effort to discuss different aspects of the base case concept. Emphasis should be put on both declarative and procedural aspects of categorizing and handling base cases as part of recursion formulation.

Categories and Subject Descriptors
F.3.3 [**Logic and Meanings of Programs**]: Studies of Program Constructs – *program and recursion schemes.*

General Terms
Algorithms.

Keywords
Recursion formulation, recursion evaluation, base case.

1. INTRODUCTION
Recursion is a central concept in computer science and is considered a powerful and useful problem-solving tool. As such, it is taught in almost every introductory course in computer science. Research studies have concluded that the concept of recursion is difficult to learn and comprehend. Moreover, students have difficulties in applying recursion in their problem-solving activities [1,6,7,9]. Comprehending the concept of recursion, and its use to solve problems, is expressed in the ability to evaluate and formulate recursive algorithms. Kahney [6] tested the hypothesis that novice programmers and experts differ in terms of their mental models of recursion as a process. He showed that experts usually possess a "copies model" of recursion, whereas novices usually possess a "loop model". Kahney and Eisendstst [7] found that novices may acquire several mental models of recursion besides the "loop model".

Various studies have concentrated on enhancing recursion evaluation ability by deepening students' understanding of recursion tracing. Wilcoks and Sanders [11] and Kann et al. [5] showed that animation that illustrates the "copies model" can enhance recursive function evaluation. George [3] claimed that the teaching of recursion may be best facilitated by teaching students how to simulate the execution of a recursive algorithm using diagrammatic traces. Importantly, he showed that diagrammatic traces eventually enhanced students' ability to evaluate embedded recursive algorithms. However, studies that concentrated on enhancing recursion formulation abilities reported different approaches. For example, Wilcoks and Sanders [11] reported that most students who used an animator to learn recursion indicated that the animator did not assist them in formulating recursive algorithms. Sooriamurthy [9] suggested that the key to comprehending recursion is to focus on the functional abstraction of recursive functions. He developed a template-based approach designed to guide students through the various steps in formulating a recursive solution. Similarly, Ginat and Shifroni [4] observed that the emphasis on the declarative approach for teaching recursion, and the emphasis on the abstract level of problem decomposition considerably improved formulating recursive programs.

In this paper we focused on a specific aspect of comprehending recursion - the role of base cases in recursion formulation.

There are two aspects of base cases. The first is based on a declarative, abstract approach that treats base cases as the smallest instances (in terms of problem size) of the problem for which we know the answer immediately, without any effort. It may be the smallest concrete entity, a boundary value, or a degenerated case. It also presents the "smallest" possible input of the problem. The second aspect is based on the procedural approach, and refers to the base case as a stopping condition. In this sense, it represents the end of decomposing the problem to smaller similar problems. In order to get a comprehensive view of the role of base cases in recursion formulation, one should adopt both the declarative and the procedural approaches.

ITiCSE'02, June 24-26, 2002, Aarhus, Denmark.

In this paper we describe student difficulties in identifying and handling base cases, and their influence on the properties of recursive algorithms. Finally, we present suggests for instruction.

2. OUR STUDY

The main goal of our study was to evaluate student conceptions of the base case as an integral component of a recursive solution. We tried to reveal students' difficulties in identifying and handling base cases in recursion formulation and evaluation. The research population consisted of pre-college beginners who learned recursion at an introductory level, and advanced college students, who learned recursive manipulation of compound data structures and abstract data types.

Our study was carried out in two stages. The first stage was designed to collect and analyze recursive algorithms that students formulated to solve seven standard problems (presented in 3.1). Besides identifying the wrong solutions, we were especially interested in "odd" though correct solutions that might have indicated alternative student conceptions regarding the role of the base case(s) in recursion formulation. The algorithms were collected through interviews with expert computer science teachers, and by reading students' notes.

The second stage of the research included an analysis of student recursion evaluation. The research subjects consisted of 42 beginners (2 classes) and 74 advanced students (3 classes), none of whom participated in the first stage. We prepared a diagnostic questionnaire of 6 questions. Each question included a description of a problem, and two recursive solutions that differed because of the identification of the problem's base case(s): a "school solution", and a wrong solution, or a correct yet complex solution. The problems and the solutions were chosen from the collection of algorithms obtained during the first stage of the study (P1-P6, see 3.1). Students were asked to assess the correctness, readability, and generality of each solution, and to indicate what solution they preferred, and to justify their views. After checking the data, we interviewed students on a personal basis, to get an in-depth view of their conceptions.

3. RESULTS

3.1 Student Recursion Formulation

In this subsection we present examples of students' common solutions of seven standard problems. Besides the "school solutions" we present examples of incorrect and overcomplicated algorithms that were formulated due to an unsuitable choice of base case. The examples are gradually presented, starting with simple problems about natural numbers, and proceed to advanced problems with compound data types.

P0. Factorial: Students were asked to formulate a recursive algorithm for computing a factorial according to the following definition: "Factorial of zero is 1. Factorial of N (N>0) is the product of all the natural numbers between 1 to N". Here we present four common solutions: Solution S0.a has only one base case (the case of N=0), and is the simplest correct solution of the problem. Although solution S0.b is correct, it includes a redundant base case (the case of N=1). Students were comfortable with S0.b because they preferred to isolate the case when N=0 from the other cases, believing, as one student put it, that "*the additional base case N=1 is essential to prevent multiplying by zero*". Solution S0.c is incorrect because it ignores the case of N=0. Moreover, solution S0.d ignores both cases of N=0 and N=1, and is based on the underlying assumption that "*the computation automatically terminates, because N is a natural number*".

Solution S0.a

```
Factorial(N)
   if N=0 then   return 1
   else return  Factorial(N-1)*N
```

Solution S0.b

```
Factorial(N)
   if N=0 then  return 1
    else
      if N=1 then   return 1
      else  return Factorial(N-1)*N
```

Solution S0.c

```
Factorial(N)
   if N=1 then   return 1
   else return   Factorial(N-1)*N
```

Solution S0.d

```
Factorial(N)
   return Factorial(N-1)*N
```

P1. Sum of digits of a natural number: The presented solutions are correct and differ only in their base cases. Solution S1.a has the base case of Num=0, and solution S1.b has the base case of one-digit number (Num<=9) that obviously includes the case of Num=0. Apparently, there is no appreciable difference between the algorithms. However, the first algorithm (S1.a) reflects students' abstract thinking: "*0 is the smallest natural number, and therefore presents the simplest instance of the problem*". Students who formulated the second algorithm (S1.b) considered any one-digit number as the simplest instance of the problem. From a procedural point of view, S1.b reflects the students' strategy of "*preventing the total destruction of the number as a consequence of its decomposition*".

Solution S1.a

```
Sum_of_Digits(Num)
   if Num=0  then  return 0
   else  return Sum_of_Digits(Num div 10) + Num mod 10
```

Solution S1.b

```
Sum_of_Digits(Num)
   if Num <= 9  then  return Num
   else    return Sum_of_Digits(Num div 10) + Num mod 10
```

P2. Is there an odd digit in a non-negative integer number? This example demonstrates how the choice of an unsuitable base case might cause code duplication. In solution S2.b a conditional statement was duplicated to handle the base case.

Solution S2.a

```
Odd_Digit?(Num)
   if Num=0  then  return False
    else
      if Num mod 2 =1 then   return True
      else  return   Odd_Digit?(Num div 10)
```

Solution S2.b

```
Odd_Digit?(Num)
   if Num <= 9 then
      if  Num mod 2 =1 then   return True
      else   return False
   else
      if  Num mod 2 =1 then   return True
      else   return Odd_Digit?(Num div 10)
```

P3. Is the array ascendant sorted? Similarly to example P2, this example illustrates the redundancy of statements, resulting from not accepting the simplest instance of a problem to be the base case. Solution S3.b reflects the students' assumption that the simplest case that justifies checking if an array is sorted is where the array has at least two elements.

Solution S3.a

```
Is_Sorted?(A, Size)
   if  Size = 1  then  return True
    else
      if A[Size] <= A[Size-1]  then    return False
       else   return   Is_Sorted?(A, Size-1)
```

Solution S3.b

```
Is_Sorted?(A, Size)
   if  Size = 2  then
      if A[Size] <= A[Size-1] then  return False
       else    return True
    else
      if A[Size] <= A[Size-1]  then   return False
       else
          return   Is_Sorted?(A, Size-1)
```

P4. Membership in a list: Students were asked to formulate a recursive algorithm for checking whether item X is a member of a list L. They were allowed to use the functions first(L) and tail(L) that return, respectively the first element in L, the tail-list of L, and the function is-empty?(L) that returns True if L is empty, and False otherwise. S4.a is the correct solution for the problem. It has two base cases: (a) the case of an empty list. In this case X is not a member of L, and consequently the returned value is False; and (b) the case where the first element in the list L equals X, and is therefore a member of L. In other cases, the membership is checked in the list's tail.

Solution S4.a

```
Member(X,L)

        if Is-Empty?(L) then   return False
         else
            if X = First(L) then  return True
             else   return     Member(X, Tail(X))
```

Solution S4.b

```
Member(X,L)

         if X = First(L)   return True
         else   return    Member(X, Tail(X))
```

Solution S4.b has only one base case. Students who formulated this solution recognized the base case as the trivial case of checking membership in a non empty list: "*If the given item X equals the first element of the list, then it is a member of the list, and then we don't have to check anymore*". But they ignored the case where the membership does not exist. Accordingly, S4.b is incorrect because it produces correct answers only in the cases where the membership holds.

P5. How many nodes are in a binary tree? This example clearly describes the effect of a wrong choice of a base case on the complexity of the entire code. Instead of recognizing that the problem has exactly one base case - the case of an empty tree (see solution S5.b), students handled four different base cases: an empty tree, a tree with a single node, a tree that has a root and a left sub-tree only, and a tree that has a root and a right sub-tree only (see solution S5.a). Apparently, students did not recognize the three last cases as instances of the general case. Consequently, they formulated a correct but a complex solution (S5.a) that includes a compound nested if statement, and is not easy to read and to comprehend.

Solution S5.a

```
Count(T)
  if  Is_Empty?(T) then   return  0
  else
      if  Is_Empty?(Left_Sub_Tree (T)) and
             Is_Empty?(Right_Sub_Tree(T)) then   return 1
    else
        if  Is_Empty?(Left_Sub_Tree (T))  then
           return  1 + Count(Right_Sub_Tree(T))
        else
            if  Is_Empty?(Right_Sub_Tree (T))  then
                return  1 + Count(Left_Sub_Tree(T))
            else
                return  1 + Count(Left_Sub_Tree(T))  +
                                Count(Right_Sub_Tree(T))
```

Solution S5.b

```
Count(T)
  if  Is_Empty?(T) then  return  0
  else
    return  1 + Count(Left_Sub_Tree(T)) +
                        Count(Right_Sub_Tree(T))
```

P6. How many different paths of N steps can a knight move on the chessboard from one square to another? This example impressively illustrates the results of avoiding the use of out-of-range values to characterize the base case. Students who formulated solution S6.a "prevented" the knight from going out of the chessboard. To do so, they had to check 8 times if the next step is "legal" (within the borders of the chessboard). In contrast, the students who considered the "out of borders" case as a possible base case formulated a much simpler solution (S6.b).

Solution S6.a

```
Chess(X,Y,Xtarget,Ytarget, N)

 if  N=0 then
    if  X= Xtarget and Y= Ytarget  then  return  1
    else   return  0
 else
      Sum  ←  0
      if   X+1 in [1..8] and  Y+2  in [1..8]  then
        Sum  ←   Sum  + Chess(X+1,Y+2,Xtarget,Ytarget, N-1)
      if   X-1 in [1..8] and  Y+2  in [1..8]  then
        Sum  ←   Sum  + Chess(X-1,Y+2,Xtarget,Ytarget, N-1)
      if   X+1 in [1..8] and  Y-2  in [1..8]  then
        Sum  ←   Sum  + Chess(X+1,Y-2,Xtarget,Ytarget, N-1)
      if   X-1 in [1..8] and  Y-2  in [1..8]  then
        Sum  ←   Sum  + Chess(X-1,Y-2,Xtarget,Ytarget, N-1)
      if   X+2 in [1..8] and  Y+1  in [1..8]  then
        Sum  ←   Sum + Chess(X+2,Y+1,Xtarget,Ytarget, N-1)
      if   X-2 in [1..8] and  Y+1  in [1..8]  then
        Sum  ←   Sum + Chess(X-2,Y+1,Xtarget,Ytarget, N-1)
      if   X+2 in [1..8] and  Y-1  in [1..8]  then
        Sum  ←   Sum + Chess(X+2,Y-1,Xtarget,Ytarget, N-1)
       if   X-2 in [1..8] and  Y-1  in [1..8]  then
         Sum  ←   Sum + Chess(X-2,Y-1,Xtarget,Ytarget, N-1)
return Sum
```

Solution S6.b

```
Chess(X,Y,Xtarget,Ytarget, N)
  if X or Y are out of borders then return 0
  else
      if N=0 then
         if X= Xtarget and Y= Ytarget then return 1
        else return 0
      else return Chess(X+1,Y+2,Xtarget,Ytarget, N-1) +
                  Chess(X-1,Y+2, Xtarget,Ytarget, N-1) +
                  Chess(X+1,Y-2, Xtarget,Ytarget, N-1) +
                  Chess(X-1, Y-2, Xtarget,Ytarget, N-1) +
                  Chess(X+2,Y+1,Xtarget,Ytarget, N-1) +
                  Chess(X-2,Y+1, Xtarget,Ytarget, N-1) +
                  Chess(X+2,Y-1, Xtarget,Ytarget, N-1) +
                  Chess(X-2,Y-1, Xtarget,Ytarget, N-1)
```

The examples presented clearly illustrate the implications of the choice of base cases on the properties of recursive algorithms. In general, the students' desire for concrete and detailed base cases usually extracts a heavy price in terms of formulating compound, long, and sometimes incorrect algorithms. Students who accept boundary, degenerated, and even out-of-range cases as possible base cases, gain the opportunity to formulate compact and elegant algorithms.

3.2 Student Recursion Evaluation

Here we presented the results of the diagnostic questionnaire. We focused on students' evaluation of the algorithms' correctness, and on their preference of alternative solutions to a given problem.

Preference of algorithms: In order to determine what algorithm they preferred, students referred to the following criteria: correctness, complexity of code, and the type of the base case(s). It seems that students gave the highest priority to the correctness of the algorithms. Where students presumed that only one of the algorithms was correct they preferred the correct one. The next priority was to check the complexity of the code, and finally the type of the base case(s). If they presumed that both algorithms were correct (or both incorrect), and had very similar code, they preferred the algorithm with the concrete base case (S1.a, ~70% of both advanced and beginners). In contrast, if they presumed that both algorithms were correct (or both incorrect), but differed in their code, they preferred the algorithm with a simpler code, even though it had a degenerated base case (e.g., S2.b: 69% advanced, 60% beginners, vs. S2.a: 26% advanced, 26% beginners).

Beginners and advanced students justified differently their preferences. Advanced students usually justified their preference in terms of correctness and efficiency considerations, whereas beginners referred mostly to readability and code length.

Correctness: The findings indicated that students had misconceptions regarding the concept of correctness. Many students indicated that an algorithm was correct even though it did not handle every possible legal instance (input) of the problem (e.g., S3.a: 76% of advanced, and 40% of beginners, whom indicated that the solution was correct). "*As long as it works all right for every case that it refers to, it is considered correct*". In contrast, students indicated that an algorithm was incorrect when it handled imperceptible, though relevant base cases (e.g., S1.b: 4% advanced, 19% beginners).

4. DISCUSSION

4.1 Classification of Student Difficulties

The examples described here illustrate four types of difficulties in determining base cases:

(a) Ignorance of boundary cases: Students tend to ignore the "small instances" of problems. They do not include boundary values in the range of the allowed values for the variables that describe the problem. Similarly, they do not refer to degenerated instances of data structures (e.g., as empty list, empty tree). This type of bug is characteristic of the problems beginners have in deciding on appropriate boundary conditions in various domains [10]. Sometimes boundary cases are hidden in other base cases (e.g., S1.b, S2.b). Here, the fact that they are explicitly ignored does not spoil the correctness of the solution. However, there are cases where the ignorance of the boundary cases yields incorrect solutions (e.g., S3.b, S4.b).

(b) Avoiding the use of out-of-range values: Sometimes, in order to simplify and to generalize the treatment of a set of possible boundary cases, "getting out of borders" is needed (e.g. S6.b). This action is very complicated for students who avoid the use of out-of-range values because they consider those values illegal. The cost of this avoidance is again expressed in terms of code complexity (e.g. S6.a).

(c) Lack of base cases: Sometimes students absolutely ignore base cases, and formulate incorrect recursive algorithms that do not include any termination conditions (e.g. S0.d).

(d) Redundancy of base cases: Redundancy of base cases occurs whenever, besides the "smallest instances" of the problem, additional cases are identified as base cases (e.g. S5.a). This results in formulating an algorithm that is much more complex than an algorithm that handles only the simplest case (e.g. S5.b)

4.2 Possible Explanations

Spohrer and Soloway [10] found that "boundary bug probably is not a result of any misunderstanding of language constructs, and appears to be symptomatic of a more general problem when they try to categorize and handle boundary points". Here we suggest some possible explanations for student difficulties with base cases. Our study supports the first explanation, and the rest should be tested in another study.

(a) Concrete vs. the abstract problem-solving approach:
Students that use an abstract approach to analyze problems mostly refer to the recursive structure of the data that should be manipulated and may easily recognize boundary and degenerated cases of that structure. However, they might ignore the procedural aspects of the recursive process, and therefore avoid essential stopping conditions. In contrast, students who have a concrete-based style of problem solving, perform the step-by-step analysis trace of a recursive process, and merely refer to case bases as stopping conditions.

(b) Mistaken use of problem-solving methods: Students may wrongly use the bottom-up problem-solving approach. They start to analyze the problem by testing simple cases, and gradually proceed to test more general cases. With recursion formulation, students eventually have to split the problem space into base case(s), and the rest to general cases. Students who have difficulties in making that distinction, usually use redundant base cases.

(c) The influence of concrete conceptual models: Teachers often use concrete conceptual models, such as the "Russian Dolls"

model to teach recursion. Although concrete conceptual models may help novice programmers to learn recursion [2,12], they might also cause some misconceptions. For example, if the Russian Doll contains a most inner doll that is not decomposable, it might mean that the base case should always be the smallest concrete case of the problem.

(d) Transfer from other programming paradigms: Students who are acquainted with different programming paradigms may transfer problem-solving techniques from one programming environment to another. We found that students who experienced list processing in Prolog tended to ignore boundary cases that control the termination of recursive computation. This can be explained by the characteristics of the Prolog language. The Prolog computational mechanism returns "no" in case of a failure and "yes" in case of a success. It enables the programmer to concentrate on the declarative and abstract aspects of problem solving, and liberates him from dealing with the procedural details of the computational process. Therefore, the programmer only has to formulate the conditions for success, and does not have to bother about the case of failure. We found a substantial support for this hypothesis in students' evaluations of solution S4.a, which is an example of an incorrect attempt to rewrite a declarative Prolog definition of the membership in terms of a procedural algorithm. The percentage of students who considered S4.a as a correct solution, and had experienced in Prolog programming (40%) was significantly higher that the percentage of the students who learned only a procedural programming language (23%) (McNemar's test: chi-square=37, $p<0.001$).

(e) Transfer from programming constrains: Teachers often warn students not to get out of arrays index range. This may prevent students from *"getting out of borders"* when needed (e.g. S6.a).

5. CONCLUDING REMARKS

In this paper we described students' difficulties with recursion regarding base cases. We demonstrated how identifying and handling of base cases in recursion formulation affects the correctness, readability, and code complexity of recursive algorithms.

We would like to conclude by suggesting some didactic recommendations for overcoming student difficulties:

* Emphasis on the declarative and abstract aspects of recursion may help eliminate students' difficulties with recursion [4,9]. We suggest that teachers make a special effort to discuss different facets of the base case concept. Emphasis should be placed on both declarative and procedural aspects of categorizing and handling base cases as part of recursion formulation. Base cases should be treated as the smallest instances of the problem's legal input, and not merely as supporting stopping conditions [8].

* Teachers should be very cautious in adapting or designing concrete models [12]. For example, when using the Russian Dolls model, the teacher should use a structure of a Russian doll that contains a most inner decomposable doll that does not contain another. This structure illustrates the possibility of null boundary values.

* Teachers may help eliminate bugs by making students explicitly aware of the problems that they may come encounter. Diagnostic questionnaires like the one presented here may be used as a learning class activity, and as a basis for a class debate about the classification and handling of base cases.

* Students who learn different programming paradigms should be guided to use self-control strategies to avoid misleading transfer from one paradigm to another.

6. REFERENCES

[1] Anderson, J.R., Prirolli, P., and Farrell, R. Learning to program recursive functions. In The Nature of Expertise. Chi, M.T., Glaser, R., and Farr, M.J. (eds.). Hillsdale, NJ: Lawrence Erlbaum Associates, 1988, 151-183.

[2] Ben-Ari, M. Recursion: From drama to program. Journal of Computer Science Education. 11(3), 1997, 9-12.

[3] George, E.C. Experience with novices: The importance of graphical representations in supporting mental models. In Blackwell, A.F. and Bilotta, E. (Eds). Proceedings of the 12th Workshop of the Psychology of Programming Interest Group, 2000, 33-44.

[4] Ginat, D. and Shifroni, E. Teaching recursion in a procedural environment - How much should we emphasize the computing model? Proceedings of the 30th SIGCSE technical symposium on computer science education, 1999, 127-131.

[5] Kann, C., Lindenman, R., and Heller, R. Integrating algorithm animation into a learning environment. Computers Educ., 28 (4), 223-228, 1997.

[6] Kahney, H. What do novice programmers know about recursion? Proceedings of the CHI '83 Conference on Human Factors in Computer Systems, 1983, 235-239.

[7] Kahney, H. and Eisenstadt, M. Programmers' mental models of their programming tasks: The interaction of real world knowledge and programming knowledge. Proceedings of the 4th Annual Conference of the Cognitive Science Society, 1982, 143-145.

[8] Segal, J. Empirical studies of functional programming learners evaluating recursive functions. Instructional Science, 22, 385-411, 1995.

[9] Sooriamurthi, R. Problems in comprehending recursion and suggested solutions. Proceedings of the 6th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, 2001, 25-28.

[10] Spohrer, J.G., and Soloway, E. Analyzing the high frequency bugs in novice programs. In Empirical Studies of Programmers. Soloway E. and Iyengar, S. (eds.). Albex Publishing Corporation, Norwood, New Jersey, 1986, 230-251.

[11] Wilcocks, D., and Sanders, I., Animating recursion as an aid to instruction. Computers Educ. 23(3), 221-226, 1994.

[12] Wu, C.C., Dale, N.B., and Bethel, L.J. Conceptual models and cognitive learning styles in teaching recursion. Proceedings of the 29th SIGCSE technical symposium on computer science education, 1998, 292-296.