

Do Senior CS Students Capitalize on Recursion?

David Ginat

CS Group, Science Education Department
Tel-Aviv University, Israel
ginat@post.tau.ac.il

ABSTRACT

CS students learn and practice recursion in CS1, Data-Structures, Introduction-to-Algorithms, and additional courses throughout the curriculum. Previous studies revealed difficulties of CS1 students with the concept and the construct of recursion. What about advanced students? They may well understand the concept and the construct of recursion; but do they invoke and utilize recursion as a problem solving means? The paper examines this aspect, with senior CS students. The students were given three algorithmic tasks, for which the suitable solution approach was recursive. The student solutions and explanations demonstrate very limited capitalization on recursion as a problem solving means. We discuss the findings and suggest pedagogical implications for teaching.

Categories & Subject Descriptors

K.3: [Computers & Education] Computer and Information Science Education – Computer Science Education.

General Terms

Algorithms, Experimentation.

Keywords

Recursion, Ways of Reasoning, Student Errors, Pedagogy.

1. INTRODUCTION

Recursion is a fundamental way for approaching mathematical and algorithmic problems. It embodies the essential problem solving heuristic of “working backwards” [10] as well as the core programming notion of specifying solutions for “larger” task instances based on solutions for “smaller” instances. Recursion is introduced in CS1 and then repeatedly utilized in further courses.

How is recursion practiced in CS1? In the paradigm of functional programming, recursion is employed from the very beginning. In procedural and object-oriented programming, recursion is usually presented at a later stage, following a variety of other language constructs. The typical tasks at this stage direct the students to invoke recursion (i.e., “Write a recursive program ...”).

Permission to make digital or hand copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, require prior specific permission and/or a fee.

ITiCSE '04, June 28-30, 2004, Leeds, United Kingdom.
Copyright 2004 ACM 1-58113-836-9/04/0006...\$5.00.

How is recursion practiced in later courses? In Data-Structures and Algorithms courses it often is encapsulated in particular recursive structures and schemes, such as trees, graphs, traversal schemes, and sorting schemes. In the rather advanced theoretical courses it is more apparent as a conceptual notion, in various definitions, reductions, and argumentations.

Recursion is not a trivial topic. Difficulties with recursion were thoroughly studied in the past two decades. The focus of all the studies was on student comprehension of the basic concept and construct of recursion (e.g., [2,3,4,5,8,13]). However, recursion is not just a concept-construct element, it also is a problem-solving heuristic, which encapsulates backward reasoning and a reverse train of thought [10,12]. Do students turn to recursion and recursive thinking as a problem-solving heuristic? Do they employ the reverse train of thought of “working backwards”? The objective of this paper is to shed light on this question.

In CS1, students are explicitly asked to employ recursion in the chapter in which it is introduced. In later courses, they employ it upon practicing particular topics that involve recursion (e.g., tree traversals), often through problems posed at the end of the topics’ chapters. However, competence with recursion invocation should not be tied to particular topics or chapters. Upon approaching a programming (algorithmic) task, a computer scientist is not told which way to go. She has to decide by herself. Sometimes, a task may be properly solved by following different, alternative “working” directions; but sometimes one particular direction may be suitable.

One may argue that experience with recursion in the different CS courses yields sufficient “maturity”, in the sense of being able to identify the suitability of the “working” direction. Is this indeed the case? The study in this paper reveals student competence in this respect and discusses its consequences.

The study was conducted with 23 CS college-level students, who completed CS1, CS2 (mostly with C++), Introduction-to-Algorithms, and additional courses that involved recursion. During a didactical course, which was part of their teaching-diploma studies, they were introduced to various pedagogical aspects of CS, which involved (among other things) the examination of different ways to approach an algorithmic task. The students were given a variety of tasks to solve (using their preferred programming language). Their solutions were examined. Some were interviewed about their solutions.

The paper displays and discusses the student solutions to three tasks, which focused on examining problem solving approaches when no explicit hint of direction is provided. The first task was a CS1 optimization task; the second – a CS1 counting task; and the

third – a CS2 counting task. Optimization and counting are common and fundamental in various CS domains.

In each of the next three sections we display a task, its solutions, and student comments on their solutions. We name solutions by their underlying approaches. We then discuss our findings and suggest relevant consequences for CS educators, based on our way of coping with the study’s results.

2. SEQUENCE OPTIMIZATION

The first task posed to the students stems from the following situation. A piece of cloth has to be measured by a sequence of operations, of two kinds – an addition of 1 centimeter to the “prefix” measured so far, or the doubling of this “prefix” (by unfolding it on the remaining cloth). The goal is to minimize the total number (the length of the sequence) of operations.

Min-Num-of-Ops. Write a program for which the input is the positive integers X and Y , $X < Y$, and its output is the minimal number of invocations of the operations $+1$ and $\times 2$ that are required to obtain Y from X .

Example. For the input 10 17, the output will be 7 (due to 7 invocations of $+1$). For the input 10 21, the output will be 2 (due to $\times 2$ and then $+1$).

More than half of the students approached the task in forward reasoning ways, and did not attempt a backward reasoning, recursive way. The interviewed students who followed the forward reasoning ways indicated that they did not consider working backwards. We display the students’ forward ways and the backward, recursive way.

Forward-Greed

Six of the 23 students solved the task according to the **greedy** principle “multiply by 2 as long as it is possible”. Their idea was to “increase X towards Y by the largest increase possible at each step”. The rationale for this idea was that the operation $\times 2$ yields a larger increase than $+1$, “so it is better to invoke it as long as possible”. The typical code was:

```
NumOfOps ← 0;
While X < Y do
  if 2X ≤ Y then X ← 2X else X ← X+1;
  NumOfOps ← NumOfOps+1
od
```

In interviews, students mentioned that they reached conviction rather quickly and intuitively, without really validating it. They “felt” that correctness was “obvious”! Even simpler examples show that this greedy approach is erroneous; e.g., for the input 10 22, the greedy-way output is 3, whereas the output should be 2 (as $+1$ and then $\times 2$ yield 22).

Forward-Stages

Four of the 23 students solved the task according to some heuristic rule that they derived from examining a few examples. These students noticed that it may sometimes be better to first apply $+1$ and only later apply $\times 2$, as $(X+1)\times 2$ “leads further” than $(X\times 2)+1$. They devised various “forward computation” schemes that progress in **stages**.

A couple of them noticed that “consecutive multiplications by 2 imply an increase by a power of 2”. Therefore, their rule was to

“repeatedly apply $+1$ on X , in a 1st stage, until it will reach a value from which a series of $\times 2$ will lead as close as possible to Y ”. The 2nd stage was the series of $\times 2$ operations, and a 3rd stage was another series of $+1$ operations. Thus, for the input 10 25, X was advanced to 11, 12, 24, 25; and for the input 10 50, X was advanced to 11, 12, 24, 48, 49, 50. The former sequence is indeed optimal, but the latter is not (11, 12, 24, 25, 50 is better). Thus, these students also derived erroneous solutions.

Forward-Recording

Four of the 23 students felt “unsafe” to form an unfounded forward-advancement rule, and looked for a “safe” way to compute forward. They noticed that it is possible to “carry forward” from each i the minimal number of operations to $i+1$ and to $2i$. They used an array of size $Y-X+1$, and yielded:

```
A[X] ← 0; A[X+1..Y] ← Y;
for i from X to Y-1 do
  A[i+1] ← min(A[i]+1, A[2i]);
  if 2i ≤ Y then A[2i] ← A[i]+1
od
```

The idea behind this solution is to compute the minimal number of operations needed for reaching each i in the range $[X..Y]$. Once this number is known for i , it is “propagated” forward with an addition of $+1$ for the cases of $i+1$ and $2i$.

It can be shown by induction on i that this solution yields the minimal number of operations for each i . Thus, this solution is correct, and actually applies a dynamic programming approach, by performing optimization for all the problem instances, from the smallest to the largest [1]. However, the complexity of the above solution is significant: $O(Y-X)$ time and space. The space complexity is particularly problematic, as for a very large Y and a small X , the above solution may require an unreasonable amount of space.

Backward-Reduction

Only nine of the 23 (less than half) of the students approached the task in the suitable way. These students approached the task by “reducing Y towards X in a minimal number of steps”. They noticed that if Y is odd, then the last operation in the sequence that yields Y must be $+1$. If Y is even, then the last operation should be $\times 2$. The key-point in their solutions was their “**reverse view**” of the operations $+1$ and $\times 2$. Going backwards, they transformed them to -1 and $/2$. Some of these students provided a recursive solution, and others – an iterative one. Yet, **reasoning backwards**, which is the core of recursive thinking, was the basis for both solution types. The typical recursive code was:

```
Num(Y, X) :
  if 2X > Y then Num ← Y-X
  else if Odd(Y) then Num ← Num(X, Y-1)+1
  else Num ← Num(X, Y/2)+1
```

The correctness of this solution is based on the argument that the number of operations for reaching N from any integer i greater than $\lfloor N/2 \rfloor$ is: $N - i$; and therefore, the “fastest” way to advance from $\lfloor N/2 \rfloor$ to N is by a multiplication by 2.

This solution is concise and very efficient – its time complexity is $O(\log(Y-X))$ and its space complexity is $O(1)$ (once the recursive concept is transformed to iteration). Notice that this solution is greedy, as the sequence of reverse operations is obtained by making the best (backwards) choice possible in each step.

Interestingly, both this “backward-greed” solution and the previously presented Forward-Greed solution are greedy, simple, and short. Yet, this solution always yields the optimal sequence of operations, whereas the other does not.

3. PATH COUNTING

The second task posed to the students involved counting of the number of different ways to reach a basketball score. Counting of the number of different ways to reach a goal is apparent in a variety of applications.

Basketball-Score. Write an algorithm for which the input is a positive integer N , indicating the number of points a team accumulated in a basketball game, and its output is the number of different ways that these points could have been accumulated. Recall that in basketball the number of points can increase by 1, 2, or 3 at a time.

Example. For the input 3, the output will be 4, since there are 4 different ways to accumulate 3: 1+1+1, 1+2, 2+1, 3.

As in the previous task, here too the majority of the students chose forward reasoning ways, and did not attempt a backward reasoning, recursive approach.

Forward-Generalization

Nine of the 23 students looked for an **arithmetic pattern**. They noticed that the output for the input 2 is 2, the output for 3 is 4, and the output for 4 is 7. A few students examined the output for the input 5 and erroneously reached 11 (rather than 13). At this stage, five of them decided that the general pattern of this sequence is 1,2,4,7,11; and therefore, “the difference between every two consecutive elements grows by 1 as we progress along the sequence”. Their typical solution was very simple:

```
Ways ← 1;
for i from 1 to N-1 do
    Ways ← Ways+i
```

Some of the students offered a mathematical formula, rather than the code. The students indicated “a feeling that there had to be a general formula ... derived from a pattern of the simple cases ...”. When asked to justify their chosen pattern, they could not offer a reasonable explanation. However, they believed that their pattern made sense, as it involved an increasing gap between successive elements. They did not consider approaching the problem from a different, possibly backwards direction.

Four students obtained the correct sequence: 1,2,4,7,13. Two of them noticed that each element is the sum of the three elements that precede it. They provided a corresponding iterative code. However, they could not clearly explain why this solution is correct. They repeated saying that the value for any score N is “propagated” forwards, but did not elaborate any further.

A few additional students attempted an enumeration approach. They tried to enumerate and count the different ways to reach N , through the scoring sequences $\langle 1,1,1,1,\dots \rangle$; $\langle 2,1,1,1,\dots \rangle$; $\langle 1,2,1,1,\dots \rangle$ up to the sequence $\langle 3,3,\dots \rangle$. At a certain point they abandoned this direction, as “it became too cumbersome”.

Backward-Decomposition

Only seven of the 23 students (less than a third) solved and properly justified their solution, by reasoning backwards. They looked at the last step that yields the score N , and noticed that it

may be a 1-point step, a 2-point step, or a 3-point step. The 1-point step occurs in a “scoring path” that ends with $\langle \dots N-1, N \rangle$; the 2-point step occurs in a path that ends with $\langle \dots N-2, N \rangle$; and the 3-point step – in a path that ends with $\langle \dots N-3, N \rangle$. They reversed the +1, +2, and +3 increments, and offered the **recursive decomposition** rule:

$$\text{Ways}(N) \leftarrow \text{Ways}(N-1) + \text{Ways}(N-2) + \text{Ways}(N-3)$$

These students were able to see that $\text{Ways}(N-3)$ contributes directly and indirectly to $\text{Ways}(N)$ – directly, as stated in the formula, and indirectly – through the values of $\text{Ways}(N-1)$ and $\text{Ways}(N-2)$. Most of them were aware of the inefficiency of implementing the recursive rule by a recursive function, and implemented it using iteration, starting from $\text{Ways}(1)$, $\text{Ways}(2)$, $\text{Ways}(3)$, and progressing “upwards”, using only $O(N)$ time and $O(1)$ space.

4. INVERSION COUNTING

The third task involved counting of the number of inversions in a list of numbers. An *inversion* is a pair $\langle i, j \rangle$ of elements in a list L of distinct elements, such that $i > j$ and i appears somewhere before j in L [6]. In examining an unordered list L , one may enquire about the number of inversions, as a means for measuring the amount of “disorder” in L .

Num-of-Inversions. Write an efficient algorithm for which the input is a list of N distinct integers and the output is the number of inversions in the list.

Example: for the input list 2 9 1 8, the output will be 3 (due to the inversions $\langle 2, 1 \rangle$, $\langle 9, 1 \rangle$, and $\langle 9, 8 \rangle$).

The vast majority of the students chose forward reasoning ways, and did not yield any backward reasoning, recursive way.

Forward-Accumulation

Fifteen of the 23 students offered the straightforward principle of separately counting for each element the number of larger elements that follow it in the list. This solution encapsulates a naïve **forward accumulation** of all the inversions of each element.

These students realized that its $O(N^2)$ computation time does not address the task’s efficiency requirement, but they could not come up with a better solution. In particular, they did not see any recursive pattern.

A few other students tried variants of Bubble-Sort and Insertion-Sort, as the “rolling” of elements in these schemes yields a one-by-one reduction of the number of inversions. However, their solutions did not improve the $O(N^2)$ time complexity.

Divide-and-Conquer

Only five of the 23 students noticed that the necessary computation can be elegantly and efficiently performed using a **recursive divide-and-conquer** scheme. Reasoning backwards, they noticed that if the list is divided into two sub-lists, and these sub-lists are ordered, then the merging of these sub-lists yields the number of inversions in the original list (which is composed from the concatenation of the two sub-lists).

Let the left, ordered sub-list be left_L and the right sub-list be right_L . The students noticed that when an element i from left_L is chosen for the ordered, merged list, it is the leftmost

among all the (`left_L` and `right_L`) elements not yet merged. However, when an element from `right_L` is chosen, it is to the right of all the remaining (not-yet-merged) elements in `left_L`, and therefore selecting it for the merged list reduces the total number of inversions by the size of `left_L` (since `i` is put in the merged list before all the elements still in `left_L`).

For example, let 5 7 9 6 8 10 be a list to be ordered, by merging its two already-ordered halves 5 7 9 (`left_L`) and 6 8 10 (`right_L`). First, 5 is chosen to be the leftmost in the merged list; then 6 is chosen to be next in the merged list, and therefore “skipped over” the two elements, 7 and 9, remaining in `left_L`; then, 7 is chosen; then 8 is chosen and “skipped over” the remaining element, 9 in `left_L`; and finally the 9 and the 10 are chosen. The total length of the “skips” is $2+1=3$, which is exactly the number of inversions in the initial list (due to the unordered pairs $\langle 7,6 \rangle$, $\langle 9,6 \rangle$, $\langle 9,8 \rangle$).

The outline of the recursive scheme is the following:

```
Inversions(L):
  if |L|=1 then Inversions ← 0
  else{ Inversions ← Inversions(left_L)
        + Inversions(right_L);
        Inversions ← Inversions
          + Merge(left_L, right_L)
        }
```

The two recursive calls `Inversions(left_L)` and `Inversions(right_L)` return `left_L` and `right_L` as ordered lists, together with the number of inversions in each of these lists. These ordered lists are then merged by `Merge`, and the number of inversions computed during the merge (as described above) is added to the total number of inversions.

All in all, the above scheme is based on the design pattern of Merge-Sort. Since the time complexity of Merge-Sort is $O(N \log N)$, the computation of the number of inversions will be done in time $O(N \log N)$, which is much better than $O(N^2)$.

4. DISCUSSION

What do we learn from the above results? We see that for all the three tasks, the majority of the students did not invoke or follow a backward reasoning, recursive approach. Instead, they attempted various forward reasoning ways, some of which were erroneous and others that were inefficient.

These students were past their CS1, CS2, and Introduction-to-Algorithms courses, in which they have seen a variety of recursive solutions. Yet, when given algorithmic tasks not explicitly tied to a particular topic, and requested to decide for themselves on the solution approach, they did not recognize and employ the suitable recursive way.

The Forward-Greed students in the first task invoked an intuitive, yet erroneous forward reasoning approach, without debating its appropriateness. The Forward-Stages students devised a forward reasoning rule, which was suitable only for particular examples. The Forward-Recording students related the solution principle to the familiar notion of progressing forward in a dynamic programming manner. The Forward-Generalization students in the second task erroneously generalized a pattern from simple cases, and the Forward-Accumulation students in the third task could not reach a pattern better than repeated forward accumulation.

We may interpret the inappropriate reasoning direction of the majority of the students in light of Lithner’s established-experiences view [9] and Schoenfeld’s monitoring component in his problem-solving model [12]. Lithner illuminated mathematics students’ tendencies to follow improper solution directions based on their established experiences, rather than plausible reasoning. Schoenfeld suggests that a primary element that distinguishes between novices and experts is the ability to properly monitor and control one’s own problem solving process.

The students’ tendency in our study to work forwards may be tied to their established experiences in solving algorithmic problems forward, and their rather limited, “technical” experience in solving problems recursively backward. The considerable gap between these two types of experiences may have led students to unsuitable reasoning directions as well as improper monitoring of the suitability of these directions. They confidently followed familiar forward directions with little hesitation and no quest for other directions.

While the students may have acquired and repeatedly employed the concept and the construct of recursion, it seems that they did not assimilate recursion as a general reasoning and problem solving means. In CS1, CS2, and the Introduction-to-Algorithms course they learnt and utilized a variety of recursive schemes. However, these schemes were usually very specific and strongly tied to very particular computations, such as tree and graph traversals. The “drill and practice” tasks, which were posed to the students with these schemes, usually focused on assimilating these schemes, and involved rather little emphasis on the general notion of reasoning backwards.

Unlike the specific experiences with recursion, forward solutions were repeatedly employed in diverse ways, throughout the students’ courses, with a broad repertoire of techniques and illustrations, including those of “forward greed” and dynamic programming. Thus, the “recursive experiences” may have yielded for many students only very specific solution schemes, whereas the forward solutions yielded both specific and more general problem solving schemes.

The recursive, backward reasoning approach is a problem solving means on which one should be able to capitalize again and again, from the very beginning of the CS studies. We believe that improved emphasis, and awareness of recursion as a problem-solving means should contribute to the development of such competence.

Following the results of the study we specified and applied with our students several guidelines for capitalizing on recursion as a problem-solving approach.

- **Recursion as a problem-solving means.** When recursion is introduced, display it not only as a concept-construct-process, but also as a means to reason about a task to solve.
- **No task cues.** Repeatedly pose tasks to students, in various places throughout their studies (course), which are not explicitly linked to the latest topic. Do not indicate or hint for any solution direction. Advise the students to explore, during solving the tasks, various approaches, including that of reasoning backwards.
- **Unsuitable ways.** Emphasize erroneous and inefficient cases of forward reasoning that could look relevant at first glance, but are actually inappropriate. Reasoning backwards and

recursion may not be the first heuristic that comes to mind, but may enclose the suitable approach.

- **Reflection on the solution process.** Upon realizing the essential role of backward reasoning in solving a given task, reflect on the process that yielded this realization. In particular, display various considerations that yielded the recursive outcome.
- **Design patterns perspective.** Upon displaying a recursive solution, demonstrate it not only as a solution to the particular problem, but also as a design-pattern, which may be used in various places for similar goals. Each of the recursive solutions of the tasks in this study may be displayed in such a way.
- **Recursive decomposition diversity.** Occasionally summarize to students the various recursive decomposition schemes that they have seen. Relate and compare these schemes to one another, and tie each of them to particular tasks.

Each of the above points can be illustrated through the three tasks that were posed to our students. In all three tasks, recursion was not just the construct by which to express the solution, but rather the underlying reasoning heuristic for solving the task. No cues appeared in the task specification that hint for recursion. Alternative forward solutions could seem at first glance very relevant, but were erroneous or inefficient.

Each of the recursive solutions of the three tasks is based on a different recursive decomposition, or design pattern. The recursive solution of the first task embodies a linear end-decomposition; the recursive solution of the second represents a “k-tuple decomposition”, and the recursive solution of the third is a divide-and-conquer decomposition.

We believe that the elaboration of recursion as a problem solving means should start in the CS1 course, and be constantly advocated in the courses that follow. Diverse tasks, such as those displayed in our study should be embedded throughout the various courses, in order to underline the role of recursion as a problem solving means. This means is essential for computer scientists first and foremost as a fundamental way of approaching programming and algorithmic tasks.

5. REFERENCES

- [1] Cormen T.H., Leiserson, C.E., & Rivest, R.L., *Introduction to Algorithms*, MIT Press, (1990).
- [2] DiCheva, D. & Close, J., Mental models of recursion. *Journal of Educational Computing Research*, 14 (1), (1996), 1-23.
- [3] Gotschi, T., Sanders, I., & Galpin, V., Mental models of recursion, *Proc of the 34th SIGCSE Symposium*, ACM Press, (2003), 346-350.
- [4] Ginat, D. & Shifroni, E., Teaching recursion in a procedural environment – how much should we emphasize the computing model? *Proc of the 30th SIGCSE Symposium*, ACM Press, (1999), 127-131.
- [5] Kahney, H., What do novice programmers know about recursion? *Proc of CHI 1983*, (1983), 235-239.
- [6] Knuth, D. E., *The Art of Computer Programming*, Addison-Wesley, (1973).
- [7] Koffman, E. & Wolz, W., *Problem Solving with Java*, Addison-Wesley, (1999).
- [8] Levi, D., Insights and conflicts in discussing recursion: a case study. *Computer Science Education*, 11 (4), (2001), 305-322.
- [9] Lithner, J., Mathematical reasoning in task solving. *Educational Studies in Mathematics*, 41, (2000), 165-190.
- [10] Polya, G., *How to Solve It*, Princeton University Press, (1957).
- [11] Savitch, W., *Problem Solving with C++*, Addison-Wesley, (1996).
- [12] Schoenfeld, A. E., Learning to think mathematically: problem solving, metacognition, and sense making in mathematics. Grouws D. A. (Ed.), *Handbook of Research on Mathematics Teaching and Learning*, (1992), 334-370.
- [13] Wilcocks, D. & Sanders, I., Animating recursion as an aid to instruction. *Computers and Education*, 23 (3), (1994), 221-226.