Pictures as Invariants

Owen Astrachan
Computer Science Dept.
Duke University
Durham, NC 27706
ola@cs.duke.edu

Abstract

The development, specification, and use of a loop invariant are useful and underutilized tools in writing code to solve a problem or implement an algorithm. Invariants are especially useful in introductory courses (CS1 and CS2) but are usually avoided because of the mathematical notation associated with them and because most textbooks have brief, if any, coverage of them. Our intent in this paper is provide several motivating examples of the use of pictures as loop invariants and to provide pointers to literature that more fully explores and develops the ideas of using loop invariants in specifying and developing programs.

1 Introduction

The development, specification, and use of a loop invariant are useful and underutilized tools in writing code to solve a problem or implement an algorithm. Invariants are especially useful in introductory courses (CS1 and CS2) but are usually avoided because of the mathematical notation associated with them and because most textbooks have brief, if any, coverage of them.

Our intent in this paper is provide several motivating examples of the use of pictures as loop invariants and to provide pointers to literature that more fully explores and develops the ideas of using loop invariants in specifying and developing programs.

A *loop invariant* is a statement that is true before and after each iteration of a loop. Such statements can be used to reason formally about the correctness of a loop and, more to the purpose of this paper, to help develop a loop to meet a specification.

There are several fundamentally different approaches to introductory Computer Science courses. These approaches differ in languages used, concepts emphasized, and even in whether computers are used [Dij89],[DPS+90]. We believe that each of these different camps has something to learn from the others.

The approach outlined here is far from the formal approach taken in [DF88], [Dij90] and [Gri81], but it borrows on many of the concepts introduced in those works . We hope to make some of these ideas more accessible and more palatable to teachers and students of introductory courses. We hope that some of the joy found by a programmer when a program runs and by a mathematician when a proof is complete can be shared by students in introductory Computer Science courses.

Invariants can serve as both aids in recalling the details of an implementation of a particular algorithm and with the construction of an algorithm to meet a specification.

We hope to show that invariants can be used in several ways in introductory courses. The first use might include the presentation of an algorithm (e.g., partition) with a diagram illustrating a loop invariant. The diagram can be used to reason about the code that implements the algorithm and as an aid to reconstructing the code. Subsequent uses of invariant diagrams can include the presentation of a problem along with an invariant that can be used in solving the problem (e.g., reversing a linked list) and the development of an invariant for a specific task.

In this paper we outline several small but

(hopefully) interesting problems. We offer terse solutions to these problems via the construction of loops from an invariant presented as a picture. All the code presented in this paper is written in Pascal with small liberties taken for purposes of clarity and succinctness. In many of our examples we use a Pascal *for* loop rather than a *while* loop. Although formal reasoning about loops requires a *while* loop (the value of the indexing variable of a Pascal *for* loop is undefined when the loop terminates), the *for* loop makes for more compact code in several of our examples.

2 Partitioning an Array

Partitioning an array about a pivot element is a fundamental part of two important algorithms: *Quicksort* and *Select*. Quicksort is an $\mathcal{O}(n \log n)$ expected time algorithm to sort n items and Select is an $\mathcal{O}(n)$ expected time algorithm to find the kth largest of n items. Both Quicksort and Select are often covered in introductory courses and are certainly covered in an algorithms course. The terse partition code shown below comes from [Ben86], a more pedagogical development of it can be found in [Kru87].

Informally, partitioning an array A involves rearranging the elements of A so that all the elements of A less than or equal to some value x precede all elements of A greater than x. Typically x is the first element of A and we will assume that this is the case in the following exposition. See [Knu73] for a full discussion about choosing the "correct" x .

More formally, we want to write a procedure *Partition* with the header

procedure *Partition* (**var** A : ArrayType;
m, n : integer;
var p : integer);

establishing

$$x = A[p] \ \wedge \ \forall_{m \leq j \leq p} \ A[j] \leq x \ \wedge \ \forall_{p < j \leq n} \ x < A[j] \tag{1}$$

Although this specifies the problem succinctly, to most students in introductory courses it does not specify the problem clearly. Consider the following diagram as a specification:

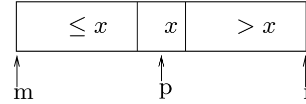

Figure 1: specification of partition

The information conveyed in the diagram is precisely that which is stated in the more formal equation 1, but understanding the diagram does not require an understanding of formal logic.

Using the diagram as a starting point, we can write an invariant for the partition algorithm that leads to a terse and easily verifiable partition procedure. Comparison of this code for partitioning an array with that found in several textbooks ([CC82],[TA81],[Kof89], [DL85],[AHU74]) shows that the code below is shorter (both in statements and number of lines) and clearer (subjectively).

The diagram for the invariant can be derived using techniques from [Gri81], our purpose in this paper is to show that the diagram is a useful tool. Intuitively, however, an invariant can often be developed from a specification by replacing a constant (in this case n which does not vary) with a variable. In this case, we maintain the picture specification above as an invariant by replacing n with i and requiring the invariant to hold through index i. The picture invariant is:

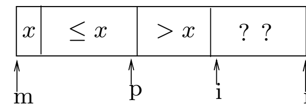

Figure 2: partition invariant

where the question marks indicate that elements in this subsection of the array have values whose relationship to x are not yet known. The first element of this subsection is referenced by i in the diagram. If $A[i] > x$ then incrementing i results in the re-establishment of the invariant while shrinking the size of the unknown section. If $A[i] \leq x$ then $A[i]$ is out of position and needs to be swapped. A brief reflection should show that incrementing p and swapping $A[i]$ with $A[p]$ leads to re-establishment of the invariant.

This results in the following code:

```
x := A[m];
p := m;
for i := m + 1 to n do
    if A[i] ≤ x then begin
        p := p + 1;
        Swap(A[p],A[i]);
    end;
```

To establish the diagram that specifies the problem (and thus the more formal equation 1) we need only swap $A[p]$ with $A[m]$ when the loop has terminated. The picture invariant of Figure 2 is equivalent to the following formula:

$$x = A[m] \ \wedge \ \forall_{m \leq j \leq p} \, A[j] \leq x \ \wedge \ \forall_{p < j < i} \, x < A[j] \tag{2}$$

Note the similarity between this formula and that of equation 1.

In teaching CS1 and CS2 courses prior to using the diagrams and code shown in this section we found that students had difficulty reconstructing code that implemented the partition algorithm. Students find it much simpler to derive the code, however, starting from the diagram of the invariant. Most can reconstruct the diagram once it has been discussed in class and use it, for example, in developing code that implements *Select*. It should be noted that the code shown above can be easily modified to squeeze in on the partition element $A[p]$ from both the right and the left (as is done in most textbooks) using a single loop with body of the form

if $A[p] \leq x$ **then** p := p + 1
else if $A[q] > x$ **then** q := q - 1
else Swap(A[p],A[q])

and an invariant of

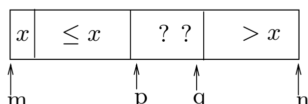

Figure 3: invariant for squeezing partition

3 Reversing the Elements of a Singly-linked list

Reversing the nodes of a singly-linked lists is a problem typical of the kind given in a CS2 or data-structures course. This problem appeared on the 1989 Advanced Placement Exam in Computer Science. Although this problem has a straightforward $\mathcal{O}(n)$ solution for a list of n elements, many students attempted complicated $\mathcal{O}(n^2)$ solutions, used auxiliary stacks, and in general (and not surprisingly) made mistakes manipulating pointers.

The problem can be specified pictorially by requiring that lists of the form shown in Figure 4

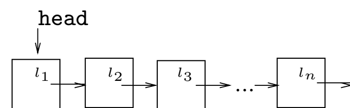

Figure 4: initial list

be transformed into lists of the form shown in Figure 5.

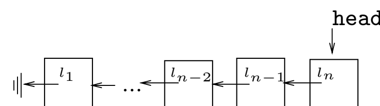

Figure 5: reversed list

We can derive a picture as an invariant with the insight that after several iterations of the loop part of the list will be reversed and the remainder of the list will be unprocessed. This can be described pictorially by the lists in Figure 6.

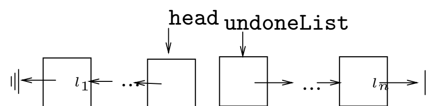

Figure 6: invariant for reversing a list

At this point, we invite the interested reader to develop a loop based on the picture invariant shown above. One need only move one node from the list referenced by `undoneList` to the list referenced by `head`.

The statements that accomplish this movement are

```
temp := undoneList↑.next;
undoneList↑.next := head;
head := undoneList;
undoneList := temp;
```

By enclosing these statements in a loop with appropriate initialization and termination statements we will have finished this problem. The appropriate initialization involves assigning values to `undoneList` and `head` so that the invariant is true before the loop iterates the first time. The termination statements may be necessary to achieve the state specified by the diagram that describes the problem.

Initially, the entire list remains to be processed (and, symmetrically, none of the list has been processed). The statements:

```
undoneList := head;
head := nil;
```

achieve this state. This results in the following final solution to the problem of reversing the nodes of a linked list.

```
undoneList := head;
head := nil;
while undoneList <> nil do begin
    temp := undoneList↑.next;
    undoneList↑.next := head;
    head := undoneList;
    undoneList := temp;
end;
```

Note that in this problem no termination statements are necessary since `head` references the first node of the reversed list as required by the picture specification of this problem. This might not have been the case, for example, if `head` had been used to point to that part of the list remaining to be processed and a pointer `alreadyDoneList` been used as a pointer to that part of the list already reversed. In this case `head` would not need to be initialized but would need to be reset when the loop terminates.

4 Compacting an Array

Consider the problem of removing the zeros from an array of n integers while leaving the order of the non-zero elements unchanged. It is similar to the more practical problem of replacing sequences of blanks by a single blank in a line of text or using run-length encoding for data compression. Removing zeros from an array was given on the 1987 Advanced Placement exam in Computer Science. There is a straightforward solution using an auxiliary array that is of complexity $\mathcal{O}(n)$, but on the AP exam students were prohibited from using such an auxiliary structure. Even so, there is a simple $\mathcal{O}(n)$ solution. In fact, the inplace solution requires at most n assignments to array elements as opposed to the solution using an auxiliary structure which may require $2n$ (to copy back to the original array). Nevertheless, most students attempted $\mathcal{O}(n^2)$ solutions or tried to process runs of consecutive zeros and missed special cases.

The simple observation that during the construction of the compacted array part of the array will be compacted and part of the array will remain unprocessed leads to the picture shown in Figure 7 as an invariant.

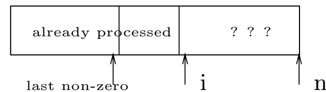

Figure 7: invariant for compacting an array

The picture illustrates that in the section of the array already processed, an index referencing the last non-zero element is maintained. As in the partition problem, we need only decide how to process an unknown element while maintaining the invariant. As in that problem there are two cases. In the first case, array element $A[i]$ is zero. In this case no array elements need be moved; incrementing i results in the re-establishment of the invariant while decreasing the size of the unknown section. In the second case, a non-zero element must be moved. The reference to the last non-zero element will be updated to reflect that this array element is the new last non-zero element. In addition, incrementing i results in re-establishment of the invariant as in the first case.

The code implementing this idea is

```
lastNonzero := 0;   {no non-zero elements yet}
for i := 1 to n do begin
    if A[i] <> 0 then begin
        lastNonzero := lastNonzero + 1;
        A[lastNonzero] := A[i]
    end;
```

5 Convex Hull

Pictures as invariants are often useful in the development of code as was observed when reversing a linked list. Pictures can also serve as an aid to students in recalling the details of how an algorithm is implemented as was seen in the partition example. The partition algorithm is relatively simple in contrast to some of the more complex algorithms encountered in CS2 courses. Pictures as invariants can be particularly useful in helping to recall the details of these more complicated algorithms.

Many interesting examples from computational geometry can be adapted for use in introductory courses [DeP88] [DBKL90]. Many of the problems from computational geometry are based on efficient sorting and searching techniques and provide contexts for studying these techniques that are interesting, novel, and visual.

The problem of finding the smallest convex polygon containing n specified points is the *convex hull* problem from computational geometry. Treatment of this problem can be found in several algorithms texts [Man89],[Sed88]. In this section we develop a version of the Graham Scan solution to the convex hull problem given in [PS85]. We show in Figure 8 a set of points on the left and the convex hull of these points on the right.

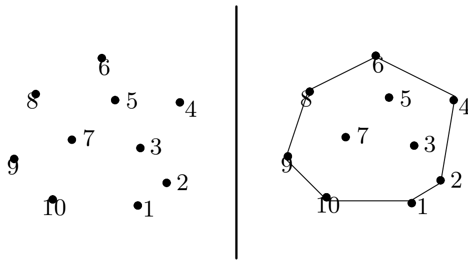

Figure 8: set of points and their convex hull

The first step of the Graham scan algorithm is to sort the points whose convex hull is being sought. The points are sorted with respect to their polar coordinates relative to a point known to be on the convex hull (or, alternatively, relative to a point known to be in the interior of the hull). We choose to sort relative to the point with smallest y-coordinate (and greatest x-coordinate if there are several points with the same smallest y-coordinate) first by angle θ and then, for points with equal values of θ, by distance. In our presentation we will assume that the sorted points are stored in a doubly-linked circular list whose first element is the point with smallest y-coordinate with respect to which the other points are sorted. Such a list is shown in Figure 9

This list is then processed using a three node "window" to determine which points are in the convex hull. As an aid to reconstructing the algorithm, the invariant shown in Figure 10 includes both the constructed hull and the window. The nodes that are labeled as "part of the convex hull so far" may or may not be part of the final convex hull as will be seen. Note that the first two nodes in the window are also part of the convex hull according to the picture. We will refer to the first node in the window as the *anchor node*. Note also that the invariant is initially true since by sorting the points we ensure that the first two points in the initial window are part of the convex hull (by proper initialization as shown below). The point to be considered for inclusion in the hull is the third point of the window.

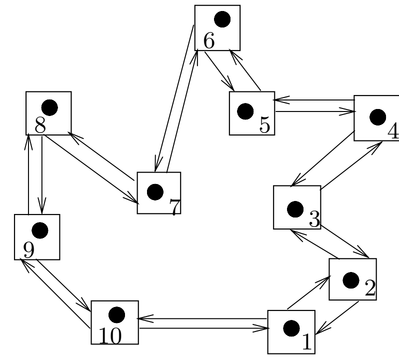

Figure 9: linked list for Graham scan

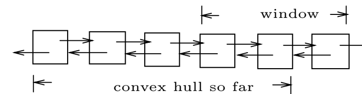

Figure 10: invariant for Graham scan

As in the other examples we have presented, the essential part of the Graham scan algorithm is developing the step that re-establishes the invariant while ensuring termination. The three points in the window (considered in order from the anchor node's point) have two possible configurations:

- they form a convex angle
- they form a reflex (non-convex) angle

In the first case (illustrated by nodes 1, 2, and 3 in Figure 9), the anchor node is advanced by one node. Since the points in the window form a convex angle, the invariant is maintained. In the second case (illustrated by nodes 2, 3, and 4 in Figure 9), the second node of the window cannot be part of the convex hull and is deleted. The anchor node is also moved back one node so that the invariant is re-established. If the anchor node is not moved back, the invariant will not necessarily remain true. The first two nodes of

the window cannot be guaranteed to be part of the convex hull of the points considered.

To see that the anchor node must be moved back, consider the points shown in Figure 11.

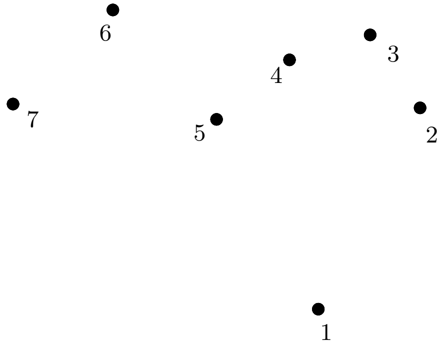

Figure 11: moving the anchor node back

When the anchor node is point 4, the invariant ensures that points 1 through 5 are part of the convex hull so far. Since points 4, 5, and 6 fall into the second category above (they form a reflex angle), point 5 will be removed from the list. If the anchor point is not moved back, node 7 will be the next node considered as it will be the third point of the window anchored at point 4. Note that the invariant will not be true since the first two points of the window — points 4 and 6 — are not part of the convex hull of points 1 through 6.

If, however, the anchor node is moved back to point 3, the invariant will be re-established. Note that the next step of the Graham scan will remove point 4 and move the anchor to point 2.

The Graham scan algorithm will terminate when all points of the list are known to be in the convex hull. The picture invariant shows that this will occur when the third node of the window is the first node of the linked list. This leads to the following implementation of the Graham scan.

```
{sort points into list as outlined above }
{let the first node of the list be denoted by first }
anchor := first;
while anchor↑.next <> first do begin
    if IsReflex(anchor, anchor↑.next,
            anchor↑.next↑.next)
    then
        anchor := anchor↑.next;
    else begin
        RemoveNode(anchor↑.next);
        anchor := anchor↑.previous;
    end;
end;
```

Since each iteration of the loop advances the window or removes a point from the list, the loop above runs in $\mathcal{O}(n)$ time for a list of n points. This makes the sorting of the points the bottleneck for this problem and the complexity for the entire algorithm is then $\mathcal{O}(n \log n)$.

6 Conclusion

We hope that interested readers will pursue the use of picture invariants and, perhaps, more formal invariants in introductory courses. When used as an aid in the development of algorithms and to remember details of algorithms rather than as a chore to develop after a loop is written, picture invariants can be made accessible to students without formal training. A comprehensive introduction to the use of more formal techniques can be found in [Gri81]. The author gives many examples, uses diagrams as invariants, and offers useful techniques for developing invariants. Invariants are used in a lower-level text in [Dro82] which is also a source of many useful examples. Finally, we urge all teachers of introductory courses to read [Har87] which covers (briefly) invariants as well as many of the most fundamental concepts of Computer Science in a manner accessible to beginning students.

References

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[Ben86] Jon Bentley. *Programming Pearls.* Addison-Wesley, 1986.

[CC82] Doug Cooper and Michael Clancy. *Oh! Pascal!* W. W. Norton & Co., second edition, 1982.

[DBKL90] N. Adlai A. DePano, Rarinaz D. Boudreau, Philip Katner, and Brian Li. Algorithmic Paradigms: Examples in Computational Geometry II. In *SIGSCE Technical Symposium on Computer Science Education*, pages 186–191, 1990.

[DeP88] N. A. A. DePano. Algorithmic Paradigms: Examples in Computational Geometry. In *SIGSCE Technical Symposium on Computer Science Education*, pages 83–87, 1988.

[DF88] Edsger W. Dijkstra and W.H.J. Feijen. *A Method of Programming*. Addison-Wesley, 1988.

[Dij89] Edsger W. Dijkstra. On the Cruelty of Really Teaching Computer Science. In *SIGSCE Technical Symposium on Computer Science Education*, pages xxv–xxxix, 1989.

[Dij90] Edsger W. Dijkstra, editor. *Formal Development of Programs and Proofs*. Addison-Wesley, 1990.

[DL85] Nell Dale and Susan C. Lilly. *Pascal plus Data Structures, Algorithms, and Advanced Programming*. D.C. Heath and Company, 1985.

[DPS+90] Edsger Dijkstra, David Parnas, William Scherlis, M.H. van Emden, Jacques Cohen, Richard Hamming, Richard M. Karp, and Terry Winograd. A Debate on Teaching Computer Science. *Communications of the ACM*, 32(12):1397–1414, December 1990.

[Dro82] R.G. Dromey. *How To Solve it by Computer*. Prentice-Hall International, 1982.

[Gri81] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[Har87] David Harel. *Algorithmics The Spirit of Computing*. Addison-Wesley, 1987.

[Knu73] Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1973.

[Kof89] Elliot B. Koffman. *Turbo Pascal*. Addison-Wesley, second edition, 1989.

[Kru87] Robert L. Kruse. *Data Structures & Program Design*. Prentice Hall, second edition, 1987.

[Man89] Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989.

[PS85] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry An Introduction*. Springer-Verlag, 1985.

[Sed88] Robert Sedgewick. *Algorithms*. Addison-Wesley, second edition, 1988.

[TA81] Aaron M. Tenenbaum and Moshe J. Augenstein. *Data Structures Using Pascal*. Prentice Hall, 1981.