Vector Programming Using Structural Recursion
An Introduction to Vectors for Beginners

Marco T. Morazán
Seton Hall University
morazanm@shu.edu

Vector programming is an important topic in many Introduction to Computer Science courses. Despite the importance of vectors, learning vector programming is a source of frustration to many students. Even though the size of a vector is a natural number, there have been no efforts to define a useful recursive data definition to help beginners design vector processing functions. This article defines the concept of a vector interval and describes how to exploit its recursive structure to design vector processing functions. The described methodology provides a context beginners can use to reason about proper vector indexing instead of leaving them adrift with this responsibility.

1 Introduction

It may very well be true that every college-trained computer programmer remembers long nights debugging programs that manipulate vectors. Perhaps, many readers of this article recall hours of work trying to determine why an index into a vector was out of range. In some programming languages, this is equivalent to figuring out why a program caused a core dump. If these same readers ponder about this long enough, they may recall that many of those long frustrating nights occurred when they were first exposed to vectors and beginning to learn how to program. Surprisingly, the same holds true for many undergraduates today. Frankly, it is a bit shocking how little we have advanced as a community in teaching vector programming to beginners.

Vector programming (a.k.a. array programming), of course, is extremely important in Computer Science. Vectors, given that they are random access, allow us to efficiently implement algorithms for applications in a myriad of fields. Vectors, for example, allow us to efficiently implement various data structures such as binary trees [6], stacks [7], and process queues [15]. They also allow us to reduce the complexity of algorithms such as, for example, finding a path in a directed graph [4] and matrix multiplication [2]. Vectors are also useful in the reduction of memory allocation by, for example, allowing us to sort, say, files or numbers in place [9]. Given the importance and versatility of vectors it is in our interest as a community to make an introduction to vector programming for beginners as frustrationless as possible. This can be achieved by providing beginners with a structured model they can use to reason about processing vectors.

The crux of the problem with much of the developed introductory material to vector programming is that it provides a structureless definition of a vector. Some material goes a step farther and presents an ADT to for vectors. Some of these ADTs describe a single operation: indexing. It is, thus, not surprising that many beginners feel as being left adrift to figure things out on their own. This is highly undesirable for at least two reasons. The first is that students get frustrated enough to quit Computer Science as a major. This is an especially important issue in universities, like in the USA, where beginning students spend a semester or two shopping around for a major. The second is that students that persevere develop bad programming habits associated with the belief that vector programming is a strict exercise in trial and error instead of an exercise in design.

To appear in EPTCS.

This article describes the work developed to introduce students to vector programming at Seton Hall University (SHU). At the heart of the approach is providing students with data definitions that help them design vector processing functions to solve problems. As the length of a vector can be of arbitrary size, these must be recursive data definitions. Such data definitions can be directly exploited using structural recursion to develop programs. The model presented to the students is that of a *vector interval* that may only contain valid indices into a vector. As the reader shall discover, this model is also useful when designing functions based on generative and accumulative recursion. The article is organized as follows. Section 2 discusses related work. Section 3 discusses the programming background of the students with whom this approach is used. Section 4 introduces and defines the concept of an interval. In addition, it provides examples of how to design functions using the different data definitions for an interval. Section 5 shows how students are introduced to vector programming by developing data definitions for a vector interval. Section 6 discusses three extended vector programming examples using the described approach. Finally, Section 7 presents concluding remarks and directions for future work.

2 Related Work

Many textbooks introduce vectors as lacking a recursive structure that can be exploited to solve problems. Readers are then introduced to how to use vectors using snippets of code that emphasize that an index into the vector must be within its bounds. For example, a vector is described as a collection of variables of the same type with each element having an index [7] or as a finite sequential list of elements of the same datatype identifying the first element, the second element, the third element, and so forth [6]. Such data definitions are inadequate, because they hide the recursive nature of the interval of valid indices into the vector and focus exclusively on the syntax to declare, create, index, and mutate vectors. They fail to provide the proper model to help beginners design functions/methods that process a vector avoiding, for example, illegal indexes into the vector. The second data definition may even be considered misleading by describing a vector as a list that has a well-known recursive structure. Vectors do not have a decomposable structure like lists. Furthermore, describing vector elements as the first, the second, the third, and so forth does not assist in any way the design of vector processing functions. We can not program “and so forth.” Even some modern approaches to make programming popular among the young address vectors in a very similar manner (e.g., [3]). In contrast, the work presented in this article aims to provide students with a decomposable data definition that beginners can use to reason about vector processing. This data definition is that of a vector interval. These intervals have a recursive structure that guides the design of vector processing functions.

The problem of only using legal indices into a vector is traditionally and summarily left to the student with no clear indication of how to accomplish this task (e.g., [14]). Some may convincingly argue that this is relatively easy when you need to process an entire vector. However, the matter is not clear when you need to process only part a the vector. Consider sorting a vector using quicksort which requires partitioning and independently sorting different parts of the vector. It is difficult for a beginner to determine or be confident that she is always correctly indexing the vector. Even worse, it is more difficult to pin down bugs when indexing errors occur if a model that helps the student reason about indexes is not provided to them. In contrast, the work presented in this article helps students reason about the indexes into a vector. A vector interval only spans the valid indexes of the part of the vector being processed. As such, when the structure of an interval is used to design a function students know that any natural number in the vector interval is a valid index. If the vector interval is empty then students know that the vector should not be indexed. Even when functions are not designed using structural recursion, reasoning about vector intervals is helpful.

Some efforts have gone beyond syntax. The textbook *How to Design Programs* (HtDP), for example, describes a vector as a well-defined mathematical class of data with some basic constructors and observers [4]. HtDP further states that we can think of vectors as functions on a small finite range of natural numbers. This begins to provide some context for vector processing, but surprisingly falls short of identifying the recursive structure of this range of natural numbers as it does so well for other types of data. It is unlikely that future editions of HtDP are to develop this given that the second edition has eliminated its introduction to vector programming [5]. In contrast, the work presented in this article tackles the recursive nature of this finite range of natural numbers and describes it as an interval. Furthermore, the fact that this range is finite is carried to its logical conclusion to obtain two data definitions for a vector interval. One data definition leads to a template that is used to design functions to process vectors from right to left (i.e., from the largest index down to the lowest index in the vector interval) and the other is used to design functions that process vectors from left to right (i.e., from the lowest index to the largest index in the vector interval).

3 Student Background

At SHU, the introductory Computer Science courses span two semesters and focus on problem solving using a computer [10, 11]. The languages of instruction are the successively richer subsets of Racket known as the student languages which are tightly-coupled with HtDP [4, 5]. No prior experience with programming is assumed. Before introducing students to vector programming, the course familiarizes students with primitive data (e.g., numbers, strings, booleans, symbols, and images), primitive functions, and library functions to manipulate images (i.e., the image teachpack). During this introduction, students are taught about variables, defining their own functions, and the importance of writing contracts and purpose statements. The next step of the course introduces students to data analysis and programming with compound data of finite size (i.e., structures). At this point, students are introduced to the first design recipe. Students develop experience in developing data definitions, examples for data definitions, function templates, and tests for all the functions they write. A great deal of emphasis is placed on all of these steps as part of the problem-solving design process. Building on this experience, students develop expertise on processing compound data of arbitrary size such as lists, natural numbers, and trees. In this part of the course, students learn to design functions using structural recursion. After structural recursion, students are introduced to functional abstraction and the use of higher-order functions such and `map` and `filter`. The first course ends with a module on distributed programming [12].

In the second course students are exposed to generative recursion, accumulative recursion, and mutation [13]. The course starts with generative recursion. At the end of this module, students get their first exposure to vector programming. Students are taught the syntax needed for vectors and are introduced to the design of vector processing functions using the material on intervals and vector intervals outlined in this article. After this, the course exposes students to accumulative recursion and iteration. The course ends with two modules on mutation that include their second exposure to vector programming. In this second exposure, students design vector mutators using vector intervals.

These topics covered follow much of the structure of HtDP [4]. There are two 75-minute lectures every week and the typical classroom has between 20 to 25 students. In addition to the lectures, the instructor is available to students during office hours (3 hours/week) and there are 20-30 hours of tutoring each week which the students may voluntarily attend. The tutoring hours are conducted by undergraduate students handpicked and trained by the author. These tutors focus on making sure students develop answers for each step of the design recipe (from writing contracts to running tests). Students must attempt to follow the steps of the design recipe prior to attending tutoring. Based on a student's work, the tutors provide guidance but do not solve problems. Students are still responsible for successfully completing all steps of the design recipe. In addition, tutors attend lectures to assist students when they get stuck with, for example, syntax errors. This type of team-teaching with undergraduate tutors has proven to be extremely well-received by students and to be an effective means to enhance the learning experience.

4 Intervals

Before introducing students to vectors, they are re-introduced to the concept of an interval. The term re-introduced is used in the same manner as students being re-introduced to natural numbers earlier in the course. That is, students in general are familiar with at least one of the following "definitions" for the set of natural numbers:

$$\mathbb{N} \;=\; \{0, 1, 2, 3, \ldots\} \qquad\qquad \mathbb{N} \;=\; \{1, 2, 3, \ldots\}.$$

Both of these definitions are inadequate, because they do not describe how to construct a natural number. Furthermore, students are left to figure out the meaning of $\ldots$. Knowing how to construct a natural number is important, because it empowers students with the knowledge needed to process such numbers by exploiting their structure. Therefore, a more useful data definition for the set of natural numbers is (e.g., adopted in HtDP):

```
A natural number (natnum) is either:
     1. 0
     2. (add1 n), where n is a natural number.
```

Such a definition exposes the structure of natural numbers and is used to define a template to write functions that process a natural number:

```
f-on-natnum: natnum → ...
Purpose: ...
(define (f-on-natnum n)
  (cond [(= n 0) ...]
        [else n...(f-on-natnum (sub1 n))])).
```

The body of this template, in essence, states that the conditional distinguishes between the varieties of natural numbers. For each variety an expression is needed to compute the result. When a natural number, n, is a constructed natural number (i.e., the second variety) the expression can manipulate the value of n and can recursively process, (sub1 n), the natural number used to construct n. This template is then specialized by students every time they need to solve a problem that requires processing a natural number. Specializing, in this context, means filling in the blanks (i.e., the different ...).

Students bring to the classroom an understanding about intervals analogous to their initial understanding of natural numbers. That is, they define an interval as:

[$i..j$], where $i \;\leq\; j$

Once again, such a definition is inadequate. It does not expose the structure of an interval that is helpful to solve problems that require processing an interval. Furthermore, the fact that an interval can be empty is well-hidden by such a definition. Given that students are already familiar with recursive data definitions, it is not much of an intellectual leap to re-define an interval, INTV, as:

```
For a given integer n, an INTV is two integers, low and high, such that
either it is:
     1. empty  (i.e., low > high)
     2. [low..high], where high = n+1 and low ≤ high
```

The natural way to represent intervals is with a structure or an object that has two fields. Choosing a two-integer representation is a concession to current practices in existing programming textbooks. To the best knowledge of the author, there are no programming books that capture in a structure or an object the lowest and highest indexes of an interval. A judgement call had to be made between representing an interval as two integers or as a structure/object. Given that beginning students will read programming books that explicitly use two indices to process a vector, the two-integer representation was ultimately chosen. It does have the advantage that it makes the material feel somewhat familiar to students that arrive in the classroom with vector programming experience.

The INTV data definition makes the structure of an interval explicit. Students know that given that there is variety in the data definition a conditional is needed to distinguish among the different varieties. Furthermore, students can observe that when the interval is not empty high is a whole number constructed using n. This means that [low..(sub1 high)] is part of the structure of [low..high]. Put differently, [low..(sub1 high)] is used to construct [low..high]. For example, the interval [-1..1] is constructed as follows:

```
[-1..1] = [[-1..0]..1]
        = [[-1..-1]..0..1]
        = [[-1..-2]..-1..0..1]
        = [empty..-1..0..1]
```

Now it becomes clear that when the interval is not empty the value of high as well as the result of recursively processing the subinterval [low..(sub1 high)] can be used. This naturally leads to the following function template to process an interval:

```
; f-on-INTV: int int → ...
; Purpose: For the given INTV, ...
(define (f-on-INTV low high)
  (cond [(empty-INTV? low high) ...]
        [else high...(f-on-INT low (sub1 high))]))
```

This template requires a function to detect that an interval is empty. For the chosen representation using two integers, this function is easily developed by students:

```
; empty-INTV?: int int → boolean
; Purpose: For the given INTV, determine if it is empty
(define (empty-INTV? low high) (> low high))
```

The template can now be used to solve problems that process an interval. For instance, consider computing the summation of all the integers in an interval. Students know to start with the template for an INTV and to develop an answer for each variety of the data starting with the non-recursive case(s). Students quickly observe that when the interval is empty the summation is 0. When the interval is not empty, they observe that high must be added to the result of recursively processing [low..(sub1 high)]. Observe how reasoning about the structure of an interval leads the programmer to a solution. Putting these ideas together leads to the following specialization of the template:

```
; sum-INTV: int int → int
; Purpose: For the given INTV, sum its elements
(define (sum-interval low high)
  (cond [(empty-INTV? low high) 0]
        [else (+ high (sum-INTV low (sub1 high)))]))
```

After working out some exercises, students realize that the template suggests that intervals must be processed from high to low (or right to left). However, many students also realize that it may be equally correct to process an interval from low to high (or left to right). This requires the development of the following data definition for an interval:

```
For a given integer n, an INTV2 is two integers, low and high, such that
it is either:
    1. empty  (i.e., low > high)
    2. [low..high], where low = n-1  and low ≤ high
```

In this data definition, low is constructed by subtracting 1 from some integer n. This leads to the following function template:

```
; f-on-INTV2: int int → ...
; Purpose: ...
(define (f-on-INTV2 low high)
  (cond [(empty-INTV2? low high) ...]
        [else low...(f-on-INTV2 (add1 low) high)]))
```

It is important to highlight to students that the above template is not an instance of generative recursion. Many students see add1 and associate it with generative recursion and mistakenly feel they must develop a termination argument for function written using this template.

Once armed with this knowledge, students can now solve problems processing the interval from left to right. For instance, summing all the integers in an interval can also be solved as follows:

```
; sum-INTV2: int int → int
; Purpose: Sum all the integers in the given interval
(define (sum-INTV2 low high)
  (cond [(empty-INTV2? low high) 0]
        [else (+ low (sum-INTV2 (add1 low) high))]))
```

Both functions to sum the elements of an interval, have a delayed operation (i.e., $+$). These functions are rewritten using an accumulator to eliminate the delayed operation when students are exposed to accumulative recursion.

5 Vector Processing

Armed with an understanding of how to process intervals, students are ready to be introduced to vectors. After introducing students to what a vector is, why it is desirable to use them, and some basic vector-syntax, students are explained that it is common to process a contiguous subset of a vector. The emphasis here is on *common* given that arrays are random access and can be processed in many different ways. Nonetheless, the reader is reminded that the goal is to expose students for the first time to vectors and, as such, intervals are useful to reason about and design programs to process a contiguous subset of a vector.

For example, for a given vector V, we may want to process the entire vector (from indices 0 to (sub1 (vector-length V)) or we may want to process only part of the vector (from indices a to b). Clearly,

```
; f-on-vector: (vector X) → ...
; Purpose: ...
(define (f-on-vector V)
  (local [; f-on-VINTV: int int → ...
          ; Purpose: For the given VINTV, ...
          (define (f-on-VINTV low high)
            (cond [(empty-VINT? low high) ...]
                  [else (vector-ref V high)
                        ...(f-on-VINTV low (sub1 high))]))

          ; f-on-VINTV2: int int → ...
          ; Purpose: For the given VINTV2, ...
          (define (f-on-VINTV2 low high)
            (cond [(empty-VINTV2? low high) ...]
                  [else (vector-ref V low)
                        ...(f-on-VINTV2 (add1 low) high)]))]
    ...))
```

Figure 1: The Template for Functions on Vectors.

processing a contiguous subset of a vector requires processing an interval. Once again, this is not a huge intellectual leap for students. Care must be taken, however, because we must avoid attempting to access a vector with an illegal index that is either negative or greater than or equal to the length of the vector. This requires developing a data definition for a *vector interval*. A vector interval is an interval that places restrictions on what values low and high may take. In general, a valid index into a vector, V, is between 0 and (sub1 (vector-length V)). Thus, we can define a vector interval as follows:

```
Given a vector of length N and a natural number n, a vector interval,
VINTV, is two integers, low >= 0 and -1 <= high <= N-1, such that it
is either:
  1. empty  (i.e., low > high)
  2. [low..high], where high=n+1 and low ≤ high
```

Observe that this data definition restricts a VINTV to only contain valid indices into the vector when it is not empty. These indices are natural numbers. Further observe that the structure of a vector interval is exactly the same as the structure of an interval. There is a difference when processing a VINTV. We are interested in processing vector elements instead of interval elements. This means that in the body of a function to process a VINTV a vector must be referenced. As with intervals, a data definition that leads to processing a vector interval from left to right is also developed.

The above observations allow for the in-class development of the function template to process a vector displayed in Figure 1. The contract states that any function that processes a vector must take as input at least a vector of any type (X is a type variable). The body of the function is a local-expression that may be used to define one or more local functions and values. Students are told that problem analysis will reveal the type of expression that is needed in the body of the local-expression. If a single value is needed from the given vector, then the expression will be one that processes a vector interval. Otherwise, the expression will be one that uses different values obtained from processing the same vector. The local definition section contains two templates: one for each direction that a vector interval can be processed

```
; avg-vector: (vectorof number) → number
; Purpose: To compute the average of the given vector
; Assumption: The vector is not empty.
(define (avg-vector V)
  (local [; sum-elems: int int → natnum
          ; Purpose: For the given interval, sum the
                     vector elements
          (define (sum-elems low high)
            (cond [(empty-interval? low high) 0]
                  [else (+ (vector-ref V high)
                           (sum-elems low (sub1 high)))]))]
    (/ (sum-elems 0 (sub1 (vector-length V)))
       (vector-length V))))
```

Figure 2: A Function to Compute the Average of a Vector of Numbers.

in. At least one of the templates is to be used to process vector elements. Observe that in each of the local templates, vector elements are processed (using vector-ref) instead of interval elements.

To make the use of the function template to process a vector concrete, consider computing the average of a vector of numbers. Problem analysis reveals that the vector cannot be empty given that division by 0 is undefined. It also reveals that it does not matter in which direction the VINTV is processed as addition is a commutative operation. Now, the template for functions on a vector from Figure 1 is specialized. The contract indicates that the input is a vector of numbers, V, and that the function returns a number. The body of the local-expression must divide the sum of the vector elements by the length of the vector. This means that we must write a function to compute the sum of vector elements. Given our problem analysis, either of the templates to process a vector interval can be used. Without loss of generality, we can choose to process from right to left (i.e., the template for VINTV). This means that when the vector interval is empty the answer is 0 and that when it is not empty we add (vector-ref V high) to the result of recusively processing the rest of the VINT (i.e., [low..high-1]). The resulting function is displayed in Figure 2. Observe that by using the template based on structural recursion it is impossible to have indexing errors.

6 Extended Examples

This section presents three extended examples of how to design functions that process vectors. The first, the dot product of two vectors [1, 16], is an example of how to process multiple vector intervals simultaneously in step. The second, the merging of two sorted vectors [9], is an example of how to process multiple vector intervals that are not processed in step. This example also shows that the design of vector mutators can benefit from exploiting the structure of vector intervals. The third, quicksort [4, 8], sorts a vector in place. This is an example of how reasoning about vector intervals assists in the design of functions that uses generative recursion and mutation.

```
; dot-product: (vector number) (vectorof number) → number
; Purpose: To compute the dot product of the two given vectors
(define (dot-product V1 V2)
  (local [; sum-products: int int → number
          ; Purpose: For the given VINTV, compute the dot product of V1 and V2
          (define (sum-products low high)
            (cond [(empty-interval? low high) 0]
                  [else (+ (* (vector-ref V1 low) (vector-ref V2 low))
                           (sum-products (add1 low) high))]))]
    (sum-products 0 (sub1 (vector-length V1)))))
```

Figure 3: Function to Compute the Dot Product of Two Vectors.

6.1 The Dot Product of Two Vectors of Numbers

Given two vectors of numbers, V1 and V2, the dot product is defined as:

`V1 · V2 =` $\Sigma_{i=0}^{N}$ `V1[i]*V2[i], where N is the length of the vectors minus 1.`

In-classroom problem analysis reveals:

- Both vectors must have the same length.
- Both vectors must be entirely processed simultaneously in step.
- Vector elements can be processed either right to left or vice versa.

Given these insights, students conclude that the function can be designed around processing a single vector interval, say for V1, and then specialize the template for functions on vectors to develop their code. The input is two vectors of numbers and the output is a number. This is reflected in the contract in Figure 3. The body of the local-expression calls a function, sum-products, to process the single interval that spans all the elements of both vectors (i.e., from 0 to (sub1 (vector-length V1))). Observe that the interval only contains valid indices into the vector.

The function, sum-products, is designed using either template for vector interval processing given the third insight above. Figure 3 uses the template that processes the interval from left to right. The code is developed by steps, as before, by formulating answers for each variety of vector interval. Students have no trouble seeing that the answer is 0 when the interval is empty. When the interval is not empty, students are explained that they must do something with the two elements, in this case, at the low end of each vector interval. This is what it means to process both vectors simultaneously in step. This action, of course, is to multiply them. To formulate the final answer, this product must be added to the result of recursively processing the rest of the vector interval.

The reader can observe that no indexing errors can arise in this example. The key to success is for students to properly define the initial VINTV2 to be processed. In this case, this task is fairly straightforward given that both vectors must be processed in their entirety. Also observe that there is no guess work involved in how to process the rest of the elements in each vector. The solution to that concern is baked into the template for functions on a vector.

6.2 Merge Two Sorted Vectors

Consider the problem of merging two vectors that are sorted in non-decreasing order into a single vector that is sorted in non-decreasing order. In-classroom problem analysis yields the following insights:

```
; merge: (vector number) (vectorof number) → (vectorof number)
; Purpose: To merge the two given sorted vectors in non-decreasing order
; Assumption: The given vectors are sorted in non-decreasing order
(define (merge V1 V2)
  (local [; res: (vectorof number)
          ; Purpose: To store the merged elements so far
          (define res (build-vector (+ (vector-length V1)
                                       (vector-length V2))
                                    (lambda (i)  (void))))

          ; combine: int int int int int int → (vectorof number)
          ; Purpose: For the given VINTVs, merge V1 and V2 into res
          (define (combine lowv1 highv1 lowv2 highv2 lowres highres)
            ...)
(combine 0 (sub1 (vector-length V1))
         0 (sub1 (vector-length V2))
         0 (sub1 (vector-length res)))))
```

Figure 4: Basic Outline for a Function to Merge Two Sorted Vectors.

- A vector to hold all the elements of the given vectors must be allocated. Given that this vector must be mutated every time an element is added, it must be a state variable.
- Three different intervals must be processed simultaneously: one for each of the input vectors and one for the result vector. These intervals are not processed in step.
- Each vector must be processed in its entirety.

Figure 4 displays the basic outline for a function to merge two sorted vectors obtained from beginning to specialize the template for a function on a vector. The contract, the purpose statement, and the assumption indicate that two sorted vectors of numbers are expected as input and a sorted vector is expected as output. The body of the local-expression calls an auxiliary function, combine, to process the three vector intervals. Three intervals are needed as input, because they are not processed in step. Observe that all three initial vector intervals span all the valid indices, respectively, for each vector. Thus, by using structural recursion indexing errors cannot occur. Locally, the state variable, res, is defined to store the result. Its invariant states that it is a vector of numbers. This vector is initialized to contain only void values to indicate that nothing in the vector has been initialized.

The task left is to develop the body of combine. Given that three intervals are not processed in step, we need to determine the different conditions that may arise during processing to augment the cond-expression that appears in the template to process a vector interval. After some class discussion, the conclusion is reached that at each step an element of one of the input vectors is placed in the result vector. Furthermore, the input vectors and the result vector can be processed from left to right or vice versa. Without loss of generality, we proceed with processing the intervals from left to right. This is an implementation choice and it is equally correct to process the intervals from right to left. These new insights and our implementation choice, in conjunction with the previous insights, lead us to five cases that must be addressed:

```
(cond [(and (empty-VINTV2? lowv1 highv1)
            (empty-VINTV2? lowv2 highv2))
       res]
      [(empty-VINTV2? lowv1 highv1)
        (begin
          (vector-set! res lowres (vector-ref V2 lowv2))
          (combine lowv1 highv1
                   (add1 lowv2) highv2
                   (add1 lowres) highres))]
      [(empty-VINTV2? lowv2 highv2)
       (begin
         (vector-set! res lowres (vector-ref V1 lowv1))
         (combine (add1 lowv1) highv1
                  lowv2 highv2
                  (add1 lowres) highres))]
      [(< (vector-ref V1 lowv1) (vector-ref V2 lowv2))
       (begin
         (vector-set! res lowres (vector-ref V1 lowv1))
         (combine (add1 lowv1) highv1
                  lowv2 highv2
                  (add1 lowres) highres))]
       [else
         (begin
           (vector-set! res lowres (vector-ref V2 lowv2))
           (combine lowv1 highv1
                    (add1 lowv2) highv2
                    (add1 lowres) highres))]))]
```

Figure 5: The Conditional for the function combine from Figure 4.

.

1. The intervals for both input vectors are empty.
2. The interval for the first input vector is empty.
3. The interval for the second input vector is empty.
4. The low element of the first input vector is less than the low element of the second input vector.
5. The low element of the second input vector is less than or equal to the low element of the first input vector.

Observe that the non-recursive case is listed first and must be the first to be solved. For this case, there are no more elements to process in either vector interval for the input vectors and the result vector is returned. For the second case, the vector interval for the first vector is empty and the process continues by placing the remaining elements left in the second vector interval into the result vector. The recursive call is made with the rest of the vector intervals for both the second input vector and the result vector. The third case is the same as the second case, but it is the vector interval for the second vector that is empty. The recursive call is made with the rest of the vector intervals for both the first input vector and

```
; qs-in-place!: (vectorof number) → (void)
; Purpose: To sort the array in non-decreasing order.
; Effect: The elements of the array are rearranged in place.
(define (qs-in-place! V)
  (local [; partition!: int int natnum → number
          ; Purpose: For the given VINTV, partition and place the pivot in
          ;          its final position.
          ; Effect: Mutate V so that all elements before the pivot are
          ;         <= pivot and all elements after the pivot are > pivot.
          (define (partition! low high pp)
            ...)
          ; qs-aux!: int int → (void)
          ; Purpose: For the given VINTV, sort V in non-decreasing order.
          ; Effect: The elements in the given interval are rearranged
          ;         in place.
          (define (qs-aux! low high)
            (cond [(empty-interval? high low) (void)]
                  [else
                     (local [(define pp (partition! low high low))]
                       (begin
                         (qs-aux! low (sub1 pp))
                         (qs-aux! (add1 pp) high)))]))])
      (qs-aux! 0 (sub1 (vector-length V)))))
```

Figure 6: Basic Outline for Quicksort in Place.

the result vector. The fourth and fifth cases place the smallest low element of the input vectors into the result vector. The recursive call is always made with the rest of the interval for the result and the rest of the interval for the input vector that had an element placed in the result. The resulting conditional is displayed in Figure 5.

The important lesson to derive from this example is that when more than one interval is processed then every recursive call must be made with the rest of the intervals that process either the low or the high interval element. Observe that if this design principle based on structural recursion is followed and the initial intervals are correctly set, an index out of bounds error cannot arise.

6.3 Quicksort In Place

Quicksort is an algorithm based on generative recursion to sort an array in place summarized as follows:

- If the vector interval is empty, stop.
- If the vector interval is not empty
  - Pick a pivot. For our purposes, we will say the low element is the pivot.
  - Partition the vector by putting the elements less than or equal to the pivot at the beginning of the vector interval and the larger elements at the end. The pivot is placed at the largest index, pp, of the smaller elements.

```
(define (partition! low high pp)
  (local
    [(define (swap i j) ...)
     (define (small-index low high pivot) ...)
     (define (large-index low high pivot) ...)

     ; separate!: int int number → natnum
     ; Purpose: For the given VINTV, separate smaller and larger elements
     ; Effect: In V move elements <= pivot before elements > pivot.
     (define (separate! low high pp)
       (local [(define s-index (small-index low high (vector-ref V pp)))
               (define l-index (large-index low high (vector-ref V pp)))]
              (cond [(<= s-index l-index) s-index]
                    [else
                     (begin (swap s-index l-index)
                            (separate! l-index s-index pp))])))
    (begin
      (local [(define pp (separate! low high low))]
        (begin (swap low pp) pp)))))
```

Figure 7: Basic Outline for partition! from Figure 6.

- Recursively sort the vector intervals: [low..pp-1] and [pp+1..high].

The basic outline for the quicksort function is displayed in Figure 6. For the readers familiar with HtDP, you will recognize the basic outline. Here its presentation has been adapted to use the concept of a vector interval. The function to quicksort a vector in place takes as input a vector of numbers and returns void as this function is only called for its effect (i.e., the mutation of the given vector). The only task that needs to be performed is the sorting of the entire vector. Therefore, the body of the local-expression only needs to call an auxiliary function, qs-in-place!, to process the correct vector interval: [0..(sub1 (vector-length))]. This auxiliary function, as required, stops if the interval is empty. If the interval is not empty, it partitions the interval and then recursively sorts the required subintervals. Observe that the recursive calls are not made with part of the structure used to construct the given interval (i.e., this is generative recursion). Instead, to new instances of the problem are created to be solved (i.e., the divide and conquer). This auxiliary function is not designed using the template based on structural recursion. Instead, it is designed using the generative recursion template in HtDP [4]. Nonetheless, the concept of a vector interval is enlightening as it explicitly has the programmer think about the varieties of an interval as reflected in the body of qs-aux!.

The function qs-aux! needs to compute the position of the pivot in the sorted vector in order to create the two new vector intervals to process. Determining this is a different task and an auxiliary function, partition!, is needed to partition the vector. It takes as input a vector interval (i.e., low and high) and the index of the element chosen as the pivot (i.e., low as per our design choice). Figure 7 displays the basic outline for the mutator partition!. For the given vector interval, this function first reorganizes the vector elements putting the elements that are less than or equal to the pivot at the beginning of the interval and the other elements at the end of the interval to compute the position of the pivot in the sorted interval. It then swaps the pivot into the computed position and returns the pivot position. Observe that this function

```
; swap: natnum natnum → (void)
; Purpose: To swap V[i] and V[j]
; Effect: Modify V[i] to contain the value of V[j] and vice versa
(define (swap i j)
  (local [(define temp (vector-ref V i))]
    (begin (vector-set! V i (vector-ref V j))
           (vector-set! V j temp))))

; small-index: int int number → natnum
; Purpose: For the given VINTV, find largest index: V[k] <= pivot
(define (small-index low high pivot)
  (cond [(empty-interval? low high) low]
        [else
          (cond [(<= (vector-ref V high) pivot) high]
                [else (small-index low (sub1 high) pivot)])]))

; larger-index: int int number → natnum
; Purpose: For the given VINTV, find the smallest index: V[k] > pivot
;          if it exists else return high
(define (larger-index low high pivot)
  (cond [(empty-interval? low high) high]
        [else
          (cond [(> (vector-ref V low) pivot) low]
                [else (larger-index (add1 low) high pivot)])]))
```

Figure 8: Auxiliary functions for partition! from Figure 7.

does not process the vector interval and, therefore, is not designed using the template for functions on a a vector interval based on structural recursion.

The auxiliary function separate! does process an interval. This function, however, does not exploit the structure of a vector interval and, therefore, is also not designed using the template for functions on a vector interval. Nonetheless, reasoning about vector intervals, once again, proves fruitful in development using generative recursion. An important observation is the separate! is performing a task for the vector interval given as input to partition! and as such always operates within this vector interval. The function separate! must find, within its given vector interval, the largest index that contains a value less than or equal to the pivot and the smallest index the that contains a value greater than the pivot (if it exists). These tasks are left to auxiliary functions, small-index and large-index, respectively. Once these indices are found, we must determine if the new vector interval defined by these indices is empty or not. If it is empty, the function returns the index of the small element, because it is the index that separates the small and the large elements in the vector interval given as input to partition!. If it is not empty, the large and small elements are swapped and the process of separation continues with this smaller vector interval defined by the two computed indices.

The remaining auxiliary functions are displayed in Figure 8. Both small-index and large-index are developed using one of the templates for functions on a vector interval. Each exploits the structure of a vector interval to find the correct index. For small-index, the code is developed using the template that processes the vector interval from right to left. When the given VINTV is empty it returns the low element of the vector interval indicating that the only number less than or equal to the pivot is the pivot. Otherwise, it checks if the vector element indexed by high is less than or equal to the pivot. If so, it returns high as it must be the largest index of an element in the given VINTV that is less than or equal to the pivot. Otherwise, it recursively process the rest of the interval. For large-index, the code is developed using the template that processes the vector interval from left to right. When the given VINTV2 is empty it returns the high element of the vector interval indicating that there are no numbers greater than the pivot in the interval. Otherwise, it checks if the vector element indexed by low is greater than the pivot. If so, it returns low as it must be the smallest index of an element in the given VINTV2 that is greater than the pivot. Otherwise, as small-index, it recursively processes the rest of the interval.

The important lesson to take away from this example is that reasoning about the structure of vector intervals is helpful when function design requires generative recursion and/or mutation. As seen in the discussion above, some functions can exploit the structure of a vector interval to develop a solution while others are developed by reasoning about the different varieties of vector intervals.

7 Concluding Remarks

This article presents a design-oriented methodology to help beginners develop vector processing functions. It is based on the concept of a vector interval that has a recursive structure. This recursive structure is exploited to develop a function template that is specialized by students to solve problems. An important issue that is addressed by this methodology is the proper indexing of vectors. The concept of a vector interval is helpful in avoiding (and resolving) out-of-bounds indexing errors. Several examples are presented to illustrate the use of the design methodology in practice.

Future work includes formulating multidimensional array processing examples that further demonstrate the usefulness of reasoning about vector intervals to design functions. The work is also being extended to demonstrate how to use vector intervals to reason about algorithms that do not process all the elements of a vector within a given interval (e.g., functions used by heap sort). Finally, the work is being extended to a course that focuses on object-oriented design.

8 Acknowledgements

Students traditionally believe that the learning process flows from the professor to the students. In my case, nothing can be further from the truth. The work presented in this article is inspired by the difficulties faced and by the questions addressed to me by my beginning students. The author thanks them for providing me with valuable lessons regarding how to teach an introduction to vector programming. In particular, I thank Josephine Des Rosiers for her many heated debates with me about designing programs.

References

[1] Sara Baase (1988): *Computer Algorithms*, second edition. Addison-Wesley Publishing, Reading, MA, USA.

[2] Richard L. Burden & J. Douglas Faires (1985): *Numerical Analysis*, third edition. Prindle, Weber, & Schmidt.

[3] Code.org (2017): *Lesson 13: Introduction to Arrays*.

[4] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt & Shriram Krishnamurthi (2001): *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, Cambridge, MA, USA.

[5] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt & Shriram Krishnamurthi (2015): *How to Design Programs*. Http://www.ccs.neu.edu/home/matthias/HtDP2e/.

[6] William Ford & William Topp (1996): *Data Structures with C++*, first edition. Prentice Hall.

[7] Michael T. Goodrich & Roberto Tamassia (2001): *Data Structures and Algorithms in Java*, second edition. John Wiley & Sons.

[8] C. A. R. Hoare (1962): *Quicksort*. *The Computer Journal* 5(1), pp. 10–16.

[9] Donald E. Knuth (1998): *The Art of Computer Programming, Volume 3: Sorting and Searching*, second edition. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

[10] Marco T. Morazán (2011): *Functional Video Games in the CS1 Classroom*. In Rex Page, Zoltán Horváth & Viktória Zsók, editors: *Trends in Functional Programming: 11th International Symposium, TFP 2010, Norman, OK, USA, May 17-19, 2010. Revised Selected Papers*, Lecture Notes in Computer Science, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 166–183, doi:`10.1007/978-3-642-22941-1_11`. Available at `http://dx.doi.org/10.1007/978-3-642-22941-1_11`.

[11] Marco T. Morazán (2012): *Functional Video Games in CS1 II*. In Ricardo Peña & Rex Page, editors: *Trends in Functional Programming: 12th International Symposium, TFP 2011, Madrid, Spain, May 16-18, 2011, Revised Selected Papers*, *Lecture Notes in Computer Science* 7193, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 146–162, doi:`10.1007/978-3-642-32037-8_10`. Available at `http://dx.doi.org/10.1007/978-3-642-32037-8_10`.

[12] Marco T. Morazán (2014): *Functional Video Games in CS1 III*. In Jay McCarthy, editor: *Trends in Functional Programming: 14th International Symposium, TFP 2013, Provo, UT, USA, May 14-16, 2013, Revised Selected Papers*, *Lecture Notes in Computer Science* 8322, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 149–167, doi:`10.1007/978-3-642-45340-3_10`. Available at `http://dx.doi.org/10.1007/978-3-642-45340-3_10`.

[13] Marco T. Morazán (2015): *Generative and Accumulative Recursion Made Fun for Beginners*. *Comput. Lang. Syst. Struct.* 44(PB), pp. 181–197, doi:`10.1016/j.cl.2015.08.001`. Available at `http://dx.doi.org/10.1016/j.cl.2015.08.001`.

[14] Robert Sedgewick & Kevin Wayne (2007): *Introduction to Programming in Java: An Interdisciplinary Approach*, 1st edition. Addison-Wesley Publishing Company, USA.

[15] Abraham Silberschatz, Peter Baer Galvin & Greg Gagne (2010): *Operating System Concepts wit Java*, eighth edition. John Wiley & Sons.

[16] Stewart Venit & Wayne Bishop (1985): *Elementary Linear Algebra*, second edition. Prindle, Weber, and Schmidt.