Verified Counting Sort in Dafny

Josh Cohen

December 7, 2020

1 Introduction

Counting Sort is an integer sorting algorithm which is simple to implement, efficient, and stable. However, its proof of correctness is extremely tricky and unlike that of other sorting algorithms. In this project, I implemented and verified counting sort in the program verifier Dafny. Dafny allows the user to prove the functional correctness of their programs by annotating the code with with preconditions, postconditions, and assertions, each of which are then checked using an SMT solver. In this report, I describe the verification process and my experiences using Dafny. The code can be found in the joscoh/verified-counting-sort GitHub repo.

1.1 Counting Sort

Counting sort takes as input an array a of length n with values of type G, a function $key : G \to int$, and a positive integer k. The algorithm sorts the array by key as long as all keys are bounded between 0 and k. Of course, it is easy to calculate k if it is not provided (it is the largest key value in the array), but in this work we assume that k is given as input.
Counting sort can then be expressed with the following psuedocode [1]:

```
b = array of k zeros
for x in a:
    b[key(x)] += 1

b[0] -= 1
for i in 1 ... k - 1:
  b[i] += b[i-1]

c = array of the same length as input
for i in a.Length - 1 ... 0:
  c[b[key(a[i])]] = a[i]
  b[key(a[i])] -= 1

return c
```

The algorithm works as follows: we treat each key as an index into the array b. In the first loop, we populate the array b such that $b[i]$ contains the number of elements with key i. In the second loop, we perform a prefix sum calculation, which results in $b[i]$ containing the number of elements in a with key $\leq i$ (we subtract an additional 1 to deal with 0-indexing).
To understand what the third loop is doing, suppose that the array a had no duplicate keys. Then, element $x := a[i]$ should occur at index $b[key(x)]$ in the sorted array (since elements occur to the left of $a[i]$ iff they have smaller keys). For the general case, there may be multiple elements with the same key. We decrement $b[key(a[i])]$ each time we see such an element to ensure that we will never put 2 elements in the same spot. Since we iterate backwards, this ensures stability of the resulting array.
Counting sort runs in time $O(n + k)$, which is easy to see: the first loop does constant work at each of its n iterations, the second does constant work at each of its k iterations, and the third again does constant work at each of its n iterations.
Our Dafny implementation (described in Section 3) is very similar, except that we separate each loop into its own function for ease of verification. The runtime is asymptotically equivalent, though we incur an extra $O(k)$ cost by creating a new array for the second loop and copying the array b before the third loop.

1.2 Dafny

Dafny[4] is a program verification tool that allows the user to write both functional and imperative code which is annotated with preconditions, postconditions, assertions, and invariants. Dafny then verifies that all of the annotations are valid and that the underlying program is safe. In addition, Dafny contains many tools and features to assist with verification tasks. Here, we give brief explanations of several that were particularly useful in this project.

- Preconditions and postconditions - These take the form of boolean expressions inside `requires` and `ensures` statements, respectively. Dafny will check that all preconditions hold when a method is called, assume the preconditions when proving any statements inside the method body, and check that the postconditions hold when the method returns.
- Assertions - Written as boolean expressions inside `assert` statements, these are propositions that Dafny will check at a particular location in the code. These are sometimes needed to help Dafny infer certain information to prove other goals and are often helpful for the programmer to help determine what Dafny can and cannot determine from a given context.
- Loop Invariants - Written as boolean expressions inside `invariant` statements, these propositions are required to hold on loop initialization and to be preserved at every iteration of the loop. Loop invariants are needed to prove anything nontrivial about loops in Dafny. Finding, using, and verifying suitable loop invariants comprised much of the difficulty of this project.
- Termination - Loop termination can be proved by supplying a `decreases` parameter. This was not too relevant to this project, since counting sort uses loops with simple bounds and clear termination.
- Functions - These are pure functional expressions with no side effects. They can be recursive but cannot modify state. Functions are particularly useful because they are allowed in propositions, unlike methods. Almost every specification in this project uses functions.
- Lemmas - Also known as ghost methods, lemmas are methods that exist soley to prove claims and cannot modify state. A lemma consists of a precondition, a postcondition, and a body (possibly empty) that allows Dafny to prove that the precondition implies the postcondition. Since lemmas are only used for verification, they are erased at runtime. We make very heavy use of lemmas in order to help Dafny prove nontrivial goals.
- Ghost variables - These variables are erased at runtime and exist only for verification purposes. They are especially helpful in keeping track of variable values before and after mutation.
- Induction - Many lemmas are proved using induction. In most cases, Dafny can infer which variable to induct on, but in other cases, this information can be manually supplied using `{: induction a}`, where a is the variable to induct on.
- Quantifiers - Most propositions that we want to verify involve one or more quantifiers. Dafny allows the use of quantifiers in propositions (such as in `assert` statements), but also provides ways of working more concretely with quantified statements. In particular, we can use the `forall` keyword to perform universal introduction (ie, prove the statement `forall x :: p(x)` by proving `p(x)` for an arbitrary x). We can also perform existential elimination with the `exists` keyword, and thus can extract a value x that satisfies p from a proof of `exists x :: p(x)`.
- Collections - Dafny have built in facilities for reasoning about lists, sets, multisets, and maps along with many standard operations (list concatenation, set union, etc). These collections are immutable and can be used in specifications. We made heavy use of sequences (immutable lists) and multisets in this project.

2 Verifying Sorting Algorithms and Related Work

Sorting algorithms are a very common use case for verification tools; for instance, mergesort, insertion sort, quicksort, and others have all been verified in Dafny, Coq, Liquid Haskell, Isabelle, and many other tools.[1] These algorithms serve as particularly good case studies or even introductory examples/exercises since their specifications are simple, their proofs of correctness are easy (using a single inductive argument or a loop invariant), and the algorithms can be written functionally (making them a good fit for tools such as Coq and Liquid Haskell).

In contrast, counting sort has a more complicated specification (due to stability), its proof of correctness is very tricky (to the point that many textbooks do not even prove correctness; they give an inutitive argument like that of Section 1.1[2]), and it relies heavily on arrays, mutability, and loops. Accordingly, counting sort is a poor fit as an introductory example or for a functional tool. However, this makes the algorithm ideal for Dafny, which can comfortably handle arrays without needing to worry about manual memory management and which has an expressive specification language that allows us to specify the correctness of the algorithm and its intermediate steps.

There has been only one previous verification of counting sort [2], completed in the KeY theorem prover and specified using the Java Modeling Language. In that work, counting sort was used to verify radix sort, and it is specialized to 2-D arrays of integers (ie, an input where each element is an array of integers). Our project comprises the first verification of counting sort in Dafny and the first verification of a generic counting sort that is parametric in the choice of type `G` and function `key : int -> G`.

Another related project is a verification of natural mergesort in Dafny [5], which is also a stable sort. Our specification of stability is inspired by that work, as is the particular method of using a generic type `G` and key function through an abstract module.

3 Project Tasks

Here we give an overview of the specification and verification, along with key challenges.

During the course of the project, I initially determined a set of invariants sufficient to verify the third loop (which was presented in class), then later realized that only a few of these invariants are needed, at the cost of some fairly difficult proofs that this more limited set of invariants implies the postconditions. The differences are explained a bit further in Section 3.5. In the master branch of the repo and in the majority of this section (in particular, Section 3.4) we take the second approach (with fewer invariants). The Appendix contains a pen-and-paper proof of the correctness of counting sort based on the first approach, which is easier to understand but requires more code in Dafny. The verified code using the first approach can be found in the more-invariants branch of the repo.

3.1 Specification

Counting sort is a sorting algorithm, so it should obey the same specification as should any such algorithm: the output must be a permutation of the input and the output must be sorted (by key). We can express each of these in Dafny as the following predicates (we work with sequences rather than arrays; we can easily turn an array into the equivalent sequence by using the `a[..]` syntax):

```
predicate permutation<T>(a: seq<T>, b : seq<T>) {
  multiset(a) == multiset(b)
}

predicate sorted(a: seq<G>) {
  forall i, j : int :: 0 <= i < |a| ==> 0 <= j < |a| && i <= j ==> key(a[i]) <= key(a[j])
}
```

Stability is harder to specify. Two arrays are stable with respect to each other if any two elements with equal keys appear in the same relative order in both arrays. We follow [5] and specify stability without directly

[1] See, for instance, mergesort in Coq, Dafny, and Isabelle, quicksort in Coq, Dafny, and Isabelle, insertion sort in Coq, Dafny, and Isabelle, and all three in Liquid Haskell.

[2] For instance, [1] gives only a brief, nonrigorous argument.

mentioning pairs of elements. First, we define a `filter` function, which acts identically to `filter` in Haskell or `List.filter` in Ocaml:

```
function filter<T>(a: seq<T>, f:T -> bool) : seq<T>
{
  if (|a| > 0) then
    if (f(a[0])) then [a[0]] + filter(a[1..], f)
    else filter(a[1..], f)
  else a
}
```

Then, we can define stability as follows:

```
predicate stable(a : seq<G>, b : seq<G>) {
  forall x : int :: filter(a, y => key(y) == x) == filter(b, y => key(y) == x)
}
```

In other words, for any given key, if we filter both lists by that key, the two filtered lists must be equal to one another. Since filter preserves the order of elements that satisfy the given predicate, this is an equivalent definition of stability.

3.2 Function Definitions

We want to be able to formalize the proof idea presented in Section 1.1 in Dafny. To do this, we need a way to talk about the number of elements with a given key, the number of elements with smaller keys, and the position of an element in the sorted array. To that end, we define the following functions in Dafny:

```
//Calculate the number of elements with key < x
function numLt(x: int, a : seq<G>) : int {
  |filter(a, y => key(y) < x)|
}
//Calculate the number of elements with key = x
function numEq(x: int, a : seq<G>) : int {
  |filter(a, y => key(y) == x)|
}
//Calculate the number of elements with key <= x
function numLeq(x: int, a : seq<G>) : int {
  numLt(x, a) + numEq(x, a)
}
```

While these definitions are very helpful for some applications, in other cases we will want different forms or equivalences, each of which we need to prove in Dafny. For instance, in some places we will want a `filter` definition for `numLeq`, which we can prove easily:

```
lemma numLeq_direct(x:int, a : seq<G>)
ensures(numLeq(x, a) == |filter(a, y => key(y) <= x)|) {
  //Dafny proves this automatically via induction on a
}
```

We will also want an inductive definition of `numLeq`, expressed as the following:

```
lemma numLeq_ind(x: int, a : seq<G>)
ensures(numLeq(x,a) == numLeq(x-1, a) + numEq(x, a)) {
  //body - calls several other lemmas
}
```

How can we specify the position of an element in the sorted array (without, of course, sorting the array first)? Note that a particular element can be in a range of locations in a sorted array, since the elements with the same key can appear in any order among themselves. However, if the array is stable as well, each element has a unique location in the sorted array. Take some element `a[i]`. Clearly, all elements with smaller keys must appear to the left of `a[i]` in the sorted array. However, since the array is stable, all elements in a with the same key that appear before position i must also occur to the left of `a[i]` in the resulting array. Thus, we have the following `position` function:

```
function position(x : int, i : int, a : seq<G>) : int {
    numLt(x, a) + numEq(x, a[..i+1]) - 1
}
```

We make the `position` function more general: it is defined for any key and for any index in the array. This allows us to better specify positions in invariants. However, `position(key(a[i]), i, a)` represents the correct position of element `a[i]` in the sorted and stable array.
Note that although we interpret the `position` function as referring to the sorted array, we must actually prove that this is the case; namely, if every element in an array is in its correct `position`, the array is sorted. We do this in Section 3.6.2.

3.3 Veryifying the First Two Loops

With these functions in mind, we can specify and verify the first two loops without much trouble. The first loop, annotated with pre and postconditions and loop invariants but without some intermediate assertions and ghost variables, is:

```
method countOccurrences(a: array<G>, k: int) returns (b: array<int>)
requires 0 < a.Length
requires 0 < k
requires (forall i: int :: 0 <= i < a.Length ==> 0 <= key(a[i]) < k)
ensures (b.Length == k)
ensures(forall i : int :: 0 <= i < k ==> b[i] == numEq(i, a[..]))
{
  b := new int[k](i => 0);
  var i := 0;
  while(i < a.Length)
  invariant (0 <= i <= a.Length)
  invariant(forall j : int :: 0 <= j < k ==> b[j] == numEq(j, a[..i])) {
    b[key(a[i])] := b[key(a[i])] + 1;
    i := i + 1;
  }
  assert(a[..i] == a[..]);
}
```

This is a straightforward Dafny program, with a simple loop invariant which is mantained quite easily (it follows essentially from the definition of `numEq`). In each step, we have accounted for all of the elements in `a[..i]`, or the array a up to and including position $i-1$. By the end, each element `b[i]` contains the number of elements in a with $key = i$, as desired.
The second loop is quite similar in difficulty. Note that we require the postcondition of the first loop as a precondition.

```
method prefixSum(a:array<G>, b : array<int>) returns (c: array<int>)
requires(0 < b.Length)
requires(forall i : int :: 0 <= i < b.Length ==> b[i] == numEq(i, a[..]))
requires (forall i: int :: 0 <= i < a.Length ==> 0 <= key(a[i]))
ensures(c.Length == b.Length)
ensures(forall i : int :: 0 <= i < b.Length ==> (c[i] == numLeq(i, a[..]) - 1));
{
  var i := 1;
  c := new int[b.Length];
  c[0] := b[0] - 1; //subtract 1 because of 0-indexing
  while(i < c.Length)
  invariant (1 <= i <= c.Length)
  invariant(forall j: int :: (0 <= j < i ==> c[j] == numLeq(j, a[..]) - 1))
  {
    numLeq_ind(i, a[..]); //need to relate numLeq(i, a[..]) with numLeq(i-1, a[..])
    c[i] := b[i] + c[i-1];
    i := i + 1;
  }
}
```

3.4 The Third Loop: Challenges and Invariants

The third loop, recall, is the following (this is mostly valid Dafny code but with no assertions or invariants - this will not compile since Dafny cannot prove that the array accesses are safe):

```
method constructSortedArray(a: array<G>, b: array<int>, default : G) returns (c : array<G>)
    {
  var b1 := ...; //copy array b into b1
  c := new G[a.Length](i => default);
  i := a.Length - 1;

  while(i >= 0) {
    c[b1[key(a[i])]] := a[i];
    b1[key(a[i])] := b1[key(a[i])] - 1;
    i := i - 1;
  }
}
```

The challenging and interesting parts of the project all arise from the third loop. We want to determine loop invariants for the while loop that allow us to prove our desired postconditions (that the array is a permutation of the input, sorted, and stable). In the following subsections, we highlight key challenges and reasons for choosing the final loop invariants.

3.4.1 Bounds

Dafny must be able to determine that all array accesses are safe. Thus, we must know that i is within bounds of a, `key(a[i])` is within bounds for $b1$, and `b1[key(a[i])]` is within bounds for c. We must be careful; however, because Dafny must also know that any array accesses within a loop invariant are safe. So we don't just need to know that all checks in `c[b[key(a[i])]]` are safe; we need more general conditions. This leads to the following invariants:

```
  invariant(-1 <= i < a.Length)
  invariant(forall j :: 0 <= j < a.Length ==> 0 <= key(a[j]) < b1.Length);
```

The first is simply the usual invariant for the loop counter. The second immediately follows from the requirement that the keys of all elements in the array are bounded between 0 and k. Note that we do not have a similar invariant for $b1$, since $b1$ changes over the course of the loop, and in fact some entries will become negative. Instead, we will need a stronger invariant that specifies exactly what is contained in $b1$. This is presented in Section 3.4.5.

3.4.2 Specifying the Completed Portion of c

One of the central challenges of this loop is that we populate the array c in an unpredictable order. For many loops, such as those in Section 3.3, we can easily specify the part of the array that has been processed by referring to `a[..i]`, since we iterate through `a[0], a[1],...,a[n-1]`. In this loop, we cannot do this, since at each iteration, there is simply some (likely non-contiguous) portion of c that has been filled in. How can we refer to this part? To help resolve this issue, we first make an additional requirement on the input type `G`: there exists an element `default` of type `G` such that `default` does not appear in the input array.

As a brief aside, we note that this condition is not very restrictive. Even if every element of type `G` appears in the array, we can simply add a new element to `G` or instead work with the type `option G`, and set `default = None`. We are then guaranteed that, since all elements in the input are non-default values, the result does not contain `None` and thus we can unwrap the options and get back an array of elements of type `G`. We do not actually implement this in Dafny, but it demonstrates that our assumption does not place any additional restrictions on the types of values we can sort.

Now that we have a `default` element, we can specify the portion of c completed so far as:

```
filter(c[..], y => y != default)
```

The portion of a dealt with so far, by constrast, is easy: `a[i+1..]`, since we iterate through a in reverse order.

3.4.3 The Length Invariant

With the above in mind, we can state the following invariant, which says that, in each iteration of the loop, we actually add a new element to the output:

```
invariant(|filter(c[..], y => y != default)| == a.Length - (i + 1));
```

We actually have some flexibility with this invariant: it follows (nontrivially, in Dafny) that permutations have the same length, so we could instead choose to use an invariant about permtuations (the preservation of which is essentially the same level of difficulty). Instead, we prove the length invariant directly, and will later show that the permutation condition follows from stronger results.
Proving that this invariant is preserved presents an immediate problem: how do we know that, in the ith iteration, we are actually populating a new index in c? In other words, how do we know that `b1[key(a[i])]` has not already been filled in during a previous iteration of the loop? Proving this central fact turned out to be one of the most difficult parts of the project.

3.4.4 Proving that the Loop Does Not Repeat

We want to show that, at each iteration of the loop, we examine a new index of c. It suffices, for our purposes, to show that `c[b1[key(a[i])]] == default` when we enter the ith iteration (since we know that we will fill this index with a non-default element).

The first attempt to prove this is simply by including it as an invariant. It is easy to show that this invariant holds on entry, since all elements of c are `default`. However, this approach runs into two issues: $b1$ is changing throughout the execution of the algorithm, and proving that the invariant is preserved is not possible, since this invariant is not nearly strong enough. In other words, we know absolutely nothing about `c[b1[key(a[i-1])]]`, which is the preservation we would need to show.

The central issue, then, is that we need something stronger; we need to be able to tell precisely which elements in c have been filled in already. But there is no easy way of specifying this either. Instead, we will take a different approach: try to specify the contents of c precisely (which we will need to do anyways for our postconditions), and then prove from this (much stronger) invariant that the index we are interested in must be empty.
How, then, should we specify the elements in c? We cannot directly say, for instance, that `c[b1[key(a[i])]] == a[i]` since $b1$ is changing, so this will not always be true. Intead, we recall the intuition that underlies the third loop: each element is placed in its correct position in the sorted array. We can state this as the following: for every element in a that we have seen so far, `c[position(key(a[i]), i, a)] = a[i]`. However, this also does not quite work, since it does not rule out the possibliity that we could have other non-default elements in c (it may be possible to combine this with the length invariant to resolve this issue, but we take a slightly different approach). Instead, we use the following similar statement: for every index j of c that has been filled in, there is some element in a (say `a[k]`) that has already been processed by the loop such that `j=position(key(a[k]), k, a)`. We formalize this in Dafny as the following:

```
invariant(forall j :: 0 <= j < c.Length ==> c[j] != default ==>
    exists k :: ((i < k < a.Length) && c[j] == a[k] &&
          j == position(key(a[k]), k, a[..])))
```

As it turns out, as long as we have an appropriate invariant for the contents of $b1$ (presented in the next section), preservation of this invariant is easy to show. This turns out to be the crucial invariant for showing that the loop considers a new element each time, as well as in proving the permutation, sorting, and stability postconditions.

One interesting thing to note is that we need an invariant that precisely describes the elements of c even to show the relatively simple fact that the loop considers a new element in each iteration. This offers a reason as to why the proof of correctness of counting sort is so tricky: it is not built from simpler pieces, but requires knowing a lot of detailed information about the execution of loop before being able to prove anything at all. In fact, the properties of sorting, permutations, and stability will follow more or less directly (though nontrivially) from this invariant, so by the time we have proved that the loop does not repeat, we have already proved everything needed to establish the postconditions.

3.4.5 b1 Array Contents

The last task needed is to specify what is contained in the $b1$ array. We would like to say that, by the time we are on iteration i, `b1[key(a[i])] = position(key(a[i]), i, a)` (we would like to put $a[i]$ in its correct position in the array c). But clearly it cannot always be the case that this condition holds: after all, we decrement `b1[key(a[i])]` immedately after.
To determine a stronger invariant, we again will specify all elements of $b1$. At the beginning of the loop, we know from the second loop's postcondition that `b[j] = numLeq(j, a)- 1`. We can equivalently think about this as the position of the last element in a with key j. Then, in each iteration of the loop, we decrement `b[j]` by 1 exactly when the element at `a[i]` has key j. Thus, we can see that $b1$ contains positions - but those positions extend to index i in the array. More precisely, we have the following invariant:

```
invariant(forall j :: 0 <= j < b.Length ==> b1[j] == position(j, i, a[..]))
```

Thus, by the time we get to iteration i and examine `b1[key(a[i])]`, we get the correct position. But before that point, $b1$ is an overestimate of the position of a given element, since there may be more elements with the same key still to come.

3.4.6 Invariant Preservation

With all of these invariants, we can now put argue very informally why the invariants are preserved.
First, we claim[3] that `position` is injective in the following sense: if `position(key(a[i]), i, a)= position(key(a[j]), j, a)`, then `i=j`. This is intuitively obvious but requires lots of case analysis in Dafny.[4]
Then, we want to claim that `c[b1[key(a[i])]] == default`. If not, by the c-structure invariant, `b1[key(a[i])] == position(key(a[k]), k, a)` for some $k > i$. But by the $b1$-invariant, `b1[key(a[i])] == position(key(a[i]), i, a)`, and by injectivity, $i = k$. This is a contradiction.
With this result, the preservation of the length invariant is easy.[5] Proving that the c-invariant is preserved is almost immediate from the $b1$-invariant (we clearly add an element at its correct position)[6]. Proving that the $b1$-invariant is preserved requires a bit of reasoning about how `position(j, i, a)` relates to `position(j, i-1, a)`, which again requires some case analysis and simple results about `position` but is not too difficult.[7]

3.5 2 Sets of Invariants

In total, then, there are 5 invariants, one of which trivially comes from the loop counter. Notice that we have said very little about the permutation condition, sorting, or stability, and in fact we prove that this simple set of invariants (mainly the c-structure invariant) implies these properties in Section 3.6.
There is also an alternate approach that was intitially used and was mentioned in the Section 3 introduction. Here, the permutation and stability conditions are themselves additional invariants, in a manner very similar to the length invariant. This makes the invariant preservation proofs a bit more difficult (particularly for stability), but it makes the proofs of the postconditions very easy. In contrast, the approach described in the previous sections makes invariant preservation easy but makes the proofs of the postconditions very difficult. On the whole, the fewer-invariant approach results in less code and fewer proof obligations, but is arguably more difficult, less intuitive, and relies on several long and complicated Dafny proofs. The alternate version, with more invariants, can be found in the more-invariants branch, and the Appendix contains a proof of correctness for counting sort based on this approach.

3.6 From Invariants to Postconditions

Now, at the end of the third loop, we want to imply that the loop invariants imply the 3 postconditions (that the output is a permutation of the input, is sorted, and is stable). We can prove that the c-structure invariant (along with the length result that tells us that every element of c is filled in), implies stability,

[3] See the Appendix (Lemma 4) for a proof
[4] The Dafny lemma is position_inj.
[5] The Dafny lemma is filter_length_invariant.
[6] The Dafny lemma is c_structure_invariant.
[7] The Dafny lemma is b_position_invariant.

though the proof is difficult (it takes about 120 lines of code in Dafny) and is omitted here.[8] There are two other tasks:

3.6.1 Stability Implies Permutations

We outline the proof that any two stable lists are permutations (a result which is independent of counting sort).[9] Let the lists be called a and b. The case when either is empty is trivial. We proceed by induction on a. We know that, since the two lists are stable, there is some index k in b such that `b[k] = a[0]` and `b[k]` is the first element in b with the same key as `a[0]`.
We claim that if we remove $a[0]$ from a and remove $b[k]$ from b, then the resulting arrays are still stable. We must show that, for any x, if we filter the two lists by keys who equal x, the filtered lists are the same. If $x = key(a[0])$, this is true, since we have removed the first element of both filtered lists (which were already assumed to be equal). If $x \neq key(a[0])$, then the two removed elements are not part of either filtered list already, so the claim follows from the assumption.
Thus, we can apply the induction hypothesis to find that a without $a[0]$ and b without $b[k]$ are permutations. Since we add the same element to both lists, the result follows.

3.6.2 c-Structure Invariant Implies Sortedness

We prove the implication in two parts. First, we prove that an alternate sorting condition implies sortedness. In particular, if each element is in any one of the possible positions for its key, then the array is sorted. That is presented in the following predicate and lemma:

```
predicate sorted_alt(a:seq<G>) {
  forall i : int :: 0 <= i < |a| ==>
      numLt(key(a[i]), a[..]) <= i <= numLeq(key(a[i]), a[..]) - 1
}

lemma sorted_alt_implies_sorted(a: seq<G>)
requires(sorted_alt(a))
ensures(sorted(a)) {
  //body
}
```

Then, we know that no elements in c are default at the end of the loop (by the length invariant). So by the c-structure invariant, we know that the following holds:

```
forall j :: 0 <= j < |c| ==> exists k :: ((-1 < k < |a|) && c[j] == a[k] &&
    j == position(key(a[k]), k, a[..]))
```

Then, we can prove, by the definition of `position`, that this condition implies `sorted_alt`:

```
lemma all_positions_implies_sorted(a : seq<G>, c : seq<G>)
requires(permutation(a, c))
requires(|a| == |c|)
requires(forall x :: x in a ==> key(x) >= 0)
requires(forall j :: 0 <= j < |c| ==> exists k :: ((-1 < k < |a|) && c[j] == a[k] && j ==
    position(key(a[k]), k, a[..])))
ensures(sorted_alt(c)) {
  //body
}
```

The proofs of these implications are fairly simple and rely on bounds for the `position` function. Note that `all_positions_implies_sorted` is actually doing a bit more: our `position` function refers only to the correct position in the sorted array made of elements in a; the `permutation` precondition, along with several (surprisingly complicated) lemmas about how permutations preserve filtered predicates, means that we can extend all of these results to c as well. In other words, c satisfies:

```
forall j :: 0 <= j < |c| ==> exists k :: ((-1 < k < |c|) && c[j] == a[k] &&
    j == position(key(a[k]), k, c[..]))
```

which then lets us apply `sorted_alt_implies_sorted` and complete the proof.

[8] The Dafny lemma is c_structure_implies_stable.
[9] The Dafny lemma is stable_implies_permutation.

4 Results

With these invariants and associated lemmas, we were able to prove that this Dafny implementation of counting sort satisfies the specification in Section 3.1.
We note the following quantiative results:

- In total, the effort takes about 1100 lines of code, including some comments. Of this, appromximately 35 lines comprise the actual counting sort algorithm that can be run; the rest is code for verification purposes that is erased at runtime.
- Dafny produces about 1324 proof obligations and verifies the obligations in about 30 seconds. However, when we add a refinement of the given module (ie, instantiating the type and key function) and a short test case, the number of obligations becomes 1878.
- In constrast, the version with more invariants is about 1800 lines of code and results in 1798 proof obligations (without any refinement or test cases), taking about the same amount of time to verify. These metrics support the observation that this approach requires more code and more proof obligations, but the proofs themselves are simpler (since the verification time does not increase significantly).
- There are about 60 lemmas, the breakdown of which are as follows:
  - 7 generic lemmas about `filter`
  - 22 lemmas about `numEq`, `numLt`, `numLeq`, and `position`
  - 5 generic results about multisets and permutations
  - 11 lemmas about how permutations interact with `numEq` and similar (many of these could have been made more generic)
  - 3 other lemmas about sequences and arrays more generally
  - 3 lemmas about sortedness conditions
  - 6 lemmas about loop invariant preservation and similar facts
  - 3 lemmas that the invariants imply the postconditions

  In the more-invariants version, there are 5-10 more lemmas.

5 Discussion

5.1 Dafny as a Verification Tool

Dafny seemed like an ideal tool in which to verify counting sort due to its great support for arrays, mutation, iteration, and loop invariants. In addition, the wide variety of verification and specification features (described in Section 1.2) makes Dafny very expressive and allows verification of rich specifications. While overall Dafny was a good choice for this project, there were also several issues and drawbacks that caused signficant frustration.

- The biggest issue concerned the time needed to verify code. Given a goal it cannot solve, Dafny often times out, and it is very difficult to determine if this is because the goal is incorrect, Dafny does not have enough information, or if more time is needed. In addition, Dafny does not consistently tell which goal timed out (this can change each time the verifier is run), making debugging difficult. Additionally, Dafny will report only the suceess or possible failiure of a goal, necessitating a highly incremental approach, in which I had to add only tiny amounts of code before verifying the result. Since Dafny has to re-verify the entire code each time, this made development, especially with timeout issues.

- Similarly, I found that seemingly small changes in how the code is organized can result in massive changes in the time needed to verify the code. Moving the invariant preservation results into separate lemmas for all but the most trivial invariants massively reduced the time needed, even though it seemed as though the proofs should be identical (perhaps Dafny had too much information in its context and could not figure out which pieces were relevant). Additionally, to make the code generic, I initially parameterized the functions with the type `G` and the function `key`, but this consistently timed out, even with massive time limits. Switching to an abstract module parameterized by `G` and `key` cut down the time needed dramatically. While this wasn't too hard to fix, I did not find any documentation as to why this would be the case.

- More generally, generics, higher order functions, and scoping in anonymous functions do not always work as intended. Dafny had particular trouble with function inputs that were then used in the body of an anonymous function. For instance, Dafny could not prove the following assertion without a separate inductive lemma:

  ```
  assert(numLt(0, a) == |filter(a, y => key(y) < 0)|)
  ```

  It appears that this should be provable by simple substitution, but this was not the case.

- Automation in Dafny is inconsistently helpful. There were some goals, even seemingly nontrivial ones, that Dafny verified with no manual effort needed, which was very welcome. For example, the following lemma verifies without a body, despite its nested quantifiers and multiple parts:

  ```
  lemma filter_fst_idx<T>(a: seq<T>, f: T -> bool)
  requires(0 < |filter(a, f)|)
  ensures(exists i : int :: 0 <= i < |a| && f(a[i]) &&
    forall j : int :: 0 <= j < i ==> !f(a[j])) {
  }
  ```

  However, for more complicated goals, since it is often difficult to tell exactly what Dafny can and cannot infer, I had to proceed with lots of assertions, and more or less sketch out the full proof, filling in steps as Dafny encountered goals it could not prove. While many of the assertions were not needed at the end, and were thus deleted, in order to prove the actual goal, I found that automation was not as helpful. However, the upside is that Dafny is very expressive, and so allowed me to carefully step through proofs and find the missing pieces needed.

Overall, I found using Dafny to be a largely enjoyable experience, though occasionally frustrating. Throughout the course of the project, I became much more adept at anticipating what Dafny could and could not infer and the sorts of intermediate steps I would need. Thus I found that there is a signficant gulf in difficulty between some of the simple tutorials on Dafny's website/the homework problems and larger-scale verification projects that require sophisticated knowledge of how to effectively use an automated tool - even if the specifications are similar.
Finally, Dafny has very good documentation, both for simple and advanced features, which helped immensely when I found myself limited and needed further features such as quantifier manipulation[10] or abstract modules [3].

5.2 Dafny vs Other Verification Tools

I have substantial experience using Coq, an interactive proof assistant, and additional experience (with a project similar in scale to this) using Liquid Haskell, another automated tool based on SMT solving. I found Dafny generally to be more enjoyable to use than Liquid Haskell, where I often felt limited by the language and presented with indecipherable error messages. However, for simpler tasks, Liquid Haskell had more complete automation, without needing annotations (beyond the refined types) or lemmas in most places. There seems to be a trade off in these two tools between expressiveness and automation, but I think that Dafny's vastly improved expressiveness more than makes up for the manual effort needed. However, it is important to note that I chose both projects to explore the tools and to prove results about complicated

[10] Documentation here.

algorithms and data structures. I may have a different view of this tradeoff had I tried to use the tools for smaller or simpler projects where lots of expressiveness is unecessary.

Dafny has a few obvious differences with interactive tools such as Coq. It can handle imperative code, allows for loop invariants, and has much more built-in automation. However, I often did not feel like I was getting the benefit of this automation, as I had to essentially step through many more complicated proofs in a manner similar to Coq.
The single biggest difference was in the handling of failed goals. In Coq, you will know exactly what step failed and (most of the time) exactly why; in Dafny, this is not the case, which can be a bit frustrating. Similarly, in Coq, one can proceed incremetally through the proof, while in Dafny, the entire module must be re-verified each time the user wants to verify a change. Dafny thus may not be as scalable for larger projects.
One area I did not get much of a chance to explore was termination checking, which is often difficult in Coq (although there are many tools and packages to help) and was the source of much frustration with Liquid Haskell. In this project, however, I used only simple loops with known bounds, so termination was not an issue.
Finally, I definitely did not try to work with propositions nearly as complex as those I have often used in Coq. Here, we had relatively simple predicates to work with, the only slight complication being some nested quantifiers. Coq allows for much richer predicates, for instance through inductively defined propositions and dependent types.
On the whole, though, Dafny is a good choice for programs that use imperative code, have specifications that can be expressed easily in Dafny's logic, and which are simple enough that automation will be useful. I did feel that for large enough projects, Dafny may not scale well without an inordinate amount of work to keep verification times small. However, for smaller projects, Dafny is a very powerful and well-designed tool that is enjoyable to use.

6 Conclusions

In this work, we verified counting sort for the first time in Dafny and provided the first verified implementation of counting sort which is generic in its choice of element and key function. Along the way, I learned a huge amount about Dafny and became much better aquainted with automated tools for program correctness. The verification was difficult, and ended up requiring several different refactorings and alternate approaches to minimize verification time, reduce the amount of repeated code, and improve the elegance of some of the proofs. Additionally, I significantly improved my understanding of counting sort, its proof of correctness, and why it is so difficult to prove such a simple algorithm correct.
An obvious extension of this work is to use the verified algorithm in another project (for instance, in radix sort) and make use of the postconditions in further proofs. Other than that, this is a fairly self-contained project.
Overall, this project was challenging but enjoyable, and I would not be surprised if I end up using Dafny and other similar tools in the future.

References

[1] Thomas H. Cormen et al. "Introduction to Algorithms, Third Edition". In: 3rd. The MIT Press, 2009, pp. 194–196.

[2] Stijn de Gouw, Frank S. de Boer, and Jurriaan Rot. "Verification of Counting Sort and Radix Sort". In: *Deductive Software Verification – The KeY Book: From Theory to Practice*. Ed. by Wolfgang Ahrendt et al. Cham: Springer International Publishing, 2016, pp. 609–618. ISBN: 978-3-319-49812-6. DOI: 10.1007/978-3-319-49812-6_19. URL: https://doi.org/10.1007/978-3-319-49812-6_19.

[3] Jason Koenig and K. Rustan M. Leino. "Programming Language Features for Refinement". In: *Electronic Proceedings in Theoretical Computer Science* 209 (June 2016), pp. 87–106. ISSN: 2075-2180. DOI: 10.4204/eptcs.209.7. URL: http://dx.doi.org/10.4204/EPTCS.209.7.

[4] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Edmund M. Clarke and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370. ISBN: 978-3-642-17511-4.

[5] Rustan Leino and Paqui Lucio. "An Assertional Proof of the Stability and Correctness of Natural Mergesort". In: *ACM Transactions on Computational Logic (TOCL)* 17.1 (Dec. 2015). URL: https://www.microsoft.com/en-us/research/publication/assertional-proof-stability-correctness-natural-mergesort/.

7 Appendix: Counting Sort Proof of Correctness

Here we provide a pen-and-paper proof of correctness based on the Dafny verification. This is based on the alternate, more-invariant approach described briefly in Section 3.5 and does not require any Dafny knowledge to understand, though the notation largely aligns with the rest of the report.

We define counting sort as the following pseudocode:

```
function countingSort<A>(a: A[], k : int, key : A -> int) : A[] {
    b = new int[k]; //initialized to 0
    for(int i = 0; i < a.Length; i++) {
        b[key(a[i])] += 1;
    }
    b[0] -= 1; //for 1 indexing
    for(int i = 1; i < k; i++) {
        b[i] += b[i-1];
    }
    c = new A[a.Length];
    for(int i = a.Length - 1; i >= 0; i--) {
        c[b[key(a[i])]] = a[i];
        b[key(a[i])] -= 1;
    }
    return c;
}
```

As expected for counting sort, we require that, for all x in a, $0 \leq key(x) < k$.
For talking about slices of arrays, we will use the notation $a[x..y]$, consisting of $a[x], a[x+1], ..., a[y-1]$.
We now define the following functions:

1. $numLt(a, x)$:= the number of elements in array a with key smaller than x.
2. $numEq(a, x)$:= the number of elements in array a with key equal to x.
3. $numLeq(a, x) := numLt(a, x) + numEq(a, x)$.
4. $position(x, i, a) := numLt(x, a) + numEq(x, a[0..i+1]) - 1$.
   This represents the position of an element in a sorted, stable array consisting of elements of a. See Section 3.2 for explanation.
5. $filter(a, f)$ filters the array a by predicate f - it keeps all elements in a that satisfy f, in order.

Now we begin the proof of correctness.

Lemma 1. *After the first loop, for all i, $b[i] = numEq(a, i)$.*

Proof. The proof is easy; for each element x in a with $key(x) = i$, we increment $b[i]$ exactly once. □

Lemma 2. *After the second loop, $b[i] = numLeq(a, i) - 1$.*

Proof. We prove the claim by induction on i. When $i = 0$, since all keys are nonnegative, $numLt(a, 0) = 0$, so $numLeq(a, i) = numEq(a, i)$. Since we subtract 1 from $b[0]$, the claim holds.
Now assume the claim is true for $i - 1$. Then $b[i] = b[i] + b[i - 1] = numEq(a, i) + numLeq(a, i - 1) - 1$. Since $numLeq(a, i - 1) = numLt(a, i)$, the claim holds. □

Now we need to handle the third loop. We will let c_i denote the portion of c that has been filled in when the loop counter is i. We will use the following loop invariants:

1. $a[(i + 1)..a.Length]$ and c_i are permutations of each other.
2. For all $0 \leq j \leq k$, $b[j] = position(j, i, a)$.
3. For all j such that $c_i[j]$ exists (ie, position j has been filled in), there exists a k such that $i < k < a.Length$ and $c_i[j] = a[k]$ and $j = position(key(a[k]), k, a)$.
4. For all integers x, $filter(a[i + 1..a.Length], y \to y = x) = filter(c_i, y \to y = x)$ (ie, the portion of a considered so far and the portion of c completed so far are stable with respect to each other).

Lemma 3. *All invariants hold before the third loop.*

Proof. Since $i = a.Length - 1$, $a[i + 1..a.Length]$ is empty, as is c_i. Thus, the only nontrivial invariant is invariant 2.
Since $i = a.Length - 1$, $position(j, i, a) = numLt(j, a) + numEq(j, a[0..a.Length]) - 1 = numLt(j, a) + numEq(j, a) - 1 = numLeq(j, a) - 1$. This is true by Lemma 2. □

Now, we will prove that each invariant is preserved during the loop by first proving the following lemmas:

Lemma 4. *position is injective in the following sense: for any i, j, if $position(key(a[i]), i, a) = position(key(a[j]), j, a)$, then $i = j$.*

Proof. Suppose not. We consider 2 cases:

1. If $key(a[i]) = key(a[j])$, let $k = key(a[i])$. Then

$$position(key(a[i]), i, a) = numLt(k, a) + numEq(k, a[0..i + 1] - 1$$
$$position(key(a[j]), j, a) = numLt(k, a) + numEq(k, a[0..j + 1]) - 1$$

   WLOG, suppose $i < j$. Then $numEq(k, a[0..i+1]) + numEq(k, a[i+1..j+1]) = numEq(k, a[0..j+1])$. Since $i < j$ and $a[j] = k$, $numEq(k, a[i + 1..j + 1]) > 0$. This contradicts the fact that the positions were equal.

2. If $key(a[i]) \neq key(a[j])$, WLOG assume $key(a[i]) < key(a[j])$. We have the following bounds straight from the definition of *position*:

$$position(key(a[i]), i, a) \leq numLeq(key(a[i]), a) - 1$$
$$numLt(key(a[j]), a) \leq position(key(a[j]), j, a)$$

   Note that we can write $numLt(key(a[j]), a) = numLeq(key(a[j]) - 1, a)$ (since we are working with integers). Now $key(a[i]) \leq key(a[j]) - 1$, so

$$\begin{aligned} position(key(a[i]), i, a) &\leq numLeq(key(a[i]), a) - 1 \\ &\leq numLeq(key(a[j]) - 1, a) - 1 \\ &= numLt(key(a[j]), a) - 1 \\ &< numLt(key(a[j]), a) \\ &\leq position(key(a[j]), j, a) \end{aligned}$$

   This again contradicts the position equality.

□

Lemma 5. *At the beginning of each iteration of the loop, $c[b[key(a[i])]]$ has not yet been filled in.*

Proof. Suppose it had, then by invariant 3, there is some k such that $i < k < a.Length$, $c_i[b[key(a[i])]] = a[k]$, and $b[key(a[i])] = position(key(a[k]), k, a)$. But by invariant 2, $b[key(a[i])] = position(key(a[i]), i, a)$. By Lemma 4, then, $i = k$, a contradiction. □

Lemma 6. *At the beginning of each iteration of the loop, for all $0 \leq j < b[key(a[i])]$, if $c_i[j]$ exists (ie, position j has been filled), then $key(c_i[j]) \neq key(a[i])$. In other words, we fill in all the values with $key = key(a[i])$ from right to left.*

Proof. Suppose not, so there is some $j < b[key(a[i])]$ where $key(c_i[j]) = key(a[i])$. By invariant 3, there is some k with $i < k < a.Length$, $c_i[j] = a[k]$, and $j = position(key(a[k]), k, a)$. By assumption, $key(a[k]) = key(a[i])$. By invariant 2, $b[key(a[i])] = position(key(a[i]), i, a)$. Thus, we get that $position(key(a[i]), k, a) < position(key(a[i]), i, a)$. Since $i < k$, this is a contradiction (ie, we cannot decrease the number of equal elements in the array by extending the range we are considering). □

Now, we can prove that each invariant is preserved.

Lemma 7. *Each invariant is preserved by the body of the third loop.*

Proof. Let b_{old} represent b at the beginning of the current iteration of the loop, and b_{new} represent b after the body of the loop.
By Lemma 5, c_{i-1} is the same as c_i, except that we fill in $c_{i-1}[b_{old}[key(a[i])]]$ with $a[i]$. All other positions are the same. Also note that $a[i..a.Length]$ is just $a[i+1..a.Length]$ with $a[i]$ added at beginning. Finally, b_{new} is the same as b_{old} except that the only change to b is that $b_{new}[key(a[i])] = b_{old}(key(a[i])] - 1$. We then consider each invariant in order.

1. By above, we add $a[i]$ to both sides, so they are still permutations of each other.
2. We consider 2 cases:
   (a) If $j = key(a[i])$, then $b_{new}[j] = b_{old}[j] - 1 = position(j, i, a) - 1$. But $position(j, i-1, a) = position(j, i, a) - 1$, since $key(a[i])$ appears at position $a[i]$.
   (b) If $j \neq key(a[i])$, then $position(j, i, a) = position(j, i-1, a)$, since the number of equal keys in the given range did not change; we just removed an unequal element.

   Thus, invariant 2 is preserved.
3. For all $j \neq key(a[i])$, the claim follows from the invariant. If $j = b_{old}[key(a[i])]$, we have $i - 1 < i < a.Length$, $c_{i-1}[j] = a[i]$, and by invariant 2, $j = position(key(a[i]), i, a)$, proving the claim.
4. If $x \neq key(a[i])$, then, since we added $a[i]$ to both sides, we did not change anything with respect to stability, so the claim continues to hold.
   If $x = key(a[i])$, then by Lemma 6, there were no elements with equal keys before index $b_{old}[key(a[i])]$; thus, if we consider the (filtered) list of elements with $key = key(a[i])$ in c_i, we have added $a[i]$ before all of the others. Likewise, we added $a[i]$ to the beginning of $a[i+1..a.Length]$, so the claim holds.

□

Therefore, we know that, once the third loop exits, the following are true (define $c_{ret} := c_{-1}$ - the array c when the loop exits):

1. a and c_{ret} are permutations.
2. a and c_{ret} are stable with respect to each other.
3. For every index j where $c_{ret}[j]$ exists, there exists a k such that $0 \leq k < a.Length$ and $c_{ret}[j] = a[k]$ and $j = position(key(a[k]), k, a)$.

The first condition implies that $|a| = |c_{ret}|$ and thus, all indices of c_{ret} are filled in (since we created c such that $c.Length = a.Length$). So the third condition is really equivalent to:

- For every $0 \leq j < c.Length$, there exists a k such that $0 \leq k < a.Length$ and $c_{ret}[j] = a[k]$ and $j = position(key(a[k]), k, a)$.

All that remains is to prove that the above condition implies sortedness. We do so in the following lemma:

Theorem 8. *c_{ret} is sorted.*

Proof. We prove this in two parts: first, we claim that, for every $0 \leq j < c_{ret}.Length$, $numLt(key(c_{ret}[j]), c_{ret}) \leq j \leq numLeq(key(c_{ret}[j]), c_{ret}) - 1$.
To prove this, we consider the k such that $0 \leq k < a.Length$, $c_{ret}[j] = a[k]$, and $j = position(key(a[k]), k, a)$.
Let $x = key(a[k]) = key(c_{ret}[j])$. Again, we know that:

$$\begin{aligned} numLt(x, a) \leq position(x, k, a) &\leq numLeq(x, a) - 1 \\ numLt(x, a) \leq j &\leq numLeq(x, a) - 1 \end{aligned}$$

It is clear by definition, that $numLt$ and $numLeq$ are preserved over permutations. Thus,

$$numLt(x, c_{ret}) \leq j \leq numLeq(x, c_{ret}) - 1$$

which is what we wanted to show.
Now we prove that the above condition implies sortedness. We want to prove that, for all $i \leq j$, $key(c_{ret}[i]) \leq key(c_{ret}[j])$.
Suppose not, so $key(c_{ret}[j]) < key(c_{ret}[i])$.
Then, using a similar idea as in Lemma 4,

$$\begin{aligned} j &\leq numLeq(key(c_{ret}[j]), c_{ret}) - 1 \\ &< numLeq(key(c_{ret}[j]), c_{ret}) \\ &\leq numLeq(key(c_{ret}[i]) - 1, c_{ret}) \\ &= numLt(key(c_{ret}[i]), c_{ret}) \\ &\leq i \end{aligned}$$

So $j < i$, a contradiction. Thus, c_{ret} is sorted. □

We have shown, therefore, that the returned array is a permutation of the input, is sorted, and is stable with respect to the input.