Software Verification Project
- AS 2015 -

Roger Koradi, Samuel Ueltschi
ETH Zürich, Switzerland

November 20, 2015

Abstract

This paper documents and discusses the authors' solutions to a project accompanying the ETH's *Software Verification* course in autumn semester 2015.

1 Introduction

Producing provably correct programs is becoming more and more important as society is being automatised and software is written that has control over safety-critical components, e.g. a car's brakes.

Proving a program correct is a tedious task, which leads to both, a high demand for tools assisting such proofs and an increased interest in research driving the development of these tools. The ETH's master course in *Software Verification* is - at the time of writing - encouraging students taking the course to complete a couple tasks requiring them to experiment with two notable verification tools: Autoproof[**autoproof**] for Eiffel and Boogie[**boogie**] for the verification language with the same name.

In a first part of the project, the students are asked, this year, to complete an Eiffel program such that Autoproof can verify it. The program can be found in the appendix (A.1). We discuss our solution in section 2 and provide full code for our solution in section 4.

A second part of the project consists of modelling a sorting algorithm that alternates between quick- and bucketsort, depending on the elements in the array passed to it, in boogie. The appendix holds a more detailed description of the algorithm in form of a boogie template (**??**). We discuss our approach in section 3, along with some issues and interesting behaviours we came across and provide full code of our solution in section 5.

2 Autoproof

We are given a class

```
class
        SV_AUTOPROOF

feature
        lst : SIMPLE_ARRAY [INTEGER]
```

And we will specify its features below in a way such that Autoproof can verify them.
We refer to Appendix(A.1) for the complete code that we modify.

2.1 wipe

Wipe(A.1.1) takes an array of integers and resets all its item to 0 We add a loop invariant

```
across 1 |..| (k-1) as i all
        x.sequence [i.item] = 0 end
```

This is sufficient, because the first postcondition,

```
x.count = old x.count
```

is already maintained by another invariant.

2.2 mod_three

Procedure *mod_three* (A.1.2) takes two integer arrays *a,b* of equal length, uses of *wipe* on both and returns *b* with its every third element set to one. First, we maintain that the amount of integers in *b* does not change:

```
b.count = b.count.old_
```

This invariant is necessary. Without it, the assignment

```
b[k] := a[k] + 1
```

may be out of bounds from Autoproof's point of view, because we iterate over the length of *a*, which is specified to be constant and only initially equal to the length of *b*.

Then, we need to postulate that each iteration over the loop can by itself change *b*:

```
modify(b)
```

Omitting this invariant will lead to Autoproof's insisting that *b* has never changed and that any further invariant claiming otherwise couldn't possibly be maintained.

Having specified that, we can now add an invariant

```
across 1 |..| (k−1) as i all
          (i.item \\ 3 = 0) implies
                    b.sequence [i.item] = 1 end
```

which will claim that every third item we already iterated over in a is one in b. The assignment may read

```
b [k] := a[k] + 1
```

but the postcondition from *wipe(a)* allows Autoproof to deduce the element's being set to one without any further specification of ours.

2.3 swapper

Swapper(A.1.4) relies on *swap*(A.1.3) to reverse *lst* (which is global).

The loop here goes

```
from
          x := 1
          y := lst.count
until
          y <= x
```

and after each iteration, x is incremented by one and y is decremented by one. This allows us to use *y ¿ 0* as a way of specifying an invariant that trivially holds before *y := lst.count* has been executed and specifies some useful property afterwards - in our case, we use it to specify that once initialised, both x and y are within the bounds of *lst*, and therefore satisfy the precondition of *swap*.

```
y > 0 implies (1 <= x and x <= lst.count and 1 <= y
    and y <= lst.count)
```

For Autoproof to be able to proof that the swapped list is a permutation of the original list, we need to specify that all items not swapped remained the same. *Swap* itself does provide such a postcondition, however, this is insufficient because the *old lst swap*'s postcondition is mentioning is in fact the *lst* at the moment *swapper* is calling *swap*, which changes with each iteration. We must link these two *"olds"* explicitly:

```
across x |..| y as i all lst.sequence[i.item] =
          lst.sequence.old_[i.item] end
```

Swapper's postcondition states

```
across 1 |..| lst.count as i all lst.sequence [i.item]
    = (old lst.sequence) [lst.count - i.item + 1] end
```

This directly motivates the addition of the following two loop invariants:

```
x > 1 implies across 1 |..| (x-1) as i all lst.
   sequence[i.item] = lst.sequence.old_[lst.count-i.
   item + 1] end

x > 1 implies across 1 |..| (x-1) as i all lst.
   sequence[lst.count-i.item+1] = lst.sequence.old_[i.
   item] end
```

Where we are again using *x ¿ 1* as a way of saying "x and y have both been initialised". We need to split the postcondition into two parts because the items in-between x and y have not been swapped yet.

However, we are using *lst.count-i.item+1* so that it matches the postcondition and Autoproof cannot proof that just yet because it's never actually using this expression in the loop. What we thus require is another invariant that specifies the index the loop is using to correlate to the arithmetic in the postcondition and the invariants:

```
y = lst.count - x + 1
```

2.4 search

Search (A.1.5) is traversing *lst* backwards and returns *True* iff it contains search key *v*. We need to specify both, postconditions and invariants, so we start with the postconditions to help us find the invariants we need.

First, we specify that *lst*, who is required to be wrapped, will remain so.

```
lst.is_wrapped
```

Since *search* does not contain a *modify*-clause for *lst*, we do not need to explicitly specify that it does not change *lst*.

We then specify the actual return value:

```
Result implies across 1 |..| lst.count as i some lst.
   sequence[i.item] = v end
(not Result) implies across 1 |..| lst.count as i all
   lst.sequence[i.item] /= v end
```

We could have replaced these two implications by an equality

```
Result = across 1 |..| lst.count as i some lst.
    sequence[i.item] = v end
```

but not doing so allowed us to check each direction individually, which helped in finding the required invariants.

The loop here goes

```
from
        k := lst.count
        Result := False
until
        Result or k < 1
```

We first specify *k* to remain within the allowed range of indices for *lst*, using *k ¿ 0* to ignore its value prior to entering the loop and after exiting the loop when *lst* does not contain *v*:

```
k > 0 implies (1 <= k and k <= lst.count)
```

Because *Result* is only ever set when the current iteration finds *v* and *k* is only decremented when we do not, specifying the case where we do find *v* becomes easy:

```
Result implies (lst.sequence[k] = v)
```

To specify the case where we do not find *v*, we take into account that *k* is being decremented, starting from the last valid index down to zero, which means at any time during the iteration, all elements with a valid index greater than *k* have been checked not to be *v*.

```
(not Result) implies (across (k+1) |..| lst.count as i
    all lst.sequence[i.item] /= v end)
```

2.5 prod_sum

Prod_sum (A.1.6) was a less complicated matter. In fact, copying the post-condition proved sufficient:

```
zz * y + xx = xx.old_
```

2.6 paly

Paly (A.1.7) takes an integer array and returns *True* iff its elements form a palindrome (i.e. represent an integer string whose every prefix is a reversed suffix). We need to specify both, postconditions and invariants, and we start again with the postconditions to help us find the invariants:

```
Result implies across 1 |..| a.count as i all a.
    sequence[i.item] = a.sequence[a.count-i.item + 1]
    end
(not Result) implies across 1 |..| a.count as i some a
    .sequence[i.item] /= a.sequence[a.count-i.item+1]
    end
```

As with *search*, we could have expressed these two implications as a single equality but choose not to for easier reasoning about it.

The loop here goes

```
from
        x := 1
        y := a.count
        Result := True
until
        x >= y or not Result
```

We first specify x and y to remain within the bounds of the array, using *y ¿ 0* to avoid violation on entry.

```
y > 0 implies (1 <= x and x <= a.count and 1 <= y and
    y <= a.count)
```

We then define a different interpretation for y to allow its translation into the arithmetic expected by the postconditions.

```
y = a.count - x + 1
```

Result is initialised to *True* and set to false as soon as we encounter a prefix that is not a reversed suffix. We exploit the fact that x is incremented with every iteration to express that *Result = True* implies "after at least one iteration" that the prefix iterated over so far is a reversed suffix:

```
(x > 1 and Result) implies across 1 |..| (x-1) as i
    all a.sequence[i.item] = a.sequence[a.count-i.item
    + 1] end
```

Because *Result* is iterated to true, we need no such trick for the other direction.

```
(not Result) implies across 1 |..| a.count as i some a
    .sequence[i.item] /= a.sequence[a.count-i.item + 1]
     end
```

3 Boogie

TODO

3.1 Quicksort Implementation

We implemented quicksort as specfified in "Introduction to Algorithms (TODO cite)". The design of our boogie implementation was mostly influenced by the bubblesort example from microsoft research (TODO cite). A global map variable `a :[int]int` is used to represent the array to be sorted. The qucksort implementation modifies this variable.
Our implementation of quicksort is divided into the following two procedures:

```
procedure qsPartition(lo : int, hi : int)
  returns (pivot_index: int, perm: [int]int) {
  ...
}

procedure qs(lo : int, hi : int) returns (perm: [int]int) {
  ...
}
```

Both procedures take arguments `lo` and `hi` to specify which part of the array is processed. `qsPartition` divides the array into two parts, a left part that is smaller than or equal to the pivot element and a right part that is greater than the pivot element. The pivot element is defined to be the right most element of the array. The return value of `qsPartition` is the final index of the pivot element and another map `perm :[int]int` which is a permutation on the range *lo* to *hi*. This permutation expresses how the original array was permutetd by the algorithm.

The second procedure `qs` first calls `qsPartition` on the entire array, then it recursively calls itselfe on the left and the right part of the array. This procedure also returns a permutation to indicate how the elements of the array were permuted.

Implementing the actual sorting algorithm was actually rather easy. We could basically copy the textbook definition to implement `qsPartition` and `qs`. The hard part was to construct a permutation that keeps track of how the array elements are mutated. In `qsPartition` this part is easy because the `perm` map just keeps track of how array elements are swapped. The difficult part was to combine the permutation that is return by `qsPartition` with the two permutations that are returned by the two recursive calls to `qs`.

First the two permutations returned by the recursive calls have to be combined to a new permutation on the entire range from *lo* to *hi*. Then this permutation is again combined with the permutation returned by `qsPartition`. This bookkeeping makes up a bigger part of the implementation.

This initial version of quicksort works but it has one major drawback. The procedure is designed to only sort a single global variable. However it is not possible to sort arbitrary arrays e.g. arrays that are passed as input argument to the procedure. Initially we overcame this shortcomming by introducing more global array variables and having multiple copies of the same qucksort procedure, each working on a different global variable. To avoid having to do this for each new array that has to be sorted, we came up with the following solution. First the `qs` procedure was changed to sort a global array variable called `a_qs`. Then to be able to sort arbitrary arrays, we introduced the following new procedure:

```
procedure quickSort(arr : [int]int, lo : int, hi : int)
  returns (arr_sorted : [int]int, perm: [int]int)
  modifies a_qs;
{
  // write input array into a_qs
  a_qs := arr;
  // let quicksort implementation sort a_qs
  call perm := qs(lo,hi);
  // write a_qs into output argument
  arr_sorted := a_qs;
}
```

This procedure uses `a_qs` as temporary variable for sorting arbitrary arrays. `quickSort` takes an array as input argument, copies it into `a_qs`, calls `qs` to sort it and then writes the now sorted array into the return value. Using this construction we are now able to sorte arbitrary arrays by having just one global variable.

3.2 Bucketsort Implementation

For our bucketsort implementation we decided to copy the procedure signature from quicksort.

```
procedure bucketSort(lo : int, hi : int) returns (perm: [int]int)
  ...
}
```

We know that `bucketSort` is only called with array elements that range from $-3 * N$ to $3 * N$ therefore the algorithm divides the array to be sorted into three buckets with elements in the range $[-3*N)$, $[-N, N)$ and $[N, 3*N)$ respectively. The three buckets are implemented as three arrays. `bucketSort` iterates over the original array and copies each element to it's corresponding bucket array. Then each bucket is sorted using our existing qucksort implementation. After that, the now sorted bucket arrays are written back to the original array yielding a sorted version of the original array.

Unfortunately we didn't have the time to implement the construction of a permutation that represents how the elements of the original array were permuted by `bucketSort`. This task proved to be more challenging than in the case of quicksort. Because the original array is divided into three buckets in a single while loop and we can't know how many elements will end up in each bucket it's hard to construct a permutation that reflects how the array elements are divided into the three buckets.

3.3 Specification

Both sort functions share the following specification:

```
procedure sort(lo : int, hi : int) returns (perm: [int]int)
  modifies a;
  requires lo <= hi;
  // perm is a permutation
  ensures (forall i: int :: lo <= i && i <= hi ==> lo <= perm[i] && perm[i] <= hi);
  ensures (forall k, l: int :: lo <= k && k < l && l <= hi ==> perm[k] != perm[l]);
  // the final array is that permutation of the input array
  ensures (forall i: int :: lo <= i && i <= hi ==> a[i] == old(a)[perm[i]]);
  // array is sorted
  ensures (forall k, l: int :: lo <= k && k <= l && l <= hi  ==> a[k] <= a[l]);
{ ... }
```

TODO: discuss specification choices ("in particular, permutation")
TODO: describe difficulties + how they were overcome
TODO: contrast Boogie - Autoproof

3.4 Verification

TODO: report "significant problems" (e.g. which procedures could be verified, and which could not
TODO: describe changes to implementation/specification made to simplify proofs
TODO: describe which parts of the specification you could not verify and why
TODO: explain how you achieved modular verification

4 Eiffel Solution Code

```
class
    SV_AUTOPROOF

feature
    lst: SIMPLE_ARRAY [INTEGER]
```

4.0.1 wipe

```
feature
wipe (x: SIMPLE_ARRAY [INTEGER])
note
        explicit: wrapping
require
        x /= Void
        modify (x)
local
        k: INTEGER
do
        from
                k := 1
        invariant
                x.is_wrapped
                x.count = x.count.old_
                across 1 |..| (k-1) as i all x.
                   sequence [i.item] = 0 end
        until
                k > x.count
        loop
                x [k] := 0
                k := k + 1
        end
ensure
        x.count = old x.count
        across 1 |..| x.count as i all x.sequence [i.
           item] = 0 end
end
```

4.0.2 mod_three

```
mod_three (a, b: SIMPLE_ARRAY [INTEGER])
note
        explicit: wrapping
require
        a /= Void
        b /= Void
        a /= b
        a.count = b.count
        a.count > 0
        modify (a, b)
local
        k: INTEGER
do
        wipe (a)
        wipe (b)
        from
                k := 1
        invariant
                a.is_wrapped and b.is_wrapped
                a.count = a.count.old_
                modify(b)
                b.count = b.count.old_
                across 1 |..| (k-1) as i all (i.item
                   \\ 3 = 0) implies b.sequence [i.
                   item] = 1 end
        until
                k > a.count
        loop
                if k \\ 3 = 0 then
                        b [k] := a[k] + 1
                end
                k := k + 1
        end
ensure
        across 1 |..| b.count as i all (i.item \\ 3 =
           0) implies b.sequence [i.item] = 1 end
end
```

4.0.3 swap

```
feature
swap (x, y: INTEGER)
note
        explicit: wrapping
require
        lst.is_wrapped
        1 <= x and x <= lst.count
        1 <= y and y <= lst.count
        modify (lst)
local
        z: INTEGER
do
        z := lst [x]
        lst [x] := lst [y]
        lst [y] := z
ensure
        lst.is_wrapped
        lst.count = old lst.count
        across 1 |..| lst.count as i all i.item /= x
           and i.item /= y implies lst.sequence [i.
           item] = (old lst.sequence) [i.item] end
        lst.sequence [x] = (old lst.sequence) [y]
        lst.sequence [y] = (old lst.sequence) [x]
end
```

4.0.4 swapper

```
swapper
note
        explicit: wrapping
require
        lst.is_wrapped
        lst /= Void
        modify (lst)
local
        x, y: INTEGER
do
        from
                x := 1
                y := lst.count
        invariant
                lst.is_wrapped
                lst.sequence.count = lst.sequence.old_
                   .count

                y > 0 implies (1 <= x and x <= lst.
                   count and 1 <= y and y <= lst.count
                   )
                across x |..| y as i all lst.sequence[
                   i.item] = lst.sequence.old_[i.item]
                    end
                        -- necessary because the "old"
                           are out of sync (swapper's
                           "old" is different from
                           swap's)

                y = lst.count - x + 1
                        --necessary for the following
                           two invariants to succeed
                x > 1 implies across 1 |..| (x-1) as i
                    all lst.sequence[i.item] = lst.
                   sequence.old_[lst.count-i.item + 1]
                    end
                x > 1 implies across 1 |..| (x-1) as i
                    all lst.sequence[lst.count-i.item
                   +1] = lst.sequence.old_[i.item] end

        until
                y <= x
```

```
        loop
                swap (x, y)
                x := x + 1
                y := y - 1
        end
ensure
        lst.is_wrapped
        lst.sequence.count = (old lst.sequence).count
        across 1 |..| lst.count as i all lst.sequence
            [i.item] = (old lst.sequence) [lst.count -
            i.item + 1] end
end
```

4.0.5 search

```
feature
search (v: INTEGER): BOOLEAN
note
        status: impure
require
        lst.is_wrapped
        lst /= Void
local
        k: INTEGER
do
        from
                k := lst.count
                Result := False
        invariant
                k > 0 implies (1 <= k and k <= lst.
                   count)
                Result implies (lst.sequence[k] = v)
                (not Result) implies (across (k+1)
                   |..| lst.count as i all lst.
                   sequence[i.item] /= v end)
        until
                Result or k < 1
        loop
                if lst [k] = v then
                        Result := True
                else
                        k := k - 1
                end
        variant
                k - if Result then 1 else 0 end
        end
ensure
        lst.is_wrapped
        Result implies across 1 |..| lst.count as i
           some lst.sequence[i.item] = v end
        (not Result) implies across 1 |..| lst.count
           as i all lst.sequence[i.item] /= v end
end
```

4.0.6 prod_sum

```
feature
xx, zz: INTEGER

prod_sum (y: INTEGER)
require
        xx >= 0
        zz >= 0
        y > 0
do
        from
                zz := 0
        invariant
                is_open
                inv
                zz * y + xx = xx.old_
        until
                xx < y
        loop
                zz := zz + 1
                xx := xx - y
        end
ensure
        zz * y + xx = old xx
end
```

4.0.7 paly

```
feature
paly (a: SIMPLE_ARRAY [INTEGER]) : BOOLEAN
note
        explicit: wrapping
require
        a /= Void
local
        x, y: INTEGER
do
        from
                x := 1
                y := a.count
                Result := True
        invariant
                y > 0 implies (1 <= x and x <= a.count
                    and 1 <= y and y <= a.count)
                y = a.count - x + 1
                        --necessary for the following
                            invariant to succeed
                (x > 1 and Result) implies across 1
                   |..| (x-1) as i all a.sequence[i.
                   item] = a.sequence[a.count-i.item +
                     1] end
                (not Result) implies across 1 |..| a.
                   count as i some a.sequence[i.item]
                   /= a.sequence[a.count-i.item + 1]
                   end
        until
                x >= y or not Result
        loop
                if a [x] /= a [y] then
                        Result := False
                end
                x := x + 1
                y := y - 1
        variant
                y - x
        end
ensure
        Result implies across 1 |..| a.count as i all
            a.sequence[i.item] = a.sequence[a.count-i.
            item + 1] end
```

```
        (not Result) implies across 1 |..| a.count as
           i some a.sequence[i.item] /= a.sequence[a.
           count-i.item+1] end
end

end -- end of class
```

5 Boogie Solution Code

TODO

6 Conclusion

TODO

Appendices

A Provided Code

A.1 Unadulterated Eiffel Code

```
class
        SV_AUTOPROOF

feature
        lst: SIMPLE_ARRAY [INTEGER]
```

A.1.1 wipe

```
feature
wipe (x: SIMPLE_ARRAY [INTEGER])
note
        explicit: wrapping
require
        x /= Void
        modify (x)
local
        k: INTEGER
do
        from
                k := 1
        invariant
                x.is_wrapped
                x.count = x.count.old_
                -- ADD MISSING LOOP INVARIANT(S)
        until
                k > x.count
        loop
                x [k] := 0
                k := k + 1
        end
ensure
        x.count = old x.count
        across 1 |..| x.count as i all x.sequence [i.
           item] = 0 end
end
```

A.1.2 mod_three

```
mod_three (a, b: SIMPLE_ARRAY [INTEGER])
note
        explicit: wrapping
require
        a /= Void
        b /= Void
        a /= b
        a.count = b.count
        a.count > 0
        modify (a, b)
local
        k: INTEGER
do
        wipe (a)
        wipe (b)
        from
                k := 1
        invariant
                a.is_wrapped and b.is_wrapped
                a.count = a.count.old_
                -- ADD MISSING LOOP INVARIANT(S)
        until
                k > a.count
        loop
                if k \\ 3 = 0 then
                        b [k] := a[k] + 1
                end
                k := k + 1
        end
ensure
        across 1 |..| b.count as i all (i.item \\ 3 =
           0) implies b.sequence [i.item] = 1 end
end
```

A.1.3 swap

```
feature
swap (x, y: INTEGER)
note
        explicit: wrapping
require
        lst.is_wrapped
        1 <= x and x <= lst.count
        1 <= y and y <= lst.count
        modify (lst)
local
        z: INTEGER
do
        z := lst [x]
        lst [x] := lst [y]
        lst [y] := z
ensure
        lst.is_wrapped
        lst.count = old lst.count
        across 1 |..| lst.count as i all i.item /= x
           and i.item /= y implies lst.sequence [i.
           item] = (old lst.sequence) [i.item] end
        lst.sequence [x] = (old lst.sequence) [y]
        lst.sequence [y] = (old lst.sequence) [x]
end
```

A.1.4 swapper

```
swapper
note
        explicit: wrapping
require
        lst.is_wrapped
        lst /= Void
        modify (lst)
local
        x, y: INTEGER
do
        from
                x := 1
                y := lst.count
        invariant
                lst.is_wrapped
                lst.sequence.count = lst.sequence.old_
                   .count
                -- ADD MISSING LOOP INVARIANT(S)
        until
                y <= x
        loop
                swap (x, y)
                x := x + 1
                y := y - 1
        end
ensure
        lst.is_wrapped
        lst.sequence.count = (old lst.sequence).count
        across 1 |..| lst.count as i all lst.sequence
           [i.item] = (old lst.sequence) [lst.count -
           i.item + 1] end
end
```

A.1.5 search

```
feature
search (v: INTEGER): BOOLEAN
note
        status: impure
require
        lst.is_wrapped
        lst /= Void
local
        k: INTEGER
do
        from
                k := lst.count
                Result := False
        invariant
                -- ADD MISSING LOOP INVARIANT(S)
        until
                Result or k < 1
        loop
                if lst [k] = v then
                        Result := True
                else
                        k := k - 1
                end
        variant
                k - if Result then 1 else 0 end
        end
ensure
        -- ADD MISSING POSTCONDITION(S)
end
```

A.1.6 prod_sum

```
feature
xx, zz: INTEGER

prod_sum (y: INTEGER)
require
        xx >= 0
        zz >= 0
        y > 0
do
        from
                zz := 0
        invariant
                is_open
                inv
                -- ADD MISSING LOOP INVARIANT(S)
        until
                xx < y
        loop
                zz := zz + 1
                xx := xx - y
        end
ensure
        zz * y + xx = old xx
end
```

A.1.7 paly

```
feature
paly (a: SIMPLE_ARRAY [INTEGER]): BOOLEAN
note
        explicit: wrapping
require
        a /= Void
local
        x, y: INTEGER
do
        from
                x := 1
                y := a.count
                Result := True
        invariant
                -- ADD MISSING LOOP INVARIANT(S)
        until
                x >= y or not Result
        loop
                if a [x] /= a [y] then
                        Result := False
                end
                x := x + 1
                y := y - 1
        variant
                y - x
        end
ensure
        -- ADD MISSING POSTCONDITION(S)
end

end --end of class
```

A.2 Boogie Template

```
// Introduce a constant 'N' and postulate that it is
   non-negative
const N: int;
axiom 0 <= N;

// Declare a map from integers to integers.
// 'a' should be treated as an array of 'N' elements,
   indexed from 0 to N-1
var a: [int]int;

// Returns true iff the elements of 'arr' are small (i
   .e. values in the range -3N to +3N)
function has_small_elements(arr: [int]int): bool
{
  (forall i: int :: (0 <= i && i < N) ==> (-3 * N <=
     arr[i] && arr[i] <= 3 * N))
}

// Sorts 'a' using bucket sort or quick sort, as
   determined by has_small_elements(a)
procedure sort() returns ()
  modifies a;
{
  if (has_small_elements(a))
  {
      // sort 'a' using bucket sort
  } else
  {
      // sort 'a' using quick sort
  }
}
```