```
1  void sort(int a[],int n)
2  {
3      int i,j,this_a;
4
5      for(i=1;i<n;i++){
6          this_a=a[i];
7          j=i-1;
8
9          while(j>=0&&this_a<a[j]){
10             a[j+1]=a[j];
11             j=j-1;
12         }
13         a[j+1]=this_a;
14         //a[0 . . . i] contains the first i+1 elements, sorted
15     }
16 }
```

Table 1: Insert sort, as seen in 1_introduction.

18 - loop invariants

Loop invariants are a tool for verifying algorithms, in a sense they present an algorithm as a recursion in the mathematical sense and make it ameanable to proof. It is easiest to approach the loop invariant through an example, the function in Table **??** does insert sort. Now consider line 14; at this point the subarray of `a` running from 0 to `i`, which we will call `a[0:i]`, contains the 0 to `i` elements of the original list, but sorted. This is the *loop invariant*, it is a statement that is true before the first loop, at the end of each iteration and the fact it is true at the end means the algorithm has achieved its goal. By checking loops for invariants we can be confident the loop will achieve its goal.

Thus a loop invariant is a statement that has three properties:

- Initialization: it is true at the start.
- Maintenance: if it is true at the start of an iteration, it is true at the end.
- Utility: when the loop terminates the loop invariant shows that the algorithm has served its purpose.

For insertion sort the sorted list is `a[0]` at the start, it is trivially sorted and contains `i=0` element of list. If `a[0:i-1]` is sorted and contains the first i elements, then the `i`th iteration adds the $i+1$ element and moves it down the list so that it is bigger than the element to its left and larger than the one to its right. This means that the new, larger, list is sorted, maintaining the invariant. Finally, when the loop terminates texttti==n-1 so the invariant tells us `a[0:n-1]` contains all the elements in the original list, sorted. Hence, this is a list invariant.

As another example, consider binary search; a code listing is given in Table **??**. In it, a sorted array `a` is searched for a key `val`, there are two markers `low` and `high` and at each iteration the marker `mid` is set to the midpoint of `low` and `high`, if `a[mid]==val` it terminates,

```
1  int search(int a[],int n, int val)
2  {
3    int mid, low=0, high=n-1;
4
5    while(low<high){
6        mid=(low+high)/2;
7        if(a[mid]==val){
8          low=mid;
9          high=mid;
10       }
11       else if(val>a[mid])
12         low=mid+1;
13       else
14         high=mid -1;
15     }
16   if(a[high]==val)
17     return high;
18   return -1;
19 }
```

Table 2: Binary search, this is modified from 3_search.

if not it exploits the fact the list is sorted and sets `low=mid+1` if `a[mid]` is less than `val` or sets `high=mid-1` otherwise. The key point is that it changes the markers so that is `val` is in the original list it is in `a[low:high]`. This gives us the loop invariant which is that if `val` is in the list it is in `a[low:high]`. This is true at the start when `a[low:high]` is the whole list, if it is true at the start of each iteration it is true at the end because the list is sorted and, finally, the loop terminates when it contains only one element, since, either this is `val` or `val` wasn't in the list, then the algorithm has served its purpose.