Sorting

1 Insertion Sort

Insertion Sort solves the sorting problem:
Input: A sequence of n numbers $\langle a_1, a_2, ..., a_n \rangle$
Output: A permutation of the input sequence $\langle a'_1, a'_2, ..., a'_n \rangle$ where $a'_1 \leq a'_2 \leq ... \leq a'_n$.

The numbers to be sorted are known as *keys*. These keys are often associated with other data, which we call *satellite data*. A key and satellite data form a *record*. When sorting with respect to some key, the satellite data is also sorted so that it stays with the associated key.

In this text we shall write algorithms in pseudocode. Real implementations in various languages can be found in the same directory as this text. Some notes regarding conventions. We shall organise compound data into *objects*, composed of *attributes*. Parameters will be passed *by value*; that is to say, the called procedure receives its own copy of the parameter. Assignments to this parameter within the called procedure will not be visible to the caller. However, if the parameter is a pointer (for example, pointer x with some attribute f), assignment $x.f = 3$ will be visible to the caller, as will changes to an array, which is passed by a pointer to the first element.

Assume boolean operators are *short circuiting*. That is, for operators *and* and *or*, the left expression will be evaluated first. The right expression will only be evaluated if the full expression's value isn't determined by the left expression. For example, if x *and* y, y won't be evaluated if x is false. Similarly, if x *or* y, y won't be evaluated if x is true.

Insertion sort is an efficient algorithm when sorting a small number of elements.

```
INSERTION–SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1:i−1]
    j = i − 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j − 1
    A[j + 1] = key
```

In words, we separate the array of elements into two *subarrays* (that is, contiguous portions of the array), where the left subarray is sorted and the right may not be. We then iterate through the right array, taking the leftmost element in the right array (call this the key) and trying to find its correct position by comparing its value with each element in the left array, from right to left. When there is a value that is less than the key, insert the key to the right of this value, shifting all the values above this along by one. Repeating this with every element in the right array will eventually result in a sorted array. See Figure 1 for a visual representation.

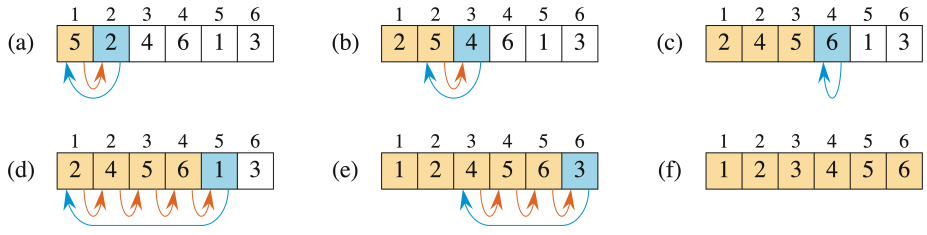

Figure 1: Visual representation of the insertion sort algorithm.

We want to prove that this is a correct algorithm. We can do this by mathematical induction using a *loop invariant*. A loop invariant is a property of a program that is true before and after each iteration of a loop. Thus, if we can formulate the desired outcome of our algorithm (in this case, for the array to be sorted) as a loop invariant, showing that this property is a valid loop invariant will be equivalent to proving correctness of the algorithm. To show that the loop invariant is corrent we must show:
Initialization: the property is true before the first iteration of the loop.
Maintenance: if the property is true before a given iteration, it is also true before the next iteration.
Termination: The loop terminates and gives a useful property that helps us to verify that the algorithm is correct.

For example, insertion sort is correct if the final array is sorted. Consider the following loop invariant:
At the start of each iteration of the for loop, the subarray `A[1:i-1]` consists of the elements originally in `A[1:i-1]`, but in correct sorted order.

To prove correctness of insertion sort, we prove the loop invariant always holds by induction:
Initialization: Before the first loop iteration[1], i = 2 so the left subarray is just `A[1:i-1]` = `A[1]`, which clearly contains the original element and is sorted.
Maintenance: Formally, we should check the while loop with a loop invariant. Informally, each interation moves the values by one until `A[i]` is placed in the correct position. This implies that `A[1:i]` is still sorted and contains the same elements. Thus, *incrementing* i preserves this.
Termination: To conclude our proof, we want to take the condition that terminates the loop and sub it into the original formulation of the loop invariant above. This should give us a statement that proves the correctness of the algorithm. The loop stops where $i > n$, or equivalently, $i = n + 1$. Subbing this in gives the following statement: *The subarray A[1:n] consists of elements originally in A[1:n] but in sorted order.* This is exactly our goal (that is, array A has been sorted) and so this proves correctness by induction.

Linear search solves the following *searching problem*:
Input: A sequence of n numbers $\langle a_1, a_2, ..., a_n \rangle$ stored in array A[1:n] and a value x.
Output: An index i such that x equals A[i] or the special value `NIL` if x does not appear in A.

Linear search scans the array from beginning to end, looking for x:

```
LINEAR-SEARCH(A, n, x)
for i in 1 to n
    if A[i] = x
        return i
return NIL
```

Prove its correctness with the following loop invariant: *At the start of each iteration, the subarray `A[1:i-1]` contains only elements not equal to x.* This is valid as the loop terminates when returning either i or `NIL`.

[1]In a `for` loop, this is between the counter initialization `int i = 2` and the condition i $\leq$ n assuming a C `for` loop: `for (int i = 2; i ≤ n; i++)`.