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.

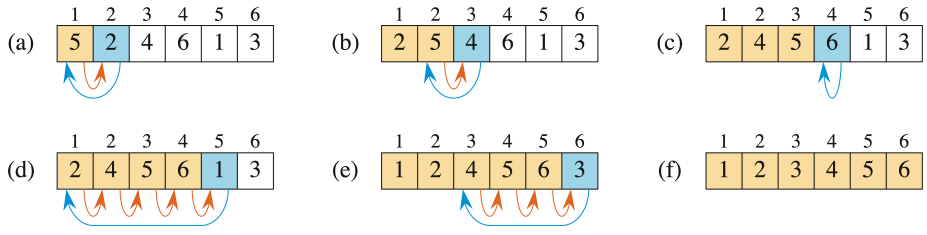

Figure 1: Visual representation of the insertion sort algorithm.

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, then 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`.

2 Analysis of Algorithms

Analysing an algorithm has come to mean predicting the resources that it requires, most commonly time (and memory). In the *random-access machine* (RAM) model of computation,

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

each instruction or data access takes a constant amount of time, and no operations occur concurrently. Note that we can't assume that every instruction takes constant time as this would lead to an unrealistic and unhelpful model. For example, an instruction that sorts in a single instruction, would be assumed to take constant time in the RAM model, which is unrealistic. Thus, the only computations that we consider to take constant time are those that can occur in real computers, especially arithmetic (standard operations, mod, floor, ceiling), data management (load, store, copy) and control (conditional & unconditional branch, subroutine call, return).

The notion of input size depends on the problem being studied. For example, in sorting, the number of items in the input is the clear choice; when multiplying two integers, the total number of bits needed to represent the input in binary may be used; for graph algorithms, both the number of vertices and edges are useful input sizes. The running time of an algorithm on a particular input is the number of instructions and data accesses executed. Assume each line of pseudocode takes constant time. Note that, while calling a subroutine takes a constant amount of time, actually executing this subroutine may (and likely will) take more.

We shall analyse Insertion Sort to demonstrate this analysis process. See figure 2 for the pseudocode of the algorithm with the cost of each step. Denote the number of times the while loop executes the `for i = 2, ..., n` with t_i. Thus, we must sum over the i to get the number of times these lines will execute. Note that the **for** and **while** loop execute one extra time than their bodies to account for the terminating check (that is, they execute $(n-1)+1$ times). The running time is the cost multiplied by the number of times this step occurs.

	INSERTION-SORT(A, n)	*cost*	*times*
1	**for** $i = 2$ **to** n	c_1	n
2	$key = A[i]$	c_2	$n-1$
3	// Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
4	$j = i-1$	c_4	$n-1$
5	**while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6	$A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n}(t_i-1)$
7	$j = j-1$	c_7	$\sum_{i=2}^{n}(t_i-1)$
8	$A[j+1] = key$	c_8	$n-1$

Figure 2: Pseudocode of insertion sort with the cost of each step.

Thus the running time for insertion sort is $T(n) = c_1 + c_2(n-1) + c_4(n-1) + c_5\sum_{i=2}^{n} t_i + c_6\sum_{i=2}^{n}(t_i-1) + c_7\sum_{i=2}^{n}(t_i-1) + c_8(n-1)$. Note that, while defining t_i in this way simplifies the function, we could have defined it in a different way if that were more useful.

In the best case scenario, the array was already sorted, so each **while** loop is only executed once. That is, $t_i = 1$ (for all i) so $T(n) = (c_1+c_2+c_4+c_5+c_8)n - (c_2+c_4+c_5+c_8) = an+b$, for some constants a and b. This is a *linear* function of n.

In the worst case scenario, the array is in reverse sorted order initially. In this case, $t_i = i$ as `A[j] > key` is true every time. We can prove the following relations by induction.

$$\sum_{i=2}^{n} i = \left(\sum_{i=1}^{n} i\right) - 1 = \frac{n(n+1)}{2} - 1$$

$$\sum_{i=2}^{n}(i-1) = \left(\sum_{i=1}^{n-1} i\right) = \frac{n(n-1)}{2}$$

If we write out T(n) again, we get a function of the form $T(n) = an^2 + bn + c$, where a, b and c depend on the constants c_k. This is a *quadratic* function of n.

Generally, we concentrate on the worst-case running time as this gives us an upper bound for the running time of our algorithm. Average-case running time can also be useful (for

example, in the above case $t_i = \frac{i}{2}$), although it's often the same as the worst-case. We are interested in rate of growth (or order of growth) of the running time, that is, only the leading term. We write the order of growth with theta notation. For example, insertion sort has $\Theta(n^2)$.

It is not common to consider the best-case running time, although there are reasons to do so. For example, the best case of insertion sort is linear time (which is the fastest conceivable comparison sort time as every element needs to be "looked at" at least once). We can use this initially to check if the array is already sorted, before applying some other sorting algorithm if it isn't. This is no more computationally expensive than just running the second algorithm immediately, which will have a running time that is greater than linear for the worst case. Thus, we can get the benefit of quickly sorting the special case for practically no cost.

2.1 Selection Sort

We now introduce another sorting algorithm: *selection sort*. This maintains the following loop invariant: *at the start of each outer for loop, the subarray A[1:i - 1] consists of the i - 1 smallest elements in the array A[1:n], and they are in sorted order*. It starts with a left (empty) subarray and the right subarray (which is just the array). It simply searches the right subarray for the lowest element and exchanges it with the first element in the right subarray. It then iterates over this, considering the left subarray to contain the former first element of the right array, thus shrinking the right array. This is made more explicit in the pseudocode below:

```
SELECTION-SORT(A, n)
for i = 1 to n - 1
    smallest = i
    for j = i + 1 to n
        if A[j] < A[smallest]
            smallest = j
    exchange A[i] with A[smallest]
```

This has average-case and worst-case running time $\Theta(n^2)$.