FIT2004: Supplementary notes for lecture 2

Analysis of Algorithms: Part II

DANIEL ANDERSON [1]

Invariants are a powerful tool for establishing the correctness of algorithms. Arising frequently and being of particular importance are *loop invariants*, which are invariants that are maintained by an iterative procedure (or loop) of a program. Loop invariants help us to analyse and establish the correctness of many simple sorting algorithms, which we will explore as an example.

Summary: Analysis of Algorithms : Part II

In this lecture, we cover:

- Using invariants to analyse algorithms
- Strong and weak loop invariants
- Analysing sorting algorithms using these invariants
- Best, average and worst-case performance of simple sorting algorithms

Recommended Resources: Analysis of Algorithms : Part II

- `http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Math/`
- `http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Sort/`
- CLRS, Introduction to Algorithms, Chapter 2
- Weiss, Data Structures and Algorithm Analysis, Chapter 7

Quadratic Sorting Algorithms

Sorting algorithms are some of the most important and fundamental routines that we will study. They are also very good case studies for the analysis of algorithms. You may be familiar with some of the quadratic ($O(N^2)$) sorting algorithms such as:

- Bubble sort
- Selection sort
- Insertion sort

There are also much faster ($O(N \log(N))$ and even $O(N)$ under certain conditions) sorting algorithms that we will deal with later. We will analyse Selection Sort and Insertion Sort, compare the two and explore their fundamental underlying properties.

Selection Sort

The Selection Sort algorithm is based on the following ideas:

[1] Faculty of Information Technology, Monash University: `daniel.anderson@monash.edu`. These notes are based on the lecture slides developed by Arun Konagurthu for FIT2004.

Key Ideas: Selection Sort

- Consider the list to be sorted as being divided into two parts
  1. The first part consists of the part of the list that is currently sorted
  2. The remaining part consists of the elements that are yet to be sorted
- Initially, the first (sorted) part is empty, while the remaining part is the entire (to be sorted) list
- We then sort the list like so:
  1. Search the unsorted part for the smallest element
  2. Swap this smallest element with the first element of the unsorted part
  3. The sorted part is now one element longer

Expressed more concretely, an implementation of Selection Sort might look this this.

Algorithm: Selection Sort

```
function SELECTION_SORT(array[1..N])
    for i = 1 to N do
        Set min = i
        for j = i+1 to N do
            if array[j] < array[min] then
                min = j
            end if
        end for
        swap(array[i], array[min])
    end for
end function
```

Let's try to understand what is going on here more logically. At some given value of `i` in the main loop of the algorithm, we have the two sublists

- `array[1..i-1]`, the sorted part
- `array[i..N]`, the yet to be sorted part

The key strategy behind selection sort simply says to progressively make `i` larger while maintaining the following invariants.

Invariant: Selection Sort

For any given value of `i` in the main loop of Selection Sort, at the beginning of the iteration, the following invariants hold:

1. `array[1..i-1]` is sorted
2. `array[1..i-1]` $\leq$ `array[i..N]`, ie. for any `x` $\in$ `array[1..i-1]` and `y` $\in$ `array[i..N]`, `x` $\leq$ `y`

Using these invariants, arguing the correctness of Selection Sort is easy.

Initial state of the invariants

Initially, the sorted part of the array is empty, so the invariants trivially hold.

Maintenance of the invariants

At each iteration of the main loop, we seek $\min\limits_{j \in [i..N]}$ `array[j]`. Since at the beginning of iteration `i`, it is true that `array[1..i-1]` $\leq$ `array[i..N]`, the value of `array[min]` is no greater than any `x` $\in$ `array[1..i-1]`, and hence the extended sorted part `array[1..i]` remains sorted after swapping `array[i]` and `array[min]`, so invariant 1 is maintained.

Since we are selecting `array[min]` as a minimum of `array[i..N]`, it is true that `array[min]` $\leq$ `array[i+1..N]` after swapping `array[i]` and `array[min]`, so invariant 2 is also maintained.

Termination

When the `i` loop terminates, the invariant must still hold, and since it was maintained in iteration `i = N`, this implies that `array[1..N]` is sorted, ie. the entire list is sorted, and we are done.

Time and Space Complexity of Selection Sort

Selection Sort always does a complete scan of the remainder of the array on each iteration. It is therefore not hard to see that the number of operations performed by Selection Sort is independent of the contents of the array. Therefore the best, average and worst-case time complexities are all exactly the same.

Theorem: Time Complexity of Selection Sort

The time complexity of Selection Sort is $O(N^2)$.

Proof

The number of iterations performed is `N - i` for each `i` from `1 to N`, therefore we perform

$$\begin{aligned}\sum_{i=1}^{N}(N-i) &= N^2 - \sum_{i=1}^{N} i \\ &= N^2 - \frac{N(N+1)}{2} \\ &= \frac{N^2-N}{2} \\ &= O(N^2)\end{aligned}$$

total operations.

We also only require one extra variable and some loop counters, so the amount of extra space required for Selection Sort is constant.

	Best case	**Average Case**	**Worst Case**
Time	$O(N^2)$	$O(N^2)$	$O(N^2)$
Auxiliary Space	$O(1)$	$O(1)$	$O(1)$

Insertion Sort

Insertion Sort is a very similar algorithm to Selection Sort, based on a similar but *weaker* invariant. The idea behind Insertion Sort, much like Selection Sort is to maintain two sublists, one that is sorted and one that is yet to be sorted. At each iteration of the algorithm, we move one new element from the yet-to-be sorted sublist into its correct position in the sorted sublist. An implementation of Insertion Sort might look something like this.

Algorithm: Insertion Sort

```
function INSERTION_SORT(array[1..N])
    for i = 2 to N do
        Set j = i
        while j $\geq$ 2 and array[j] < array[j-1] do
            swap(array[j], array[j-1])
            j = j - 1
        end while
    end for
end function
```

Selection Sort was based on the idea that we maintain two invariants with respect to the sorted and yet-to-be-sorted sublists. We required that the first sublist be sorted, and that the second sublist contain elements greater or equal to those in the sorted sublist. Insertion Sort maintains a weaker invariant.

Invariant: Insertion Sort

For any given value of `i` in the main loop of Insertion Sort, at the beginning of the iteration, the following invariant holds:

1. `array[1..i-1]` is sorted

Ie. Insertion Sort maintains just one of the two invariants that Selection Sort did, hence why we call it "weaker."

Initial state of the invariant

Again, the sorted sublist is empty at the beginning of the main loop, so the invariant trivially holds.

Maintenance of the invariant

At the beginning of iteration `i`, the sublist `array[1..i-1]` is sorted. The inner while loop of Insertion Sort swaps the item `array[j]` one place to the left until `array[j-1]` $\leq$ `array[j]`. Since `array[1..i-1]` was originally sorted, it is therefore true that after the termination of the while loop, `array[1..j]` is sorted. Since the sublist `array[j+1..i]` was originally sorted, and for each `x` $\in$ `array[j+1..i]`, we had `array[j]` $<$ `x`, it is also true that `array[j..i]` is sorted. Combining these observations, we can conclude that `array[1..i]` is now sorted, and hence the invariant has been maintained.

Termination

At each iteration of the inner while loop of Insertion Sort, `j` is decremented, therefore either the loop condition `j` $\geq$ `2` must eventually be false, or it will be the case that `array[j-1]` $\leq$ `array[j]`. Since the loop invariant was maintained when `i = N`, it is true that `array[1..N]` is sorted, and we are done.

Time and Space Complexity of Insertion Sort

Unlike Selection Sort, Insertion Sort behaves differently depending on the contents of the array.

Suppose we provide insertion sort with an already sorted list. The inner while loop will therefore never iterate since it will always be true that `array[j-1]` $\leq$ `array[j]`. Therefore the only operations required by Insertion Sort are to perform the outer loop from from `1 to N`. Therefore in the best case, Insertion Sort only takes $O(N)$ time.

In the worst case, suppose we provide an array of distinct elements that is sorted in reverse order. We can see that in this case, the inner while loop of Insertion Sort will have to loop the entire way from `i to 1` since it will always be true that `array[j]` $<$ `array[j-1]`. We observe that in this case, the amount of work done is the same as Selection Sort, since we perform `i - 1` operations for each `i` from `2` to `N`. Therefore in the worst case, Insertion Sort takes $O(N^2)$ time.

For a random list, in the average case we will have to swap `array[j]` with half of the proceeding elements, meaning that we need to perform roughly half of the amount of operations as the worst case. This means that in the average case, Insertion Sort still takes $O(N^2)$ time. If you are not satisfied by the lack of rigour in this explanation, we will see a more rigorous proof of Insertion Sort's average case time complexity later.

Finally, just like Selection Sort, the amount of extra space is constant, since we keep only one extra variable and a loop counter.

	Best case	**Average Case**	**Worst Case**
Time	$O(N)$	$O(N^2)$	$O(N^2)$
Auxiliary Space	$O(1)$	$O(1)$	$O(1)$

Properties of Sorting Algorithms

Stable Sorting

A sorting algorithm is said to be *stable* if the relative ordering of elements that compare equal is maintained by the algorithm. What does this mean? Say for example that we are going to sort a list of people's names **by their length**. If multiple names have the same length, a stable sorting algorithm will guarantee that for each class of names with the same length, that their ordering is preserved. For example, if we want to sort the names

`Bill, Bob, Jane, Peter, Kate` by their length, the names `Bill, Jane, Kate` all share the same length. A stable sorting algorithm will produce the following sorted list

`Bob, Bill, Jane, Kate, Peter`

noticing that `Bill, Jane, Kate` are relatively in the same order as the initial list. An *unstable* sorting algorithm may have shuffled the order of `Bill, Jane, Kate` while still producing a list of names that is sorted by length.

Insertion sort is an example of a stable sorting algorithm, since when comparing the keys we use the condition `array[j]` $<$ `array[j-1]` rather than `array[j]` $\leq$ `array[j-1]`. The former guarantees that two elements that compare equal are never swapped.

Selection Sort on the other hand is not a stable sorting algorithm, since swapping `array[i]` with `array[min]` may cause `array[i]` to be swapped out of its relative ordering within its class of equivalent elements.

Online Sorting

An algorithm (sorting or otherwise) is called *online* if it can process its input sequentially without knowing it all in advance, as opposed to an *offline* algorithm which must know the entire input in advance.

Insertion Sort is an example of an online sorting algorithm. If you only know some of the input, you can begin to run insertion sort, and continue to read in the rest of the input while continuing to sort, and the algorithm will still work.

On the other hand, Selection Sort is not an online algorithm, since if any new elements are added to the input, the algorithm would have to start all over.