Algorithms I – supervision 1

James Wood

January 26, 2015

1 Insertion sort

(a)

```
def insertSort(a):
    for i from 1 included to len(a) excluded:
        j = i - 1
        this = a[i]
        while j >= 0 and a[j] > this:
            a[j + 1] = a[j]
            j = j - 1
        a[j] = this
```

(b) All of `a[j + 1:i]` are greater than `this`, and are sorted.

(c) $O(n^2)$

2 Selection sort

`[7,6,5,4,3,2,1]`

3 Bubble sort

Keep track of the number of passes by introducing the variable `i`:

```
def bubbleSort(a):
    i = 0
    repeat:
        didSomeSwapsInThisPass = False
        for k from 0 included to len(a) - 1 excluded:
            if a[k] > a[k + 1]:
                swap(a[k], a[k + 1])
                didSomeSwapsInThisPass = True
        i = i + 1
    until didSomeSwapsInThisPass == False
```

The statement is then that, once the algorithm has completed, $i \leq n$.

Lemma 0

Items `a[n - i:n]` are in the correct place at the start of each pass.

Proof

`a[n - i:n]` are in the correct place iff it is a sorted list containing elements greater than or equal to every element in `a[0:n - i]`. This is equivalent to saying that `a[n - i:n]` is sorted and `a[n - i]` is greater than or equal to every element in `a[0:n - i]`, given that every element in the sorted list `[n - i:n]` is greater than or equal to `a[n - i]`.

We proceed by induction. In the base case, where $i = 0$, the list in question is empty. Thus, it is sorted and any predicate is true for all of its elements. Then, given that `a[n - i:n]` satisfy the conditions, consider `a` after another pass. Let `x` denote the largest element in `a[0:n - i]`, and suppose that it starts in position `j`. When `k == j`, either `k + 1 == n - i`, in which case `x` was already in the correct place, or `a[k] > a[k + 1]` is true by the fact that `x` is greater than every other element in `a[0:n - i]`. In the latter case, `x` moves to position `k + 1`. By repeating this process, it is seen that `x` is moved to position `n - i - 1`. Hence, `a[n - i - 1:n]` are in the correct place, QED.

Using this lemma, we note that, at the start of the `n`th pass, the elements of `a[n - n:n]`, equivalently `a`, are in the correct place. The elements are ordered in such a way that `a[k] > a[k + 1]` is always false, so `didSomeSwapsInThisPass` remains `False` and the algorithm terminates.

4 Mergesort

(a) The standard `min` function loses the information of which of the two numbers it chose. Hence, it becomes difficult to update `i1` and `i2` to reflect which sublist was chosen from. To solve this problem, I would write the function `minE : (Ord a) => (a,a) -> Either a a` defined by:

```
def minE(x, y):
    if x <= y:
        Left x
    else:
        Right y
```

I would then case split on its result:

```
case minE(a1[i1], a2[i2]):
    Left x:
        a3[i3] = x
        i1 = i1 + 1
    Right y:
        a3[i3] = y
        i2 = i2 + 1
```

The main problem with this approach is that it requires a language that is so Haskell-like that the imperative pseudocode algorithm would have to be completely restructured before it could be implemented. Alternatively, the information conveyed in the choice of `Left` vs `Right` could be conveyed in a boolean value, returned in a tuple with the minimum. Then the only disadvantage is the use of slightly more space and time than a function `smallest` that mutates its arguments.

(b) When merging, start storing the merged elements in the $n/2$-sized spare array. Once half of the elements have been merged, swap the elements from the first sorted subarray with the merged elements, and move the elements from the second sorted subarray to the end of the main storage array. Then, continue merging until there are no more elements in the spare array to consider. The remaining elements from the second sorted subarray will then be in the correct place, and the whole array will be sorted.

5 Quicksort

First-element-pivot or last-element-pivot quicksort's worst case comes about when the input list is monotonic. In these cases, random-element-pivot quicksort improves performance on average by more often picking a non-extremal element. However, the worst case remains the same, namely when random-element-pivot quicksort happens to emulate the other variants. In fact, it is possible for any input list to become pathological in random-element-pivot quicksort. However, in general, the expected performance is the same, except that random numbers have to be generated (though only to the order of n in the worst case).

6 Comparisons

(a) $n - 1$

(b) $2 \cdot n - 3$