Universität Innsbruck - Institut für Informatik
Prof. Clemens Ballarin, Robert Binna, Friedrich Neurauter, Francois Scharffe and Sarah Winkler

3 June 2008

Proseminar Algorithmen und Datenstrukturen

Exercise Sheet 10

Exercise 1 (Master Theorem)

Give a recurrence equation for the worst-case running time and a tight asymptotic (Θ-notation) bound on the worst-case running time of the following algorithms:

a) Binary search in a sorted array of size n.

Searching for a value in a sorted array of size n, we examine the middle value (this is a constant time operation, hence it is in $O(1)$). Either this is the value we are looking for and we are done, or we must search for the value in either the left or the right half of the array. Thus we obtain a smaller problem of the same type as we had before. The recurrence we obtain is

$$T(n) = T(\frac{n}{2}) + 1$$

An application of the master theorem yields

$$T(n) \in \Theta(n^{\log_2(1)} \log(n)) = \Theta(\log(n))$$

b) $StoogeSort(A, 1, n)$ where n is the size of the array A.

In total we have three recursive calls of $StoogeSort$, each of which operates on a subarray of length $\frac{2}{3}n$. All other statements are independent of n, hence constant time operations. Thus the recurrence we obtain is

$$T(n) = 3 \cdot T(\frac{2n}{3}) + 1$$

and an application of the master theorem yields

$$T(n) \in \Theta(n^{\log_{3/2}(3)}) = \Theta(n^{\log(3)/\log(3/2)}) = \Theta(n^{2.7})$$

Listing 1 *StoogeSort*

Input: array A, start index i, stop index j
Output: the array elements $A[i..j]$ are sorted!

```
1: if A[i] > A[j] then
2:    exchange A[i] ↔ A[j]
3: if i + 1 ≥ j then
4:    return
5: k := ⌊(j − i + 1)/3⌋
6: StoogeSort(A, i, j − k)
7: StoogeSort(A, i + k, j)
8: StoogeSort(A, i, j − k)
```

Exercise 2 (Stooge Sort)

Prove that $StoogeSort(A, 1, length(A))$ correctly sorts the input array A by induction on the length of A.

Clearly, the algorithm works correctly for one-element and two-element arrays, which establishes the base case of the induction.

In the inductive step we see that the recursive calls of *StoogeSort* in Lines 6,7 and 8 operate on strict subarrays of the input array, and we may thus assume that each of these calls correctly sorts its input (by the induction hypothesis!). So the call in Line 6 correctly sorts the first two thirds of the array.

Having executed this call, clearly every element of the second third of the array is no smaller than any element of the first third. Thus none of the elements of the first third should actually end up in the last third, and by inspection of Lines 7 and 8 that is clearly the case.

At this point we know that all elements that actually belong to the last third (i.e. the largest elements) are contained in the last two thirds of the array. Hence, executing the call at Line 7, which sorts the last two thirds of the array, has the effect that all those elements are now moved to their final position inside the last third.

At this point we know that all elements that actually belong to the first two thirds are contained in the first two thirds of the array. Hence, executing the call at Line 8, which sorts the first two thirds of the array, has the effect that all those elements are now moved to their final position.

Exercise 3 (Quicksort)

Construct a non-trivial example for which quicksort will use $\Omega(n^2)$ comparisons when the pivot is chosen by taking the median of the first, last and middle elements of the sequence (the median of a finite list of numbers can be found by arranging them from lowest value to highest value and picking the middle one, e.g. $median(4, 3, 5) = 4$).

Quicksort runs in $\Omega(n^2)$ time when in every step of the recursion the pivot divides the array into two parts, one of which is empty or contains only few elements and the other of which contains the remaining elements (and thus is much bigger). The exact shape of such an input array depends on the partition algorithm that is used.

Here is an example that works for the partition algorithm you learnt in the lecture:

$$1, 3, 5, 7, 9, 2, 4, 6, 8$$

Running quicksort manually, we see that in every step the array is split into two parts such that one side always has one element and the other has the rest (minus the pivot element). For example, at the beginning $quicksort(1, 3, 5, 7, 9, 2, 4, 6, 8)$ chooses 8 as the pivot element and leads to the following two recursive calls, $quicksort([1, 3, 5, 7, 2, 4, 6])$ and $quicksort([9])$. Similarly, $quicksort([1, 3, 5, 7, 2, 4, 6])$ chooses 6 as the pivot element and leads to the following two recursive calls, $quicksort([1, 3, 5, 2, 4])$ and $quicksort([7])$; and so on and so forth.

And finally, this is an example that works for Hoare's original partition algorithm:

$$1, 3, 5, 7, 9, 2, 6, 4, 8$$

Exercise 4 (Sorting Algorithms)

a) Implement *mergesort* in C and incorporate it into the framework of Exercise 3 of last week.

See listing *mysort.c*

b) Implement *quicksort* in C and incorporate it into the framework of Exercise 3 of last week.

See listing *mysort.c*