S. Schwoon / D. Suwimonteerabuth Winter Term 2008/09

Fundamental Algorithms

Exercise Sheet 5

1. Let a and b be integers having n digits. *Long multiplication* is a natural way of multiplying a by b taught in schools: multiply a by each digit of b and then add up all the properly shifted results. The operations needed are multiplications for single digits, shifts, and additions. The time complexity is obviously $\Theta(n^2)$.

   Devise a multiplication algorithm based on the divide-and-conquer paradigm that has a better complexity, using only three operations above.

   Hints: if n is an even number, we can rewrite $a = 10^{n/2}u + v$ where u and v are first and last $n/2$ digits of a, respectively, and $b = 10^{n/2}x + y$, where x and y are first and last $n/2$ digits of b, respectively.

2. A *comparison sort* is a type of sorting algorithm that determines the sorted order based only on comparisons between input elements. For instance, insertion sort, heapsort, and quicksort are comparison sorts; whereas counting sort and radix sort are not.

   In the lecture, it has been shown that any comparison sort must make $\Omega(n \log n)$ comparisons in the worst case to sort an array of n elements. The proof assumes that all elements are distinct; hence, there are $n!$ permutations, only one of which is in sorted order. Since each comparison has only two possible outcomes, if the sorting algorithm is always able to find the right permutation after at most h comparisons, we know that it cannot distinguish more than 2^h cases. Therefore, $2^h \geq n!$, or equivalently $h \geq \log_2 n!$. From Stirling's approximation, we have $\log_2 n! \in \Omega(n \log n)$, and the corresponding lower bound.

   We know, however, that sometimes comparison sorts can perform better than $\Omega(n \log n)$ in *best* cases. For instance, insertion sort requires only $\Theta(n)$ when the array is already sorted.

   Show that there is no comparison sort whose running time is $\Theta(n)$ for a fraction of $1/2^n$ of the inputs of length n.

3. In this exercise, we are interested in two algorithms that work together. The first algorithm takes an array a of length n whose elements are in the range 1 to k as the parameter. It processes a and returns something for the second algorithm in $\mathcal{O}(n + k)$ time. The second algorithm takes the output of the first algorithm together with two integers u and v as the parameters, and returns the number of elements in a that fall into the range $[u..v]$.

   Design both algorithms such that the second algorithm always takes $\mathcal{O}(1)$ time. What is your output of the first algorithm?