Solution of Problem Set no. 4

Exercise 4.1 (a) Show that the constants $C_1 = C_1(k,b)$ and $C_2 = C_2(k,b)$ of slide 28 can be constructed in $O(\log k)$ time without the use of multiplication (and division). (b) Show that C_1 can be constructed in $O(1)$ if the use of multiplication and division is allowed. (c) Show that if C_1 is available, then C_2 can be constructed in $O(1)$ time with the use of multiplication.

Solution 4.1 We assume for simplicity that k is a power of 2. This assumption can be removed at the cost of some non-essential technical complications.
(a) We first show that $C_1(k,b)$ can be constructed in $O(\log k)$ time. Initialize $C_1 \leftarrow (1 \ll b)$ and $\ell \leftarrow b+1$. Now, repeat $C_1 \leftarrow (C_1 \ll \ell) \vee C_1$, and $\ell \leftarrow \ell \ll 1$, $\lg k$ times. In a similar manner, we can also construct $C_2(k,b)$ in $O(\log k)$ time. Initialize $A \leftarrow 1$, $C_2 \leftarrow 0$, $\ell \leftarrow b+1$, $i \leftarrow 0$. Now, repeat $A \leftarrow (A \ll \ell) \vee A$, $C_2 \leftarrow ((C_2 + (A \ll i)) \ll \ell) \vee C_2$, $\ell \leftarrow \ell \ll 1$, $i \leftarrow i+1$, $\lg k$ times.
(b) Note that $C_1 = \frac{2^{b+1}(2^{2k(b+1)}-1)}{2^{b+1}-1}$. If k is a power of 2, we do not even need multiplication. Also not that the division here by $2^{b+1}-1$ is exact, i.e., without remainder.
(c) Note that $C_1/2^b = \sum_{i=0}^{2k-1} 2^{(b+1)i}$. Hence, $(C_1 \gg b)^2 \bmod 2^{2k(b+1)} = \sum_{i=0}^{2k-1}(i+1)2^{(b+1)i}$. Thus, $C_2 = ((C_1 \gg b)^2 \wedge ((1 \ll (2k(b+1))) - 1)) - (C_1 \gg b)$.

Exercise 4.2 Show that n integers, each of $w/(\log n \log\log n)$-bits, where w is the word size, can be sorted in $O(n)$ time. (Randomization is allowed.)

Solution 4.2 In class we saw a sorting algorithm that starts by partitioning the n items to be sorted into groups of size $k = \Theta(\log n \log\log n)$, sorts each group naively in $O(k \log k)$ time, and then packs each group into a word. The time of this step is $O((n/k) \cdot k \log k) = O(n \log k) = O(n \log\log n)$. Afterwards, the packed groups were sorted in $O(n)$ time. Thus, to get an $O(n)$-time algorithm, we only need to implement the first step of the algorithm in $O(n)$ instead of $O(n \log\log n)$ time. We can do that as follows. We pack each group of k items into a word in an unsorted manner. On each word we simulate Batcher's bitonic sort in $O((\log k)^2)$ time. The packing takes only $O(n)$ time. The sorting takes $O((n/k)(\log k)^2) = o(n)$. Thus, the whole step takes $O(n)$ time, as required. (We saw in class how to simulate a bitonic merge network in $O(\log k)$ time. Using very similar ideas we can simulate a bitonic sort network in $O(\log^2 k)$.) No randomization is actually required.

Exercise 4.3 (a) How much space is used by a straightforward implementation of the $O(n \log\log n)$-time integer sorting algorithm given in class? (b) Show that the space requirement can be reduced to $O(n)$ while maintaining the $O(n \log\log n)$ running time.

Solution 4.3 (a) The algorithm that we saw (see pseudo-code on slide 26) is composed of $O(\log\log n)$ recursive calls to range reduction steps that reduce the number of bits in each key from w to $w/(\log n)^2$. Each recursive call has its own hash table of buckets (B), which may require $\Theta(n)$ space. Thus, the straightforward implementation requires $O(n \log\log n)$ space.
(b) Each recursive call halves the number of bits in each key. Thus, after the first recursive call, we can work with half words instead of words, after the second recursive call with quarter words, etc. Thus, the space needed for the hash table in the i-th recursive call is only $O(n/2^i)$, and the total space used is $O(n)$. (If items have associated information, then this information should be stored once and not copied during each recursive call.)

Exercise 4.4 Show that for every integer $r \geq 1$, there is a *deterministic* $O(n \log\log n)$-time algorithm for sorting n integers that uses $O(n + 2^{w/r})$ space.

Solution 4.4 Suppose at first that $b = w/r$, where $r \geq 1$ is some integer. The $O(n \log\log n)$-time algorithm we saw in class uses a hash table with b-bit keys. To avoid randomization, we can simply use an array of size $2^b = 2^{w/r}$. (We use the trick of working with an uninitialized array.) Combining this with the previous exercise, we get an $O(n \log\log n)$-time deterministic algorithm that uses only $O(n + 2^{w/r})$ space. Now, if $b = w$, we can simply use radix sort, treating each w/r consecutive bits as a character, to get an $O(rn \log\log n)$-time deterministic algorithm that uses only $O(n + 2^{w/r})$ space. (Note that for every fixed integer r we have $O(rn \log\log n) = O(n \log\log n)$.)

Exercise 4.5 In the description of signature sort, on slide 85, it is claimed that $nq \leq n^2$. (a) Show that this is not always the case. (b) What (minor) change is needed to make the algorithm and its analysis correct?

Solution 4.5 (a) On slide 75 we define $q = \Theta(w/((\log n)^2 \log\log n))$. As w may be arbitrary large with respect to n, so can q, and it is clearly not necessarily true that $nq \leq n^2$. In particular, this is not true if $w = \omega(n(\log n)^2 \log\log n)$.
(b) The recurrence relation of the running time of the algorithm (see slide 80) is $T(n, b) = O(n) + T(n, q \log n) + T(n, b/q)$. The choice $q = \Theta(w/((\log n)^2 \log\log n))$ maximizes q while still making sure that $T(n, q \log n) = O(n)$. However, we there is no reason of choosing a value of q larger than $\log n \log\log n$, as this already ensures that $T(n, b/q) = O(n)$. Thus, we can simply let $q = \min\{w/((\log n)^2 \log\log n), \log n \log\log n\}$. Everything works as before, and now $nq \leq n^2$. (Note that $q = \Theta(w/((\log n)^2 \log\log n))$ remains the choice of q when $w = (\log n)^{2+\varepsilon}$, which is the most 'challenging' value of w.)