**Homework 5**

**C-8.2** Let A be a collection of objects. Describe an efficient method for converting A into a set. That is, remove all duplicates from A. What is the running time of this method?

**Sol.** First we sort the objects of A. Then we can walk through the sorted sequence and remove all duplicates. This takes $O(n \log n)$ time to sort and $n$ time to remove the duplicates. Overall, therefore, this is an $O(n \log n)$-time method.

**C-8.5** Suppose we are given a sequence S of n elements, each of which is colored red or blue. Assuming S is represented as an array, give an in-place method for ordering S so that all the blue elements are listed before all the red elements. Can you extend your approach to three colors?

**Sol.** For the red and blue elements, we can order them by doing the following. Start with a marker at the beginning of the array and one at the end of the array. While the first marker is at a blue element, continue incrementing its index. Likewise, when the second marker is at a red element, continue decrementing its index. When the first marker has reached a red element and the second a blue element, swap the elements. Continue moving the markers and swapping until they meet. At this point, the sequence is ordered. With three colors in the sequence, we can order it by doing the above algorithm twice. In the first run, we will move one color to the front, swapping back elements of the other two colors. Then we can start at the end of the first run and swap the elements of the other two colors in exactly the same way as before. Only this time the first marker will begin where it stopped at the end of the first run. The complexity of the algorithm is $O(n)$, where n is the number of elements.

**C-8.6** Suppose we are given two sequences A and B of n elements, possibly containing duplicates, on which a total order relation is defined. Describe an efficient algorithm for determining if A and B contain the same set of elements (possibly in different orders). What is the running time of this method?

**Sol.** We can use the solution from C-8.2 to obtain the distinct elements from A and B, respectively. This step takes $O(n \log n)$. Then A and B contain the same set of elements if and only if the distinct elements from A and B are identical, assuming they are sorted. The second step takes $O(n)$.

**C-8.11** Let A and B be two sequences of n integers each. Given an integer x, describe an $O(n \log n)$-time algorithm for determining if there is an integer a in A and an integer b in B such that $x = a + b$.

**Sol.**

Solution C8-11 (A,B,x) {
    A = mergesort (A)
    B = mergesort (B)
    i = 0
    j = B.length – 1
    while i < A.length and j >= 0 {
        if (A[i] + B[j] == x) return true
        else if (A[i] + B[j] < x) i+=1
    }
    }
return false;
}

C-8.12 Given a sequence of numbers \((x_1, x_2, \ldots, x_n)\) the mode is the value that appears the most number of times in this sequence. Give an efficient algorithm to compute the mode for a sequence of \(n\) numbers. What is the running time of your method?

Sol. Sort the numbers by non-decreasing values. Next we can scan the sequence to keep track, for each subsequence of numbers that are all the same, how long that subsequence is. In addition, we can store the length and xi-value for the longest sequence we have seen so far. When we complete the scan of the sequence, this xi value is the mode. The running time for this method is dominated by the time to sort the sequence, which can be done in \(O(n \log n)\) time in the worst-case by using the merge-sort algorithm.

C-9.1 Show that any comparison based sorting algorithm can be made to be stable, without affecting the asymptotic running time of this algorithm.

Sol. Change the way elements are compared with each other, by replacing each \(x_i\) with the pair, \((x_i, i)\). Now do all comparisons lexicographically. This approach guarantees that if \(x_i = x_j\), then the comparison will be resolved by using the lexicographic rule that \((x_i, i) < (x_j, j)\), if \(i < j\).

C-9.2 Suppose we are given two sequences A and B of \(n\) elements, possibly containing duplicates, in the range of 1 to 2n. Describe a linear time algorithm for determining if A and B contain the same set of elements (possibly in different orders).

Sol. Since a range is given, we can use the idea of counting sort with two arrays, C and D, of size 2n. Run time of that function will be \(O(n+2n) = O(3n) = O(n)\).

```
Linear_algo (A,B) {
  for ( i=0; i < 2n; i++) { C[i] = D[i] = false; }
  for ( i=0; i < n; i++) { C[A[i]] = true; D[B[i]] = true; }
  for ( i=0; i < 2n; i++) if (C[i] ≠ D[i]) return false;
  return true;
}
```

C-9.3 Suppose we are given a sequence S of \(n\) elements, each of which is an integer in the range \([0, n^2 - 1]\). Describe a simple method for sorting S in \(O(n)\) time.

Sol. To sort S, do a radix sort on the \(n\) elements, viewing them as pairs \((i, j)\) such that \(i\) and \(j\) are integers in the range \([0, n - 1]\). The details are given in the sample solution of midterm 1 of the morning session.
C-9.10 Suppose you are given two sorted lists, A and B, of n elements each, all of which are distinct. Describe a method that runs in $O(\log n)$ time for finding the median in the set defined by the union of A and B.

**Sol.** Given two sorted lists, A and B, the basic idea of finding the median of the combination of the two lists in $O(\log n)$ time is as follows. First, find the medians of A and B, say $m_A$ and $m_B$, if they are equal then we have found the median. Otherwise, if $m_A < m_B$ then the median is either in the latter half of A or in the first half of B. If $m_A > m_B$ then the converse is true, the median is either in the first half of A or the latter half of B. This process is repeated until either the median is found or both A and B have size two or one. If both lists have size two or one, the median is given as the max of the two first elements. Since the process can be repeated at most $O(\log n)$ times and each step takes constant time, the complexity is $O(\log n)$.

```plaintext
FindMedian(A, B) {
    findMed(A, 0, n-1, B, 0, n-1);
}

findMed(A, la, ha, B, lb, hb) {
    // look for median in A[la..ha] and B[lb..hb], where A and B are sorted and ha-la = hb-lb.
    if (ha-la <= 1) return max(A[la], B[lb]);
    ma = (ha+la+1)/2;
    mb = (hb+lb+1)/2;
    if (A[ma] == B[mb]) return A[ma];
    pack = (ha-la)%2 // if (ha-la)%2 = 1, number of elements in A[la..ha] is even
    // pack is used to make sure ha-la = hb-lb in recursive calls.
    if (A[ma] < B[mb]) {
        return findMed(A, ma, ha, B, lb, mb - pack);
    } else {
        return findMed(A, ml, ma - pack, B, mb, hb);
    }
}
```

The recursive calls are tail recursion and can be removed.

C-9.11 Given a set of n elements that come from a total order, show that you can find the second smallest element in this set using $n + \log \ceil(n) - 2$ comparison

**Sol.**
Employing binary tree can make this happen. The algorithm named by \textit{findSecondSmallest} can be described as follows.

\begin{algorithm}
\textbf{findSecondSmallest}(A)
\begin{algorithmic}[1]
\State Initialize a perfect binary tree with height $\lceil \log n \rceil$
\State Store all elements in the array to the keys at the bottom level (with height 0) of the tree
\State $h \leftarrow 0$
\While {$h < \lceil \log n \rceil$} \Do
\For {All elements at height $h$} \Do
\State Compare each two elements who have the same parent, and store the smaller value as the key of their common parent
\EndFor
\State $h \leftarrow h + 1$
\EndWhile
\State $B \leftarrow \emptyset$
\While {(a.leftChild! = null) AND (a.rightChild! = null)} \Do
\If {(a.leftChild.key $==$ root.key)} \Then
\State $B \leftarrow B \cup \{a.rightChild.key\}$
\State $a \leftarrow a.leftChild$
\Else
\State $B \leftarrow B \cup \{a.leftChild.key\}$
\State $a \leftarrow a.rightChild$
\EndIf
\EndWhile
\State \Return {\textit{findMin}(B)}
\end{algorithmic}
\end{algorithm}

Finding the smallest element needs $n - 1$ comparisons, and finding the second smallest element needs $\lceil \log n \rceil - 1$ comparisons. Hence, totally the above algorithm needs $n + \lceil \log n \rceil - 2$ comparisons.