Purdue University
CS 381

S. Hambrusch - C. Guerra
Fall 1999

Solution Sketches to Assignment 5

1) (Graded by Tian Luan) Assume you are given a sequence of n elements $a_1, a_2, \cdots, a_n$ to be sorted. The sequence has the property that the location of element a_i in the sorted sequence is at most d positions away from position i; i.e., if a_i ends up at position j, then $|i - j| \leq d$. Describe and analyze an algorithm to sort the sequence $a_1, a_2, \cdots, a_n$ in $O(n \log d)$ time when you know the value of d.

Assume we are to generate a non-decreasing sequence. To achieve $O(n \log d)$ time, create a min-heap using the first d elements in array A. The minimum element of this heap of height $\log d$ is the smallest element in A. We extract the minimum and then insert the next element of A not yet in the heap. This process continues: extract-min, followed by an insertion. Once all elements of A have been inserted into the heap, we perform extract-min operations until the heap is empty. The running time is $O(n \log d)$ since every element is inserted and deleted from a heap containing at most d elements.

Another $O(n \log d)$ solution is to partition the array into $\lceil n/d \rceil$ pieces, sort each piece in $O(d \log d)$ time, and perform a 2-step merge as follows: merge every even numbered piece with its left neighbor and then merge with its right neighbor.

2) (Graded by Tian Luan) Consider the following modification to the base case of *Mergesort* when sorting an array A of size n. Given is a parameter k, $1 \leq k \leq n$; when the array to be sorted has size less than or equal to k, insertion sort is invoked. This corresponds to ending the recursion of Mergesort when the array size is $\leq k$.

(a) What is the worst-case time complexity of this sorting algorithm?

(b) What is the largest value of k as a function of n that gives the same time asymptotic complexity as the standard mergesort?

Assume n and k powers of 2. The recursion of mergesort ends after $\log n - \log k = \log \frac{n}{k}$ steps, thus the time complexity of the mergesort is $O(n \log \frac{n}{k})$. (Consider the recursion tree; there are $\log \frac{n}{k}$ levels in the tree and the row sum is n at each level). Then insertion sort is invoked. There are $\frac{n}{k}$ sub-arrays of size k to be sorted by

insertion sort. The worst-case time complexity of insertion sort for each such sub-array is $O(k^2)$ for a total of $O(k^2 \times \frac{n}{k}) = O(kn)$ time. In conclusion, the overall worst-case time complexity of the proposed modified mergesort algorithm is $O(n \log \frac{n}{k} + kn)$.

The algorithm has same time asymptotic complexity as the standard mergesort for $k = O(\log n)$.

3) (Graded by Biana Babinsky) (i) Consider a max-heap of size n. Describe an efficient algorithm for finding the 4-th largest element.

The 4-th largest element can be found on $O(1)$ time. It cannot be the element at the root, but it can be any one of the next 14 locations in the array. Copy these elements into an array of size 14. To find the 4-th largest element we either sort these 14 elements, or we create a heap of size 14 and make 4 extract-min operations, or we simple scan the array 4 times. Any one of these actions costs $O(1)$ time.

(ii) Let S be a set of n elements which contains only 8 distinct elements. You do not know these 8 elements. Describe and analyze an efficient algorithm to sort set S under this assumption.

We can sort the n elements in $O(n)$ time as follows. Note first that we do **not** know the keys of the 8 elements (as some of you assumed in their solution). The first step is to create eight buckets and to identify the eight distinct keys. Use an array B of size 8. Array B contains two fields: one will hold a key, the other a pointer to a list containing the entries having this key.

Scan the input list. For every entry, scan array B until either an entry with the same key is found or the end of the current keys is reached. In the latter case we create a new key. After all n elements have been put into one of the lists in array B, we sort the eight keys in array B. This costs $O(1)$ time. Finally, we output the sorted list of size n. The total time used is $O(n)$.