



## CPE 11040830: Algorithms, Homework #3 Solution

**Dr. Khalil Ahmad Yousef**

(1) CLR 6.1-6

No,  $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$  is not a heap because the heap property does not hold between the 4th element and its second child, the 9th element (i.e.,  $6 < 7$ ).

(2) CLR 6.5-8

Without loss of generality, assume that the  $k$  lists are sorted in descending order. Assume that the function  $\text{GET-NEXT}(i)$  returns the next element from the sorted list  $i$ , initially starting with the first element. If list  $i$  is empty it returns  $NULL$ .

When merging several sorted lists, the basic operation to be performed at each step is to get the maximum of the current front-most elements of the  $k$  lists and output it. Then consider the next element in the list (if it exists) from which the maximum was output. We can perform this operation efficiently using a heap. We maintain a heap of the current largest elements from the  $k$  lists. We also store its list number along with its value. So each element in the heap is represented by an ordered pair  $\langle x, y \rangle$ , where  $x$  is the value and  $y$  is the list number.

The algorithm is a straightforward implementation of the description above. We store the heap in the array  $A$ , and we store the output array in  $B$ . Lines 1-3 perform the initialization of the heap. The *for* loop in line 4 extracts the maximum from the heap and inserts the next element from the list where the maximum came from.

### MERGE-K-LISTS

```
1. for  $i \leftarrow 1$  to  $k$ 
2.   do  $A[i] \leftarrow \langle \text{GET-NEXT}(i), i \rangle$ 
3. BUILD-HEAP( $A$ )
4. for  $i \leftarrow n$  downto 1
5.   do  $\langle x, y \rangle \leftarrow \text{HEAP-EXTRACT-MAX}(A)$ 
6.      $B[i] \leftarrow x$ 
7.      $z \leftarrow \text{GET-NEXT}(y)$ 
8.     if  $z \neq NULL$ 
9.       then  $\text{HEAP-INSERT}(A, \langle z, y \rangle)$ 
10.    else  $\text{heap-size}(A) \leftarrow \text{heap-size}(A) - 1$ 
```



The first loop in lines 1-2 clearly takes  $O(k)$  time. The call to BUILD-HEAP in line 3 also takes  $O(k)$  time. The call to HEAP-EXTRACT-MAX is  $O(\lg k)$ , as is the call to HEAP-INSERT. The rest of the statements in the *for* loop are all  $\Theta(1)$ . Since the loop is repeated  $n$  times, the entire loop is  $O(n \lg k)$ , and thus the entire algorithm is  $O(n \lg k)$ .

### (3) CLR 6-2

- (a) A  $d$ -ary heap can be represented in a 1-dimensional array as follows. The root is kept in  $A[1]$ , its  $d$  children are kept in order in  $A[2]$  through  $A[d + 1]$ , their children are kept in order in  $A[d + 2]$  through  $A[d^2 + d + 1]$ , and so on. The two procedures that map a node with index  $i$  to its parent and to its  $j^{\text{th}}$  child (for  $1 \leq j \leq d$ ), respectively, are:

D-ARY-PARENT( $i$ )

**return**  $\lfloor \frac{i-2}{d} + 1 \rfloor$

D-ARY-CHILD( $i, j$ )

**return**  $d(i-1) + j + 1$

To convince yourself that these procedures really work, verify that D-ARY-PARENT(D-ARY-CHILD( $i, j$ )) =  $i$ , for any  $1 \leq j \leq d$ . Notice that the binary heap procedures are a special case of the above procedures when  $d = 2$ .

- (b) Since each node has  $d$  children, the height of a  $d$ -ary heap with  $n$  nodes is  $\Theta(\log_d n) = \Theta(\frac{\lg n}{\lg d})$ .
- (c) The procedure HEAP-EXTRACT-MAX given in the text for binary heaps works fine for  $d$ -ary heaps too. The change needed to support  $d$ -ary heaps is in MAX-HEAPIFY, which must compare the argument node to all  $d$  children instead of 2 children. The running time of HEAP-EXTRACT-MAX is still the running time of MAX-HEAPIFY, but that now takes worst-case time proportional to the product of the height of the heap by the number of children examined at each node (at most  $d$ ), namely  $\Theta(d \log_d n) = \Theta(d \frac{\lg n}{\lg d})$ .
- (d) The procedure MAX-HEAP-INSERT given in the text for binary heaps works fine for  $d$ -ary heaps too. The worst-case running time is still  $\Theta(h)$ , where  $h$  is the height of the heap. (Since only parent pointers are followed, the number of children a node has is irrelevant.) For a  $d$ -ary heap, this is  $\Theta(\log_d n) = \Theta(\frac{\lg n}{\lg d})$ .
- (e) INCREASE-KEY can be implemented in almost the same way as HEAP-INCREASE-KEY on page 140 of the book (need to modify for the max test).

HEAP-INCREASE-KEY( $A, i, key$ )

1.  $A[i] \leftarrow \max(A[i], key)$

2. **while**  $i > 1$  **and**  $A[PARENT(i)] < A[i]$

3.     **do exchange**  $A[i] \leftrightarrow A[PARENT(i)]$

4.      $i \leftarrow PARENT(i)$

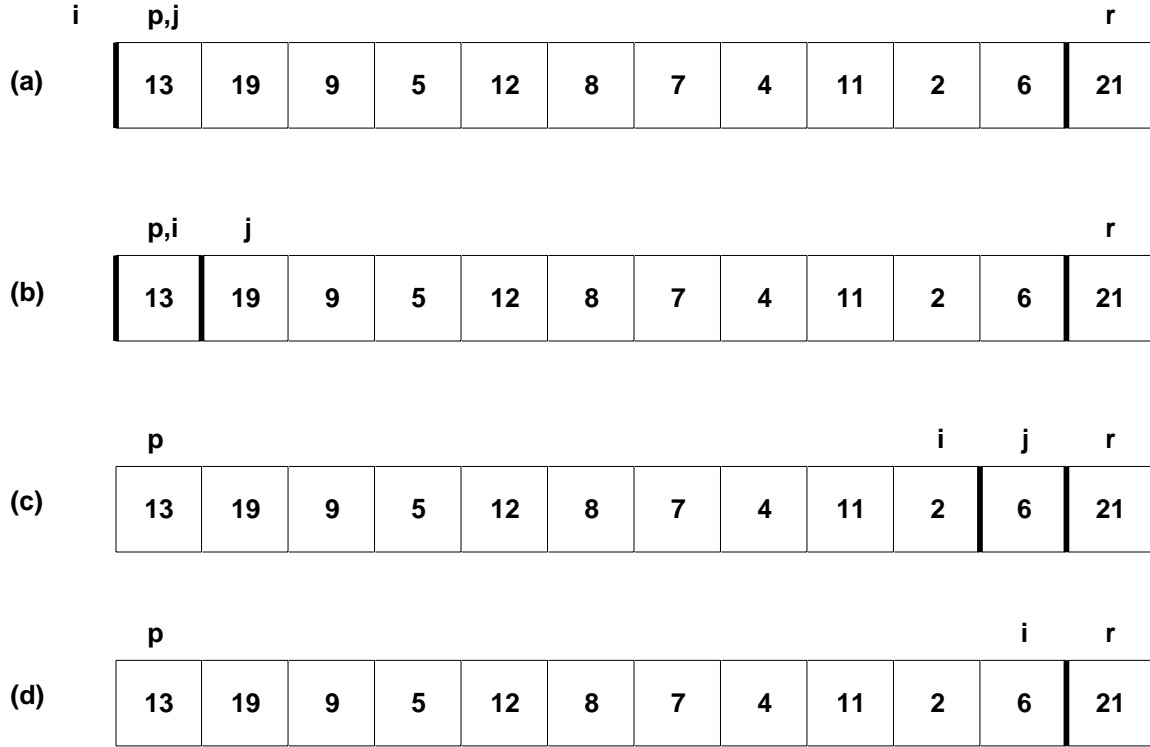


Figure 1: The operation of PARTITION on A.

In the worst case, the entire height of the tree must be traversed, so the worst-case running time for a  $d$ -ary heap is  $\Theta(h) = \Theta(\log_d n) = \Theta(\frac{\lg n}{\lg d})$ .

#### (4) CLR 7.1-1

See Figure 1

(a) The initial array and position of  $i$  and  $j$  indices at the beginning of the first iteration of the for loop. Because  $A[j] \leq x$ ,  $i$  is incremented by one to  $p$  and  $A[p]$  is exchanged with itself (nothing changes).

(b) At the start of the second iteration in the for loop, we show the placement of the  $i$  and  $j$  indices. This iteration is much like the first, as are all the iterations up to the last one. (c)

At the start of the last iteration in the loop,  $i$  is at position  $r - 2$  and  $j$  is at position  $r - 1$ . Because  $A[j] \leq x$ ,  $i$  is incremented by one to  $r - 1$  and  $A[r - 1]$  is exchanged with itself.

(d)  $i + 1 = r$  is returned at line 8 after  $A[r]$  is exchanged with itself.

#### (5) CLR 7.1-4

To make QUICKSORT to return  $A$  in non-increasing order, we simply replace PARTITION with PARTITION-NONINCREASING in the below.



PARTITION-NONINCREASING( $A, p, r$ )

```
1   $x \leftarrow A[r]$ 
2   $i \leftarrow p - 1$ 
3  for  $j \leftarrow p$  to  $r - 1$ 
4      do if  $A[j] \geq x$ 
5          then  $i \leftarrow i + 1$ 
6          exchange  $A[i] \leftrightarrow A[j]$ 
7  exchange  $A[i + 1] \leftrightarrow A[r]$ 
8  return  $i + 1$ 
```

(6) CLR 7.2-2

It is a special case of an array which is sorted. According to the partition procedure, at each recursive level, the pivot is always the same as all the other elements. The first partition with values no greater than the pivot will always have  $r - p$  elements, which is less than the number of total elements in the subarray by one. So the worst-case unbalanced partitioning will always happen with recursive running time  $T(n) = T(n - 1) + \theta(n)$ . The running time is therefore  $\theta(n^2)$ .

(7) CLR 7.2-3

Analyze QUICKSORT when the array  $A$  is in decreasing order. The initial call to the PARTITION( $A, p, r$ ) sets the  $q \leftarrow p$  which gives the worst case partition of 0 and  $r - p$ . On the next recursion of the QUICKSORT, the largest element is at the last cell of the subarray. PARTITION( $A, p, r$ ) sets  $q = r$  which also gives a worst case partition. Now we have another subarray with decreasing order elements. Hence, QUICKSORT repeats steps above until exhausting the elements of  $A$ . The corresponding recursion is:

$$T(n) = T(n - 1) + \Theta(n) = \Theta(n^2)$$

(8) CLR 7.4-4



Since the running time of RANDOMIZED-QUICKSORT is bounded by  $n$  calls to PARTITION and  $X$  comparisons within all the calls to PARTITION, we will show that the expected running time of RANDOMIZED-QUICKSORT is  $\Omega(n \lg n)$  by demonstrating that  $E[X] = \Omega(n \lg n)$ .

To do the analysis, we rename the elements of  $A$  as  $z_1, z_2, \dots, z_n$  with  $z_i$  being the  $i$ th ranked element, as in the upper bound analysis of the expected running time. Our analysis then uses the indicator random variable,  $X_{ij} = I\{z_i \text{ is compared to } z_j\}$  to calculate a lower bound on the number of comparisons:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}$$

Taking the expectation of  $X$  and using linearity and Lemma 5.1, we obtain:



$$\begin{aligned}
E[X] &= E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n E[X_{ij}] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \Pr\{z_i \text{ is compared to } z_j\} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} \\
&\geq \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{1}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{1}{j-i+1} \quad \triangleright \text{Let } k = j - i + 1 \text{ with } k \text{ ranging from } 2 \text{ to } n - i + 1 \\
&= \sum_{i=1}^{n-1} \left( \sum_{k=1}^{n-i+1} \frac{1}{k} - 1 \right) \\
&\geq \sum_{i=1}^{n-1} \left( \sum_{k=1}^{n-i} \frac{1}{k} - 1 \right) \\
&= \sum_{i=1}^{n-1} (\Omega(\lg(n-i)) - 1) \quad \triangleright \text{By A.2-3} \\
&= \sum_{i=1}^{n-1} \Omega(\lg(n-i)) - (n-1) \\
&\geq \sum_{i=1}^{n-1} \Omega(\lg(n-i)) - n \\
&= \Omega\left(\sum_{i=1}^{n-1} \lg(n-i)\right) - n \\
&= \Omega\left(\sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \lg(n-i) + \sum_{i=\lfloor \frac{n}{2} \rfloor + 1}^{n-1} \lg(n-i)\right) - n \\
&\geq \Omega\left(\sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \lg\left(n - \frac{n}{2}\right) + \sum_{i=\lfloor \frac{n}{2} \rfloor + 1}^{n-1} \lg(n - (n-1))\right) - n \\
&= \Omega\left(\sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \lg\left(\frac{n}{2}\right)\right) - n \\
&= \Omega\left(\sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} (\lg n - \lg 2)\right) - n \\
&= \Omega\left(\lfloor \frac{n}{2} \rfloor \lg n - \lfloor \frac{n}{2} \rfloor\right) - n \\
&= \Omega\left(\lfloor \frac{n}{2} \rfloor (\lg n - 1)\right) - n \\
&= \Omega(n \lg n)
\end{aligned}$$



Hence, we can conclude the expected running time is  $\Omega(n \lg n)$ .

(9) CLR 7.4-5

Assuming that the input is nearly sorted (locally sorted), i.e., neighboring elements are sorted, although not sorted globally.

So we can expect average running time of Quicksort. Since we always stop to partition further more when the number of elements in a subarray is less than  $k$ . The number of recursion levels should be  $O(\lg(n/k))$  (at each partition level, the number of partition is  $2^L = n/k$ ). the running time of the Quicksort is  $O(n \cdot \lg(n/k))$ .

We can regard the  $k$ -elements subarrays as macro elements, after Quick sort. They are sorted, i.e., every element in the macro element with a larger index is larger than every element in the macro element with a smaller index. When Insertion Sort is called in its  $N$  iterations, only  $O(k)$  comparisons are needed (only the  $k$  elements in the current macro element need to be compared). So insertion sort running time is  $O(nk)$ .

By combining these two results, the overall running time is  $O(nk + n \cdot \lg(n/k))$ .

Theoretically,  $k$  is 1 in order to minimize the the function  $f(k) = nk + n \lg n/k$ , and hence the running time.

In practice,