600.363/463 Solution to Homework 2
Due: Wednesday, 9/24

6.4-3

We want the output of heap sort to be in increasing order. Even if the input array A is sorted in increasing order, each call to MAX_HEAPIFY$(A, 1)$ in the procedure HEAPSORT(A) still takes $O(\log n)$ time. So the running time of heap sort is dominated by this and hence is $O(n \log n)$.

The cost of MAX_HEAPIFY does not change for the case where the input is sorted in decreasing order and heap sort still runs in $O(n \log n)$ time.

6-2 *Analysis of d-ary heaps*

1. A d-ary heap can be represented in a 1-dimensional array as follows. The root is keep 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 jth child (for $1 \leq j \leq d$), respectively, are:

   D-ARY-PARENT(i)
   return $\lfloor (i-2)/d + 1 \rfloor$

   D-ARY-CHILD(i, j)
   return $d(i-1) + j + 1$

   (To convenience 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 eap procedures are a special case of the above procedures when d=2.)

2. Since each node has d children, the height of a d-ary heap with n nodes is $\Theta(\log_d n)$=$\Theta(\log n / \lg d)$.

3. 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 HEAPIFY, which must compare the argument node to all d children instead of just 2 children. The running time of HEAP-EXTRACT-MAX is still the running time for 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 $\Theta(d \log_d n) = \Theta(d \lg n / \lg d)$.

4. The procedure HEAP-INSERT given in the text for binary heaps works find 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, thenumber of children a node has is irrelevant.) For a d-ary heap this is $\Theta(\log_d n) = \Theta(\lg n/\lg d)$.

5. HEAP-INCREASE-KEY can be implemented as a slight modification of HEAP-INSERT (only the first two lines are different). Increasing an element may make it larger than its paren, in which case it must be moved higher up in the tree. This can be done just as for insertion, traversing a path from the increased node toward the root. In the worst case, the entire height of the tree must be traversed, so the worst-case running time is $\Theta(h) = \Theta log_d n) = \Theta(\lg n/\lg d)$.

```
HEAP-INCREASE-KEY(A, i, k)
    if A[i] ≥ k
        then return
    while i > 1 and A[PARENT(i)] < k
        do A[i] ← A[PARENT(i)]
            i ← PARENT(i)
    A[i] ← key
```

7.2-3

If the input array is sorted in decreasing order, each call to PARTITION divides the input array in to sub-arrays of size 0 and $n-1$. So the recurrence relation for the runtime of quick sort would be $T(n) = T(n-1) + O(n)$, the $O(n)$ factor comes because of the time for PARTITION. The solution to the above recurrence relation is $T(n) = O(n^2)$ which can be obtained by using the iteration method or substitution method.

7.4-2

Let $T(n)$ be the best-case time for Quicksort on on input of size n. We have the recurrence:

$$T(n) = \min_{1 \le q \le n-1}(T(q) + T(n-q)) + \Theta(n).$$

We argue that $T(n) \ge cn\lg n$ for some constant c. Substituting this guess into the recurrence. We obtain:

$$\begin{aligned} T(n) &\ge \min_{1 \le q \le n-1}(cq\lg q + c(n-q)\lg(n-q)) + \Theta(n) \\ &= c\min_{1 \le q \le n-1}(q\lg q + (n-q)\lg(n-q)) + \Theta(n) \end{aligned}$$

The expresson $q\lg q + (n-q)\lg(n-q)$ achieves a minimum over the range of $1 \le= q \le= n-1$ at the midpoint, $q = n/2$, since the first derivative of the expression with respect to q is 0 when $q = n/2$ and second derivative of the expression is positive. This gives us the bound:

$\min_{1 \le q \le n-1}(q\lg q + (n-q)\lg(n-q)) \ge n/2\lg(n/2) + (n-n/2)\lg(n-n/2) = n\lg(n/2)$

Continuing with our bounding of $T(n)$, we obtain

$T(n) \ge cn\lg(n/2) + \Theta(n) = cn\lg n - cn + \Theta(n) \ge cn\lg n$, since we can pick

the constant c small enough so that cn is dominated by the $\Theta(n)$ term. Thus the best runnig time of quicksort is $\Omega(n\lg n)$.

7-4 Stack depth for quicksort

a. QUICKSORT' does exactly what QUICKSORT does, hense it sorts correctly. QUICKSORT and QUICKSORT' do the same partitioning, and then each calls itself with arguments A, p, q. QUICKSORT then calls itself again, with arguments A, $q+1$, r. QUICKSORT' instead sets $p \leftarrow q+1$ and reexecutes itself. This executes the same operations as calling itself with A, $q+1$, r, because in both cases the first and third arguments (A, r) have the same values as before and p has the old value of $q+1$.

b. The stack depth of QUICKSORT' will be $\Theta(n)$ on an n-element input array if there are $\Theta(n)$ recursive calls to QUICKSORT'. This happens if every call to PARTITION(A, p, r) returns $q = r - 1$. The sequence of recursive calls in this scenario is: QUICKSORT'$(A, 1, n)$,QUICKSORT'$(A, 1, n-1)$, QUICKSORT'$(A, 1, n-2)$, ..., QUICKSORT'$(A, 1, 1)$.

If you want to, you can construct a specific array that causes this behavior: The PARTITION shown in the book will behave this way, for example, on the array $(n, 1, 2, ...n-1)$, which it partitions into $(n-1, 1, 2, ...n-2)$ and (n). Each time PARTITION is given an array with the largest element at the front and the rest of the elements sorted in increasing order, the high side of the resulting partition has just the largest element and the low side has the rest of the elements, again with the largest at the front and the rest increasing order.

b. The problem demonstrated by the scenario in (b) is that each invocating of QUICKSORT' calls QUICKSORT' again with almost the same range. To avoid such behavior, we must change QUICKSORT' so that the recursive call is on a smaller interval of the array. The following variation of QUICKSORT' checks which of the two subarrays returned from PARTITION is smaller and recurses on the smaller subarray, which is at most half the size of the current array. Since the array size is reduced by at least half on each revursive call, the number of recursive calls, and hence the stack depth, is $\Theta(\lg n)$ in the worst case. Note that this works no matter what PARTITION algorithm we use.

```
    QUICKSORT"(A, p, r)
1   while p < r
2       do ▷ Partition and sort the small subarray first
3          q ← PARTITION(A, p, r)
4          if q − p + 1 < r − q
5             then QUICKSORT"(A, p, q)
6                  p ← q + 1
7             else QUICKSORT"(A, q + 1, r)
8                  r ← q
```

The expected running time is not affected, because exactly the same work is done as before: The same partitions are produced, and the same subarrays are sorted.