Assignment 2 4041 Algorithms and Data Structures
DUE: Thursday, Sept 29th, 10:00 pm Fall 2016

Wyatt Kormick

1. Answer the following questions about the properties of heaps and priority queues.
   a. What is the run time of HEAPSORT on an array in sorted order? Briefly justify.
      Heapsort's run time is O(n log n). It doesn't matter if the array was already sorted, because the algorithm will take the time to mix it up by turning it into a max heap before sorting it.

   b. Increase-Key defined in 6.5 has an input parameter of the index of the element being increased. Increase-Key is O(lg n). There is no indication of how that index is known. What is the run time for finding the element with key *key* in a given heap? Briefly justify.
      The runtime for finding the index element key in a heap would be O(lg n) as it involves tracing down one series of branches down to the index rather than searching the entire array.

   c. Why set the key of the inserted node to -infinity in line 2 of Max-Heap-Insert when the intent is to insert this node into the heap with the key value *key*?
      Max-Heap-Insert has to expand the heap to be able to include the new key before calling Heap-Increase-Key on the new leaf. This also allows for the new element to be moved to its proper position as its current spot has a lower priority than the spot that it probably should be in.

2. Prove that an n-element heap has height floor(lg n).

H(0) = 2^0 leaves
H(1) = 2^1 leaves
H(h) = 2^h = n leaves
h = lg n
h is height, n is the number of leaves at most
If there are less than n for height h the height is the same and must be an integer, hence the floor(lg n)

There are at most 2^h leaves after h splits in the tree

3. Give an O(nlgk)-time algorithm to merge k sorted lists into one monotonically decreasing list, where n is the total number of elements. Justify the runtime. Assume the lists are sorted in decreasing order. For any algorithm from the textbook, you can use the known bounds without justification.

```
LIST-MERGE(K) //Where K is a list containing k lists
    Let M be a new array
    for i = 1 to k       //k
        HEAP-INSERT(M, K[i]) //k*n
    MIN-HEAPSORT(M) //k*n*log k

Min-HEAPSORT(H) //O(n log n)
    BUILD-MIN-HEAP(H)
    for i = H.length downto 2
        Swap(H[1], H[i])
        H.heap-size = H.heap-size-1
        MIN-HEAPIFY(H, 1)

BUILD-MIN-HEAP(K]) //Like BUILD-MAX-HEAP for a MIN HEAP O(n log n)
MIN-HEAPIFY(H, 1) //Like MAX-HEAPIFY for a min heap O(log n)

HEAP-INSERT(H, x) //O(n)
    for i = 1 to H.length
        H.heap-size = H.heap-size+1
        H[H.heap-size] = x[i]
```

O(k + kn + knlogk) = O(cn log k)

4. Use indicator random variables to compute the expected value of the sum of n dice.

    X_{1i}: I{Rolled 1 on die i} … X_{6i}: I{Rolled 6 on die i}

    E(X)=$\sum_{i=1}^{n}((1/6)*1+(1/6)*2+(1/6)*3+(1/6)*4+(1/6)*5+(1/6)*6)=3.5n$

5. QuickSort run time.
    a. Explain why the running time of QuickSort is O(n^2) when the array A contains distinct elements sorted in decreasing order.

        Quicksort always makes the last element in the array the pivot location. This means that on one side of the pivot are numbers larger than it, and on the other, smaller. When the array is in sorted order, no numbers are going to be placed on one side of the pivot, so the next partition call still works on the entire list minus the pivot, effectively making it an O(n^2) operation to divide and place the objects in the array. With an even split each partition will work on a smaller part of the array making it an O(n lg n) operation

    b. Explain why, even if given the same array A of distinct elements sorted in decreasing order, the running time of Randomized-Quicksort is typically defined as O(nlgn).

        The order of the original array doesn't matter in Randomized Quicksort because the pivots are chosen at random so it is equivalent to Quicksort on a randomized array. The problem with sorted arrays in Quicksort was that the pivot chosen was always the worst one because it didn't choose one at random.

6. Given a sorted array of 100 scores, the quartiles are at index 25, 50, and 75, which divide the data into even rankings each of which are the bottom, 2 middle, and top 25% of all scores. Write an algorithm with expected running time of O(n) that, given an array A of distinct scores NOT sorted, returns an array in which elements A[1] to A[n/4] are the bottom 25%, A[3/4n+1] to A[n] are the top 25%, etc., but the scores are not necessarily sorted within their respective ranking group. HINT: Consider QuickSort and see Randomized-Select(A,p,r,i) in Chapter 9, which can find the ith order statistic (i.e. the ith-smallest element) in the subarray A[p] to A[r]. Briefly justify the runtime.