Name: _____

1. (3 points) What is the running time of QUICKSORT in each of the following cases?

(a) Average Case	$\Theta(n \lg n)$
(b) Input A contains distinct elements sorted in decreasing order	$\Theta(n^2)$
(c) All the elements of input A have the same value	$\Theta(n^2)$

2. (1 point) Justify your answer to (1.c).

Solution

Since the elements are identical, the pivot, $A[r]$, is equal to every other element of A. Therefore, the conditional on line 4 of PARTITION is always true and i will be incremented with every iteration of the loop, resulting in $i == j$ at every iteration, and so the exchange on line 6 does nothing. The loop terminates with $i = r - 1$, so the final exchange on line 7 also does nothing, leaving the pivot in its original position, and returning r to QUICKSORT as the "midpoint" of the array. QUICKSORT will therefore make a *single* recursive call with start and end indices p and $r - 1$; that is, the recursive call will be one element smaller, which is maximally imbalanced. We have shown that QUICKSORT with maximally imbalanced partitions has a quadratic running time.

(continued on other side)

3. (3 points) It is important that the main loop of BUILD-MAX-HEAP go from $\lfloor n/2 \rfloor$ *down* to 1 rather than from 1 up to $\lfloor n/2 \rfloor$.

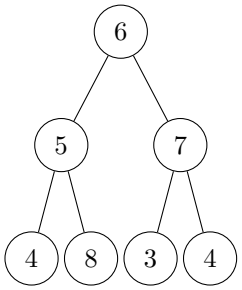
(a) Explain why this is important for the operation of BUILD-MAX-HEAP.

(b) Construct a small example array A for which looping from 1 to $\lfloor n/2 \rfloor$ will produce incorrect results.

Solution

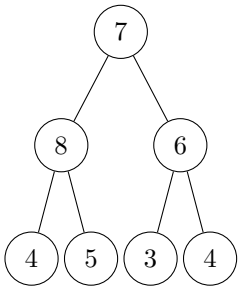
A call to MAX-HEAPIFY(A, i) is only guaranteed to leave the subtree rooted at i a max-heap if the subtrees rooted at LEFT(i) and RIGHT(i) are already max-heaps. By looping from $\lfloor n/2 \rfloor$ down to 1, BUILD-MAX-HEAP guarantees that this condition holds for each call to MAX-HEAPIFY.

Looping from 1 to $\lfloor n/2 \rfloor$ will not give the correct answer for the following heap:



1. Call to MAX-HEAPIFY$(A, 1)$ will swap nodes 1 and 3 (values 6 and 7), leaving the value 7 in node 1
2. Call to MAX-HEAPIFY$(A, 2)$ will swap nodes 2 and 5 (values 5 and 8), leaving the value 8 in node 2
3. Call to MAX-HEAPIFY$(A, 3)$ will do nothing

This leaves the heap in the following state, which is not a max-heap:



4. (3 points) You are given an array A of positive integers, where different integers may have different numbers of digits, but the total number of digits over *all* the integers is n. Show that the following algorithm sorts the array in $O(n)$ time:

1. Compute the number of digits in each integer.
2. Sort the array by number of digits using Counting-Sort, grouping the integers by digit length.
3. Sort each group using Radix-sort.

Solution

First, we need to compute the number of digits in each number. This can be done by, for example, looping over the list of numbers, and for each number, performing integer division by 10 until we reach zero (you can think of this as digit-wise right shift for each number). This requires one division by 10 for each digit of each number, or one division per digit, so this step can be completed in $O(n)$ time.

Now we must sort the numbers into groups by number of digits, which can be done using Counting-Sort in $O(m+n)$ time, where m is the number of integers in the array, and n is an upper bound on the number of digits in any one integer (n plays the role of k in the pseudocode for Counting-Sort presented in class). Since every integer must have at least one digit, $m = O(n)$, and $O(m+n) = O(n)$.

Now, the possible digits length of any one integer is between 1 and n. Suppose there are m_i digits of length i for $1 \leq i \leq n$. Note that $\sum_{i=1}^{n} i \cdot m_i = n$. It is possible that some of the m_i are zero. The time to sort m_i integers of length i using Radix-Sort is $\Theta(i \cdot m_i)$ and so the time to sort all the integers, grouped by digit length, is

$$\sum_{i=1}^{n} \Theta(i \cdot m_i) = \Theta\left(\sum_{i=1}^{n} i \cdot m_i\right) = \Theta(n).$$

Since all of the steps are $O(n)$, their sum is $O(n)$ and we conclude that the sort can be completed in linear time.

(pseudocode on other side)

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)

PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          Exchange A[i] and A[j]
7  Exchange A[i + 1] with A[r]
8  return i + 1

MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap_size and A[l] > A[i]
 4      largest = l
 5  else largest = i
 6  if r ≤ A.heap_size and A[r] > A[largest]
 7      largest = r
 8  if largest ≠ i
 9      Exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)

BUILD-MAX-HEAP(A)
1  A.heap_size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)

COUNTING-SORT(A, B, k)
 1  Let C[0 . . k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  for i = 1 to k
 7      C[i] = C[i] + C[i − 1]
 8  for j = A.length downto 1
 9      B[C[A[j]]] = A[j]
10      C[A[j]] = C[A[j]] − 1

RADIX-SORT(A, d)
1  for i = 1 to d
2      Sort A on digit i using a stable sort
```