Answer the questions in the spaces provided on the exam. If you run out of room for an answer, continue on the back of the page.
Note that you must justify all answers!

Name: _____

1. You have 140 minutes to complete the exam.
2. The exam has 5 questions, printed on 5 pages (plus one page for grading). You may use the back of the page if you run out of room for an answer.
3. I recommend you use pencil on the exam, but blue or black ink are also acceptable.
4. You may NOT use calculators, cell phones or other electronic devices while taking the exam.
5. The exam is closed book. You may use one page of prepared notes (double-sided).

1. Suppose the functions $f_1(n)$, $f_2(n)$, ..., and $g_1(n)$, $g_2(n)$, ..., are such that $f_1(n) = O(g_1(n))$, $f_2(n) = O(g_2(n))$, etc.

   (a) [10 points] Use the formal definition of Big-Oh to prove $f_1(n)f_2(n) = O(g_1(n)g_2(n))$.

   Solution:

   Proof. Since $f_1(n) = O(g_1(n))$, there are positive constants c_1 and n_1 such that $f_1(n) \leq c_1 g_1(n)$ for all $n \geq n_1$, and since $f_2(n) = O(g_2(n))$, there are positive constants c_2 and n_2 such that $f_2(n) \leq c_2 g_2(n)$ for all $n \geq n_2$. Thus, for all $n \geq \max\{n_1, n_2\}$, both inequalities will hold, so $f_1(n)f_2(n) \leq c_1 c_2 g_1(n)g_2(n)$ for all $n \geq \max\{n_1, n_2\}$. Hence, there exist positive constants $c_3 = c_1 c_2$ and $n_3 = \max\{c_1, c_2\}$ such that $f_1(n)f_2(n) \leq c_3 g_1(n)g_2(n)$ for all $n \geq n_3$, so $f_1(n)f_2(n) = O(g_1(n)g_2(n))$ by the formal definition of Big-Oh. □

   (b) [10 points] Prove that if the product of $f_1(n)f_2(n)\cdots f_k(n) = O(g_1(n)g_2(n)\cdots g_k(n))$ for some $x \geq 2$, then $f_1(n)f_2(n)\cdots f_k(n)f_{k+1}(n) = O(g_1(n)g_2(n)\cdots g_k(n)g_{k+1}(n))$. In product notation, prove that if $\prod_{i=1}^{k} f_i(n) = O\left(\prod_{i=1}^{k} g_i(n)\right)$ for some $k \geq 2$, $\prod_{i=1}^{k+1} f_i(n) = O\left(\prod_{i=1}^{k+1} g_i(n)\right)$. You may use the result of part (a) in your proof, but you should not use any of the other Big-Oh properties.

   Solution:

Proof. Suppose $\prod_{i=1}^{k} f_i(n) = O\left(\prod_{i=1}^{k} g_i(n)\right)$, for some $k \geq 2$. Let $F_k(n) = \prod_{i=1}^{k} f_i(n)$ and $G_k(n) = \prod_{i=1}^{k} G_i(n)$. Thus, $F_k(n) = O(G_k(n))$.

Consider $F_{k+1}(n) = \prod_{i=1}^{k+1} f_i(n)$. By the definition of F_{k+1} and F_k, $F_{k+1}(n) = F_k(n) f_{k+1}(n)$. Since we know that $F_k(n) = O(G_k(n))$ and $f_{k+1}(n) = O(g_{k+1}(n))$, $F_k(n) f_{k+1}(n) = O(G_k(n) g_{k+1}(n))$ by part (a). Hence, $\prod_{i=1}^{k+1} f_i(n) = O\left(\prod_{i=1}^{k+1} g_i(n)\right)$, so the claim is true for all $x \geq 2$ by induction.

□

2. (a) [10 points] Simulate the algorithm below on the input $data = [-4, -9, 1, 8, 5, 6, 7, 10, -3, 2]$, and list the Union operations that are performed by the algorithm, in the order they are performed.

```
   Input: data: integer array of size n
   Input: n: the size of data
1  Algorithm: UnionMystery
2  uf = UnionFind(n)
3  for i = 2 to n do
4  |  if data[i] · data[i − 1] > 0 then
5  |  |  uf.Union(i, i − 1)
6  |  end
7  end
8  return uf
```

Solution: Union(2, 1) Union(4, 3) Union(5, 4) Union(6, 5) Union(7, 6) Union(8, 7)

(b) [10 points] Simulate these operations on the union-find uf, and indicate the contents of the union-find array at the end of the algorithm. Show your work.

You do not need to represent the array that keeps track of the size of each tree. For example, if the algorithm joined together the first five and the last five values, you might respond with

1	1	1	1	1	6	6	6	6	6

Solution: Expected answer:

2	2	4	4	4	4	4	4	9	10

Several variations possible, including using 1 and 3 for roots instead of 2 and 4. Alternatively, could be using an array with 0-based indexing (subtract 1 from all integers).

3. Answer the following questions about the recurrence $T(n) = k^2 T(n/k) + \Theta(n^2)$, where $k \geq 2$ is an integer.

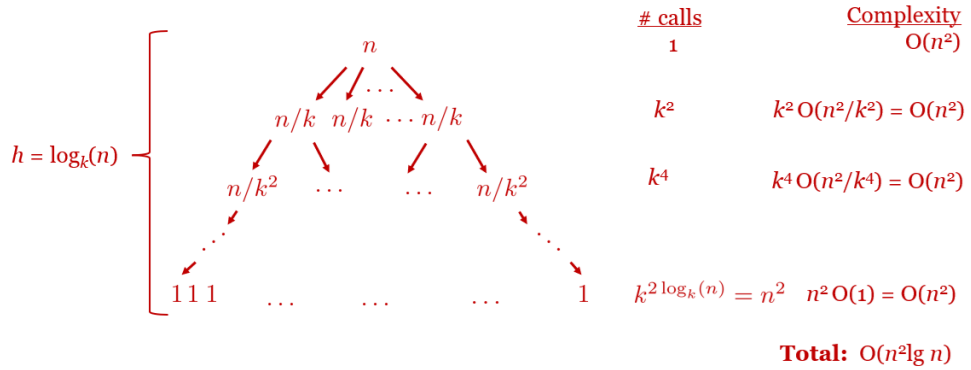
(a) [5 points] Sketch a recursion tree for $T(n)$. You should use ellipsis (. . .) to represent k^2, and you should indicate the height of the tree (in terms of n and k).

(b) [5 points] How many recursive calls are on each level of your tree? Label at least the first, second, and bottom levels of the tree.

(c) [5 points] Analyze the time complexity for each level of your tree. Show your work.

(d) [5 points] What is the overall time complexity for $T(n)$? Show your work.

Solution:




```
   Input: data: an array with n positive integers
   Input: n: size of data
   Input: t: a positive integer
 1 Algorithm: ArrayMystery
 2 heap = MaxHeap()
 3 max = 2^n
 4 for i = 1 to max do
 5 |   sum = 0
 6 |   for j = 1 to n do
 7 |   |   if bit j in the binary rep'n of i is 1 then
 8 |   |   |   sum = sum + data[j]
 9 |   |   end
10 |   end
11 |   if sum ≥ t then
12 |   |   heap.Insert(sum)
13 |   end
14 end
15 while heap is not empty do
16 |   heap.DeleteMax()
17 end
18 return the array containing heap
```

4. (a) [10 points] What is the worst case time complexity of the for loop in lines 5–15? Show your work. You may assume that you can test bits in $\Theta(1)$ time.

Solution: The inner loop iterates n times, at $\Theta(1)$ per iteration, for a total time of $\Theta(n)$. Line 12 takes time that is logarithmic with respect to the size of *heap* (see part b). If *heap* is $O(2^n)$, this line is $O(n)$. The for loop and heap insertion dominate the $\Theta(1)$ for lines 5 and

11, so each iteration of the outer for loop is $\Theta(n)$. The outer for loop iterates 2^n times, so the overall complexity will be $\Theta(n2^n)$.

(b) [5 points] How large is *heap* after the for loop in lines 5–15 in the worst case?

Solution: $O(2^n)$

(c) [5 points] What is the worst case time complexity of the while loop in lines 16–18? Show your work.

Solution: $O(n2^n)$
Each iteration is $O(n)$ (based on the answer to part b), and the loop iterates $O(2^n)$ times (again based on part b), so the overall complexity will be $O(n2^n)$.

5.

```
   Input: data: array of n elements to be sorted
   Input: n: size of data
   Output: permutation of data such that
                data[1] ≤ data[2] ≤ ... ≤ data[n]
 1 Algorithm: BucketMergeSort
 2 Let min and max be the min and max of data
 3 Divide the range of data into n buckets
 4 Let bucket be an array of n lists, initially empty
 5 for i = 1 to n do
 6 |   Let k be the bucket data[i] belongs in
 7 |   Append data[i] to bucket[k]
 8 end
 9 sorted = Array()
10 for i = 1 to n do
11 |   MergeSort(bucket[i])
12 |   Append bucket[i] to sorted
13 end
14 return sorted
```

(a) [5 points] Analyze the worst-case time complexity of lines 2–8 of BucketMergeSort. Show your work.

Solution: Lines 2 and 4 take $\Theta(n)$, while line 3 probably takes $\Theta(1)$. Lines 6 and 7 take $\Theta(1)$, and the loop in line 5 iterates n times, for a total of $\Theta(n)$. Thus, lines 2–8 take $\Theta(n)$.

(b) [5 points] Analyze the time complexity of lines 9–14 of BucketMergeSort in the case where the data values are distributed proportionally (i.e., each bucket is size $\Theta(1)$).

Solution: This loop iterates n times. If the values are distributed proportionally, each bucket is $\Theta(1)$, so calling MergeSort and appending these values to a list both take $\Theta(1)$ time. This

loop takes $\Theta(n)$ time total in this case.

(c) [5 points] What would be the worst-case time complexity of lines 9–14 of BucketMergeSort? Describe the worst case and show your complexity analysis.

Solution: The worst case would be that every value (or at least $O(n)$ of them) end up in the same bucket (e.g., if all values in the array were duplicates). In this case, calling MergeSort would take $\Theta(n \lg n)$ for some bucket (but not all n buckets*). Appending all of the buckets will always take $\Theta(n)$ total because every entry of the array appears in exactly one bucket. However, the total cost will be dominated by the $O(n \lg n)$ time for the most expensive MergeSort call, for a total of $O(n \lg n)$ time worst case.

* There can only be $\Theta(1)$ buckets of size $O(n)$; otherwise, the input array would need to contain more than n elements.

(d) [5 points] How does BucketMergeSort compare to the standard BucketSort, in which we use InsertionSort to sort each bucket instead of MergeSort?

Solution: BucketMergeSort has a better worst-case complexity than standard BucketSort (depends on answer to previous part). InsertionSort is the fastest sorting algorithm for small arrays, so the Big-Oh coefficient may be somewhat higher than the standard BucketSort in the best/expected case.

Just like BucketSort, BucketMergeSort is stable but not in-place.