1. (7 pts) In class before the midterm we discussed building heaps from \( n \) elements using a divide and conquer approach. The basic operation is “check that a root has more priority than its children and swap if necessary.” We had used the function \( T(n) \) defined by \( T(1) = 0, T(2) = T(3) = 1 \) and for \( n > 1 \), \( T(n) = 2T(\lfloor n/2 \rfloor) + \lg n \) as an upper bound on the number of these basic operations needed to build an \( n \) element heap. However, this analysis assumed that every parent’s two subtrees are equal size (or equivalently, that the heap was a complete binary tree - pg. 1090 of the text).

First (2 pts), if a heap contains \( n \) nodes, what is the largest fraction of \( n \) that can appear in a single subtree? Justify your answer.

Next (2 pts) consider the following way of building a heap of \( n \) elements. If \( n = 2^k - 1 \), then the heap is complete binary tree and we can use the divide and conquer process discussed in class. On the other hand, if \( 2^k - 1 < n < 2^k - 1 \) then we can add \( 2^k - 1 - n \) “dummy” items with lower priority than any possible “real” items, creating a new \( 2^k - 1 \) element build-heap problem. This technique of adding dummy elements to get a convenient input size is called “padding”. Show that building a heap with padding requires at most \( 2n \) basic operations (where \( n \) is the number of original, unpadded elements).

Finally (3 pts), from the first part you can get a recurrence relation like \( T(n) = T(\alpha_n n) + T((1 - \alpha_n)n) + \lg n \) where \( \alpha_n \) is some constant between 1/2 and your answer to the first part. (The \( \alpha \)’s are subscripted as the fraction of elements on each side depends on \( n \).) Use induction to prove that This \( T(n) \) is at most some \( f(n) \in \Theta(n) \). (Hints: Consider \( f(n) = cn \)−something. For my proof I found it easier to use several base cases so that the inductive step didn’t have to deal with any values less than 5 or 6.)

2. (8 pts) A 101 assignment requires the implementation of a hash table. One of the students in the class thinks that by making the hash table big enough, the chance of collision will be negligible, and thus the bug in his linked list will go unnoticed. Since the program is given \( k > 2 \), the number of elements to be inserted ahead of time, he decides to have his hash table contain \( k^2 \) locations.

- (2 pts) How many different ways can the \( k \) different elements be hashed into the \( k^2 \) different table locations?
- (3 pts) How many of these ways result in no collisions?
- (2 pts) If each way has the same probability, then what is the probability of one or more collisions? (You may want to use the product notation, \( \prod \), in this answer).
- (1 pt) Compute this probability for \( k = 5 \).

3. (3 pts) Assume \( n = 2^k \) and find a way to multiply two (positive) \( n \)-bit numbers using three multiplications of \( n/2 \)-bit numbers. Describe (in pseudo-code) and analyze a recursive divide-and-conquer algorithm for multiplication and analyze its running time. You can assume that the computer has a “shift” instruction that takes 1 unit of time (see page 22), and adding two \( n \)-bit numbers takes \( \Theta(n) \) time. To simplify the analysis, assume that there no “carry-outs” happen, so the addition of two \( \ell \)-bit numbers always gives a result fitting in \( \ell \)-bits.
4. (7 pts) Assume that you have an array of $n$ objects, and want to determine which of the objects (if any) is a “majority element”, occurring at least $\lfloor n/2 \rfloor + 1$ times in array. These objects are not ordered, so “<” or “>” comparisons are not possible, but you can test two objects for equality. Discover and analyze a divide and conquer algorithm for this problem that uses $O(n \lg n)$ equality tests on the objects.

(Hard!) For extra credit (0 pts), find a divide and conquer algorithm using only $O(n)$ equality tests. For the extra credit part, you can assume that $n$ is a convenient size (i.e. $c^k$ for some small integer $c$). The algorithm I have in mind does not strictly fit the “divide and conquer” template, as the outermost recursion does some extra work that is not done at every level.