**Problem 1.** (25 points [writing/solving recurrence relations])

A CS 141 student has been trying to speed-up Karatsuba’s divide-and-conquer integer multiplication algorithm. Given two numbers $x, y$ with $n$ bits each, her/his algorithm (1) first divides both $x$ and $y$ into four equal-length pieces, then (2) expresses the product $x \cdot y$ using $4$ multiplications of these $n/4$-bit pieces, followed by a merging step that takes $\Theta(n^p)$ where $p > 1$. What condition on $p$ would give her/him a faster algorithm than the Karatsuba’s algorithm covered in class? Justify your answer using the Master Theorem.

**Solution:** The new divide and conquer algorithm has the following recurrence relation:

$$T(n) = 4T(n/4) + n^p$$

which is Master Theorem case III, $n^p \in \Omega(n^{\log_4 4 + \epsilon})$, given that $p > 1$ then $\epsilon = p - \log_4 4 > 0$. The second condition requires that $af(n/b) \leq \delta f(n)$, that is $4(n/4)^p \leq \delta n^p$ that is $\delta \geq 4^{1-p}$ if we pick $\delta = 4^{1-p} < 1$ as needed. The conclusion is that $T(n) \in O(n^p)$. We need to find a condition on $p$ such that this algorithm is asymptotically better than $O(n^{\log_2 3})$, i.e., we need $p < \log_2 3 \approx 1.58$.

**Problem 2.** (25 points [writing/solving recurrence relations])

In the algorithm SELECT described in class (linear-time selection), the input elements are divided into $\lceil n/5 \rceil$ groups of 5. Suppose you modify the algorithm to divide the input elements into $\lceil n/3 \rceil$ groups of 3 instead. Let $T(n)$ denote the worst-case running time of the modified algorithm as a function of the input size $n$. Write a recurrence relation for $T(n)$, but do NOT solve it.

**Answer:**

For groups of 3, the number of elements greater than $x$ (and the number of element less than $x$) is at least

$$2 \left( \lceil \frac{n}{2} \rceil - 2 \right) \geq \frac{n}{3} - 4$$

and the recurrence relation becomes

$$T_3(n) \leq T_3(\lceil n/3 \rceil) + T_3(2n/3 + 4) + O(n)$$

The algorithm no longer works in linear time. One can show that $T_3(n) \geq n \log n$ by induction. In fact, only group of odd size $\geq 5$ make the algorithm work in linear time.

**Problem 3.** (18 points [divide & conquer])

In homework 3 you have solved the problem of the majority element using divide and conquer. This is a variant on that problem. We say that an array $A$ has a one-third-majority item if that item appears at least $n/3$ times in $A$. Given an unsorted array $A[1\ldots n]$ of $n$ items we want to determine whether $A$ has one-third-majority element, and if so, return such an item. If there is more than one one-third-majority element in $A$, we are OK with any one of them. Consider the following divide and conquer algorithm for this problem:
Algorithm **Find-One-Third-Majority** \((A: \text{array})\)

1. \(n \leftarrow |A|\)
2. **if** \(n \leq 3\) **then return** \(A[1]\)
3. Partition \(A\) into \(A_1, A_2, A_3\) where \(A_1\) is the first third of \(A\), \(A_2\) is the second third of \(A\), and \(A_3\) is the last third of \(A\)
4. \(a_1 \leftarrow \text{Find-One-Third-Majority}(A_1)\)
5. \(a_2 \leftarrow \text{Find-One-Third-Majority}(A_2)\)
6. \(a_3 \leftarrow \text{Find-One-Third-Majority}(A_3)\)
7. **if** the number of occurrences of \(a_1\) in \(A\) is \(\geq n/3\) **then return** \(a_1\)
8. **else if** the number of occurrences of \(a_2\) in \(A\) is \(\geq n/3\) **then return** \(a_2\)
9. **else if** the number of occurrences of \(a_3\) in \(A\) is \(\geq n/3\) **then return** \(a_3\)
10. **else return** \(\text{False}\)

Is this algorithm correct i.e., does it always return a one-third-majority element, if \(A\) has one in it (or \(\text{False}\) otherwise)? Give a counterexample if your answer is “No”, a brief argument of correctness if your answer is “Yes”. Assume \(n\) is a power of 3.

The algorithm does not work. Consider \(A = \{1, 2, 3, 4, 2, 5, 6, 2, 7\}\), which contains the one-third-majority element 2 (appears three time in \(A\) and \(|A| = 9\)). The recursive call on \(A_1 = \{1, 2, 3\}\) will return \(a_1 = 1\) (which is correct, since 1 is one of the one-third-majority element in \(A_1\)), but it appears only once in \(A\). The recursive call on \(A_2 = \{4, 2, 5\}\) will return \(a_2 = 4\) (4 is one of the one-third-majority element in \(A_2\), but it appears only once in \(A\). The recursive call on \(A_3 = \{6, 2, 7\}\) will return \(a_3 = 6\) (6 is one of the one-third-majority element in \(A_3\), but it appears only once in \(A\). The element 2 is missed, despite being one-third-majority element in \(A\).

**Problem 4.** (25 points [divide & conquer])

Suppose you are given an array \(A = \{a_1, a_2, \ldots, a_n\}\) of \(n\) distinct integers. You are told that the sequence of values \(a_1, a_2, \ldots, a_n\) is unimodal, that is for some index \(p \in [1, n]\), the values in the array increase up to position \(p\), and then decrease the remainder of the way until position \(n\). Give an algorithm to find the position \(p\) in \(O(\log n)\) time. You can assume \(n\) to be a power of 2.

**Answer:** Compare the elements \(A[n/2]\), \(A[n/2 - 1]\) and \(A[n/2 + 1]\) to decide whether to search on the left, on the right, or whether we are done. More specifically

- if \(A[n/2 - 1] < A[n/2] < A[n/2 + 1]\), then search recursively in the entries \(A[n/2 + 1 \ldots n]\)
- if \(A[n/2 - 1] > A[n/2] > A[n/2 + 1]\), then search recursively in the entries \(A[1 \ldots n/2 - 1]\)

The algorithm has the same structure of binary search, its recurrence relation is \(T(n) = T(n/2) + O(1)\), which has solution \(O(\log n)\).