Problem 1
Developing efficient methods to "split" the data is fundamental in the design of efficient divide and conquer algorithms. Read Section 2.4, pages 64-66 of "Algorithms" by Dasgupta, Papadimitriou and Vazirani.
(a) In the median-finding algorithm of Section 2.4 of the above textbook, a basic primitive is the SPLIT operation, which takes as input an array $S$ and a value $u$ and then divides $S$ into three sets of elements: the elements less than $u$, the elements equal to $u$, and the elements greater than $u$. Show how to perform this SPLIT operation in place, that is, without allocating new memory. [This is Exercise 2.15 in the above textbook.]
(b) On page 66 of the above textbook there is a high-level description of the quicksort algorithm. Write down the pseudocode for quicksort. [This is Exercise 2.24 part (a) in the above textbook.]
Comment: Realize that, in view of the fact that the SPLIT operation can be performed in place, the quicksort algorithm essentially sorts in place, in the sense that the only additional memory allocation involves the stack needed to keep track of the recursion. This is an additional reason why, in practice, quicksort is very often the algorithm of choice.

Problem 2
An array $A(1 \ldots n)$ is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element. The elements of the array are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form is $A(i) > A(j)$? (Think of the array elements as GIF files, say.) However you can answer questions of the form: is $A(i) = A(j)$? in constant time.
(a) Show how to solve this problem in $O(n \log n)$ time.
Hint: Split the array $A$ into two arrays $A1$ and $A2$ of half the size. Does knowing the majority elements of $A1$ and $A2$ help you figure out the majority element of $A$? If so, you can use a divide-and-conquer approach.
(b) Can you give a linear-time algorithm? Hint: Here's another divide-and-conquer approach:
- Pair up the elements of $A$ arbitrarily, to get $n/2$ pairs.
- Look at each pair: if the two elements are different, discard both of them; if they are the same, keep just one of them.
Show that after this procedure there are at most $n/2$ elements left, and that they have a majority element if $A$ does. [This is Exercise 2.23 of "Algorithms" by Dasgupta, Papadimitriou and Vazirani.]
Comment: After you design your time-efficient algorithms, you may want to start considering space efficiency (this is optional).

Problem 3
In Section 2.1 of "Algorithms" by Dasgupta, Papadimitriou and Vazirani we saw an $O(n^{\log_2 3})$ algorithm for integer multiplication. $O(n^{\log_2 3})$ is the solution to the recurrence $T(n) = 3T(n/2) + cn$. In this problem you are asked to design and analyze a faster algorithm for integer multiplication.
Hint: On input $n$-bit integers $x$ and $y$, let $x_2$ be the $n/3$ leading bits of $x$, $x_1$ be the $n/3$ middle bits of $x$, and $x_0$ be the $n/3$ final bits of $x$, so that $x = x_2 2^\frac{2n}{3} + x_1 2^\frac{n}{3} + x_0$. Similarly $y = y_2 2^\frac{2n}{3} + y_1 2^\frac{n}{3} + y_0$. Can you infer the product $xy$ using 5 multiplications involving $n/3$-bit numbers, and a constant number of additions? That would lead to a recurrence $T(n) = 5T(\frac{n}{3}) + cn$ and an $O(n^{\log_2 5})$ algorithm.
Problem 4
There are $2^n$ 0-1 $n$-bit numbers. Give an algorithm that, on input $n$, reserves an array $W(1 \ldots n)$, and successively writes on $W$ all $2^n$ 0-1 $n$-bit numbers in increasing order. For example, for $n = 3$, $W$ should successively become 000, 001, 010, 011, 100, 101, 110, 111. Your algorithm will clearly take time exponential in $n$. However, your algorithm should allocate additional memory no more than a polynomial in $n$. Hint: A recursive algorithm will satisfy the memory requirement.  

Comment: In practice, algorithms need to go through counts of 1 to $N$ all the time. If you think of $N = 2^m$, the above problem indicates that each one of such counts can be implemented by allocating additional memory $O(\log N)$ or perhaps $O(\log^2 N)$. 
