You should solve the “Basic Problems” 1–4 at home, and attend one of the tutorial sessions in the given week, prepared to present your solutions at class if requested. You earn one tutorial credit point for each of these problems that you have solved, and marked as solved at the tutorial session. The “Advanced Problems” 5–6 will be solved collaboratively at the tutorial session, directed by the TA. Although you do not earn credit points for these, you may wish to think about them before attending the session.

Basic problems:

1. [Dasgupta et al., Ex. 2.5] Solve the following recurrence relations and give an \( O(f(n)) \) bound for each of them.
   
   (a) \( T(n) = 2T(n/3) + 1 \)
   (b) \( T(n) = 8T(n/2) + n^3 \)
   (c) \( T(n) = T(n-1) + n^c \), where \( c \geq 1 \) is a constant
   (d) \( T(n) = T(n-1) + c^{n-1} \), where \( c > 1 \) is a constant
   (e) \( T(n) = 2T(n-1) + 1 \)
   (f) \( T(n) = T(\sqrt{n}) + 1 \)

   Assume in each case that \( n \) has an algebraically convenient form to ease the solution. For example, you may assume in (a) that \( n = 3^k \) for \( k = 1, 2, \ldots \) and in (f) that \( n = 2^{2^k} \) for \( k = 0, 1, 2, \ldots \). The base case is \( T(1) = 1 \) in (a)–(e) and \( T(2) = 1 \) in (f).

2. Recall that in Strassen’s efficient divide-and-conquer algorithm for matrix multiplication, two \( n \times n \) matrices are partitioned into \( \frac{n}{2} \times \frac{n}{2} \) submatrices that are then multiplied together using 7 recursive calls, instead of 8 as would be suggested by a straightforward approach. This results in an algorithm with a running time of \( O(n^\log_2{7}) \). Now suppose that you came up with an idea for multiplying \( 3 \times 3 \) matrices together using \( m \) < 27 multiplications. What would be the running time of a divide-and-conquer algorithm for multiplying \( n \times n \) matrices based on this idea? How small should the value of \( m \) be so that your algorithm would be asymptotically faster than Strassen’s method?

3. [Dasgupta et al., Ex. 2.14] You are given an array of \( n \) elements, and you notice that some of the elements are duplicates; that is, they appear more than once in the array. Show how to remove all duplicates from the array in time \( O(n \log n) \).

4. [Dasgupta et al., Ex. 2.23] Array \( A[1 \ldots n] \) contains a majority element \( a \) if \( A[i] = a \) for more than half of all the indices \( i \in \{1, \ldots, n\} \). Suppose that the elements of \( A \) do not necessarily come from an ordered domain, and so they can only be tested for equality (“is \( A[i] = A[j] \)”), but not compared (“is \( A[i] > A[j] \)”). Design an algorithm that determines in time \( T(n) = O(n \log n) \) if a given array \( A[1 \ldots n] \) contains a majority element, and if so then what it is. For simplicity, you may assume that \( n \) is a power of 2. (Hint: Consider first splitting the array in two halves and determining their majority elements, if any. What can you conclude from this information concerning the majority element, if any, of the full array?)
One can even achieve $T(n) = O(n)$; can you see how? (Hint: First prune in linear time at least half of the elements away, and then make a single recursive call with at most $n/2$ elements.)

**Advanced problems:**

5. You are given an array $x[1...n]$ of integers with $0 \leq x_i \leq 2^b - 1$ for $i = 1, 2, ..., n$. Assume that the array elements can be copied and moved in time $O(1)$. Design an algorithm that sorts the array in time $O(bn)$. Why doesn’t the $\Omega(n \log n)$ lower bound on sorting apply here?

6. [Dasgupta et al., Ex. 2.24] Consider the quicksort algorithm presented at Lecture 4, with a uniformly random selection of the pivot element.

   (a) Show that the worst-case runtime of the algorithm on an array with $n$ elements is $\Theta(n/2)$.

   (b) Show that the expected runtime $T(n)$ satisfies the recurrence relation

   $$T(n) \leq cn + \frac{1}{n} \sum_{i=0}^{n-1} (T(i) + T(n-i)),$$

   for some constant $c$. Deduce from this that $T(n) = O(n \log n)$.