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 \(n = 2^k\) for \(k = 0, 1, 2, \ldots\). The base case is \(T(1) = 1\) in (a)–(e) and \(T(2) = 1\) in (f).

   **Hints:** For (a) and (b), apply the “master theorem” for divide-and-conquer recurrences, Lecture 3, slide 20. For (c), (d) and (e), unwind the recurrence. In (c), upper-bound the resulting sum by its biggest term and the total number of terms. (You may also want to lower-bound it based on the middle term.) In (d) and (e), apply the formula for geometric sums. In (f) you may want to first solve the recurrence for the transformed sequence \(T'(k) = T(2^k)\), and then apply the inverse transformation \(T(n) = T'(\log_2 \log_2 n)\).

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?

   **Hint:** Master theorem for divide-and-conquer recurrences.

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 \(O(n \log n)\).

   **Hint:** What algorithms for dealing with sequences, of complexity \(O(n \log n)\), do you know?

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. \( \text{(Hint: Consider first splitting the array in two halves and determining their majority elements, if any. What can you conclude from this information concerning the majority element, if any, of the full array?)} \)

One can even achieve \( T(n) = O(n) \); can you see how? \( \text{(Hint: First prune in linear time at least half of the elements away, and then make a single recursive call with at most} \ n/2 \text{elements.)} \)

**Advanced problems:**

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

**Solution:** Let us write \( x_i \) in base 2 as \( x_i = \sum_{j=0}^{b-1} 2^{b-1-j} x_{ij} \), with \( x_{ij} \in \{0, 1\} \) for all \( j = 0, 1, \ldots, b-1 \). Note that we have \( x_{i1} < x_{i2} \) if and only if there exists a \( k = 0, 1, \ldots, b-1 \) such that \( x_{i1,k} < x_{i2,k} \) and for all \( j = 0, 1, \ldots, k-1 \) we have \( x_{i1,j} = x_{i2,j} \). Equivalently, \( x_{i1} < x_{i2} \) holds if and only if, viewing \( x_{i1} \) and \( x_{i2} \) as \( b \)-bit strings, the first difference between \( x_{i1} \) and \( x_{i2} \) occurs in the \( k \)-th most significant bit, which is a 0 in \( x_{i1} \) and a 1 in \( x_{i2} \).

We can turn this observation into a divide-and-conquer sorting algorithm as follows. Suppose our task is to sort the subarray \( x[\ell \ldots h] \), and we are promised that the elements of \( x[\ell \ldots h] \) all agree pairwise in their most significant \( k-1 \) bits. (Initially, take \( k = 1, \ell = 1, h = n \). In particular, the promise is trivial.)

Let us carry out the sort as follows. First, split the array \( x[\ell \ldots h] \) so that no element with the \( k \)-th bit set to 1 is to the right of an element with the \( k \)-th bit set to 0. This splitting can be implemented in \( O(h - \ell) \) time. Indeed, scan \( x[\ell \ldots h] \) from the left towards the right until an element with the \( k \)-th bit set to 1 is found, and from the right towards the left until an element with the \( k \)-th bit set to 0 is found, or until the two scans meet. Whenever such a pair of elements is found, transpose the elements and continue the scan. When the two scans meet, the split is complete. After the split, the left part of the array consists of elements that agree in the most significant \( k \) bits, with the \( k \)-th bit set to 0, and the right part of the array consists of elements that agree in the most significant \( k \) bits, with the \( k \)-th bit set to 1. We can thus sort the left and right parts using two recursive calls. (Indeed, the promise holds for \( k+1 \) in the left and right parts.) Note, however, that one of the parts may be empty, in which case only one recursive call is required. The base case is at \( k = b+1 \), at which point we observe that the array \( x[\ell \ldots h] \) consists of identical elements and hence is sorted.

There are \( b+1 \) levels of recursive calls, and the time required at each level \( k \) is bounded by the total time consumed by the split operations at that level, which is \( O(n) \) since the parts at each level are pairwise disjoint. Thus, the running time is \( O(bn) \).

The \( \Omega(n \log n) \) lower bound on sorting does not apply here because we are using the structure of the encoding of integers with at most \( b \) bits. Recall that the lower bound holds if the sorting process is based *exclusively* on pairwise comparison between elements.

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

\[
T(n) \leq cn + \frac{1}{n} \sum_{i=1}^{n-1} (T(i) + T(n-i)),
\]
for some constant $c$. Deduce from this that $T(n) = O(n\log n)$.

**Solution:** The quicksort python implementation from lecture 4 is presented here in Listing 1. For (a), we need to show that $n^2$ is both a lower and an upper bound for the worst-case runtime of quicksort.

For the lower bound, consider the case where the partition returns $k = i + 1$ in line 4. This occurs if for instance the input is a sorted array and the randomized pivoting in line 10 yields $p = i$. In such a case, the sorting for the left partition, in line 5, will always be on an array of size 1, while the sorting for the right partition, in line 6, will always be on an array of size $n - 1$. Hence, the run-time of the algorithm will be $T(n) = T(1) + T(n - 1) + T_{14-21}$, where $T_{14-21}$ is the number of elementary operations carried out in the partition function. But, $T_{14-21} \geq T_{17} + T_{19}$, where $T_{17}$ and $T_{19}$ are the number of times lines 17 and line 19 are executed, respectively. Since at the end of the while loop $l = r$, it must be the case that $i + T_{17} = j - T_{19}$, or equivalently $T_{17} + T_{19} = j - i = n - 1$. Hence, $T_{14-21} \geq n - 1$ and $T(n) \geq T(1) + T(n - 1) + n - 1$. Expanding and setting $T(1) = 0$ we get,

$$
T(n) \geq T(1) + T(n - 1) + n - 1 \\
\geq T(1) + T(1) + T(n - 2) + n - 2 + n - 1 \\
\geq T(1) + T(1) + T(1) + T(n - 3) + n - 3 + n - 2 + n - 1 \\
\geq (n - 1)T(1) + (1 + 2 + \ldots + n - 2 + n - 1) \\
\geq \frac{(n - 1)n}{2} \\
= \Omega(n^2).
$$

For the upper bound, we count the number of times the quicksort is called and use an upper bound for each call. Let $x$ denote the number of quicksort calls where $i < j$ in line 2 and $y$ where $i = j$. Clearly, the total number of calls is $x + y$. On the other hand, each instance where $i < j$ makes two quicksort calls in lines 5 and 6, while each instance where $i = j$ makes no calls. Noting that the initial quicksort with $i = 1$ and $j = n$ is not called by anyone but must be counted, we can deduce the total number of calls is $2x + 1$. Thus, $2x + 1 = x + y$; or equivalently, $x = y - 1$. Further, observe that there are exactly $n$ independent cases where $i = j$, corresponding to indices $\{1, 2, \ldots, n\}$, and hence $y = n$. Thus, the total number of quicksort calls is $2n - 1$.

Now, a quicksort call on an array of size $m$ takes at most $m + c$ elementary operations. Indeed, $T_{17} + T_{19} = j - i = m - 1$ and $T_{10}$, $T_{14}$ and $T_{21}$ all take constant time. Since each quicksort call operates on an array (range) of size $n$ or less, each call takes at most $n + c$ elementary operations. With a total of $2n - 1$ calls, the upper bound on the total number of elementary operations is $(2n - 1)(n + c)$ and hence $T(n) = O(n^2)$. 

3
Listing 1: Quicksort python implementation

```python
def quicksort(A, i, j):
    if i < j:
        v = pivot(A, i, j)
        k = partition(A, i, j, v)
        quicksort(A, i, k-1)
        quicksort(A, k, j)

def pivot(A, i, j):
    # randomised pivoting
    p = random.randint(i, j)
    return A[p]

def partition(A, i, j, v):
    l, r = i, j
    while l < r:
        while (A[l] <= v) and (l < r):
            l = l + 1
        while (A[r] > v) and (l < r):
            r = r - 1
        if l < r:
    return l
```

For (b), let us consider the runtime of quicksort on array of size $n$ as a random variable $t(n)$ which depends on the randomly selected pivot $v = A[p]$. Recalling the linear upper bound on the number of elementary operations in the partition function, we can see that $t(n) \leq cn + t(k - 1) + t(n - k + 1)$, where $c$ is some constant and $k$ is the value returned in line 4. We see that $k$ is a random variable dependent on the number of array elements which are less than or equal to $A[p]$.

Let us denote by $m$ the size of the set $\{i : A[i] \leq A[p] \}$. If $m = n$, then $l$ is incremented all the way up-to $n$ in line 17 and $r$ is not decremented at all. Thus, the partition function returns $k = n$. If however $m < n$, then it can be checked that all the elements less than or equal to $A[p]$ will be exchanged to positions less than the returned $k$ value and thus $k = m + 1$.

Since we are taking $p$ at random from 1 to $n$, $m$ will also take a random value between 1 and $n$ with probability $1/n$. However, $k$ will take value between 2 and $n - 1$ with probability $1/n$. On the other hand, it has value $n$ with probability $2/n$ ($1/n$ for the case where $m = n$ and $1/n$ for the case with $m = n - 1$.) We can now find the expectation of $t(n)$ using conditional expectations:
\[
T(n) = E[t(n)] = \sum_{i=2}^{n} E[t(n)|k = i]Pr[k = i]
\]  
(1)

\[
\leq \frac{cn + T(n-1) + T(1)}{n} + \frac{1}{n} \sum_{i=2}^{n-1} (cn + T(i-1) + T(n-i+1)) \frac{1}{n}
\]  
(2)

\[
\leq \frac{cn + T(n-1) + T(1)}{n} + \frac{1}{n} \sum_{i=2}^{n} (cn + T(i-1) + T(n-i+1)) \frac{1}{n}
\]  
(3)

\[
\leq \frac{cn + T(n-1) + T(1)}{n} + c(n-1) + \frac{1}{n} \sum_{i=2}^{n} T(i-1) + T(n-i+1)
\]  
(4)

\[
\leq c'n + \frac{1}{n} \sum_{i=1}^{n-1} T(i) + T(n-i)
\]  
(5)

In step (2), we are splitting the sum to the cases where \(k = n\) and \(k < n\). In step (3), we reintroduce half of the first term to the summation. In step (4), we are using the fact that \(\sum_{i=2}^{n}(cn)\frac{1}{n} = c(n-1)\). In step (5), we use the fact that \(T(n-1) \leq T(n) = O(n^2)\) to bound the whole expression before the sum.

To show that \(T(n) = O(n \log n)\), we first observe that the term \(T(j)\) would appear twice in the expansion of the sum and hence \(T(n) \leq cn + \frac{2}{n} \sum_{i=1}^{n-1} T(i)\). Iteratively expanding, we obtain
\[ T(n) \leq cn + \frac{2}{n} \sum_{i=1}^{n-1} T(i) \]
\[ \leq cn + \frac{2}{n} (T(n-1) + \sum_{i=1}^{n-2} T(i)) \]
\[ \leq cn + \frac{2}{n} (c(n-1) + \frac{2}{n-1} \sum_{i=1}^{n-2} T(i) + \sum_{i=1}^{n-2} T(i)) \]
\[ \leq cn + \frac{2}{n} (c(n-1) + \frac{n+1}{n-1} \sum_{i=1}^{n-2} T(i)) \]
\[ \leq cn + \frac{2}{n} (c(n-1) + \frac{2(n+1)}{n(n-1)} (T(n-2) + \sum_{i=1}^{n-3} T(i)) \]
\[ \leq cn + \frac{2}{n} (c(n-1) + \frac{2(n+1)}{n(n-1)} (c(n-2) + \frac{2}{n-2} \sum_{i=1}^{n-3} T(i) + \sum_{i=1}^{n-3} T(i)) \]
\[ \leq cn + \frac{2}{n} (c(n-1) + \frac{2(n+1)}{n(n-1)} (c(n-2) + \frac{n}{n-2} \sum_{i=1}^{n-3} T(i)) \]
\[ \leq cn + \frac{2}{n} (c(n-1) + \frac{2c(n+1)(n-2)}{n(n-1)} + \frac{2(n+1)}{(n-1)(n-2)} \sum_{i=1}^{n-3} T(i) \]
\[ \leq cn + \frac{2}{n} (c(n-1) + \frac{2c(n+1)(n-2)}{n(n-1)} + \frac{2c(n+1)(n-3)}{(n-1)(n-2)} + \frac{2(n+1)}{(n-2)(n-3)} \sum_{i=1}^{n-4} T(i) \]
\[ \leq c'n + 2c(n+1) (\frac{(n-2)}{n(n-1)} + \frac{(n-3)}{(n-1)(n-2)} + \frac{(n-4)}{(n-2)(n-3)} + \ldots + \frac{2}{(4)(3)} + \frac{2(n+1)}{(3)(2)}) T(1) \]
\[ \leq c''n + 2c(n+1) \sum_{i=1}^{n-3} \frac{i+1}{(i+3)(i+2)} \]
\[ \leq c''n + 2c(n+1) \sum_{i=1}^{n-3} \frac{1}{i+2} \]
\[ \leq c''n + 2c(n+1) \sum_{i=1}^{n} \frac{1}{i} \]
\[ \leq c''n + 2c(n+1)H_n \]

where \(H_n\) is the \(n\)th harmonic number. Asymptotically, \(H_n = O(\log n)\). Hence, \(T(n) = O(n \log n)\).