Homework problems:

3 [Dasgupta et al., Ex. 2.5] Solve the following recurrence equations and give an $O(f(n))$ bound for each of them.

(a) $T(n) = 7T(n/7) + n$
(b) $T(n) = 9T(n/3) + n^2$
(c) $T(n) = T(n - 1) + n^c$, where $c \geq 1$ is a constant
(d) $T(n) = T(n - 1) + cn^{c-1}$, where $c > 1$ is a constant
(e) $T(n) = T(\sqrt{n}) + 1$

Assume in each case that $n$ has an algebraically convenient form to ease the solution. For example, you may assume in (a) that $n = 7^k$ for $k = 1, 2, \ldots$ and in (e) that $n = 2^{2^k}$ for $k = 0, 1, 2, \ldots$. The base case is $T(1) = 1$ in (a)–(d) and $T(2) = 1$ in (e).

Hint. For (a) and (b), apply the “master theorem” for divide-and-conquer recurrences, Lecture 3, slide 20. For (c) and (d), 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), apply the formula for geometric sums. In (e) you may want to first solve the recurrence for the transformed sequence $T'(k) = T(2^{2^k})$, and then apply the inverse transformation $T(n) = T'(\log_2 \log_2 n)$.

4 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. It suffices to directly apply master’s theorem on the recursive relation to obtain a complexity notation. Value of $m$ can be found by solving the inequality.

5 [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. Is there any case where the major element is not returned by either of the two halves? What if the two sub problems return different majority elements? How much computation are you allowed to combine the two subproblems?

Demonstration problems:
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 $T(n) = O(n\log n)$.

**Solution:** The quicksort python implementation from lecture 4 is presented here below. 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 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,

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

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 $T(n) = O(n^2)$.
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:
\[ T(n) = E[t(n)] = \sum_{i=2}^{n} E[t(n) | k = i] Pr[k = i] \]  
\[ \leq (cn + T(n-1) + T(1)) \frac{2}{n} + \sum_{i=2}^{n-1} (cn + T(i-1) + T(n-i+1)) \frac{1}{n} \] (2) 
\[ \leq (cn + T(n-1) + T(1)) \frac{1}{n} + \sum_{i=2}^{n} (cn + T(i-1) + T(n-i+1)) \frac{1}{n} \] (3) 
\[ \leq (cn + T(n-1) + T(1)) \frac{1}{n} + c(n-1) + \frac{1}{n} \sum_{i=2}^{n} T(i-1) + T(n-i+1) \] (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
\[ T(n) \leq cn + \frac{2}{n} \sum_{i=1}^{n-1} T(i) \]
\[ \leq cn + \frac{2}{n} (T(n-1) + \sum_{i=1}^{n-2} T(i)) \]
\[ \leq cn + \frac{2}{n} (c(n-1) + \frac{2}{n-1} \sum_{i=1}^{n-2} T(i) + \sum_{i=1}^{n-2} T(i)) \]
\[ \leq cn + \frac{2}{n} (c(n-1) + \frac{n+1}{n-1} \sum_{i=1}^{n-2} T(i)) \]
\[ \leq cn + \frac{2}{n} (c(n-1) + \frac{2(n+1)}{n(n-1)} (T(n-2) + \sum_{i=1}^{n-3} T(i)) \]
\[ \leq cn + \frac{2}{n} (c(n-1) + \frac{2(n+1)}{n(n-1)} (c(n-2) + \frac{2}{n-2} \sum_{i=1}^{n-3} T(i) + \sum_{i=1}^{n-3} T(i)) \]
\[ \leq cn + \frac{2}{n} (c(n-1) + \frac{2(n+1)}{n(n-1)} (c(n-2) + \frac{n}{n-2} \sum_{i=1}^{n-3} T(i) \]
\[ \leq cn + \frac{2}{n} (c(n-1) + \frac{2c(n+1)(n-2)}{n(n-1)} + \frac{2(n+1)}{(n-1)(n-2)} \sum_{i=1}^{n-3} T(i) \]
\[ \leq cn + \frac{2}{n} (c(n-1) + \frac{2c(n+1)(n-2)}{n(n-1)} + \frac{2(n+1)}{(n-1)(n-2)} \sum_{i=1}^{n-3} T(i) \]
\[ \leq c'n + 2c(n+1) \left( \frac{(n-2)}{n(n-1)} + \frac{(n-3)}{(n-1)(n-2)} + \frac{(n-4)}{(n-2)(n-3)} + \ldots + \frac{2(n+1)}{(3)(2)} \right) \]
\[ \leq c'n + 2c(n+1) \sum_{i=1}^{n-3} \frac{i+1}{(i+3)(i+2)} \]
\[ \leq c'n + 2c(n+1) \sum_{i=1}^{n-3} \frac{1}{i+2} \]
\[ \leq c'n + 2c(n+1) \sum_{i=1}^{n} \frac{1}{i} \]
\[ \leq c'n + 2c(n+1)H_n \]

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