Pre-tutorial questions

Do you know the basic concepts of this week's lecture content? These questions are only to test yourself. They will not be explicitly discussed in the tutorial, and no solutions will be given to them.

1. Divide and Conquer
   (a) What is the general idea of a Divide-and-Conquer algorithm? Can you break up the general structure into three steps?
   (b) Why are recursions usually needed to analyse the running time of a Divide-and-Conquer algorithm?
2. Algorithms
   (a) Describe the general idea of merge sort.
   (b) What are the three main steps in the algorithm for counting inversions?
3. Recurrences
   (a) State the master method.
   (b) Try to explain the master method using a recursion tree.

Tutorial

Problem 1
Solve the following recurrences:

1. $T(n) = 4T(n/2) + n^2$
2. $T(n) = T(n/2) + 2^n$
3. $T(n) = 16T(n/4) + n$
4. $T(n) = 2T(n/2) + n \log n$
5. $T(n) = \sqrt{2}T(n/2) + logn$
6. $T(n) = 3T(n/2) + n$
7. $T(n) = 3T(n/3) + \sqrt{n}$

Solution: First we state the Master Theorem.
The Master Theorem applies to recurrences of the following form: $T(n) = aT(n/b) + f(n)$ where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function. There are three cases:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ with $\varepsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$. Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

1. $T(n) = 4T(n/2) + n^2 \rightarrow T(n) = \Theta(n^2 \log n)$ (Case 2)
2. $T(n) = T(n/2) + 2^n \rightarrow \Theta(2^n)$ (Case 3)
3. $T(n) = 16T(n/4) + n \rightarrow T(n) = \Theta(n^2)$ (Case 1)
4. $T(n) = 2T(n/2) + n \log n \rightarrow T(n) = \Theta(n \log^2 n)$ (Case 2)
5. $T(n) = \sqrt{2}T(n/2) + logn \rightarrow T(n) = \Theta(\sqrt{n})$ (Case 1)
6. $T(n) = 3T(n/2) + n \rightarrow T(n) = \Theta(n^{\lg 3})$ (Case 1)
7. $T(n) = 3T(n/3) + \sqrt{n} \rightarrow T(n) = \Theta(n)$ (Case 1)

Problem 2
Consider the following algorithm.

Algorithm 1 REVERSE

```
function REVERSE(A)
    if |A| = 1 then
        return A
    else
        Let B and C be the first and second half of A
        return concatenate REVERSE(C) and REVERSE(B)
    end if
end function
```

Let $T(n)$ be the running time of the algorithm on a instance of size n. Write down the recurrence relation for $T(n)$ and solve it by unrolling it.

Solution:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n) = O(n \log n)$$

Problem 3
The product of two $n \times n$ matrices X and Y is a third $n \times n$ matrix $Z = XY$, where the (i, j) entry of Z is $Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}$. Suppose that X and Y are divided into four $n/2 \times n/2$ blocks each:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \text{ and } Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}.$$

Using this block notation we can express the product of X and Y as follows

$$XY = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}.$$

In this way, one multiplication of $n \times n$ matrices can be expressed in terms of 8 multiplications and 4 additions that involve $n/2 \times n/2$ matrices. Let $T(n)$ be the time complexity of multiplying two $n \times n$ matrices using this recursive algorithm.

1. Derive the recurrence for $T(n)$. (Assume adding two $k \times k$ matrices takes $O(k^2)$ time.)
2. Solve the recurrence by unrolling it.

Solution:

1. Breaking each matrix into four sub-matrices takes $O(n^2)$ time and so does putting together the results from the recursive call. Therefore the recurrence is

$$T(n) = 8T(n/2) + O(n^2).$$

2. $T(n) = O(n^3)$. Therefore, this is no better that the standard algorithm for computing the product of two n by n matrices.

Problem 4

Your friend Alex is very excited because he has discovered a novel algorithm for sorting an array of n numbers. The algorithm makes three recursive calls on arrays of size $\frac{2n}{3}$ and spends only $O(1)$ time per call.

Algorithm 2 NEW-SORT

```
1: procedure NEW-SORT(A)
2:     if |A| < 3 then
3:         sort A directly
4:     else
5:         NEW-SORT(A[0], ..., A[2n/3])
6:         NEW-SORT(A[n/3], ..., A[n])
7:         NEW-SORT(A[0], ..., A[2n/3])
8:     end if
9: end procedure
```

Alex thinks his breakthrough sorting algorithm is very fast but has no idea how to analyze its complexity or prove its correctness. Your task is to help Alex:

1. Find the time complexity of NEW-SORT
2. Prove that the algorithm actually sorts the input array

Solution:

1. Let $T(n)$ be the running time of the algorithm on an array of length n. Then $T(n) = 3T(\frac{2n}{3})+O(1)$, which is $O\left(n^{\log_{\frac{3}{2}} 3}\right)$. That's roughly $O(n^{2.71})$. Much worse that bubble sort! Alex is heartbroken...

2. The algorithm, however, is correct. Think of the bottom $\frac{1}{3}$ of the elements in sorted order. After Line 6 is executed we are guaranteed that these element lie in the first $\frac{2}{3}$ of the array and thus Line 7 correctly places them in the first $\frac{1}{3}$ of the array. A similar argument can be made about the top $\frac{1}{3}$ of the elements in sorted order ending in the $\frac{1}{3}$ of the array. Therefore, the middle $\frac{1}{3}$ of the elements in sorted order are placed in the middle $\frac{1}{3}$ of the array. Within each $\frac{1}{3}$ of the array the elements are sorted, so the array is indeed sorted at the end of the algorithm.

Problem 5

Given an array A holding n objects, we want to test whether there is a *majority* element; that is, we want to know whether there is an object that appears in more than $n/2$ positions of A.

Assume we can test equality of two objects in $O(1)$ time, but we cannot use a dictionary indexed by the objects. Your task is to design an $O(n \log n)$ time algorithm for solving the majority problem.

1. Show that if x is a majority element in the array then x is a majority element in the first half of the array or the second half of the array

2. Show how to check in $O(n)$ time if a candidate element x is indeed a majority element.

3. Put these observation together to design a divide an conquer algorithm whose running time obeys the recurrence $T(n) = 2T(n/2) + O(n)$

4. Solve the recurrence by unrolling it.

Solution:

1. If x is a majority element in the original array then, by the pigeon-hole principle, x must be a majority element in at least one of the two halves. Suppose that n is even. If x is a majority element at least $n/2+1$ entries equal x. By the pigeon hole principle either the first half or the second half must contain $n/4+1$ copies of x, which makes x a majority element within that half.

2. We scan the array counting how many entries equal x. If the count is more than $n/2$ we declare x to be a majority element. The algorithm scans the array and spends $O(1)$ time per element, so $O(n)$ time overall.

3. We break the input array A into two halves, recursively call the algorithm on each half, and then test in A if either of the elements returned by the recursive calls is indeed a majority element of A. If that is the case we return the majority element, otherwise, we report that there is "no majority element".

   To argue the correctness, we see if there is no majority element of A then the algorithm must return "no majority element" since no matter what the recursive call return, we always check if an element is indeed a majority element. Otherwise, if there is a majority element x in A we are guaranteed that one of our recursive call will identify x and the subsequent check will lead the algorithm to return x.

   Regarding time complexity, breaking the main problem into the two subproblems and testing the two candidate majority elements can be done in $O(n)$ time. Thus we get the following recurrence

   $$T(n) = 2T(n/2) + O(n).$$

4. $$T(n) = O(n \log n).$$

Problem 6

Suppose we are given an array A with n distinct numbers. We say an index i is locally optimal if $A[i] < A[i-1]$ and $A[i] < A[i+1]$ for $0 < i < n-1$, or $A[i] < A[i+1]$ for if $i = 0$, or $A[i] < A[i-1]$ for $i = n-1$.

Design an algorithm for finding a locally optimal index using divide an conquer. Your algorithm should run in $O(\log n)$ time.

Solution: First we test whether $i = 1$ or $i = n$ are locally optimal entries. Otherwise, we know that $A[1] > A[2]$ and $A[n-1] < A[n]$. If $n \leq 4$, it is easy to see that either $i = 2$ or $i = 3$ is locally optimal and we can check that in $O(1)$ time. Otherwise, pick the middle positions in the array for example $i = \lceil n/2 \rceil$ and test whether i is locally optimal. If it is we are done, other wise $A[i-1] < A[i]$ or $A[i] > A[i+1]$; in the former case we recurse on $A[1, \ldots, i]$ in the latter we recurse on $A[i, \ldots, n]$. Either way, we reduce the size of the array by half. Rather than creating a new array each time we recurse (which would take $O(n)$ time) we can keep the working subarray implicitly by maintaining a pair of indices (b, e) telling us that we are working with the array $A[b, \ldots, e]$. Thus, each call demands $O(1)$ work plus the work done by recursive calls. This leads to the following recurrence,

$$T(n) = T(n/2) + O(1),$$

which solves to $T(n) = O(\log n)$.

Problem 7

[**Hard**] Given two sorted lists of size m and n. Give an $O(\log(m+n))$ time algorithm for finding the kth smallest element in the union of the two lists.

Solution: **(Sketch.)** Let A be the larger array with n elements and B be the smaller array with m elements. Let us assume for simplicity that all numbers are distinct.
Let $rank(x)$ be the rank of x in the union of A and B. Notice that for each i we have $i \leq rank(A[i]) \leq i+m$. (Why?) Furthermore, if we compare $A[i]$ to $B[1]$, $B[m/4]$, $B[m/2]$, $B[3m/4]$, and $B[m]$ we can better estimate $i+a \leq rank(A[i]) \leq i+b$ where $b-a \leq m/4$. (Why?) Let us picture this estimate as an interval $I_i = [i+a, i+b]$.
Now observe that if $k < I_i$ (if k is smaller than the left endpoint of I_i) then $k < rank(A[i])$ so we can ignore every item larger than $A[i]$ from the large array and search for the kth element in the smaller instance. Similarly, if $k > I_i$ then $k > rank(A[i])$ so we can ignore everything smaller than $A[i]$ from the large array and search for the $(k-i)$th element in the smaller instance.
The only issue is what to do if $k \in I_i$. To get around this, we find the intervals of two indices. Notice that $rank(A[3n/4]) - rank(A[n/4]) \geq n/2$ and that $|I_{n/4}|, |I_{3n/4}| \leq m/4 \leq n/4$, therefore, $I_{n/4}$ and $I_{3n/4}$ must be disjoint and therefore either $k > |I_{n/4}|$ or $k < |I_{3n/4}|$. That means that we can always remove the elements that are either smaller or than $A[n/4]$ or larger than $A[3n/4]$.
For the time complexity we note that we do not really need to create a new array each time we recurse, we can simply keep two pairs of indices, one for A and one for B, that tell us the parts of the array where we are still searching on. Then, in each iteration we get rid of $n/4 \geq (m+n)/8$ elements in $O(1)$ time. Therefore, the combined length of the intervals is at most $\frac{7}{8}$ their previous combined length. Thus, if we let N measure the combined length of the intervals, we get the recurrence

$$T(N) = T(7N/8) + O(1),$$

which solves to $T(N) = O(\log N)$. Therefore, the algorithm runs in $O(\log(n+m))$ time.

Problem 8
[**Advanced**] Can you prove a lower bound on the approximation ratio of the greedy algorithm for Set Cover?

Problem 9
[**Advanced**] Can you prove a lower bound on the approximation ratio of the "MST-approach" (doubling the MST) for the Travelling Salesman Problem?