COL351 Holi2023: Tutorial Problem Set 2

1. Solve the following recurrences and express your answer using the $\Theta(\cdot)$ notation.

   (a) $T(n) = 2T(n/2) + n^2$.

   (b) $T(n) = 4T(n/2) + n$.

   (c) $T(n) = 4T(n/2) + n^2$.

   (d) $T(n) = 3T(n/2) + n$.

   (e) $T(n) = 7T(n/3) + n$.

   (f) $T(n) = 2T(\sqrt{n}) + \log_2 n$.

   A result popularly called the *master theorem* will provide answers to most of the above. You are advised to not use that theorem and try to solve by hand. Once you are done, read up the statement of the master theorem and prove it, using the intuition you built up while solving the above recurrences.

2. Let a, b, c, n_0, T_0 be positive constants such that $a + b < 1$. Prove that the solution to the recurrence

$$T(n) = \begin{cases} T(an) + T(bn) + cn & \text{if } n > n_0 \\ T_0 & \text{otherwise} \end{cases}$$

   satisfies $T(n) = O(n)$. (Find a constant c', possibly dependent on a, b, c, n_0, T_0 but independent of n, such that $T(n) \leq c'n$ for all $n > n_0$.)

3. [Kleinberg-Tardos Chapter 5 Exercise 6] Consider an n-node complete binary tree T, where $n = 2^d - 1$ for some d. Each node v of T is labeled with a real number x_v. You may assume that the real numbers labeling the nodes are all distinct. A node v of T is a *local minimum* if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge.

   You are given such a complete binary tree T, but the labeling is only specified in the following *implicit* way: for each node v, you can determine the value of x_v by *probing* the node v. Show how to find a local minimum of T using only $O(\log n)$ probes to the nodes of T.

4. You have two jobs, say J_1 and J_2, which you want to run on a total of $n+1$ processors in parallel. Jobs J_1 and J_2 take time a_k and b_k respectively to complete when run on k processors, where $a_1 \geq \cdots \geq a_n$ and $b_1 \geq \cdots \geq b_n$ (naturally, with more computational power, a job can only finish earlier). You may allocate m processors to run J_1 and the remaining $n+1-m$ processors to run J_2, where $m \in \{1, \ldots, n\}$, and this results in an overall completion time of $\max(a_m, b_{n+1-m})$.

   Since you don't know the values $a_1, \ldots a_n$ and $b_1, \ldots, b_n$, you wish to find out the best m^* which minimizes the overall completion time experimentally. In each experiment, you can run any one of the two jobs on any number m of machines, and measure its completion time (a_m or b_m). Design a strategy to determine m^* by performing $O(\log n)$ such experiments. Prove that your strategy indeed finds the correct m^*, and that it performs $O(\log n)$ experiments.

5. Design an $o(n^2)$-time algorithm which, given an array $A[1..n]$ of n numbers, finds the maximum possible sum of a sub-array of A. Formally, the algorithm is required to compute $\max_{i,j \in \{1,\ldots,n\}: i \leq j}(a_i + \cdots + a_{j-1})$. You might find several algorithms for this, but try to design one using the divide-and-conquer strategy. Specifically, figure out what extra work your recursive algorithm needs to do to make the "conquer" step faster.

6. Consider an n-node binary tree T where each edge e of T is labeled with a number x_e. Design an algorithm to find the maximum possible weight of a path in the tree, where the weight of a path is the sum of the weights of edges in the path. Your algorithm must run in time $O(n)$. (Observe that the previous problem is a special case of this problem where the tree is just a path.) Note that a path can contain only one node (and therefore, zero edges). For example, if the tree does not contain any edge e with $x_e > 0$, then the maximum weight path is the empty path with weight 0.

7. [Kleinberg-Tardos Chapter 5 Exercise 3, rephrased] Let A be an array of n objects. An object x is called a majority element of A if x occurs **strictly more than** $n/2$ times in A. Note that the objects need not be from an ordered set, so you can only check whether $A[i] = A[j]$ or not, for any two indices i, j. Design an $O(n \log n)$-time algorithm to determine whether an input array of size n has a majority element, and if so, to output that element. (By the way, if you think harder, you will actually get an $O(n)$-time algorithm.)