COL351: Analysis and Design of Algorithms

Tutorial Sheet - 7

October 17, 2022

Question 1 Design a divide-and-conquer algorithm to merge k sorted arrays, each with n elements, into a single sorted array of kn elements. What is the time complexity of this algorithm, in terms of k and n?

Solution

Algorithm 1: Merge$(A_1, \ldots, A_k)$

$B_1 = Merge(A_1, ..., A_{\frac{k}{2}})$;

$B_2 = Merge(A_{1+\frac{k}{2}}, ..., A_k)$ `/*` `size of` $B1$ `and` B_2 `is` $(nk/2)$ `*/`

B = Merge B_1 and B_2 in $O(nk)$ time by using two pointers as in merge-sort;

Return B;

Let $T(n,k)$:= Time to merge k arrays of size n.

$$\begin{aligned}
T(n,k) &= 2T\Big(n, \frac{k}{2}\Big) + cnk \\
&= 2\Big(2T(n, \frac{k}{4}) + cn\frac{k}{2}\Big) + cnk \\
&= 4T\Big(n, \frac{k}{4}\Big) + 2 \cdot cnk \\
&= \vdots \\
&= kT\Big(n, 1\Big) + \log_2 k \cdot cnk \\
&= O(nk \log_2 k)
\end{aligned}$$

Question 2 You are given an n-node **complete** binary tree T of height h, so $n = 2^h - 1$. The nodes of T are labelled with distinct real numbers. A node in T is a local minimum if its label is smaller than the label of its neighbours. Design an algorithm to find a local minimum of T in $O(\log n)$ time.

Solution

Algorithm 2: MINIMA-IN-SUBTREE(x)

if *(x is leaf)* **then** Return LABEL(x);

$L = x.left\text{-}child$;

$L = x.right\text{-}child$;

if $(\text{LABEL}(x) < \text{LABEL}(L), \text{LABEL}(R))$ **then** Return LABEL(x);

if $(\text{LABEL}(L) < \text{LABEL}(x), \text{LABEL}(R))$ **then** Return MINIMA-IN-SUBTREE(L);

if $(\text{LABEL}(R) < \text{LABEL}(x), \text{LABEL}(L))$ **then** Return MINIMA-IN-SUBTREE(R);

Time complexity is order of height of tree, i.e. $O(\log_2 n)$.

The correctness follows from the following lemma and corollary.

Lemma 1. *If* MINIMA-IN-SUBTREE(x) *is invoked then*

1. *either x is root, or*
2. LABEL$(x) <$ LABEL$(parent(x))$.

Corollary 1. *If* MINIMA-IN-SUBTREE(x) *is invoked then there exists a local minima in subtree rooted at x.*

Question 3 Given an n sized array A, the *Inversion Count* of A is the number of pairs (i, j) such that $A[i] > A[j]$ and $i < j$. So if A is already sorted, then the inversion count is 0, but if A is sorted in the reverse order, the inversion count is ${}^{n}C_2$. Design a divide-and-conquer algorithm to compute *Inversion Count* of an array A of size n in $O(n \log n)$ time.

Solution The following algorithm computes the number of inversions and sorts the array. The time complexity satisfies the relation $T(n) = 2T(n/2) + O(n)$, and thus $T(n) = O(n \log n)$.

Algorithm 3: InvCountNsort(A)

```
n ← LEN(A);
if n = 1 then Return 0;
Copy in B_1 the sub-array A[1, n/2];
Copy in B_2 the sub-array A[1 + n/2, n];
Ans = InvCountNsort(B_1) + InvCountNsort(B_2)  /* B_1, B_2 are sorted now */
x, y, pos ← 1;
while x ⩽ LEN(B_1) or y ⩽ LEN(B_2) do
    if B_1[x] ⩽ B_2[y] and x < LEN(B_1) then
        A[pos] ← B_1[x];
        Increment x, pos by 1;
    else
        A[pos] ← B_2[y];
        Increment y, pos by 1;
        Ans = Ans + (1 + LEN(B_1) − x);
        /* Added term= no. of elements in B_1 larger than B_2[y] */
    end
end
```

Question 4 Show that the randomized quick sort can be implemented by just using $O(1)$ extra space.

Solution Implementation of Randomized Quick Sort:

Algorithm 4: Randomized-Quick-Sort(A, L, R)

1 **if** $(R \leqslant L)$ **then** Return;

2 $q \leftarrow$ Random-index-from-interval($[L, R]$) `/* Pivot is` $A[q]$ `*/`

3 $k \leftarrow$ Partition(A, L, R, q);

4 Randomized-Quick-Sort$(A, L, k-1)$;

5 Randomized-Quick-Sort$(A, k+1, R)$;

Algorithm 5: Partition(A, L, R, q)

1 $k \leftarrow L$ + (No. of elements in $A[L, R]$ smaller than $A[q]$);

2 Swap(A, q, k) `/* Put pivot at correct index */`

3 **while** $(L < k < R)$ **do**

4   **while** $(A[L] < A[k])$ **do** $L = L + 1$;

5   **while** $(A[k] \leqslant A[R])$ **do** $R = R - 1$;

6   **if** $(L < k < R)$ **then** Swap(A, L, R);

7 **end**

8 Return k;

Main Idea We first put pivot at its final index. This is achieved by step 2. Next we keep two pointers L, R that scan sub-array from left and right end respectively, as follows:

- If $A[L] < A[k]$, we increment L.
- If $A[k] \leqslant A[R]$, we increment R.
- We perform a swap at indices L, R if we have $(A[L] \geqslant A[k] > A[R])$.

Question 5 Analyze the time complexity to compute Median of a list using Medians-of-Median algorithm (covered in Lecture 24) when the chunk size is (i) 3, and (ii) 7.

Solution
Let L be input list of size n. Suppose the chunk size is k, for some odd integer k. Then the new list U will have size n/k.

Let x be median of U.

Then in U, roughly $\frac{n}{2k}$ elements are larger (resp. smaller) than x.

This implies that in L, roughly $\frac{k+1}{2} \cdot \frac{n}{2k} = \frac{n(k+1)}{4k}$ elements are larger (resp. smaller) than x.

This implies that the size of list L_1 (L_2) is at most $\frac{n(3k-1)}{4k}$.

The recurrence relation for time complexity will be:

$$T(n) \;=\; T(n/k) \;+\; T\Big(\frac{n(3k-1)}{4k}\Big) \;+\; O(n)$$

Case k $=$ 3: We have $T(n) = T(n/3) + T(2n/3) + O(n)$, on solving gives $T(n) = O(n \log n)$.

Case k $=$ 7: We have $T(n) = T(n/7) + T(5n/7) + O(n)$, on solving gives $T(n) = O(n)$.