ESO207A: Data Structures and Algorithms

Practice Set *1: Divide and Conquer* –

Problem 1. Unimodal Array Maximum. You are given an array $A = [a_1, \ldots, a_n]$ of n distinct numbers that is *unimodal*, that is, for some index $p \in \{1, 2, \ldots, n\}$, the values in the values in the array entries increase up to index p and then decrease from index $p+1$ till index n. The problem is to find the peak entry p by reading as few array entries as possible. Show how to find the peak index p in time $O(\log n)$.

Note. 1. Use a divide and conquer approach, taking cue from the fact that the solution to the recurrence equation $T(n) = T(n/2) + \Theta(1)$ is $T(n) = O(\log n)$. Design an algorithm so that after a constant amount of work, you can discard one half of the current sub-array, as in binary search. 2. If you were to plot a unimodal array, with array indices j on the x-axis and array entries $A[j]$ on the y-axis, the plotted points will rise until x-value p where it attains a maximum, and then falls from thereon.

Solution Outline. First let us assume that the array is strictly unimodal, that is $2 \le p \le n-1$. This can then be extended for the other cases when the array is completely sorted.

In general and for $l < r$, let $[l \ldots r]$ be the array index segment such that the peak entry p lies in this segment. Initially $l = 1$ and $r = n$ satisfies the invariant. Let $m = (l+r)/2$ be the middle element. For now, assume $1 < m < n$. Now we check the values in the three consecutive indices $A[m-1], A[m]$ and $A[m+1]$. The following three cases (only) can occur since the array is unimodal.

1. If $A[m-1] < A[m]$ and $A[m] > A[m+1]$, then, m is the peak point and we are done.
2. If the three are in ascending order, $A[m-1] < A[m] < A[m+1]$, then, the peak point is in the range $[m+1 \ldots, r]$ and we recurse there.
3. If the three are in descending order $A[m-1] > A[m] > A[m+1]$, then, the peak point is in the range $[l, m-1]$ and we recurse there.

This is written below as pseudo-code.

```
FindPeakUniModal(A, l, r) // Assume A is strictly unimodal
 if  l == r return l
elseif  r == l + 1 {
          if A[r] > A[l] return r
          else return l
     }
else {
     m = (l + r)/2
     if A[m − 1] < A[m] and A[m] > A[m + 1]
          return m
     else if A[m − 1] < A[m] and  A[m] < A[m + 1]
          return  FindPeakUniModal(A, m + 1, r)
          return  FindPeakUniModal(A, l, m − 1)
```

Time Complexity: Lines 1, 2-5 and 7-9 spend $\Theta(1)$ time checking for termination cases. The array is split into at most 1/2 the size. Hence the recurrence relation is dominated by $T(n) = T(n/2)+\Theta(1)$, whose solution (by unrolling) is $T(n) = \Theta(\log n)$.

Problem 2. Counting Significant Inversions. Given an array $A = [a_1, a_2, \ldots, a_n]$ of n integers, we say that a pair (i, j) with $i < j$ is a *significant inversion* if $a_i > 2a_j$. Give an $O(n \log n)$ algorithm to compute the number of significant inversions in A.

Solution Outline. This is a direct modification of the counting inversions problem. As before, given $A[1 \ldots n]$ we will sort A in place and return a count of significant inversions. This is done by overloading Merge-Sort routine, particularly, the Merge routine. Consider the routine *Merge*(A, p, q, r). This assumes that $A[p \ldots q]$ is sorted and $A[q+1, \ldots, r]$ is sorted. The routine not only merges $A[p \ldots r]$ into a sorted array, but also counts the number of significant cross inversions, namely,

$$|\{(i, j) \mid p \leq i \leq q \text{ and } q + 1 \leq j \leq r \text{ and } A[i] \geq 2A[j]\}|$$

The only point to note is the following. As we are merging the lists, we keep an index i of the left list and j of the right list, as usual, where, $A[p \ldots, i - 1] \cup A[q + 1 \ldots j - 1]$ have been copied into the merged array. i is the smallest index of $A[p \ldots q]$ not yet merged and j is the smallest index of $A[q + 1 \ldots r]$ not yet merged. We also keep an index k, $i \leq k \leq q + 1$ and maintain the following invariant: k is the smallest index satisfying $A[k] > 2A[j]$.

Due to the sorted nature of the lists $A[p \ldots q]$ and $A[q+1 \ldots r]$, if $A[k] > 2A[j]$, then, $A[k+1] > 2A[j]$ and so on. So when $A[j]$ is copied into its proper position in the merged array, that is, $A[j] < A[i]$, it should contribute $q-k+1$ to the count of significant cross inversions, which is added to a running counter.

The invariant is easily maintained: initially, a scan is done starting at p through q to find the earliest index k such that $A[k] > 2A[i]$.

The number of inversions in the left list and the number of inversions in the right list are recursively computed while calling Merge-Sort on the left and right lists.

Problem 3. Median of union of two sorted arrays. Given two arrays $A = [a_1, a_2, \ldots, a_n]$ and $B = [b_1, b_2, \ldots, b_m]$ that are each individually sorted in increasing order. Assume that the numbers in $A \cup B$ are all distinct. Find the median of $A \cup B$ in time $O(\log n)$.

Solution outline. Let both arrays have the same size $m = n$. Also let m, n be a power of 2, for simplicity. Let $a_{n/2}$ and $b_{n/2+1}$ be the median elements of A and B respectively. There are 3 cases, either $a_{n/2} < b_{n/2+1}$ or, $a_{n/2} > b_{n/2+1}$ and $a_{n/2} = b_{n/2+1}$.

If $a_{n/2} = b_{n/2+1}$, then, $a_1, \ldots, a_{n/2-1}$ and $b_1, \ldots, b_{n/2}$ are each less than $a_{n/2}$. Thus is the union $A \cup B$, there are $n/2 + n/2 - 1 = n - 1$ elements less than $a_{n/2}$. Hence in the union array, $a_{n/2}$ has rank n when sorted, that is, it is the median.

Now suppose $a_{n/2} > b_{n/2+1}$. Then there are at most $a_{n/2+1}, \ldots a_n$ and $b_{n/2+2}, \ldots, b_n$ who are more than $a_{n/2}$. So $a_{n/2}$ has sorted rank at least $2n - n/2 - (n/2 - 1) = n + 1$ in the union $A \cup B$. Similarly, it is argued that the sorted rank of $b_{n/2+1}$ is at most $n + 1$. Thus, the actual median lies between $[b_{n/2+1}, a_{n/2}]$. Consider the subarrays $A' = A[1 \ldots n/2]$ and $B' = B[n/2 + 1, \ldots n]$.

Let m be the median of $A' \cup B'$. Then m has say rank $n/2$ and there are exactly $n/2 - 1$ elements in $A' \cup B'$ smaller than it and $n/2$ elements larger than it. Then, in $A \cup B$, there are $n/2$ additional elements $b_1, \ldots, b_{n/2}$ less than it. This makes its rank n, which means m is the median of $A \cup B$.

The *divide and conquer* step. In $\Theta(1)$ time, we have reduced A to A' and B to B' which are each half of the original array. Thus, we get the recurrence $T(n) = T(n/2) + \Theta(1)$ whose solution is $\Theta(\log n)$.

Problem 4. Hadamard Matrices. Hadamard matrices H_n are square $2^n \times 2^n$ matrices and are defined as follows.

1. H_0 is the 1×1 matrix $[1]$.
2. For $k \geq 1$, H_k is the $2^k \times 2^k$ matrix $H_k = \frac{1}{\sqrt{2}} \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix}$

Show that if v is a column vector of length $n = 2^k$, then the matrix-vector product $H_k v$ can be calculated in $O(n \log n)$ operations. Assume that the numbers in v are small enough so that basic arithmetic operations like addition and multiplication take unit time. (*Note.* An interesting property of the Hadamard matrices is that it is (real) orthonormal, that is, $H_k^T H_k = I$, analogous to the DFT matrix F_n.)

Solution Outline. Let $v = \begin{bmatrix} v_0 \\ v_1 \end{bmatrix}$ where v_0 and v_1 are each $n/2 = 2^{k-1}$ dimensional vectors. Then,

$$H_k v = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \end{bmatrix} = \begin{bmatrix} H_{k-1} v_0 + H_{k-1} v_1 \\ H_{k-1} v_0 - H_{k-1} v_1 \end{bmatrix}$$

We can compute $H_k v$ as follows.

1. Compute $x = H_{k-1} v_0$ and $y = H_{k-1} v_1$. This recursively takes time $2T(n/2)$.
2. Compute $x + y$ and $x - y$. This takes time $\Theta(n)$.

The recurrence relation is $T(n) = 2T(n/2) + \Theta(n)$.

Problem 5. Longest increasing contiguous subarray. Given an array $A[1, \ldots, n]$, a subarray $A[p \ldots q]$ is said to be an increasing contiguous subarray if $A[p] < A[p+1] < A[p+2] < \ldots < A[q]$ and is of length $q - p + 1$. The problem is to find the length of the *longest* increasing contiguous subarray. (*Note*: A divide and conquer approach similar to the maximum contiguous subarray sum problem can be designed to work in $O(n \log n)$ time.)

(*Note 2*: For $1 \leq i \leq n$, let $L[i]$ denote the length of the longest increasing contiguous subarray ending at i. Then, the following recurrence equation holds $L[i+1] = L[i] + 1$ if $L[i+1] > L[i]$; otherwise, $L[i+1] = 1$ (corresponding to the singleton subarray $[i+1, ..., i+1]$. This dynamic programming algorithm takes time $\Theta(n)$.

Solution Outline. The solution is similar to maximum contiguous subarray sum problem. Let $LIC(A, p, r)$ denote the longest increasing contiguous subarray (LICS) in $A[p \ldots r]$. A divide and conquer solution would be as follows. Let $q = (p + q)/2$ be the midpoint. The LICS for $A[p \ldots r]$ may either lie completely in $A[p \ldots q]$ or completely in $A[q + 1 \ldots r]$ or may overlap the border. Let *LICS-crossing*(A, p, q, r) denote the subroutine that finds the length of the LICS overlapping $A[q \ldots q + 1]$. Clearly, $A[q]$ must be $< A[q + 1]$, otherwise, this length is 0.

LICS-crossing(A, p, q, r) {
if $A[q] \geq A[q+1]$ **return** 0
$k = q$
while $k - 1 \geq p$ and $A[k-1] < A[k]$
    $k = k - 1$
leftlen $= q - k + 1$
$k = q + 1$
while $k + 1 \leq r$ and $A[k] < A[k+1]$
    $k = k + 1$
rightlen $= q + 1 - k + 1$
return *leftlen* + *rightlen* + 1
}

Problem 6. Divide and Conquer: Monge Arrays [Problem 4-6 from CLRS.] An $m \times n$ array A of real numbers is a *Monge array* if for all i, j, k and l such that $1 \leq i < k \leq m$ and $1 \leq j < l \leq n$, we have,

$$A[i,j] + A[k,l] \leq A[i,l] + A[k,j] \ .$$

In other words, whenever we pick two rows and two columns of a Monge array and consider the four elements at the intersections of the rows and columns, the sum of the upper-left and lower-right elements is less than or equal to the sum of the lower-left and upper-right elements.

a. Prove that an array is Monge if and only if for all $i = 1, 2, \ldots, m-1$ and $j = 1, 2, \ldots, n-1$, we have,

$$A[i,j] + A[i+1, j+1] \leq A[i, j+1] + A[i+1, j]$$

(*Hint*: For the "if" part, use induction separately on rows and columns.)

Soln outline. The "only if" part is obvious. Let us prove the Monge array property by induction on $l - j$. For $= 1$, this is the base case and is given. Suppose, we have for a fixed k that $A[i,j] + A[i+1, j+l] \leq A[i+1, j] + A[i, j+l]$. By assumption, we have, $A[i, j+l] + A[i+1, j+l+1] \leq A[i+1, j+l] + A[i, j+l+1]$. Adding both, we have,

$$\begin{aligned} A[i,j] + A[i+1, j+l] + A[i, j+l] + A[i+1, j+l+1] \\ \leq A[i+1, j] + A[i, j+l] + A[i+1, j+l] + A[i, j+l+1] \end{aligned}$$

or, by cancellation, we get $A[i,j] + A[i+1, j+l+1] \leq A[i+1, j] + A[i, j+l+1]$. This proves the induction case. Hence, we have proved that for any $1 \leq i \leq m-1$ and $1 \leq j < l \leq n$ that $A[i,j] + A[i+1, l] \leq A[i+1, j] + A[i, l]$. Now extend this by inducing on the gap between the row indices $k - i$. This will complete the proof of the property.

b. Let $f(i)$ be the index of the column containing the leftmost minimum element of row i. Prove that for any $m \times n$ Monge array, $f(1) \leq f(2) \leq \cdots \leq f(m)$.

Soln outline. By way of contradiction, let $f(i) > f(i+1)$. By the Monge property,

$$A[i, f(i+1)] + A[i+1, f(i)] \leq A[i+1, f(i+1)] + A[i, f(i)]$$

Now, since $f(i)$ is the left most minimum element of the ith row, we have, $A[i, f(i+1)] > A[i, f(i)]$. Also, since $f(i+1)$ is the left most min element of $i+1$st row, $A[i+1, f(i)] \geq A[i+1, f(i+1)]$, Adding, we have,

$$A[i, f(i+1)] + A[i+1, f(i)] > A[i+1, f(i+1)] + A[i, f(i)]$$

which contradicts the Monge property. By way of contradiction, this proves the property.

c. The following describes a divide-and-conquer algorithm that computes the leftmost miminum element in each row of an $m \times n$ Monge array A:

> Construct a submatrix A' of A consisting of the even-numbered rows of A. Recursively determine the leftmost minimum for each row of A'. Then compute the leftmost minimum in the odd numbered rows of A.

Explain how to compute the leftmost minimum in the odd-numbered rows of A (given that the leftmost minimum of the even-umbered rows is known) in $O(m+n)$ time.

Soln. outline Suppose we determine the left most minimum for each even numbered row of A. Thus, we have computed $f(2) \leq f(4) \leq \ldots f(m)$, assuming m is even.

To determine $f(1)$, find the left most minimum between $1 \ldots f(2)$, for $f(3)$, find the leftmost minimum between $f(2)$ and $f(4)$ and so on. The time taken is $f(2) - 1 + 1 + (f(4) - f(2) + 1) + \ldots + (f(m) - f(m-2) + 1) + (n - f(m) + 1) = \Theta(m+n)$.

d. Write the recurrence describing the running time of the algorithm described in part (d). Show that its solution is $O(m + n \log m)$.

Soln. The recurrence relation is $T(m) = T(m/2) + \Theta(m+n)$, with $T(1) = \Theta(n)$. The solution to this is $\Theta(n \log m)$.

Problem 7. Closest pair of points in 2-dimensions. Given n points in the plane, the problem is to find the pair that is the closest in terms of Euclidean distance. Design an $O(n \log n)$ algorithm for this problem.

Note. You may assume that each point is a pair (x, y) and has an id (between $1, 2, \ldots, n$). Let P be an array of points. Assume also that the Euclidean distance between two points can be calculated in constant time. Note that an $O(n^2)$ algorithm is trivial, since one checks the distance between all $\binom{n}{2}$ pairs of points and returns the pair that has the minimum distance. The $O(n \log n)$ time algorithm is a divide and conquer algorithm with a careful geometric analysis for the combine phase. A solution may be found in the text CLRS Section 33.4.

The overall approach is the following. Let P_x be a copy of P sorted on the x-coordinate and P_y be a copy of P sorted on the y coordinate. The two copies are mutually synced in the following sense (or an equivalent implementation of it). For every point p in P_x, there is a field of p that points to the copy of p in P_y and vice-versa. Partition P by the x coordinate into two equal halves Q (the left half) and R (the right half). Q contains the first $n/2$ (actually, $\lceil n/2 \rceil$) points (by x-coordinate) of P_x and R contains the remaining $n/2$ ($\lfloor n/2 \rfloor$) points. Now in $O(n)$ time, using the mutual syncing of P_x and P_y, in $O(n)$ time, we can create four lists Q_x and Q_y, which are points in Q sorted by x and y coordinates respectively and mutually synced, and lists R_x and R_y, which are points in R sorted by x and y coordinates respectively and mutually synced. Let L be a vertical line passing through the $\lceil n/2 \rceil$ th point, that is the point with the highest x-coordinate in Q.

Now recursively determine the closest pair of points in Q (using the lists Q_x and Q_y) and R respectively. Let q_0^*, q_1^* be returned as the closest pair in Q an r_0^*, r_1^* be returned as the closest pair in R. Let δ be the minimum of $d(q_0^*, q_1^*)$ and $d(r_0^*, r_1^*)$. This value of δ is the closest pair distance unless there is a closer pair (q, r), $q \in Q$ and $r \in R$, where, $d(q, r) < \delta$. We only need to look for "cross-pairs" (q, r).

Show that it suffices to look at only point pairs (q, r) that lie in a 2δ-band, where, $q \in Q$ is at a distance of at most δ from L (to its left) and $r \in R$ is a within at most a distance of δ from L (to its right).

We can now delete from Q all points that are at a distance greater than δ to the left of the line L and similarly, drop all points of R that are at a distance greater than δ to the right of L. This can be done in time $O(n)$ and leaves the remaining points in Q and R sorted by y coordinate. Let $S = Q \cup R$, after the deletions have been done. Merge S into a single sorted list by y-coordinate in $O(n)$ time.

Now (the meticulous part), show that if $s, s' \in S$ satisfying $d(s, s') < \delta$, then, s and s' are within 15 positions of each other in the sorted list S_y. (Here, 15 is used to refer to an absolute constant, CLRS argues it down to 7). Hence, the merging can be done in time $O(n)$.