1. (a) Given n elements and $n^{3/2}$ CRCW processors, show how to compute the minimum in $O(1)$ time.

Solution: In order to compute minimum of n elements in $O(1)$ time, we need $O(n^2)$ CRCW processors. So we partition n elements into $\sqrt{n}$ blocks each consisting of $\sqrt{n}$ elements. We assign $O(n)$ processors to each block and compute minimum of each block in $O(1)$ time. Total number of processors needed for this step are $n \times \sqrt{n} = n^{3/2}$.

Now we have $O(\sqrt{n})$ candidate elements and minimum of these can be computed using $O(n)$ processors in $O(1)$ time.

(b) Extend the previous idea to compute minimum of n elements in $O(1)$ time using $n^{1+\epsilon}$ CRCW processors for any $0 < \epsilon < 1$.

Solution: The main issue here, as was in the above problem, is that we do not have enough processors to do it in one parallel step. So we partition the elements into blocks such that all the blocks can be processed in $O(1)$ parallel time. Then for the next round we have fewer candidate elements and we again partition the elements into blocks of manageable size. The termination condition is when number of processors is square of candidate elements and hence we can compute minimum in one step. The number of parallel steps required should be function of ϵ.

We partition n elements into $n^{1-\epsilon}$ blocks each consisting of n^{ϵ} elements. We assign $O(n^{2\epsilon})$ processors to each block and compute minimum of each block in $O(1)$ time. Total number of processors needed for this step are $n^{2\epsilon} \times n^{1-\epsilon} = n^{1+\epsilon}$.

For the j^{th} round we have $n^{1-(2^j-1)\epsilon}$ blocks with $n^{2^{j-1}\epsilon}$ elements each.

Total number of rounds: Let $(j+1)^{st}$ be the round at which square of number of elements is same as the total number of processors. That is,

$$n^{2\times(1-(2^j-1)\epsilon)} = n^{1+\epsilon}$$

The number of rounds are $\log_2(\frac{1}{\epsilon}+1)$ which is $O(1)$ as ϵ is a constant.

2. Given an array of n elements $a_1, a_2 \ldots a_n$, the *nearest smaller value* of any element a_i is defined as

$$NSV(a_i) = argmin_{j>i}\{a_j < a_i\}$$

The all nearest value problem (ANSV) is to compute for each element a_i, its *nearest smaller value*.

(a) Design a linear time sequential algorithm for ANSV.

Solution:

procedure NearestSmallerValue(a_i, j)
  if $j = -1$ **then**
    $NSV(i) \leftarrow -1$
  else if $j = n+1$ **then**
    $NSV(i) \leftarrow -1$
  else if $a_i > a_j$ **then**
    $NSV(i) \leftarrow j$
  else if $a_i = a_j$ **then**

```
        NSV(i) ← NSV(j)
    else if a_i < a_j then
        NearestSmallerValue(a_i, NSV(j))
    end if
end procedure

function Main(array a)
    for i ← n to 1 do
        NearestSmallerValue(a_i, i + 1)
    end for
end function
```

Proof of Correctness: We prove by contradiction. Let a_i be an element such that $\forall\, l\ (i < l \leq n)$, $NS(l)$ is correct. Let $NS(i) = k$ and there exists an element a_j such that $j < k$ and $a_j < a_i$. That is $NS(i)$ should have been j. Now

$$
\begin{array}{lll}
 & a_i < a_l, \quad \forall\, l \quad i < l < j & \text{(By definition of NS)} \\
 & a_j < a_l, \quad \forall\, l \quad i < l < j & \text{(as } a_j < a_i\text{)} \\
\implies & NS(l) \leq j & \\
\implies & NS(i+1) \leq j & \\
\implies & NS(i) \leq j & \text{(By our Algorithm)} \\
\implies & k \leq j & \text{(Contradiction)}
\end{array}
$$

■

Analysis: Consider two elements a_i and a_j such that $i < j$. So $NSV(j)$ is computed before $NSV(i)$.

Claim: Only one of the recursive calls NearestSmallerValue(a_i, k) or NearestSmallerValue(a_j, k) is possible while computing $NSV(i)$ and $NSV(j)$ respectively.

Proof by Contradiction:
Assume both the calls occured and while computing $NSV(i)$ we compare a_i to a_l and $NSV(l) = k$. So the next recursive call is NearestSmallerValue(a_i, k), therefore $a_k < a_l$. Also $a_l \leq a_j$ else $NSV(l) = j$. Therefore $a_k < a_l \leq a_j$. Due to the call NearestSmallerValue(a_j, k), $a_j < a_k$ and hence $a_j < a_k < a_l \leq a_j$. A Contradiction. ■

Since each element is part of exactly one recursive call, the running time of the algorithm is $O(n)$.

(b) Design a polylog time $O(n)$ processors CRCW PRAM algorithm for ANSV problem.

Solution:
The procedure NearestSmallerValueParallel divides the array into two equal halves and computes NSV and *Prefix Minimum* of the two halves recursively and in parallel. The NSV of the two halves is then combined. NSV of the right half is fixed as the elements in the left half does not affect it by definition of NSV. There may be some elements in left half whose NSV does not exist in the left half but may exists in right half. Here we use the $PrefixMin$ values of the elements of right half to search for NSV of these elements.

Assume number of elements is a power of 2.

```
1: procedure NearestSmallerValueParallel(array a, index i, index j)
2:     if j = i then
3:         NSV(i) ← −1
4:     else if j = i + 1 then
5:         if a_j < a_i then
6:             NSV(i) ← j
7:         else
8:             NSV(i) ← −1
9:         end if
10:        NSV(j) ← −1
11:    else
12:        mid ← (i + j)/2
13:
14:        do in parallel
15:            NearestSmallerValueParallel(a, i, mid)
16:            NearestSmallerValueParallel(a, mid + 1, j)
17:        end parallel
18:
19:        do in parallel
20:            PrefixMin[i ··· mid] ← PrefixMinimumParallel(a, i, mid)
21:            PrefixMin[mid + 1 ··· j] ← PrefixMinimumParallel(a, mid + 1, j)
22:        end parallel
23:
24:        ▷ Form PrefixMin of the array a[i ··· j]
25:        for k ← mid + 1 to j do in parallel
26:            if PrefixMin[k] > PrefixMin[mid] then
27:                PrefixMin[k] ← PrefixMin[mid]
28:            end if
29:        end for
30:
31:        ▷ Combining the results of two halves to compute NSV of a[i ··· j].
32:        for k ← i to mid do in parallel
33:            if NSV(k) = −1 then
34:                BinarySearch(a, k, mid + 1, j)
35:            end if
36:        end for
37:
38:    end if
39: end procedure
40:
41: procedure BinarySearch(array a, index k, index i, index j)
42:    if j = i + 1 then
43:        if PrefixMin(i) = a_i ∧ a_i < a_k then
44:            NSV(k) ← i
45:        else if PrefixMin(j) = a_j ∧ a_j < a_k then
46:            NSV(k) ← j
47:        end if
48:    else
49:        mid ← (i + j)/2
50:        if PrefixMin(mid) < a_k then
```

```
51:             j ← mid
52:         else if PrefixMin(mid) ≥ a_k then
53:             i ← mid
54:         end if
55:         BinarySearch(a, k, i, j)
56:     end if
57: end procedure
58:
```

Proof of Correctness: We will prove by induction on the size of the array n.

- *Base Case:* If $n = 1$ then due to line 2, the $NSV(i)$ is set to -1, which is correct. If $n = 2$ then due to line 4, the $NSV(i)$ is set to j if $a_j < a_i$ else it is set to -1.
- *Inductive Hypothesis:* Assume NSV of the two sub-arrays $a[1 \cdots mid]$ and $a[mid{+}1 \cdots n]$, both of size n, is correct.
- *Inductive Step:* Consider an array of size $2n$. Algorithm divide the array into two equal size sub-arrays and compute the NSVs of the two sub-arrays independently. It then combines the two NSVs array to form a final solution. The NSV values of the elements in right sub-array will remain same in the final solution. Consider an element a_k in left sub-array such that $NSV(k) = -1$. Let there be an element a_l in right half such that $NSV(k) = l$. The standard BinarySearch procedure with $PrefixMin$ values as key will set the $NSV(k)$ to l. It is trivial to see that if condition at line 50 is satisfied then $NSV(k)$ is to left of the mid else its on $right$. The correctness of the BinarySearch procedure can be established by simple inductive argument on the size of the array.

■

Analysis: The $PrefixMin$ computation of n elements takes $O(log(n))$ time. The modification of the $PrefixMin$ values of the right half in parallel loop at line 25 takes $O(1)$ time. The BinarySearch procedure takes $O(log(n)$ time and for element in left half it is called in parallel. So the recurrence equation is

$$T^{\|}_{ANSV}(n, n) = T^{\|}_{ANSV}(n/2, n/2) + O(log(n/2))$$

Therefore, $T^{\|}_{ANSV}(n, n) = O(log(n))$

3. (a) Show how to obtain a better processor-time bound for the two versions of the prefix computation. Recall that the first algorithm uses $n \log n$ processors and the second one uses n processors to obtain the same parallel time bound of $O(\log n)$.

Solution: Given n elements $a_1, a_2, \cdots a_n$, perform prefix computation to obtain $S_1, S_2, \cdots S_n$ where $S_i = \bigodot_{k=1 \text{ to } i} a_k$. Using $n \log n$ processors:

1. Divide n elements into $\frac{n}{log(n)}$ blocks each of size $log(n)$.
2. For each block, perform partial prefix computation using a single processor. Let partial prefix computation values of i^{th} block be $S^i_1, S^i_2, \cdots S^i_{log(n)}$. Number of processors needed for this step are $\frac{n}{log(n)}$ and time required is $O(log(n))$.
3. So final $S_{(i-1)\,log(n)+k} = S^i_{(i-1)\,log(n)+k} \odot \bigodot_{j=2}^{i} S^{j-1}_{log(n)}$. The second term can be thought

of as a prefix computation value performed with $S^1_{log(n)}, S^2_{log(n)} \cdots S^{n/log(n)}_{log(n)}$ as elements. These prefix computation values can be performed using $O(\frac{n}{log(n)} log(\frac{n}{log(n)}))$ processors and time required is $O(log(\frac{n}{log(n)}))$.

4. Each processor can update the partial prefix computation values of its block to obtain final $S_{(i-1)\,log(n)+k} = S^i_{(i-1)\,log(n)+k} \odot \bigodot_{j=2}^{i} S^{j-1}_{log(n)}$. Number of processors needed are $\frac{n}{log(n)}$ and time required is $O(log(n))$.

Processor-Time Bound:

$$\begin{aligned} Number\ of\ processors &= \frac{n}{log(n)} + \frac{n}{log(n)} log(\frac{n}{log(n)}) + \frac{n}{log(n)} \\ &= O(n) \end{aligned}$$

$$\begin{aligned} Time\ taken &= log(n) + log(\frac{n}{log(n)}) + log(n) \\ &= O(log(n)) \end{aligned}$$

Therefore **Processor-Time bound** is $O(n\,log(n))$.

Using n processors: Step 3 of the above algorithm to perform prefix computation over the partial sums can be done using $O(n)$ processors and $O(log(n))$ time version algorithm. So the required number of processors

Processor-Time Bound:

$$\begin{aligned} Number\ of\ processors &= \frac{n}{log(n)} + \frac{n}{log(n)} + \frac{n}{log(n)} \\ &= O(\frac{n}{log(n)}) \end{aligned}$$

$$\begin{aligned} Time\ taken &= log(n) + log(\frac{n}{log(n)}) + log(n) \\ &= O(log(n)) \end{aligned}$$

Therefore **Processor-Time bound** is $O(n)$.

(b) Generalize the technique of clubbing k (a parameter between 1 and n) contiguous values, compute the prefix recursively and then generate the missing values as a function of k and n.

Solution:

1. Divide n elements into $\frac{n}{k}$ blocks each of size k.
2. For each block, perform partial prefix computation using a single processor. Let partial prefix computation values of i^{th} block be $S^i_1, S^i_2, \cdots S^i_k$. Number of processors needed for this step are $\frac{n}{k}$ and time required is $O(k)$.

3. So final $S_{(i-1)\,k+j} = S^i_{(i-1)\,k+j} \odot \bigodot_{l=2}^{i} S_j^{l-1}$. The second term can be thought of as a prefix computation value performed with $S_k^1, S_k^2 \cdots S_k^{n/k}$ as elements. These prefix computation values can be performed recursively. The time required is $T(n/k)$.

4. Each processor can update the partial prefix computation values of its block to obtain final $S_{(i-1)\,k+j} = S^i_{(i-1)\,k+j} \odot \bigodot_{l=2}^{i} S_j^{l-1}$. Number of processors needed are $\frac{n}{log(n)}$ and Time required is $O(k)$.

Time Analysis:

$$\begin{aligned} T(n) &= O(k) + T(n/k) + O(k) \\ &= O(k\, log_k n) \end{aligned}$$

4. Show how to sort n integers in the range $[1 \cdots \sqrt{n}]$ using $\sqrt{n}$ processors in $O(\sqrt{n})$ parallel steps. Specify which PRAM model is used.

Solution:

1. Divide n elements into $\sqrt{n}$ blocks each of size $\sqrt{n}$ and assign each block to a processor.
2. Each processor p_i performs a single pass over its block to count number of occurrences of each integers in the range $[1 \cdots \sqrt{n}]$. Let C_i^j be the number of occurrences of integer i in block j. Computing C_i^j's value of a block takes $O(\sqrt{n})$ time.
3. Final count of the number of occurrences of an integer is the sum of its number of occurrences in each block computed in above step. So the number of occurrences of integer i is $C_i = \sum_{j=1}^{\sqrt{n}} C_i^j$. Computing all C_i's takes $O(\sqrt{n})$ time.
4. Sorted sequence can be obtained by computing prefix sum over C_i's. This step also takes $O(\sqrt{n})$ time using single processor.

Time taken is $O(\sqrt{n})$ and the **Number of processors** required are also $O(\sqrt{n})$.
PRAM model used here is EREW as each of the C_i^j's and C_i's are written and read exclusively by single processor at a time.