1. Write pseudocode for a non-recursive pre_x-sums algorithm that is similar to the one studied in class but that does not use the auxiliary variables B and C. The input array A should hold the pre_x sums when the algorithm terminates.

Solution:

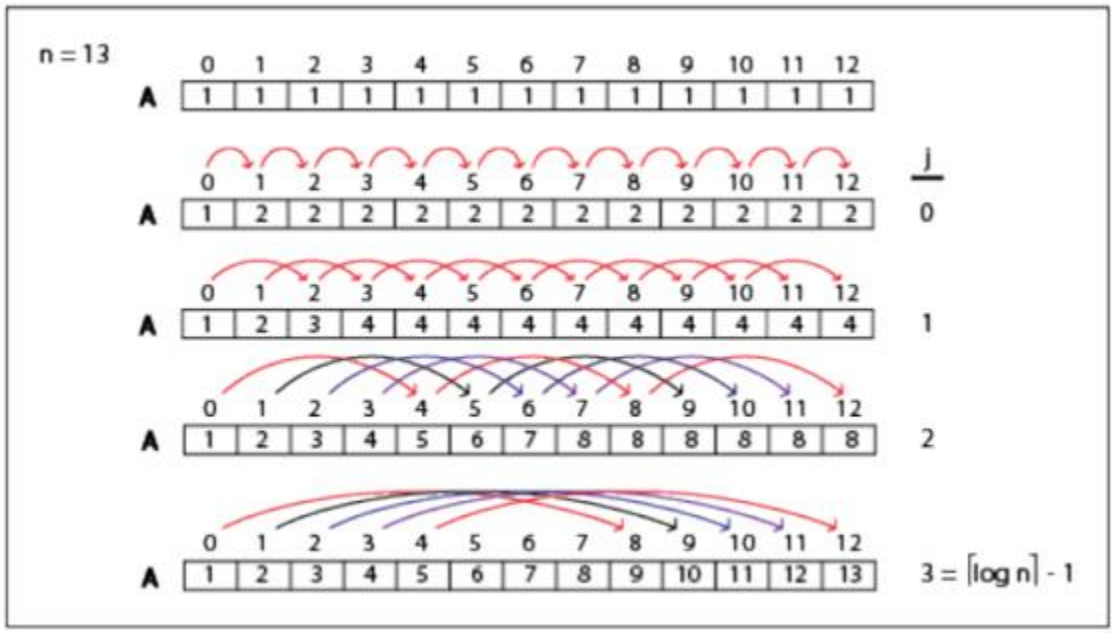
Given an array A, with n elements, the prefix sums algorithm returns an array B, such that:

$$B[i] = \sum_{k=1}^{i} A[k]$$

In other words, each element of B in position I, is equal to the sum of all the elements from 0 to i in A.

Prefix sums inherently seems like an iterative problem since each entry depends upon those that come before it. However, we want to be able to perform this task in parallel. We assume that the input array has a size of 2n + 1 and has index positions i, for -n <= i < n, where elements in position i < 0 are equal to zero.

```
Algorithm: ParallelInPlacePrefixSums(A[0....(n-1)])
-----------------------------------------------------
begin
    for j = 0 to ceil(logn - 1) do
        for i = 2 ^ j to n - 1 in pardo
            return A[i] = A[i] + A[i - 2 ^ j]
end
```



Let's say that we have the input array A, with size, n = 14, and the elements initialized to one. In the first iteration, j = 0, and the elements with indices i, from $2^j = 2^0 = 1$ to n – 1, get the sum of its own value and that of the element to its left. On the next iteration, j = 1, and now the elements from 2 <= i <= n – 1, get the sum of its own value with that of the element 2 indices away on the left. We continue this until j = ceil (log n) – 1. In our case, this is when j = 3.

We can see that the runtime for the inner for loop will be constant time since the operation is done by all the processors in parallel. The outer for loop runs from zero to logn – 1, and so, requires logn time to execute. This tells us that our runtime is T(n) = O(logn)

To calculate work, we need to include the steps done in the inner for loop. We know that I goes from 2 ^ j to n – 1 while the outer for loop goes from j = 0 to log n. So we can describe work as follows:

$$W(n) = \sum_{j=0}^{\log n - 1} \sum_{i=2^j}^{n-1} 1$$

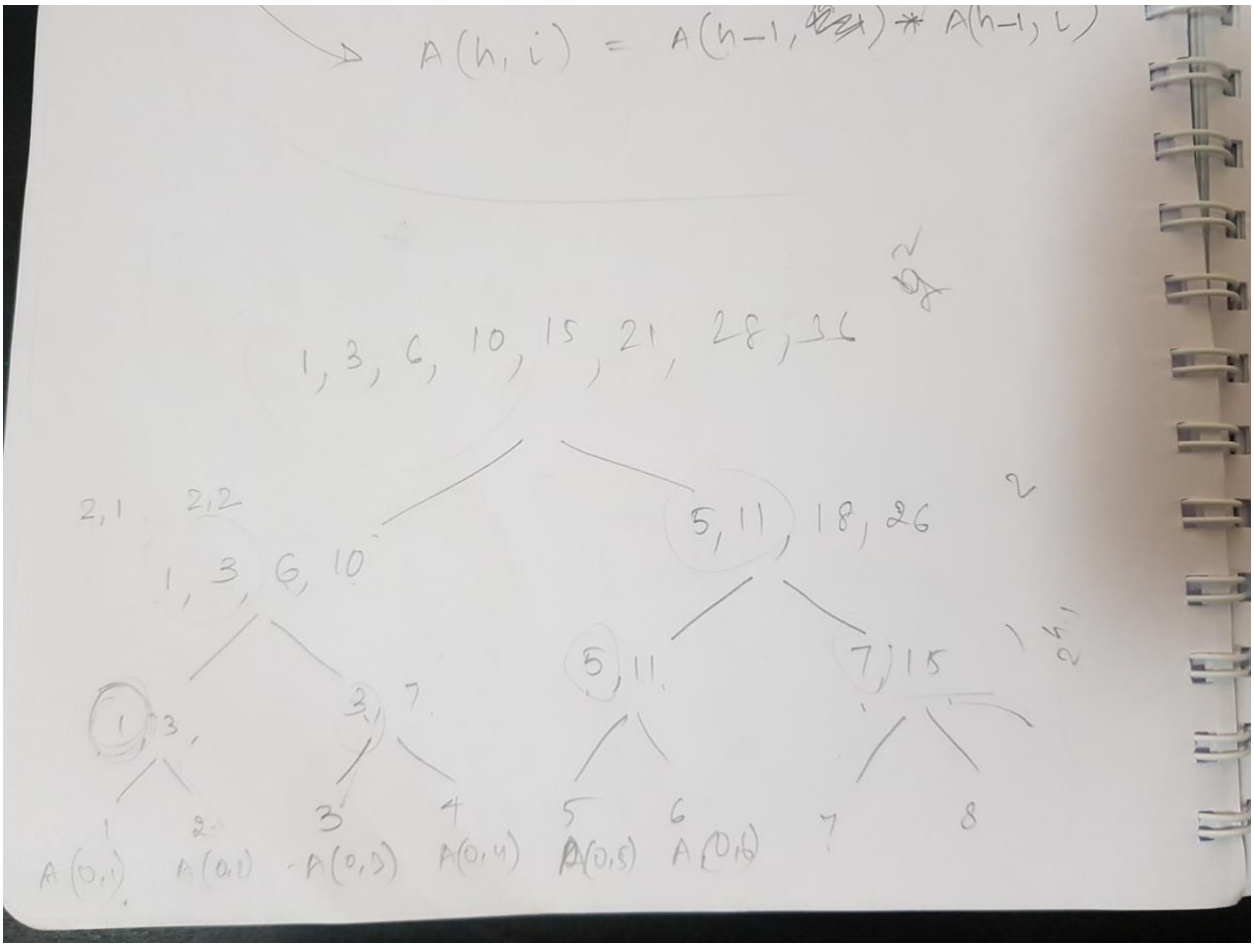
$$= n \log n - \sum_{j=0}^{\log n - 1} 2^j$$

$$= n \log n - (n - 1)$$

$$= \theta(n \log n)$$

So, the work is equal to $\Theta(n \log n)$

Alternative Solution:




```
Algorithm: ParallelInPlacePrefixSums(A[0][0....(n - 1)])
-----------------------------------------------------------
begin
    for h = 1 to log n do
        for 1 <= j <= n/2 ^ h pardo
            for i = 1 to 2 ^ h - 1 in A do
                // set 2 ^ h - 1 elements same from previous step
                A(h, i) = A(h - 1, i)
            for i = 2 ^ h - 1 to 2 ^ h in A do
                // Every next set added to the last element of the previous step
                A(h, i) = A(h - 1, 2 ^ h - 1) * A(h - 1, i)
end
```

In the above approach, we make use of the height of the tree to calculate the prefix sum. We set the elements to same values for half of the elements in the set of elements for that height. The rest half of the elements will be replaced by adding the last element value from previous set to each element of the next set.

At the end of the execution of the algorithm the variable A holds prefix sums of the given array.

The running time of the algorithm is O(n log n) as there are two for loops effectively.

The work done is $W(n) = O(\{\log n(2n^2)\}) = O(n^2 \log n)$

2. We are given an array of colors $A = [a_1; a_2; : : : ; a_n]$ drawn from k colors $fc_1; c_2; : : : ; c_k g$, where k is a constant. We wish to compute k indices $i_1; i_2; : : : ; i_k$, for each element a_i, such that i_j is the index of the closest element to the right of a_i whose color is c_j. If no such element exists, then set $i_j = 0$. Write pseudocode for solving this problem in O(log (n)) using a total of O(n) operations.

Solution:

Let c(i) = i be the initial coloring assigned to the vertices of the directed cycle. For each vertex, we assign a color value based on the significant bit(binary value). The number of colors is reduced from 15 to six.

By the new coloring scheme, no two adjacent vertices will have the same color. No two vertices of the same color are adjacent. We can solve the problem in more optimal and efficient manner by just using 3 colors and in time O(log(n)) and total of O(n) operations.

Input: An array A with n elements

Output: 3-coloring of the elements

begin

1. for 1<= i < n pardo
   Set C(i): = i
2. Apply Basic Coloring Algorithm once
3. Sort the vertices by their colors
4. for i = 3 to 2logn do
   for all elements ai of color i pardo

   Color ai with the smallest color from {0,1,2} that is different from the colors of its two neighbors.

end

After sorting the vertices by their colors, we can assume that all the vertices with the same color are in consecutive memory locations, and that we know the locations of the first and the last vertices of each color. Let n_i be the number of vertices of color i. Recoloring these vertices

takes O(1) parallel time, using O(n) operations. Therefore assignment of colors from {0, 1, 2} takes O(log n) time, using a total of O(n) operations.

3. Suppose that we have an algorithm A to solve a given problem P of size n in O(log (n)) time on the PRAM model using O(n log (n)) operations. On the other hand, an algorithm B exists that reduces the size of P by a constant fraction in $O(_{\log (n)/\log \log (n))}$ time using O(n) operations without altering the solution. Derive an O(log (n)) time algorithm to solve P using O(n) operations.

Solution:

Strategy A works in O(n log n) operations with a complexity of O(log n) – which concludes that the algorithm is running using a divide and conquer strategy on n processors using a shared memory

Strategy B runs in $O(_{\log (n)/\log \log (n))}$ time using O(n) operations

To prove that there exists an algorithm that runs in O(log n) using O(n) operations. It can proved if we can show that $O(_{\log (n)/\log \log (n))}$ is of O(log n)

Proof:
Suppose f(n) = O(log (n)/ log log n)
and we want to prove that f(n)=O(log n).

Assume f(n) is a positive function. By the definition of the big O notation, f(n) = O(log (n)/ log log n) implies that there exists a N_0 and a positive constant k such that

$f(n) \leq k \cdot \log (n)/\log \log n, \forall\ n \geq N_0$

Since log (n)/log log n < = log n; for sufficiently large n, there must exist a N_1 such that
$f(n) \leq k \cdot \log n, \forall\ n \geq N_1$

thus f(n) = O(log n) and there exists an algorithm that runs in O(log n) using O(n) operations to solve P.
Alternative Proof:

We have to prove that f = O(logn) <=> (log n)/ (log(log n)) = O(logn)
So, we need to find c and N_0 such that $0 \leq (\log n) / (\log (\log n)) \leq c * \log n$ for all $n \geq N_0$. Let's suppose that the logarithm base is b (it doesn't matter, but you can consider b in {2, e, 10}). If you choose c = 1 and $N_0 = b \wedge b \wedge 2$, $0 \leq (\log n)/ (\log (\log n)) \leq \log n$ for all $n \geq b^{b^2}$.

- the first part is true, because $\log n \geq \log b^{b^2} = b^2 \geq 0$ and $\log (\log n) \geq \log (\log b^{b^2}) = 2 \geq 0$
- the second part is also true, because it becomes $\log (\log n) \geq 1$ and $\log (\log n) \geq \log(b^2) = 2 \geq 1$.