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.**

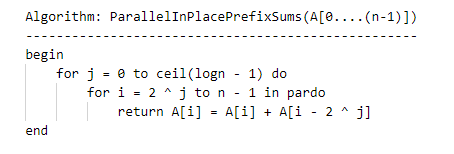
**Solution:**

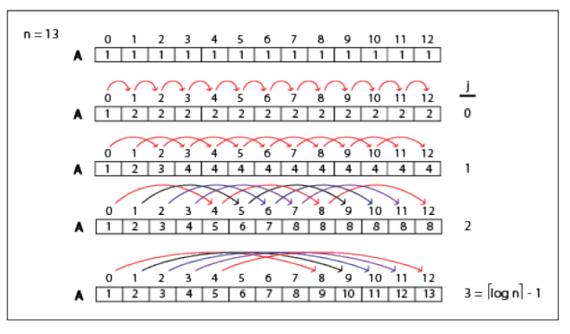
Given an array A, with n elements, the prefix sums algorithm returns an array B, such that:

B[i] = ∑­­ik = 1A[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.





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 2j = 20 = 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:

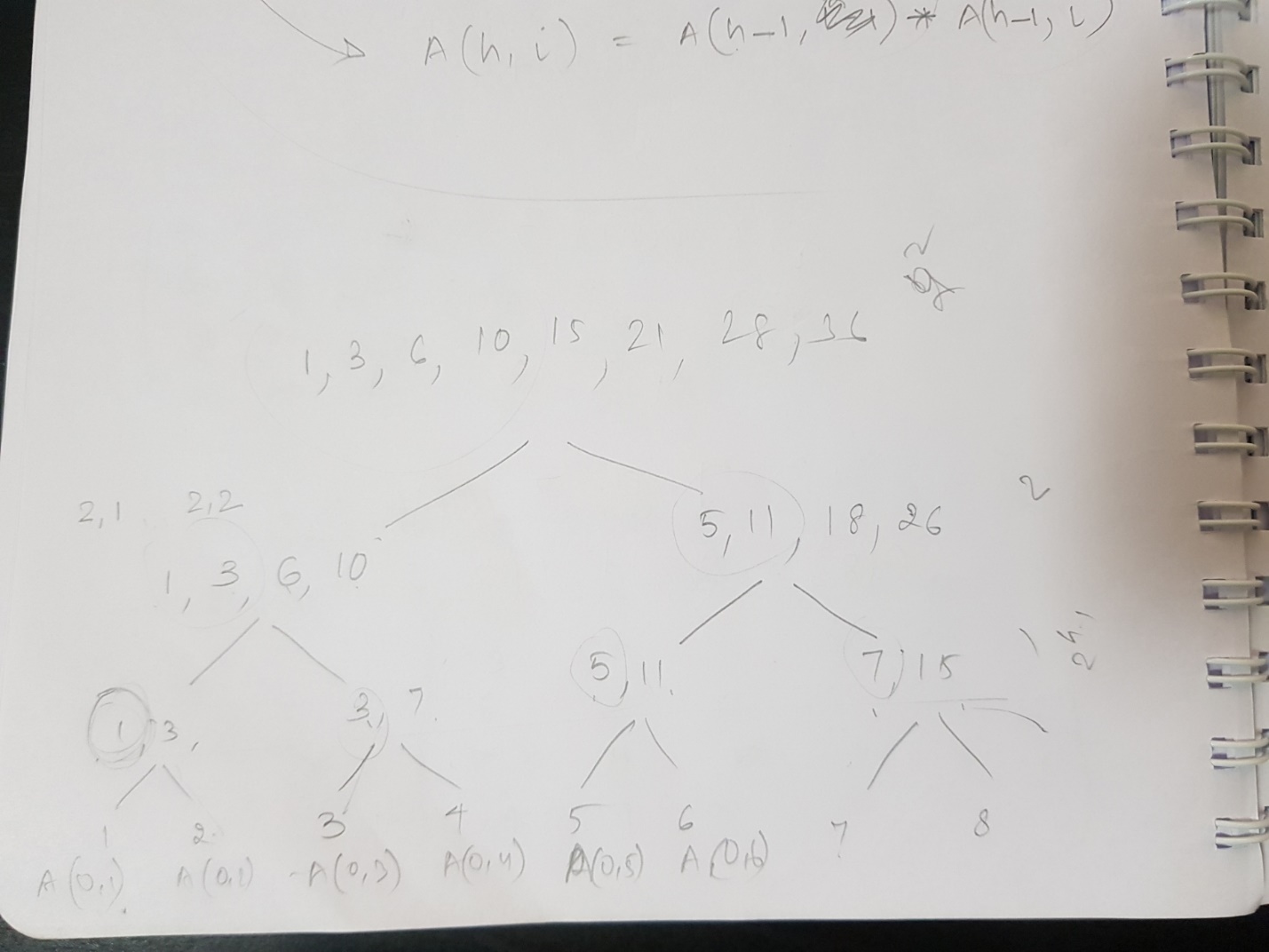
W(n) = ∑ j = 0 log n -1∑ i = 2j n – 1 1

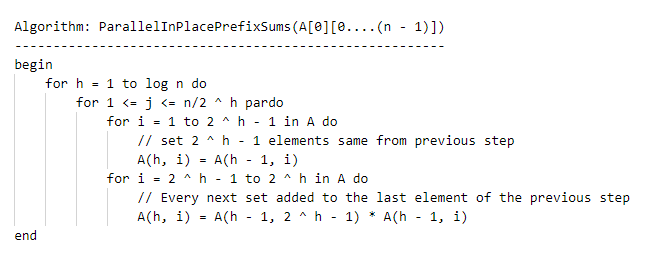
= n log n - ∑ j = 0 log n -12j

= n log n – (n - 1)

= ɵ(n log n)

So, the work is equal to ɵ(n log n)

**Alternative Solution:**



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(2n2)}) = O(n2 log n)

1. **We are given an array of colors A = [a1; a2; : : : ; an] drawn from k colors**

**fc1; c2; : : : ; ckg, where k is a constant. We wish to compute k indices i1; i2; : : : ; ik, for**

**each element ai, such that ij is the index of the closest element to the right of ai whose**

**color is cj . If no such element exists, then set ij = 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.

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 n­i 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.

1. **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 N0 and a positive constant k such that

f(n) ≤ k ⋅ log (n)/log log n, ∀ n ≥ N0

Since log (n)/log log n < = log n; for sufficiently large n, there must exist a N1 such that

f(n) ≤ k ⋅ log n, ∀ n ≥ N1

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 N0 such that 0 ≤ (log n) / (log (log n)) ≤ c \* log n for all n ≥ N0. 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 N0 = b ^ b ^ 2, 0 ≤ (log n)/ (log (log n)) ≤ log n for all n ≥ b^b^2.

* the first part is true, because log n ≥ log b^b^2 = b^2 ≥ 0 and log (log n) ≥ log (log b^b^2) = 2 ≥ 0
* the second part is also true, because it becomes log (log n) ≥ 1 and log (log n) ≥ log(b^2) = 2 ≥ 1.