Let $A = (a_0, a_1, a_2, \ldots, a_{N-1})$ be a **shared** array of N integers. The *prefix sums* of A are defined as:

$$\begin{aligned} s_0 &= a_0, \\ s_1 &= a_0 + a_1, \\ s_2 &= a_0 + a_1 + a_2, \\ &\ldots \\ s_{N-1} &= a_0 + a_1 + a_2 + \cdots + a_{N-1}. \end{aligned}$$

We want to compute the prefix sums of A *in place* (that is, in the array A itself). For this, we follow the recursive procedure discussed now. Break A into two halves, the first (A_1) containing the first $N_1 = \lfloor N/2 \rfloor$ elements of A, and the second (A_2) containing the last $N_2 = \lceil N/2 \rceil = \lfloor (N+1)/2 \rfloor = N - N_1$ elements of A. Recursively compute the prefix sums of A_1 and A_2 in place. Let these be $s'_0, s'_1, s'_2, \ldots, s'_{N_1-1}$ and $s'_{N_1}, s'_{N_1+1}, s'_{N_1+2}, \ldots, s'_{N-1}$, respectively. The prefix sums of the original array A are then computed as:

$$s_i = \begin{cases} s'_i & \text{if } 0 \leqslant i \leqslant N_1 - 1, \\ s'_{N_1-1} + s'_i & \text{if } N_1 \leqslant i \leqslant N-1. \end{cases}$$

This is not the most efficient way of solving the given problem, but we will use this algorithm only.

Besides N and the initial array elements $a_0, a_1, a_2, \ldots, a_{N-1}$, the user also specifies a maximum level number $L \geqslant 1$. Assume that $N \geqslant 2^{L+1}$. Each of the following two parts constructs a full binary tree of processes with levels $0, 1, 2, \ldots, L$ (so there are $L+1$ levels). The initial process is at the root of the process tree, and is at level 0. The leaf processes are at level L, and there are 2^L of them. The parents of the leaf processes are at level $L-1$; there are 2^{L-1} of them.

In general, for $l \in \{0, 1, 2, \ldots, L\}$, there are 2^l processes at level l. These are numbered $0, 1, 2, \ldots, 2^{l-1}$ from left to right. This means that the level l and a number $i \in [0, 2^l - 1]$ uniquely identify a process. Denote this process as $P_{l,i}$. With this notation, $P_{0,0}$ is the root (that is, the initial) process, the leaf processes are $P_{L,0}, P_{L,1}, P_{L,2}, \ldots, P_{L,2^L-1}$, and their parent processes are $P_{L-1,0}, P_{L-1,1}, P_{L-1,2}, \ldots, P_{L-1,2^{L-1}-1}$.

Part 1

Write a function **`prefixsum1()`** with appropriate arguments to define the work of the process $P_{l,i}$. Let C be the chunk of A, that comes to this process, and n the number of elements in the chunk C.

If $P_{l,i}$ is a leaf process (that is, $l = L$), it creates no further processes. It instead computes in place the prefix sums of its chunk C sequentially. After this computation, the process terminates.

If $P_{l,i}$ is not a leaf process (that is, $l < L$), the process divides its chunk into two parts C_1, C_2 consisting of the first $\lfloor n/2 \rfloor$ and the last $\lceil n/2 \rceil$ elements of C. It then forks two processes $P_{l+1,2i}$ and $P_{l+1,2i+1}$, and assigns the chunks C_1 and C_2 to them, respectively. The process $P_{l,i}$ then waits for its two child processes to terminate. After the wait is over, the process $P_{l,i}$ computes the prefix sums of its chunk C by the merging formula given above. Finally, if $P_{l,i}$ is not the root process (that is, $0 < l < L$), it terminates. The root process should return to the **`main()`** function for further work.

The synchronization of a (non-leaf) process with its children is to be achieved by the **`wait()`** system calls. You may design **`prefixsum1()`** as an iterative function or as a recursive function.

Part 2

This part is computationally similar to Part 1. That is, the leaf processes $P_{L,i}$ sequentially process their respective chunks, whereas a non-leaf process $P_{l,i}$ creates two child processes $P_{l+1,2i}$ and $P_{l+1,2i+1}$, and when these child processes are done with their computations, $P_{l,i}$ merges the two subchunks. The difference is that now a leaf process $P_{L,i}$ does not terminate immediately after processing its chunk C. It keeps on waiting until the entire array A is updated in place. It then prints its own chunk in the *final* (fully updated) array A, and then it terminates. This introduces the following additional synchronization issues.

First, a non-leaf process $P_{l,i}$ can wait for the termination of its two children so long as $0 \leqslant l \leqslant L-2$. The parents of the leaf processes (these are at level $L-1$) use a separate synchronization mechanism, since the leaf processes remain alive even after the entire A is updated. When a leaf process is done with the sequential computation on its chunk, it sends an end-of-work notification to its parent. A process at level $L-1$, after receiving the notifications from both its children, proceeds to its merging task, and then terminates.

Second, we require the chunks of the leaf processes be printed sequentially from left to right (that is, for $i = 0, 1, 2, \ldots, 2^L - 1$ in that order). The synchronization is to be achieved by interacting with the root process $P_{0,0}$. When $P_{0,0}$ finishes merging its two subarrays into the entire array, it wakes up the leaf processes *one by one* in the sequence mentioned above. If $P_{0,0}$ wakes up all the leaf processes at once, the scheduling of the woken up leaf processes will not be in your control. You must ensure that after $P_{0,0}$ wakes up a leaf process, $P_{0,0}$ waits until the woken up leaf process signals the root process *after* finishing the printing of its chunk.

Implement these new synchronization requirements using sets of semaphores (not by pipes or any other IPC mechanism). Write a recursive or iterative function **`prefixsum2()`** for implementing Part 2.

The main() function (for the root process only)

- Read N, L, and the array elements $a_0, a_1, a_2, \ldots, a_{N-1}$ from the user. Recall that A must be stored in **shared memory**. Keep a copy of A in a **local** array B.
- Call **`prefixsum1()`** to update A in place. When this function returns, print the updated array A.
- Copy back the local array B to the shared array A.
- Call **`prefixsum2()`** to update A in place. This time, the chunk-wise printing is to be done by the leaf processes, whereas the root process prints nothing.
- **Remove** all shared-memory segments and semaphore arrays created and used by your program.

Sample output

```
+++ N = 50
+++ L = 3

+++ Initial array
         6   13  -22   13  -20    7   23   21   16  -11  -10    1    7    7  -15
        19    5  -20    5    0  -17  -19   24  -11  -18    2   -8  -16  -22    4
       -16   13   -3  -14    2    1   17    5   -3   13   18   17  -11    5   -1
        -2   -1  -16    2  -21

*** PART 1
+++ Root process: Final array
         6   19   -3   10  -10   -3   20   41   57   46   36   37   44   51   36
        55   60   40   45   45   28    9   33   22    4    6   -2  -18  -40  -36
       -52  -39  -42  -56  -54  -53  -36  -31  -34  -21   -3   14    3    8    7
         5    4  -12  -10  -31

*** PART 2
+++ Leaf process 0: Final array segment
         6   19   -3   10  -10   -3
+++ Leaf process 1: Final array segment
        20   41   57   46   36   37
+++ Leaf process 2: Final array segment
        44   51   36   55   60   40
+++ Leaf process 3: Final array segment
        45   45   28    9   33   22    4
+++ Leaf process 4: Final array segment
         6   -2  -18  -40  -36  -52
+++ Leaf process 5: Final array segment
       -39  -42  -56  -54  -53  -36
+++ Leaf process 6: Final array segment
       -31  -34  -21   -3   14    3
+++ Leaf process 7: Final array segment
         8    7    5    4  -12  -10  -31
```

Submit one C/C++ file. Write your name, roll no, and PC no as comments in your code.

Let $A = (a_0, a_1, a_2, \ldots, a_{N-1})$ be a **shared** array of N integers. The *suffix minima* of A are defined as:

$$\begin{aligned} s_{N-1} &= a_{N-1}, \\ s_{N-2} &= \min(a_{N-2}, a_{N-1}), \\ s_{N-3} &= \min(a_{N-3}, a_{N-2}, a_{N-1}), \\ &\ldots \\ s_0 &= \min(a_0, a_1, a_2, \ldots, a_{N-1}). \end{aligned}$$

We want to compute the suffix minima of A *in place* (that is, in the array A itself). For this, we follow the recursive procedure discussed now. Break A into two halves, the first (A_1) containing the first $N_1 = \lfloor N/2 \rfloor$ elements of A, and the second (A_2) containing the last $N_2 = \lceil N/2 \rceil = \lfloor (N+1)/2 \rfloor = N - N_1$ elements of A. Recursively compute the suffix minima of A_1 and A_2 in place. Let these be $s'_0, s'_1, s'_2, \ldots, s'_{N_1-1}$ and $s'_{N_1}, s'_{N_1+1}, s'_{N_1+2}, \ldots, s'_{N-1}$, respectively. The suffix minima of the original array A are then computed as:

$$s_i = \begin{cases} s'_i & \text{if } N_1 \leqslant i \leqslant N-1, \\ \min(s'_i, s'_{N_1}) & \text{if } 0 \leqslant i \leqslant N_1 - 1. \end{cases}$$

This is not the most efficient way of solving the given problem, but we will use this algorithm only.

Besides N and the initial array elements $a_0, a_1, a_2, \ldots, a_{N-1}$, the user also specifies a maximum level number $L \geqslant 1$. Assume that $N \geqslant 2^{L+1}$. Each of the following two parts constructs a full binary tree of processes with levels $0, 1, 2, \ldots, L$ (so there are $L+1$ levels). The initial process is at the root of the process tree, and is at level 0. The leaf processes are at level L, and there are 2^L of them. The parents of the leaf processes are at level $L-1$; there are 2^{L-1} of them.

In general, for $l \in \{0, 1, 2, \ldots, L\}$, there are 2^l processes at level l. These are numbered $0, 1, 2, \ldots, 2^{l-1}$ from left to right. This means that the level l and a number $i \in [0, 2^l - 1]$ uniquely identify a process. Denote this process as $P_{l,i}$. With this notation, $P_{0,0}$ is the root (that is, the initial) process, the leaf processes are $P_{L,0}, P_{L,1}, P_{L,2}, \ldots, P_{L,2^L-1}$, and their parent processes are $P_{L-1,0}, P_{L-1,1}, P_{L-1,2}, \ldots, P_{L-1,2^{L-1}-1}$.

Part 1

Write a function **`suffixmin1()`** with appropriate arguments to define the work of the process $P_{l,i}$. Let C be the chunk of A, that comes to this process, and n the number of elements in the chunk C.

If $P_{l,i}$ is a leaf process (that is, $l = L$), it creates no further processes. It instead computes in place the suffix minima of its chunk C sequentially. After this computation, the process terminates.

If $P_{l,i}$ is not a leaf process (that is, $l < L$), the process divides its chunk into two parts C_1, C_2 consisting of the first $\lfloor n/2 \rfloor$ and the last $\lceil n/2 \rceil$ elements of C. It then forks two processes $P_{l+1,2i}$ and $P_{l+1,2i+1}$, and assigns the chunks C_1 and C_2 to them, respectively. The process $P_{l,i}$ then waits for its two child processes to terminate. After the wait is over, the process $P_{l,i}$ computes the suffix minima of its chunk C by the merging formula given above. Finally, if $P_{l,i}$ is not the root process (that is, $0 < l < L$), it terminates. The root process should return to the **`main()`** function for further work.

The synchronization of a (non-leaf) process with its children is to be achieved by the **`wait()`** system calls. You may design **`suffixmin1()`** as an iterative function or as a recursive function.

Part 2

This part is computationally similar to Part 1. That is, the leaf processes $P_{L,i}$ sequentially process their respective chunks, whereas a non-leaf process $P_{l,i}$ creates two child processes $P_{l+1,2i}$ and $P_{l+1,2i+1}$, and when these child processes are done with their computations, $P_{l,i}$ merges the two subchunks. The difference is that now a leaf process $P_{L,i}$ does not terminate immediately after processing its chunk C. It keeps on waiting until the entire array A is updated in place. It then prints its own chunk in the *final* (fully updated) array A, and then it terminates. This introduces the following additional synchronization issues.

First, a non-leaf process $P_{l,i}$ can wait for the termination of its two children so long as $0 \leqslant l \leqslant L-2$. The parents of the leaf processes (these are at level $L-1$) use a separate synchronization mechanism, since the leaf processes remain alive even after the entire A is updated. When a leaf process is done with the sequential computation on its chunk, it sends an end-of-work notification to its parent. A process at level $L-1$, after receiving the notifications from both its children, proceeds to its merging task, and then terminates.

Second, we require the chunks of the leaf processes be printed sequentially from left to right (that is, for $i = 0, 1, 2, \ldots, 2^L - 1$ in that order). The synchronization is to be achieved by interacting with the root process $P_{0,0}$. When $P_{0,0}$ finishes merging its two subarrays into the entire array, it wakes up the leaf processes *one by one* in the sequence mentioned above. If $P_{0,0}$ wakes up all the leaf processes at once, the scheduling of the woken up leaf processes will not be in your control. You must ensure that after $P_{0,0}$ wakes up a leaf process, $P_{0,0}$ waits until the woken up leaf process signals the root process *after* finishing the printing of its chunk.

Implement these new synchronization requirements using sets of semaphores (not by pipes or any other IPC mechanism). Write a recursive or iterative function **`suffixmin2()`** for implementing Part 2.

The main() function (for the root process only)

- Read N, L, and the array elements $a_0, a_1, a_2, \ldots, a_{N-1}$ from the user. Recall that A must be stored in **shared memory**. Keep a copy of A in a **local** array B.
- Call **`suffixmin1()`** to update A in place. When this function returns, print the updated array A.
- Copy back the local array B to the shared array A.
- Call **`suffixmin2()`** to update A in place. This time, the chunk-wise printing is to be done by the leaf processes, whereas the root process prints nothing.
- **Remove** all shared-memory segments and semaphore arrays created and used by your program.

Sample output

```
+++ N = 50
+++ L = 3

+++ Initial array
      171  211  532  579  604 1154  758 1193  939 1262 1627 1672 2026 1752 1695
     2427 2427 2742 2143 3390 4152 3691 4064 3263 4859 4675 3615 4152 4133 3150
     6064 4502 3649 3403 6498 4732 7121 7375 4201 4956 6043 7675 7021 4543 6072
     4754 8554 8532 8956 9742

*** PART 1
+++ Root process: Final array
      171  211  532  579  604  758  758  939  939 1262 1627 1672 1695 1695 1695
     2143 2143 2143 2143 3150 3150 3150 3150 3150 3150 3150 3150 3150 3150 3150
     3403 3403 3403 3403 4201 4201 4201 4201 4543 4543 4543 4543 4543 4754
     4754 8532 8532 8956 9742

*** PART 2
+++ Leaf process 0: Final array segment
      171  211  532  579  604  758
+++ Leaf process 1: Final array segment
      758  939  939 1262 1627 1672
+++ Leaf process 2: Final array segment
     1695 1695 1695 2143 2143 2143
+++ Leaf process 3: Final array segment
     2143 3150 3150 3150 3150 3150 3150
+++ Leaf process 4: Final array segment
     3150 3150 3150 3150 3150 3403
+++ Leaf process 5: Final array segment
     3403 3403 3403 4201 4201 4201
+++ Leaf process 6: Final array segment
     4201 4201 4543 4543 4543 4543
+++ Leaf process 7: Final array segment
     4543 4754 4754 8532 8532 8956 9742
```

Submit one C/C++ file. Write your name, roll no, and PC no as comments in your code.