From Algorithms to Programs:- *Case Study of Merge Sort*

SUNIL SITARAM SHELKE

MergeSort(A, low, high)

If($low < high$)

$mid = \frac{low+high}{2}$

MergeSort(A, low, mid)

MergeSort($A, mid + 1,\ high$)

Merge($A, low, mid, high$)

Merge(A, low, mid, high)

Create temporary arrays $Left$ and $Right$ of lsize $= mid - low + 1$ and rsize $= high - mid$ elements each

Copy $A[low..mid]$ into $Left$ array and $A[mid + 1..high]$ into $Right$ array

For($\mathrm{x} = 0, y = 0,\ k = low;\ \ x \le lsize\ \ and\ y \le rsize;$)

if($Left[j] \le Right[j]$)

$A[k++] = Left[x];\ \ x++;$

else

$A[k++] = Right[y];\ \ y++;$

If($x == lsize$)

while($y < rsize$)

$A[k++] = Right[y++]$

Else

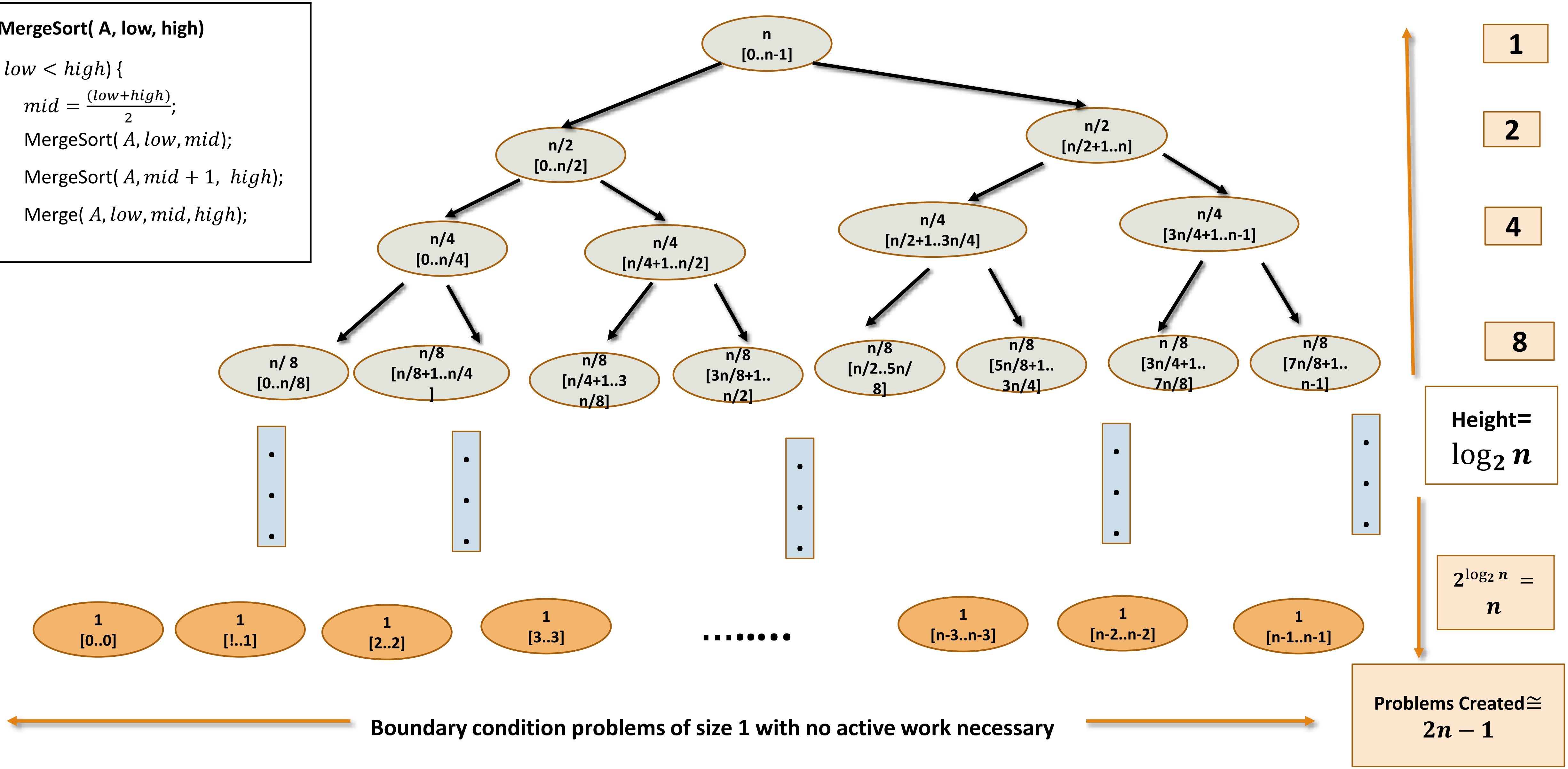
while($x < lsize$)

$A[k++] = Left[x++]$

Initial Call =
$MergeSort(A, 0, n - 1)$

MergeSort :- Boundary Condition of Recursion and Useless Function Calls

```
void MergeSort( A, low, high)
If( low < high) {
        mid = (low+high)/2;
        MergeSort( A, low, mid);
        MergeSort( A, mid + 1,  high);
        Merge( A, low, mid, high);
}
```



- Number of boundary condition problems= n Each of size=1 (when low==high) require no active work to be done
- Total Number of function calls $\cong 2n-1$ *Boundary condition function calls* $=n \cong 50\%$ *total calls*

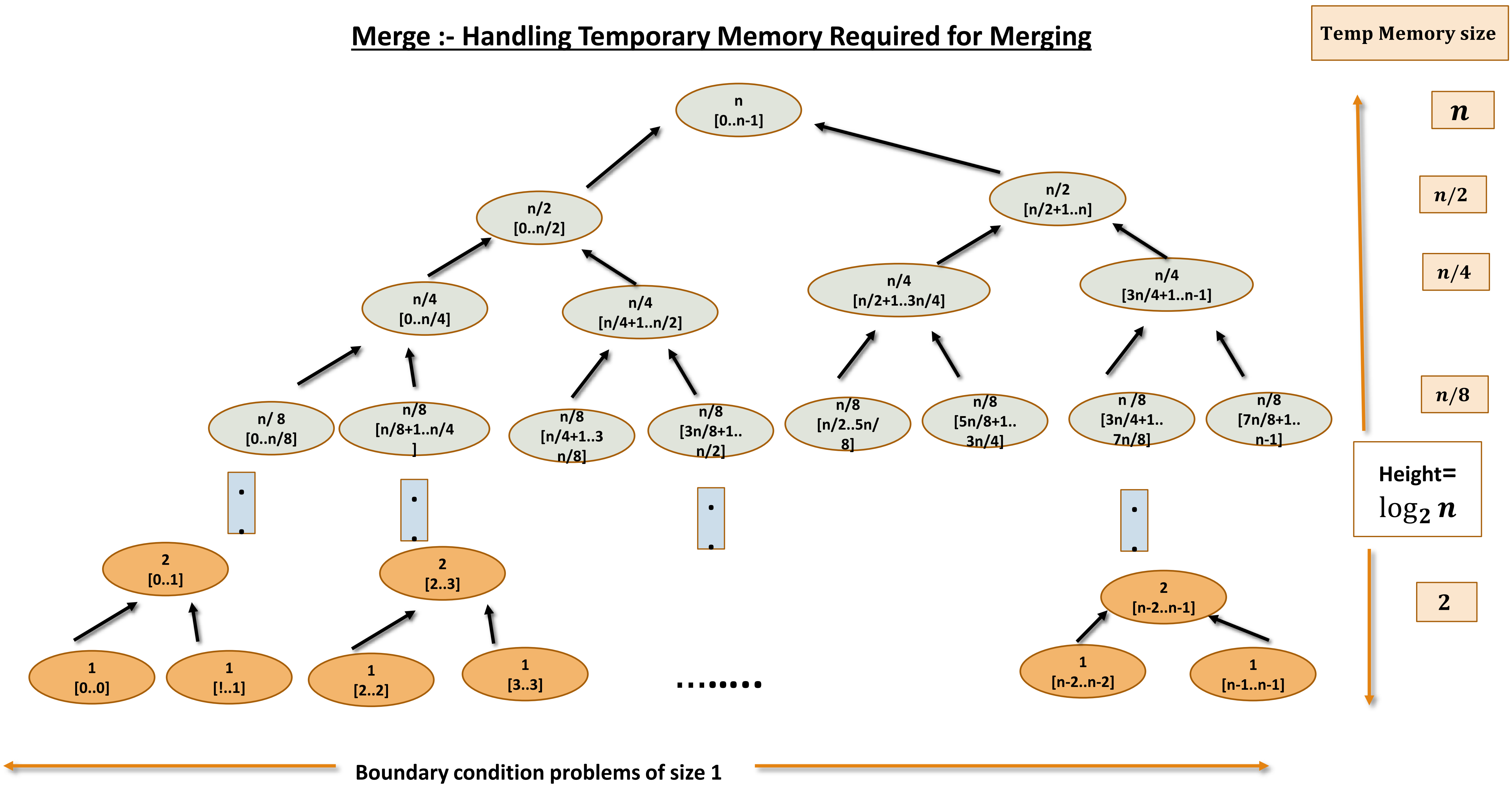
```
void MergeSort(int * A, int low, int high) {
If( low < high) {
        mid = (low + high)/2;
        MergeSort( A, low, mid);
        MergeSort( A, mid + 1,  high);
        Merge( A, low, mid, high);
      }
   }
  // Before
 // Number of function calls= 2n − 1
```

```
void MergeSort(int * A, int low, int high) {
  mid = (low + high)/2;
If( low < mid) {
       MergeSort( A, low, mid); }
If( mid + 1 < high) {
       MergeSort( A, mid + 1,  high);
   }
   Merge( A, low, mid, high);
 }
// After Call Reduction : n − 1
 // 50% increase in efficiency
```

```
void MergeSort(int * A, int low, int high) {
  mid = low + (high − low)/2;
If( low < mid) {
       MergeSort( A, low, mid); }
If( mid + 1 < high) {
       MergeSort( A, mid + 1,  high);
   }
   Merge( A, low, mid, high);
 }
// After avoiding overflow too
```

- Assume we consider positive values and *int* is of 4 bits. Max positive value ≤ 15
- Consider $\boldsymbol{low = 9}$ $\boldsymbol{high = 15}$ We have $mid = \frac{9+15}{2} = 24/2$
- Now $15 < 24$ so due to overflow we get $mid = \frac{8}{2} = \mathbf{4}$ .. **WRONG !!**

Merge :- Handling Temporary Memory Required for Merging



```
Void Merge( int A, int low, int mid, int high ) {

1    int * Left, *Right, lsize = mid − low + 1, rsize = high − mid, x, y, k ;

     Left=( int *) malloc( lsize ∗ sizeof(int) );
2
     Right=( int *) malloc( rsize ∗ sizeof(int) );
     for( x = 0; x < lsize; x + +)
3
          Left[x] = A[low + x];
     for( y = 0; y < rsize; y + +)

          Right[y] = A[mid + 1 + y];

4    for( x = 0, y = 0,  k = low;  x < lsize  and y < rsize; ) {

5             if( Left[x] ≤ Right[y])
6                  A[k + +] = Left[x + +];
7              else
8                  A[k + +] = Right[y + +];   }
8    if(x == lsize)
10            while(y < rsize)
11                 A[k + +] = A[y + +];
12   else
13            while(x < lsize)
                   A[k + +] = A[x + +];

14   free(Left);  free(Right); }
```

```
int * temp=( int *) malloc( n ∗ sizeof(int) );   // Global Temp array


Void Merge( int A, int low, int mid, int high ) {

1    int lsize = mid − low + 1, rsize = high − mid, x, y, k ;

     for( x = 0; x < lsize; x + +)
2
          temp[x] = A[low + x];
     for( y = mid + 1; y < (rsize + mid + 1); y + +)

          temp[y] = A[mid + 1 + y];

3    for( x = 0, y = mid + 1,  k = low;  x < lsize  and y < (rsize + mid + 1); ) {

4             if( temp[x] ≤ temp[y])
5                  A[k + +] = temp[x + +];
6              else
7                  A[k + +] = temp[y + +];   }
8    if(x == lsize)
9             while(y < (rsize + mid + 1))
10                 A[k + +] = temp[y + +];
11   else
12            while(x < lsize)
                   A[k + +] = temp[x + +];

     }
```

Further improvements in implementation of Merge Sort

1. **Early Recursion Termination:-**
   - Stop when problem size reduces to some tolerable value, say, $k > 1$.
   - Invoke another sorting algorithm which is very fast for such small values for eg. ***Insertion Sort*** on small arrays
   - Value of k can be chosen so as to maintain asymptotic complexity of traditional merge sort i.e $\mathbf{\Theta(n \log n)}$

2. **Bottom-Up/Iterative implementation:-**
   - **Recursive** implementation requires system **stack of height** $\boldsymbol{O(\log n)}$ to maintain recursion.
   - **Iterative** left to right bottom up merging can be done with maintaining time complexity $\mathbf{\Theta(n \log n)}$ by **avoiding system stack** requirement.

Thank You !!