2021-2022

Data Structures and Algorithms (II) – Sorting Algorithms

Dr. Dapeng Dong

Merge Sort

- Merge sort is an efficient comparison-based, divide-and-conquer recursive algorithm.

- Merge sort runs in $\boldsymbol{O(N\lg N)}$ both worst-case and average-case running time and the number of comparisons used by the algorithm is nearly optimal.

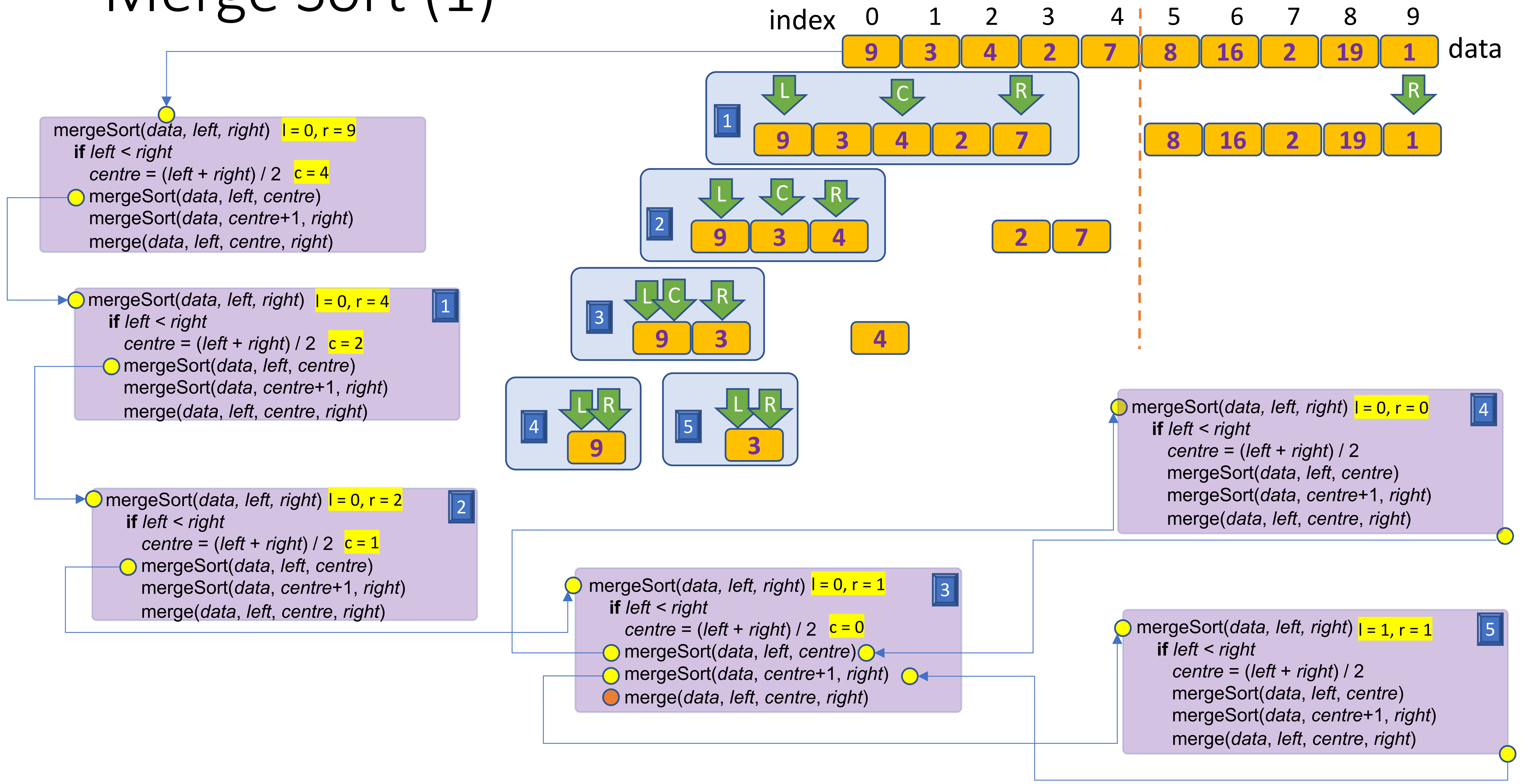
- Many implementations of merge sort are stable, i.e., the order of equal elements is the same in the input and output.

Divide-and-Conquer

The merge sort algorithm follows the divide-and-conquer paradigm

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem
  - ***Merge sort***: divide the $\boldsymbol{N}$-element sequence to be sorted into two sub-sequences of $\boldsymbol{\frac{N}{2}}$ elements each
- **Conquer** the subproblems by solving them recursively.
  - ***Merge sort***: sort the two sub-sequences recursively
- **Combine** the solutions to the subproblems into the solution for the original problem
  - ***Merge sort***: merge the two sorted sub-sequences to produce the sorted answer

Merge Sort (1)

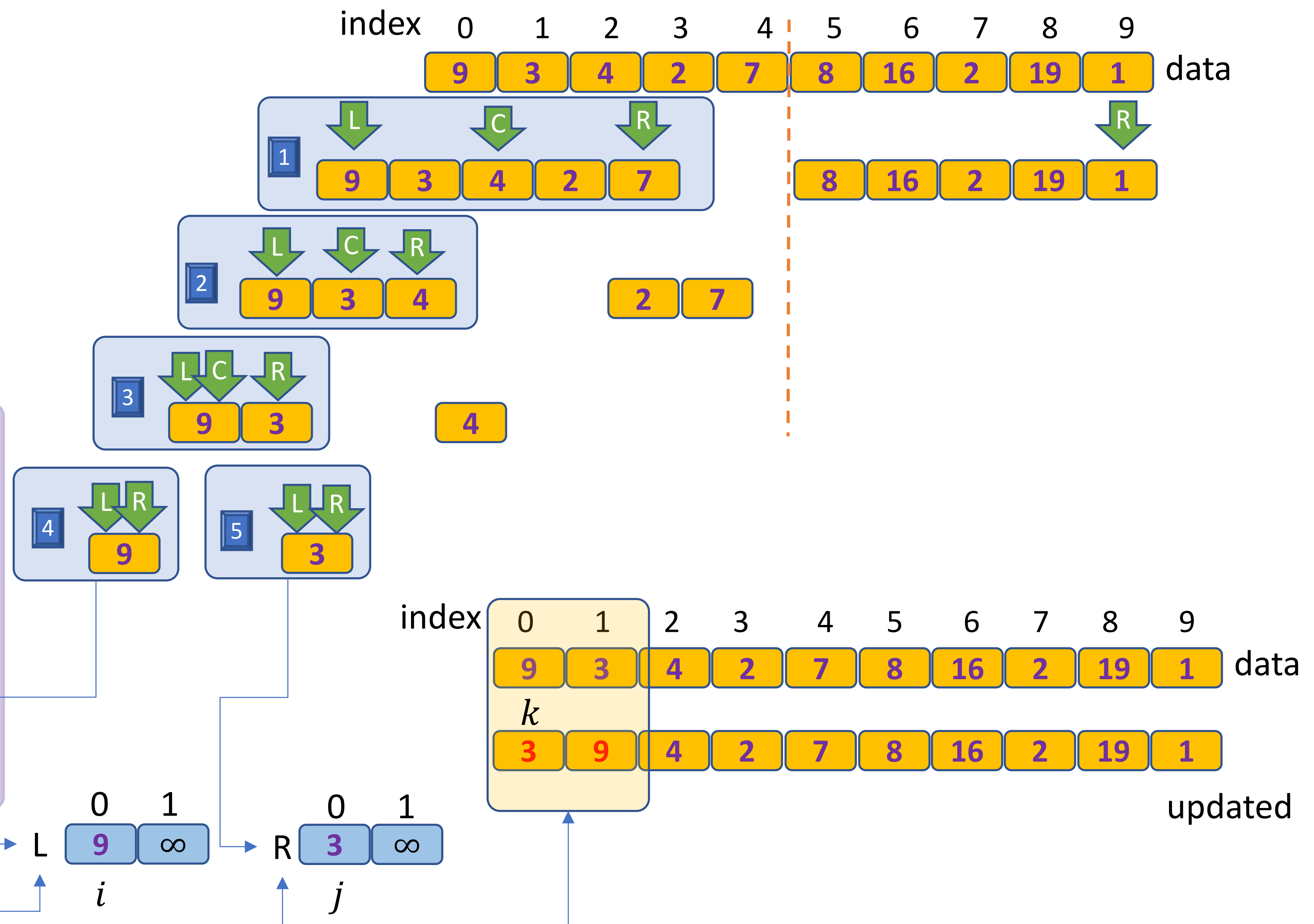

index 0 1 2 3 4 5 6 7 8 9
9 3 4 2 7 8 16 2 19 1
data
L C R
1
9 3 4 2 7
R
8 16 2 19 1
2
L C R
9 3 4
2 7
3
L C R
9 3
4
4
L R
9
5
L R
3
mergeSort(data, left, right) l = 0, r = 9
if left < right
centre = (left + right) / 2 c = 4
mergeSort(data, left, centre)
mergeSort(data, centre+1, right)
merge(data, left, centre, right)
mergeSort(data, left, right) l = 0, r = 4
1
if left < right
centre = (left + right) / 2 c = 2
mergeSort(data, left, centre)
mergeSort(data, centre+1, right)
merge(data, left, centre, right)
mergeSort(data, left, right) l = 0, r = 2
2
if left < right
centre = (left + right) / 2 c = 1
mergeSort(data, left, centre)
mergeSort(data, centre+1, right)
merge(data, left, centre, right)
mergeSort(data, left, right) l = 0, r = 1
3
if left < right
centre = (left + right) / 2 c = 0
mergeSort(data, left, centre)
mergeSort(data, centre+1, right)
merge(data, left, centre, right)
mergeSort(data, left, right) l = 0, r = 0
4
if left < right
centre = (left + right) / 2
mergeSort(data, left, centre)
mergeSort(data, centre+1, right)
merge(data, left, centre, right)
mergeSort(data, left, right) l = 1, r = 1
5
if left < right
centre = (left + right) / 2
mergeSort(data, left, centre)
mergeSort(data, centre+1, right)
merge(data, left, centre, right)

Merge Sort (2)

```
mergeSort(data, left, right)  l = 0, r = 1        3
    if left < right
        centre = (left + right) / 2   c = 0
        mergeSort(data, left, centre)
        mergeSort(data, centre+1, right)
        merge(data, left, centre, right)
```

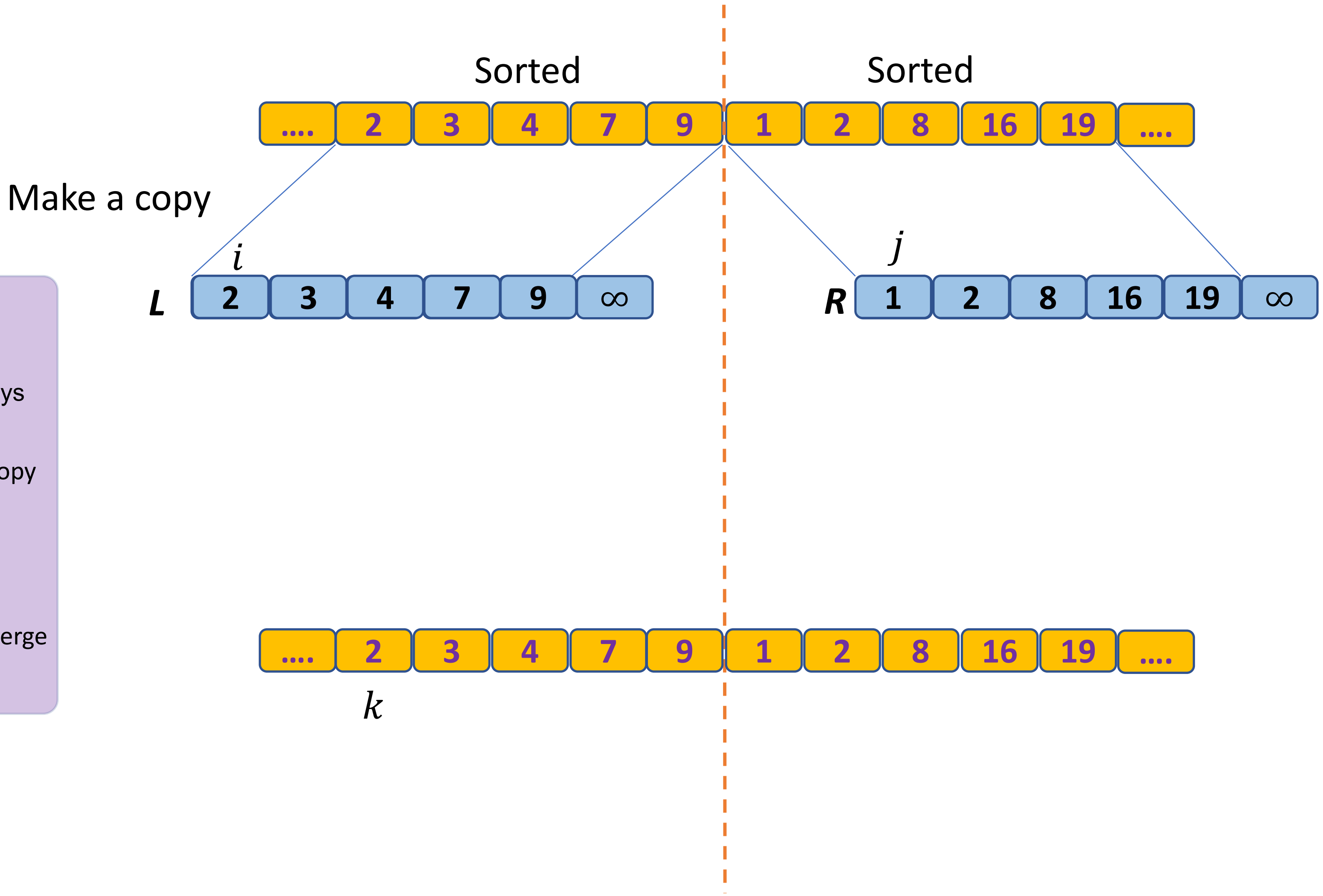
0 0 1

```
merge(data, left, centre, right)
    n1 = centre – left + 1   1
    n2 = right - centre      1
    let L[ 0..n1 ] and R[ 0.. n2 ] be new arrays
    for i = 0 to n1
        L[ i ] = data[ left + i ]
    for j = 0 to n2
        R[ j ] = data[ centre + j ]
    L[ n1 ] = R[ n2 ] = ∞
    i = j = 0
    for k = left to right
        if L[ i ] ≤ R[ j ]
            data[ k ] = L[ i++ ]
        else data[ k ] = R[ j++ ]
```



The General Case for Merging

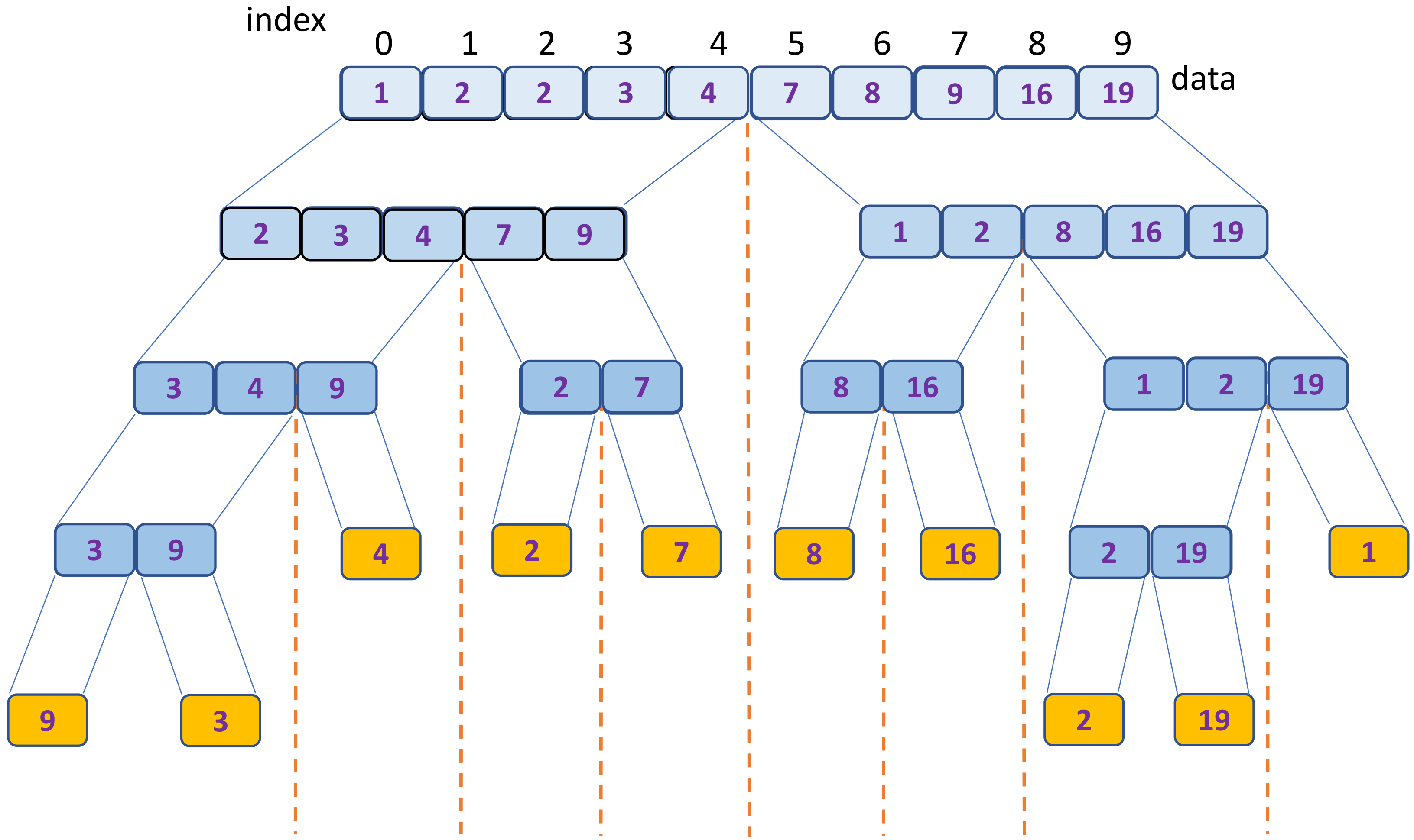
```
merge(data, left, centre, right)
   n1 = centre – left + 1
   n2 = right - centre
   let L[ 0..n1 ] and R[ 0.. n2 ] be new arrays
   for i = 0 to n1                                  Split & Copy
      L[ i ] = data[ left + i ]
   for j = 0 to n2
      R[ j ] = data[ centre + j ]
   L[ n1 ] = R[ n2 ] = ∞
   i = j = 0
   for k = left to right                            Sort & Merge
      if L[ i ] ≤ R[ j ]
         data[ k ] = L[ i++ ]
      else data[ k ] = R[ j++ ]
```



Merge Sort Example

mergeSort(*data*, *left*, *right*)
  if *left* < *right*
    centre = (*left* + *right*) / 2
    mergeSort(*data*, *left*, *centre*)
    mergeSort(*data*, *centre*+1, *right*)
    merge(*data*, *left*, *centre*, *right*)

merge(*data*, *left*, *centre*, *right*)
  n_1 = *centre* – *left* + 1
  n_2 = *right* - *centre*
  let $L[\ 0..n_1\ ]$ and $R[\ 0..\ n_2\ ]$ be new arrays
  for i = 0 **to** n_1
    $L[\ i\ ]$ = $data[\ left + i\ ]$
  for j = 0 **to** n_2
    $R[\ j\ ]$ = $data[\ centre + j\ ]$
  $L[\ n_1\ ] = R[\ n_2\ ] = \infty$
  $i = j = 0$
  for k = *left* **to** *right*
    if $L[\ i\ ] \leq R[\ j\ ]$
      $data[\ k\ ] = L[\ i{+}{+}\ ]$
    else $data[\ k\ ] = R[\ j{+}{+}\ ]$



Analysis (1)

1 In merge sort, each divide step yields two subsequence of size exactly $\frac{N}{2}$ (assume the original problem size is a power of 2).

Divide:
Let $D(N)$ denotes the time used for dividing subarrays. Since the divide step just computes the middle of the subarray, which takes constant time. Thus, $D(N) = \theta(1)$.

Conquer:
Let $T(N)$ denotes the time used for solving the problem of size N. We recursively solve two subproblems, each of size $\frac{N}{2}$, which contributes $2T(\frac{N}{2})$ to the running time.

Combine:
Merging two $\frac{N}{2}$ subarrays takes time $\theta(N)$, thus the time used for combing is $C(N) = \theta(N)$.

```
mergeSort(data, left, right)
  if left < right
Divide     centre = (left + right) / 2
Conquer    mergeSort(data, left, centre)
           mergeSort(data, centre+1, right)
Combine    merge(data, left, centre, right)
```

```
merge(data, left, centre, right)
  n1 = centre – left + 1
  n2 = right - centre
  let L[ 0..n1 ] and R[ 0.. n2 ] be new arrays
  for i = 0 to n1
    L[ i ] = data[ left + i ]
  for j = 0 to n2
    R[ j ] = data[ centre + j ]
  L[ n1 ] = R[ n2 ] = ∞
  i = j = 0
  for k = left to right
    if L[ i ] ≤ R[ j ]
      data[ k ] = L[ i++ ]
    else data[ k ] = R[ j++ ]
```

Analysis (2)

2 If there is only one value in the array, it will take a constant time $\boldsymbol{c}$ to complete; if there are more than one values to be sorted, it will take time $\boldsymbol{2T\left(\frac{N}{2}\right) + D(N) + C(N)}$.

3 Since, $\boldsymbol{D(N)} = \boldsymbol{\theta(1)}$ and $\boldsymbol{C(N) = \theta(N)}$, given the rule $\boldsymbol{T(N) = \theta(max(f(N), g(N))}$, we have

$$T(N) = \begin{cases} c & if\ N = 1, \\ 2T\left(\frac{N}{2}\right) + cN & if\ N > 1, \end{cases} \quad \overset{\text{Ignore the constant}}{\Longrightarrow} \quad T(N) = \begin{cases} 1 & if\ N = 1, \\ 2T\left(\frac{N}{2}\right) + N & if\ N > 1, \end{cases}$$

4 Dividing by $\boldsymbol{N}$

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + 1$$

Analysis (3)

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + 1$$

5 Telescoping,

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + 1$$

$$\frac{T(N/2)}{N/2} = \frac{T(N/4)}{N/4} + 1$$

$$\frac{T(N/4)}{N/4} = \frac{T(N/8)}{N/8} + 1$$

$$\vdots$$

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

6 Collecting the terms and rearrange,

$$\frac{T(N)}{N} = \frac{T(1)}{1} + \underbrace{1 + 1 + \ \ldots + 1 + 1}_{\Downarrow \ \left\langle N, \frac{N}{2}, \frac{N}{4}, \frac{N}{8}, \ldots, 4, 2 \right\rangle}$$

6.1 Based on the assumption that we have made, $\boldsymbol{N}$ is a power of $\mathbf{2}$, let $\boldsymbol{N} = \mathbf{2}^{\boldsymbol{k}}$

$$\left\langle N, \frac{N}{2}, \frac{N}{4}, \frac{N}{8}, \ldots, 4, 2 \right\rangle \implies \left\langle 2^k, 2^{k-1}, 2^{k-2}, \ldots, 2^2, 2^1 \right\rangle$$

Thus, $\boldsymbol{k}$ indicates the number of $\mathbf{1}$s in 6

6.2 Taking logarithm to the base 2 on both sides,

$$N = 2^k \implies \lg N = k$$

7 Given that $T(1) = 1$,

$$T(N) = N\lg N + N$$

7.1 Ignoring the lower-term,

$$T(N) = \theta(N\lg N)$$

Analysis -- Using Recursion Tree

$$T(N) = \begin{cases} c & \text{if } N = 1, \\ 2T\left(\frac{N}{2}\right) + cN & \text{if } N > 1, \end{cases}$$

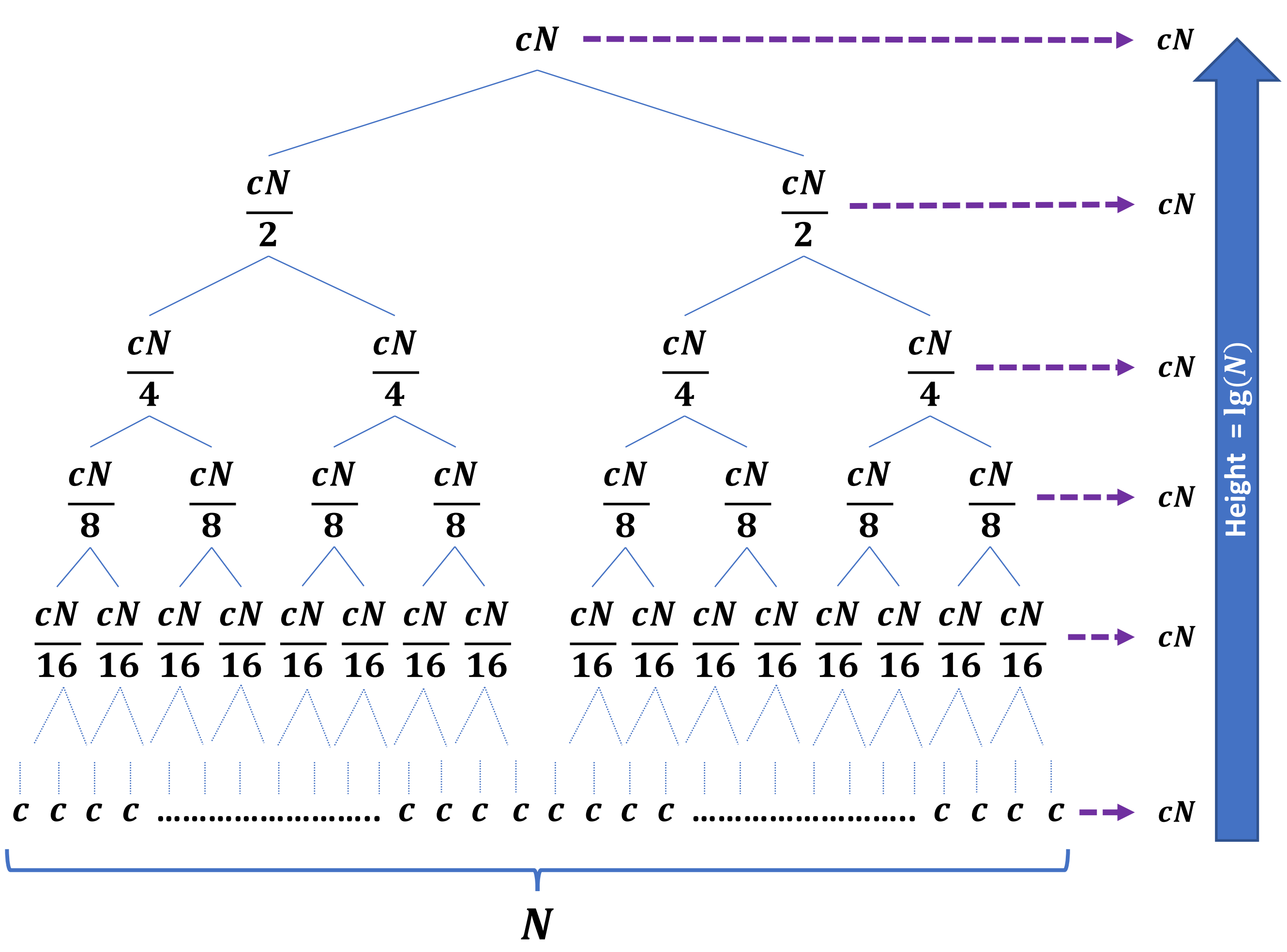
1. From the recursion tree, given the height of the tree $\mathbf{lg}N$, there are in total $\mathbf{lg}N + \mathbf{1}$ levels

   Thus, $T(N) = \mathbf{c}N\mathbf{lg}N + cN$

2. Ignoring the low-order term and the constant c, thus $N\mathbf{lg}N$ is an asymptotically tight bound for $T(N)$, i.e.,

$$T(N) = \theta(N\mathbf{lg}N)$$

Recursion Tree



Merge Sort Implementation using an Auxiliary Array

```java
private static <E extends Comparable<E>> void mergeSort(E[] data, E[] auxArray, int left, int right) {
  if (left < right) {
    int centre = (left + right) / 2;
    mergeSort(data, auxArray, left, centre);
    mergeSort(data, auxArray, centre + 1, right);
    merge(data, auxArray, left, centre + 1, right);
  }
}
```

```java
private static <E extends Comparable<E>> void merge(E[] data, E[] auxArray, int leftPos, int rightPos, int rightEnd) {
  int leftEnd = rightPos - 1;
  int tempPos = leftPos;
  int numElements = rightEnd - leftPos + 1;

  while (leftPos <= leftEnd && rightPos <= rightEnd) {
    if (data[leftPos].compareTo(data[rightPos]) <= 0)
      auxArray[tempPos++] = data[leftPos++];
    else
      auxArray[tempPos++] = data[rightPos++];
  }
  while (leftPos <= leftEnd) // Copy rest of first half
    auxArray[tempPos++] = data[leftPos++];
  while (rightPos <= rightEnd) // Copy rest of right half
    auxArray[tempPos++] = data[rightPos++];
  for (int i = 0; i < numElements; i++, rightEnd--) { // Copy auxArray back
    data[rightEnd] = auxArray[rightEnd];
}
```