Department of Computer Science and Engineering

Data Structures and Object-Oriented Design

(CSE – 2050)

Hasan Baig

Office: UConn (Stamford), 305C
email: hasan.baig@uconn.edu

Module 7

Divide and Conquer

Divide and Conquer

Divide and Conquer is a paradigm for algorithm design and consists of the following three steps:

1. ***Divide:*** Divide the input data D into two or more disjoint subsets, D_1 and D_2.
2. ***Conquer:*** Recursively solve the subproblems associated with the subsets, D_1 and D_2.
3. ***Combine:*** Take the solutions to the subproblems, D_1 and D_2, and merge them into a solution to the original problem D.

Base case: Base case for the recursion are subproblems of size 0 or 1.

Merge-Sort Algorithm

It is a sorting algorithm based on the divide-and-conquer paradigm

Algorithm Development

- Divide: Partition *D* into two sequences D_1 and D_2, having n/2 elements in each.
  - If *D* has zero or one item, return D as it is considered sorted
- Conquer: Recursively sort D_1 and D_2.
- Combine: Merge D_1 and D_2 into a sorted sequence.

```
Algorithm mergeSort(D)
 Input sequence D with n
       elements
 Output sequence D sorted

if D.size() > 1
 (D1, D2) ← partition(D, n/2)
 mergeSort(D1)
 mergeSort(D2)
 D ← merge(D1, D2)
```

Merge-Sort Algorithm

85	24	63	45	17	31	96	50

```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Merge-Sort Algorithm

Divide

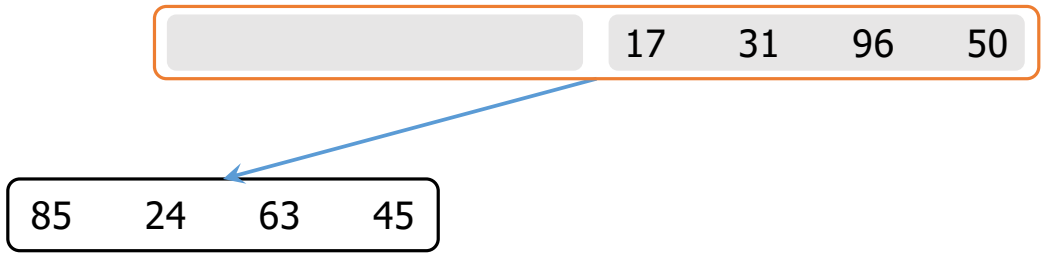
85	24	63	45	17	31	96	50

```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Merge-Sort Algorithm

Conquer

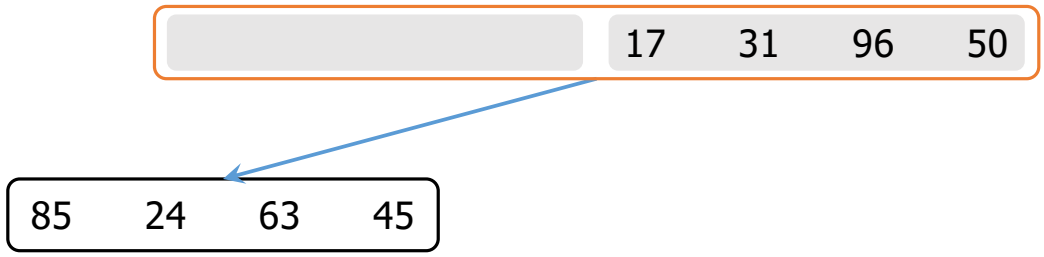



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Merge-Sort Algorithm

Conquer

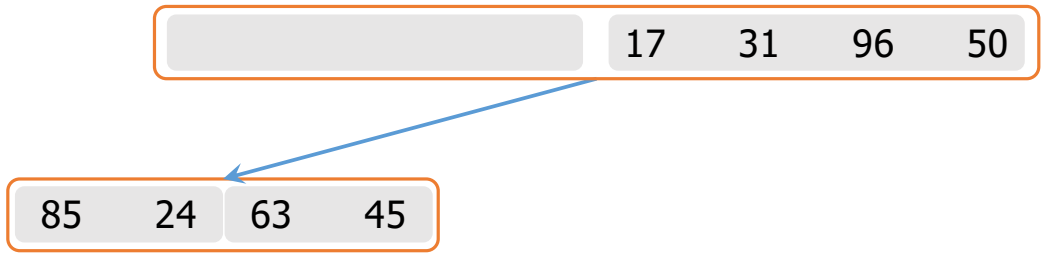



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Merge-Sort Algorithm

Divide

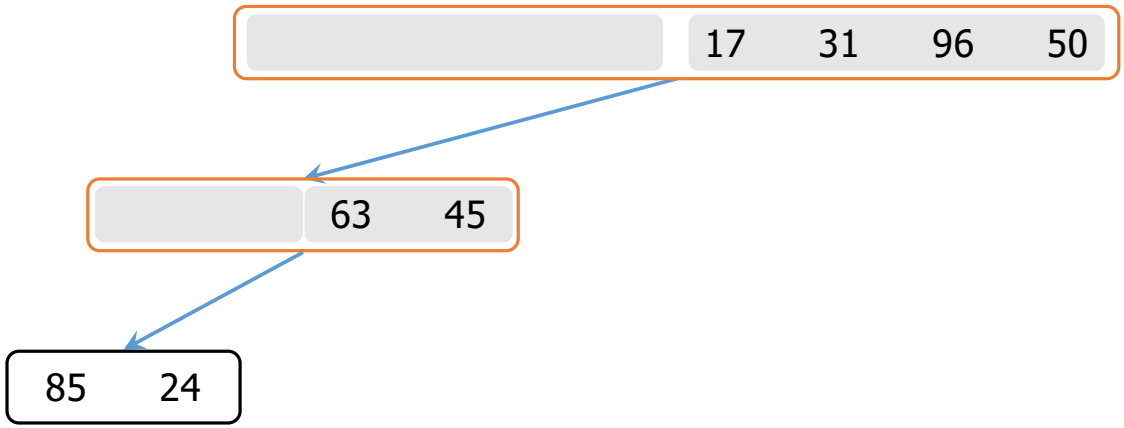



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Merge-Sort Algorithm

Conquer

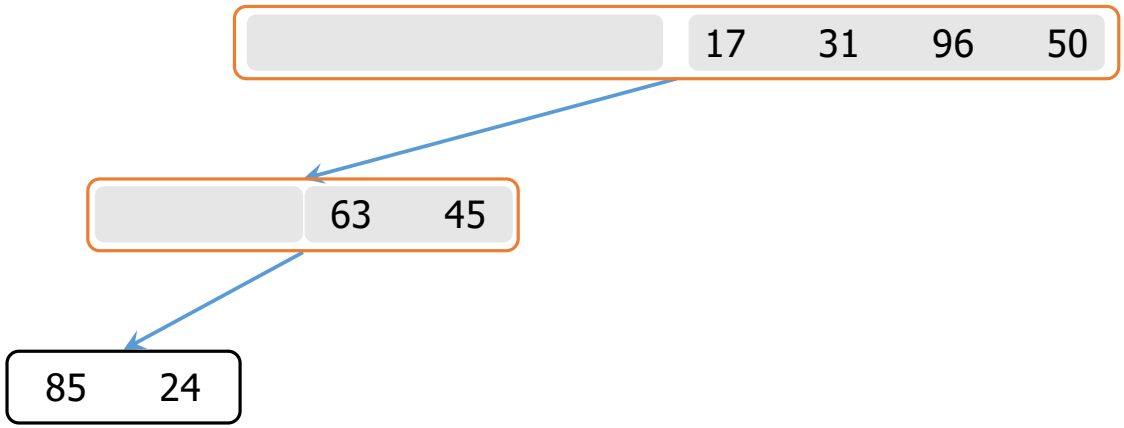



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Merge-Sort Algorithm

Conquer

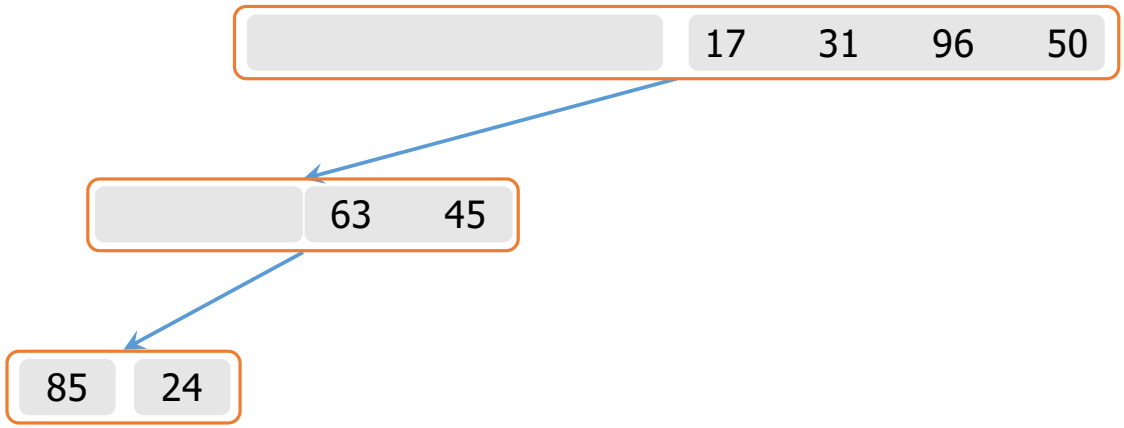



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Merge-Sort Algorithm

Divide

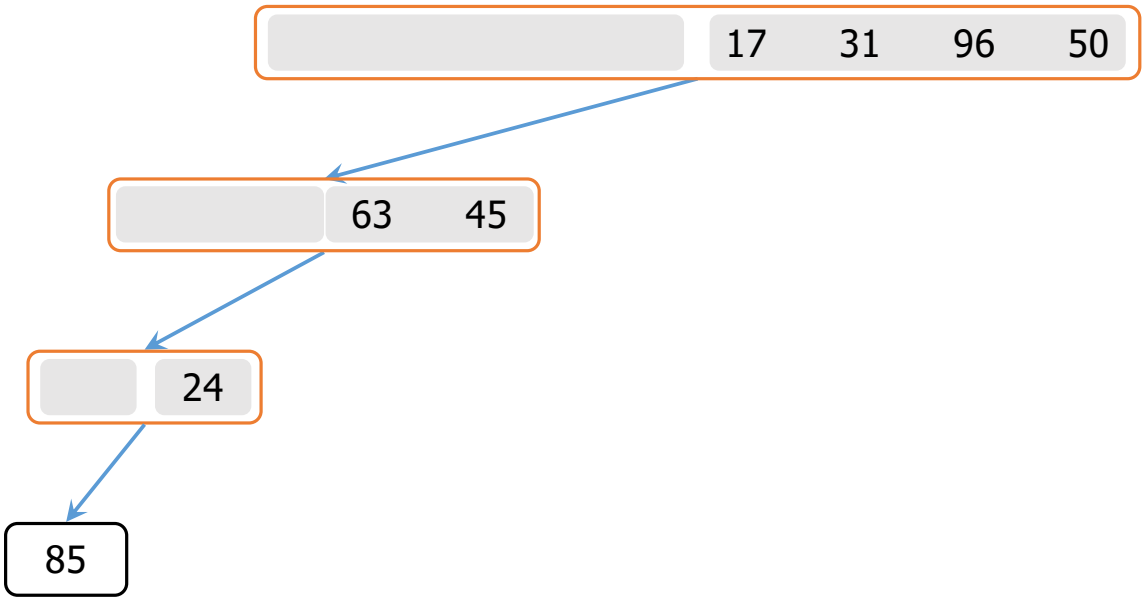



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Merge-Sort Algorithm

Conquer

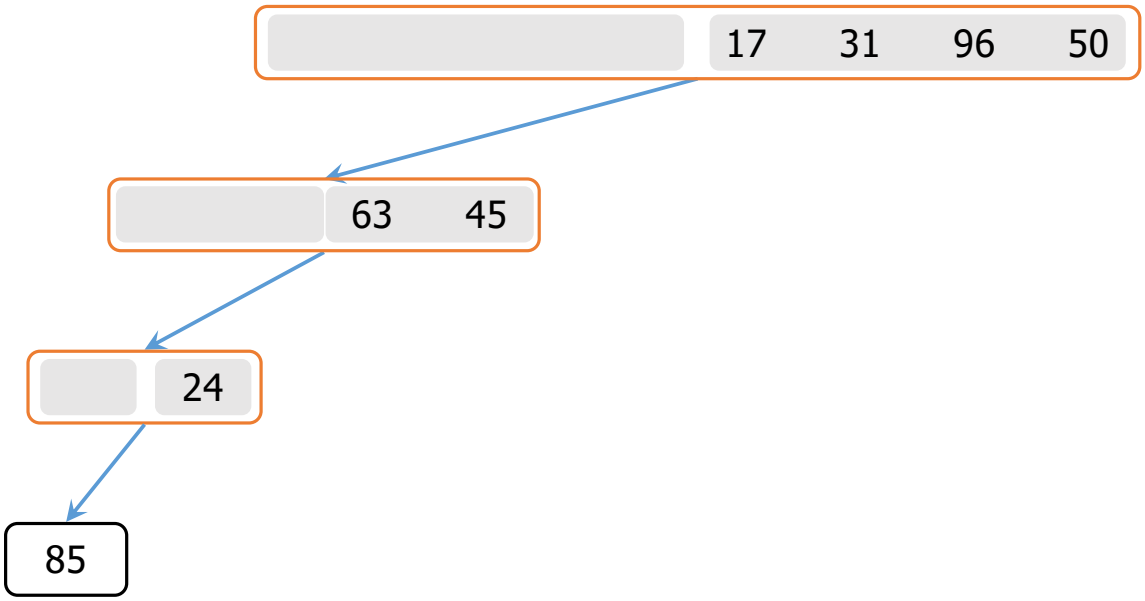



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Merge-Sort Algorithm

Base case

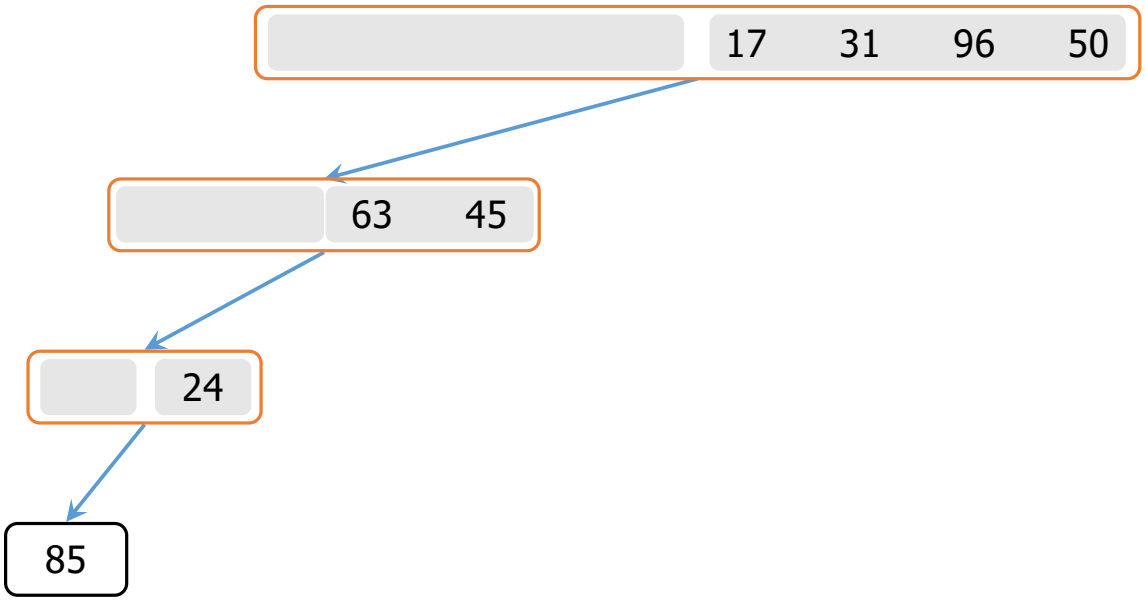



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

  if D.size() > 1
    (D1, D2) ← partition(D, n/2)
    mergeSort(D1)
    mergeSort(D2)
    D ← merge(D1, D2)
```

Merge-Sort Algorithm

Base case

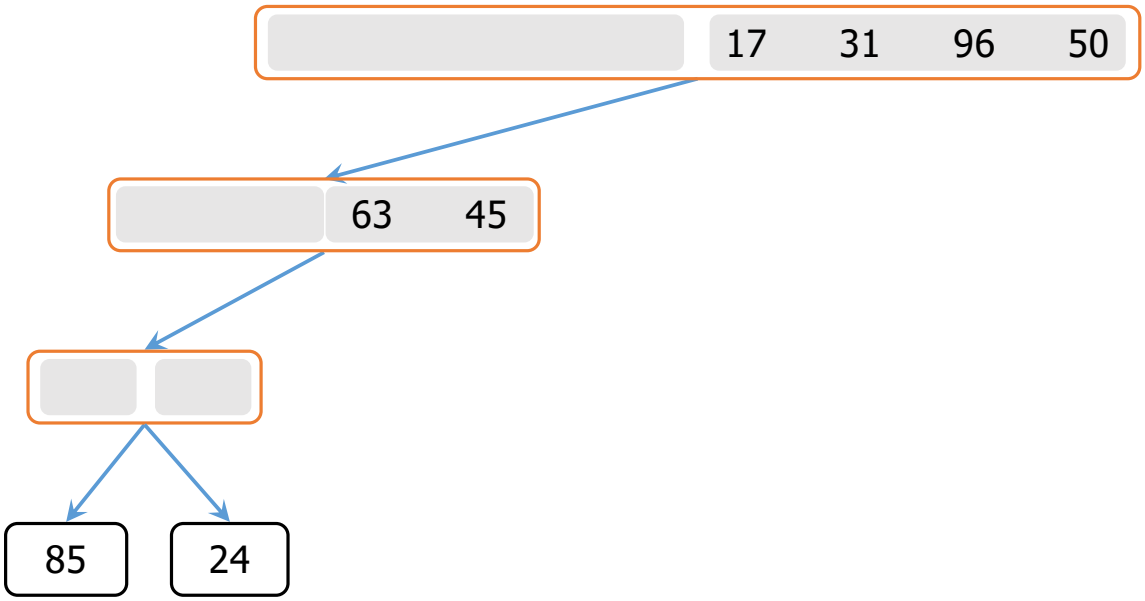



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

  if D.size() > 1
    (D1, D2) ← partition(D, n/2)
    mergeSort(D1)
    mergeSort(D2)
    D ← merge(D1, D2)
```

Merge-Sort Algorithm

Conquer

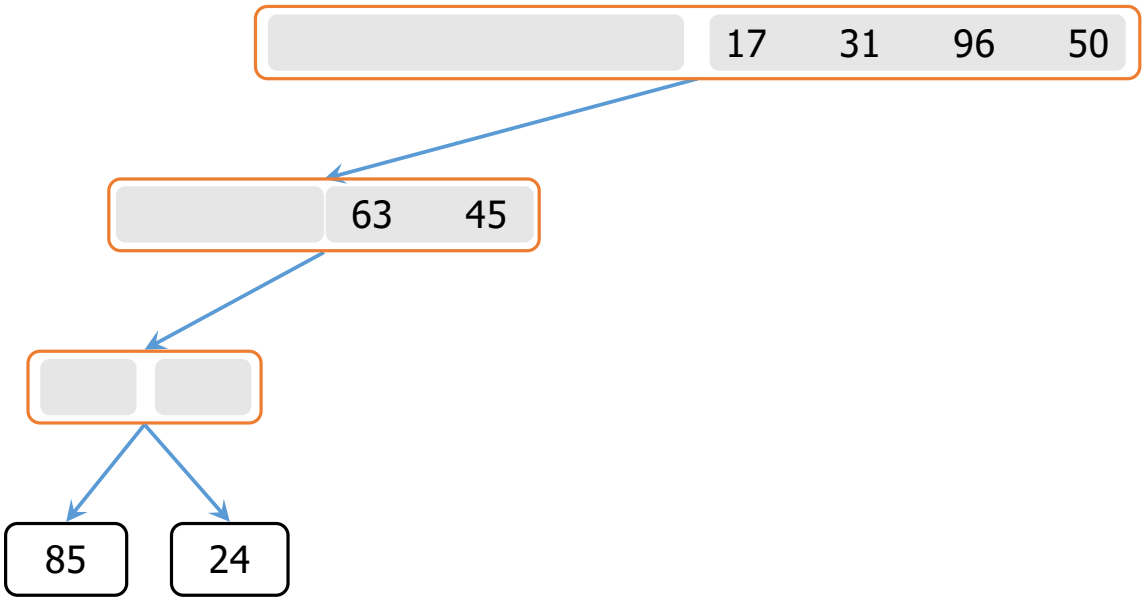



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Merge-Sort Algorithm

Base case

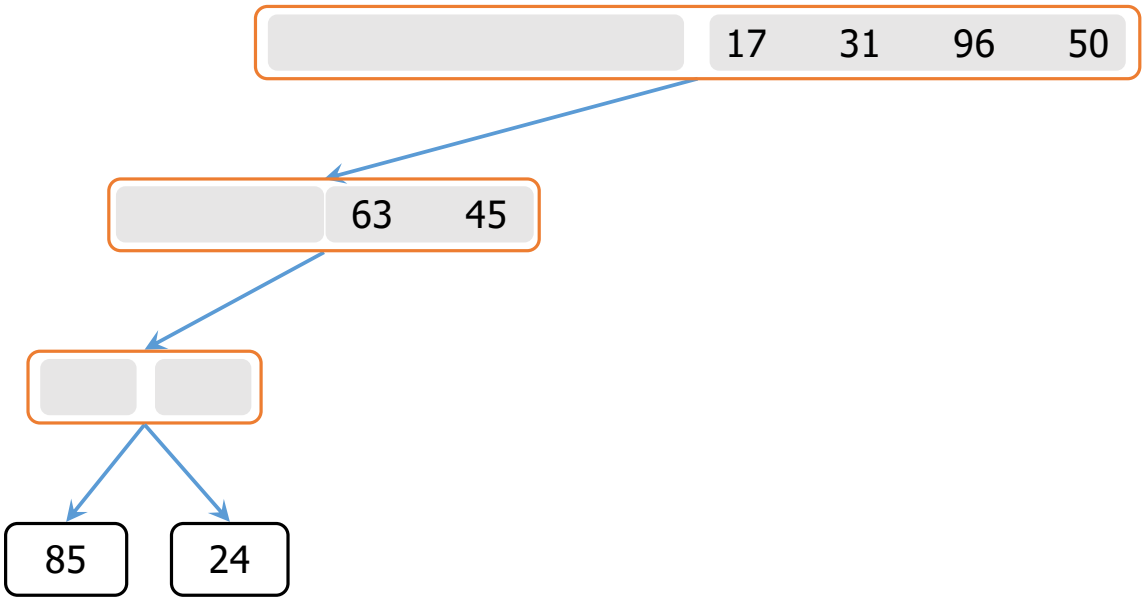



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

  if D.size() > 1
    (D1, D2) ← partition(D, n/2)
    mergeSort(D1)
    mergeSort(D2)
    D ← merge(D1, D2)
```

Merge-Sort Algorithm

Base case

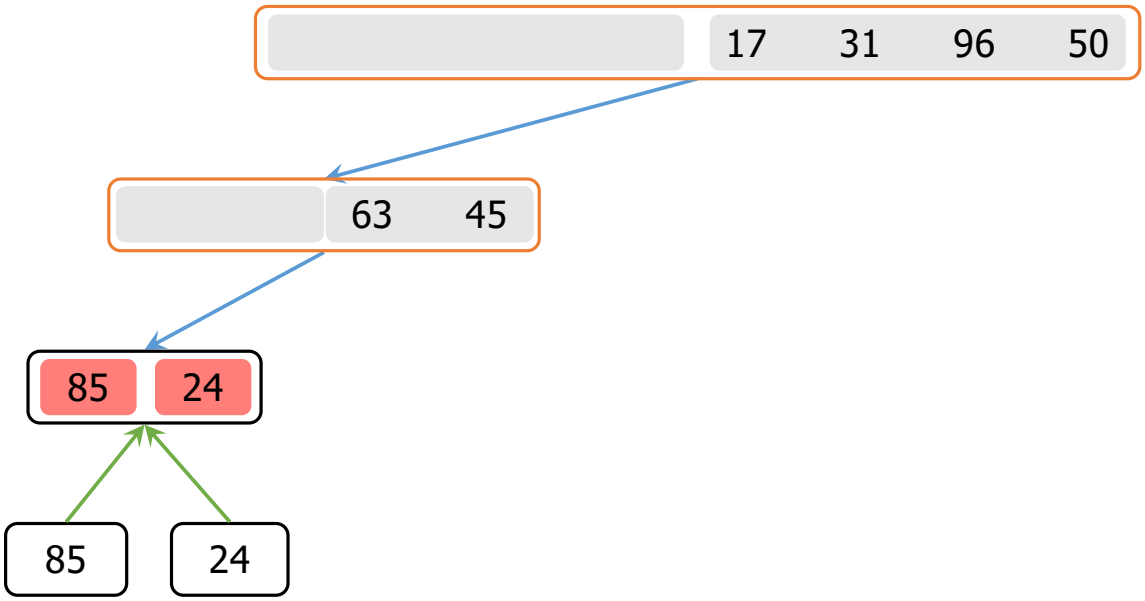



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

  if D.size() > 1
    (D1, D2) ← partition(D, n/2)
    mergeSort(D1)
    mergeSort(D2)
    D ← merge(D1, D2)
```

Merge-Sort Algorithm

Merge

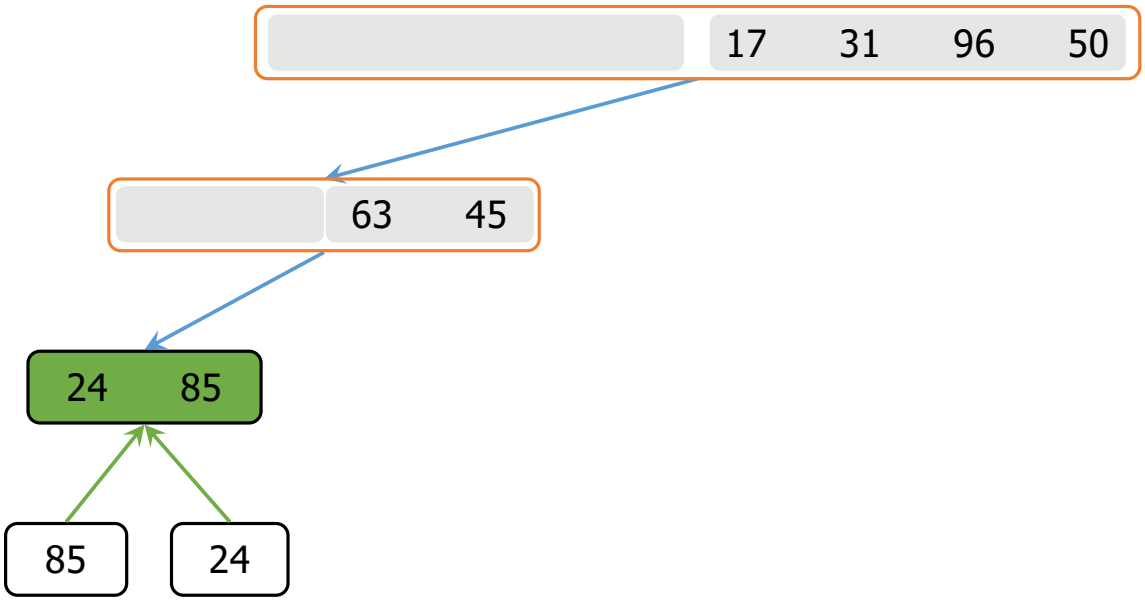



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Merge-Sort Algorithm

Merge

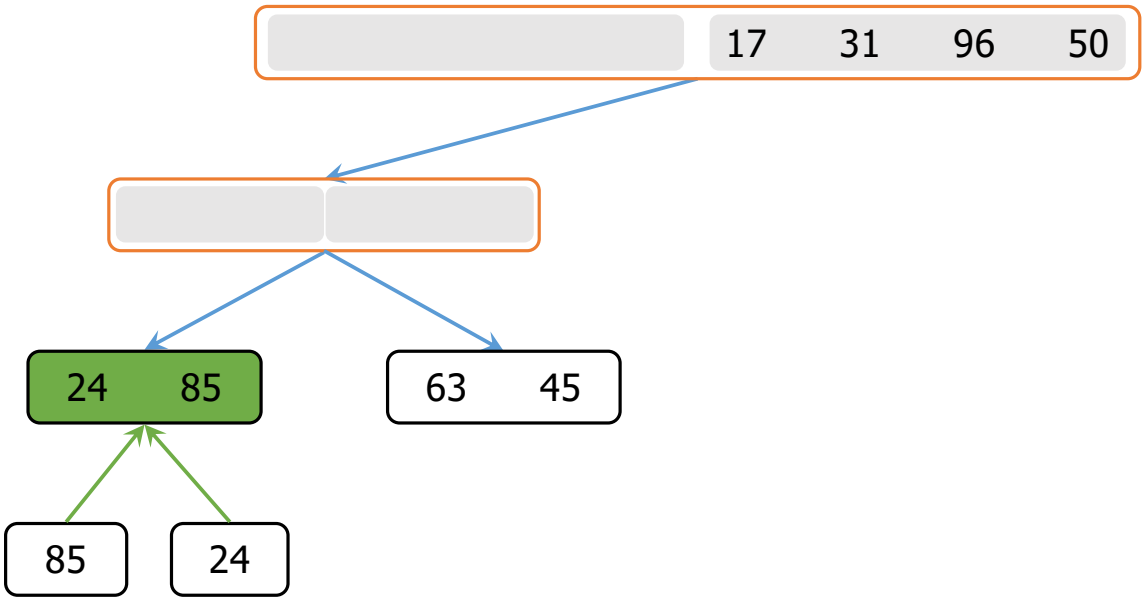



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Merge-Sort Algorithm

Conquer

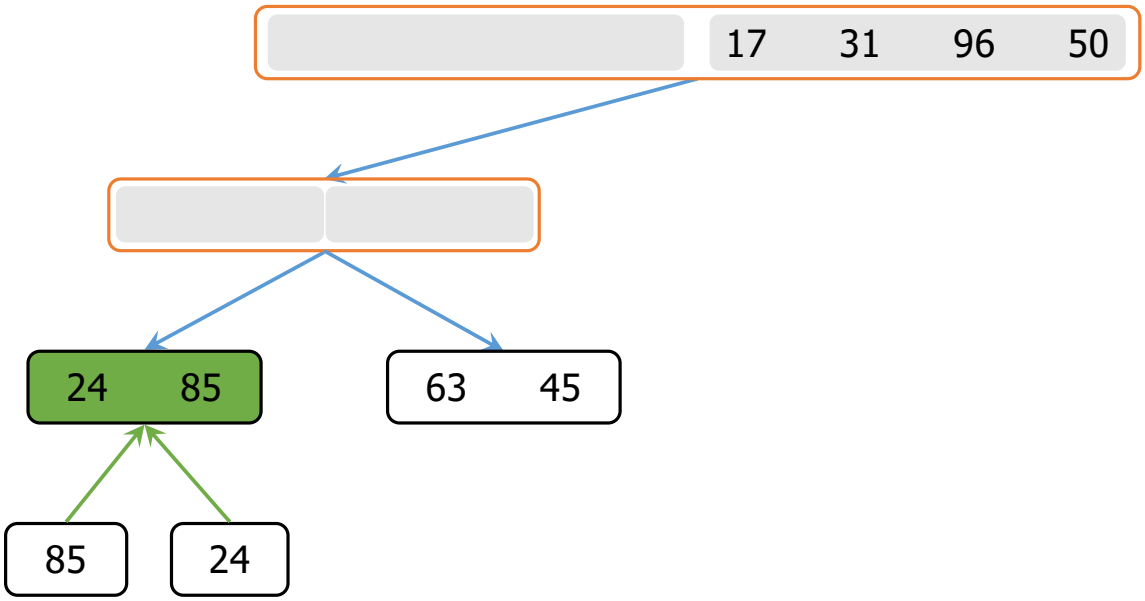



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Merge-Sort Algorithm

Conquer

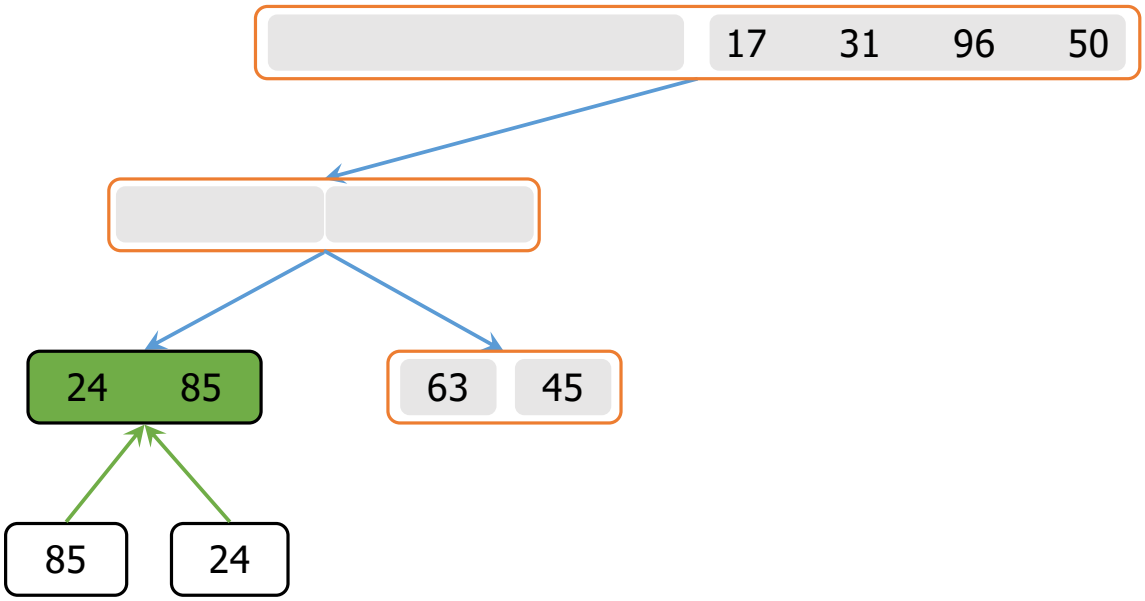



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

  if D.size() > 1
    (D1, D2) ← partition(D, n/2)
    mergeSort(D1)
    mergeSort(D2)
    D ← merge(D1, D2)
```

Merge-Sort Algorithm

Divide

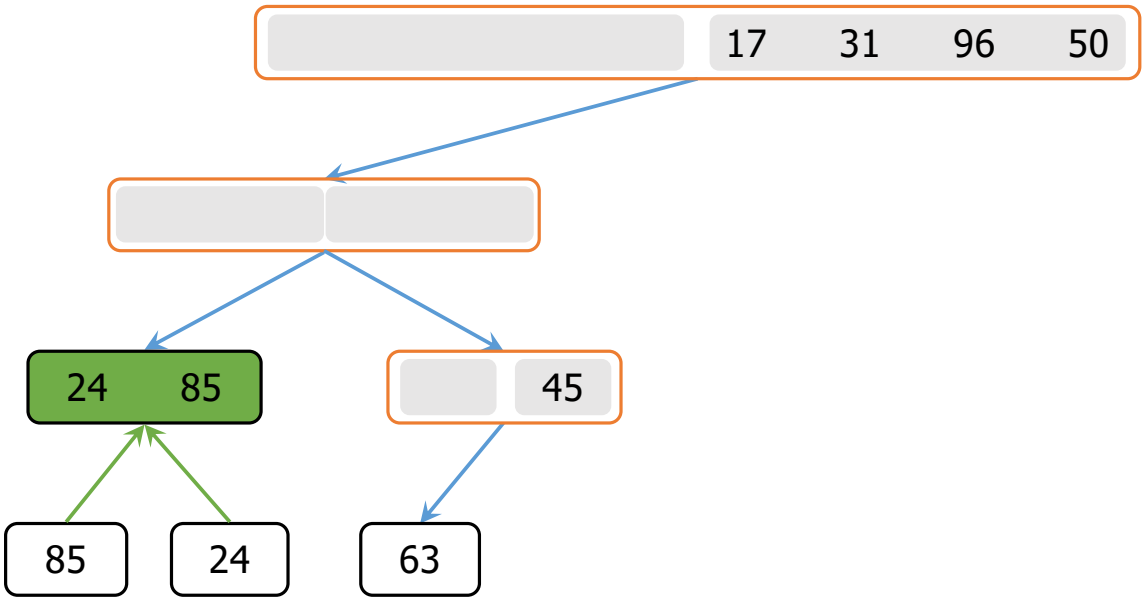



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Merge-Sort Algorithm

Conquer

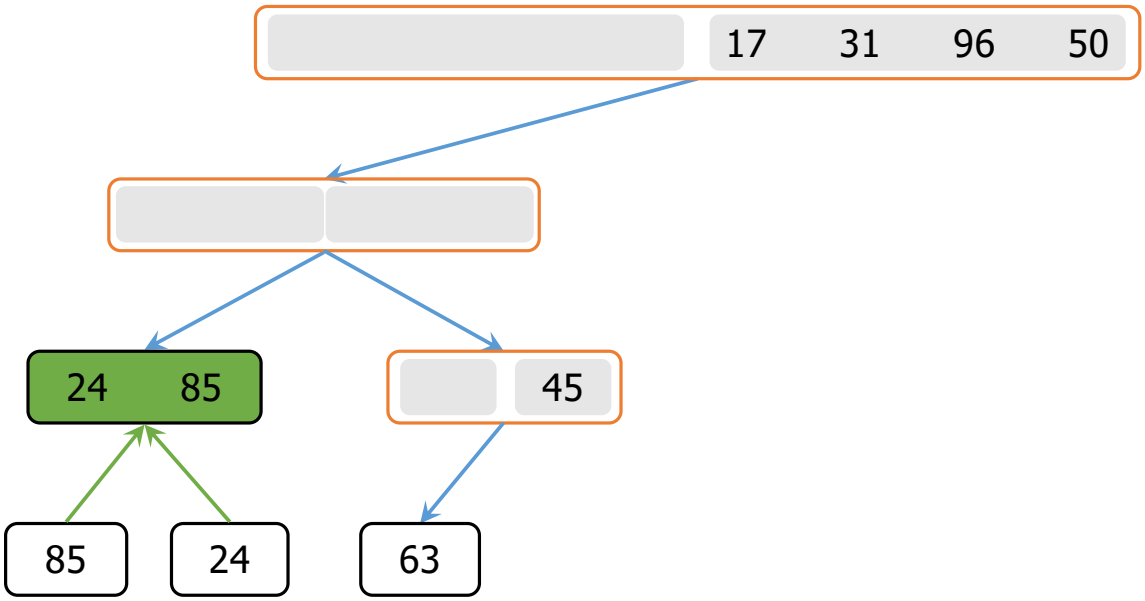



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Merge-Sort Algorithm

Base case

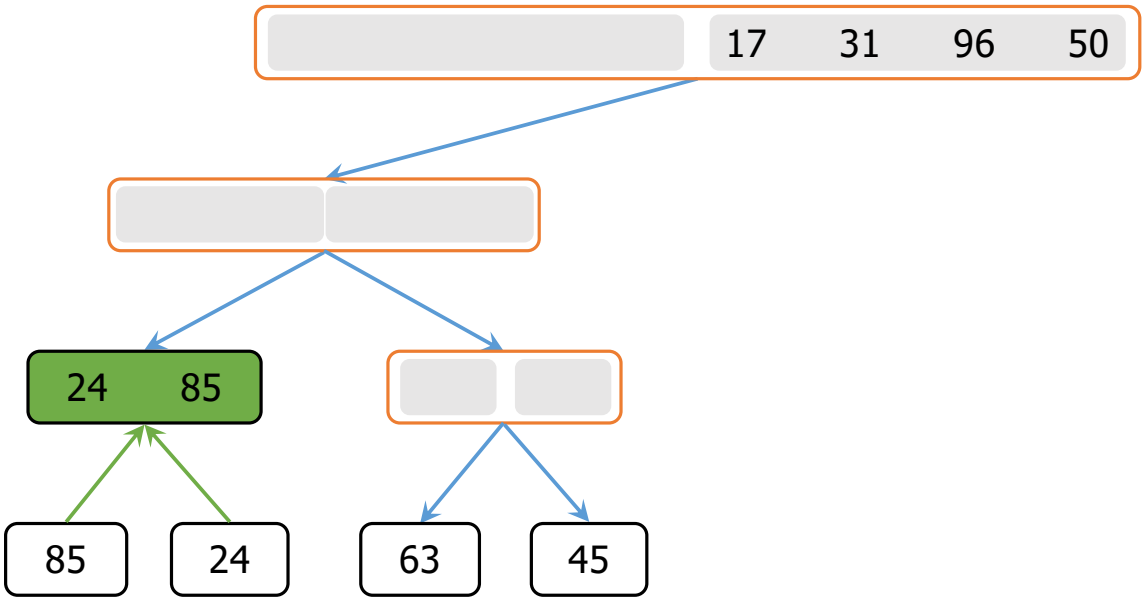



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

  if D.size() > 1
    (D1, D2) ← partition(D, n/2)
    mergeSort(D1)
    mergeSort(D2)
    D ← merge(D1, D2)
```

Merge-Sort Algorithm

Conquer

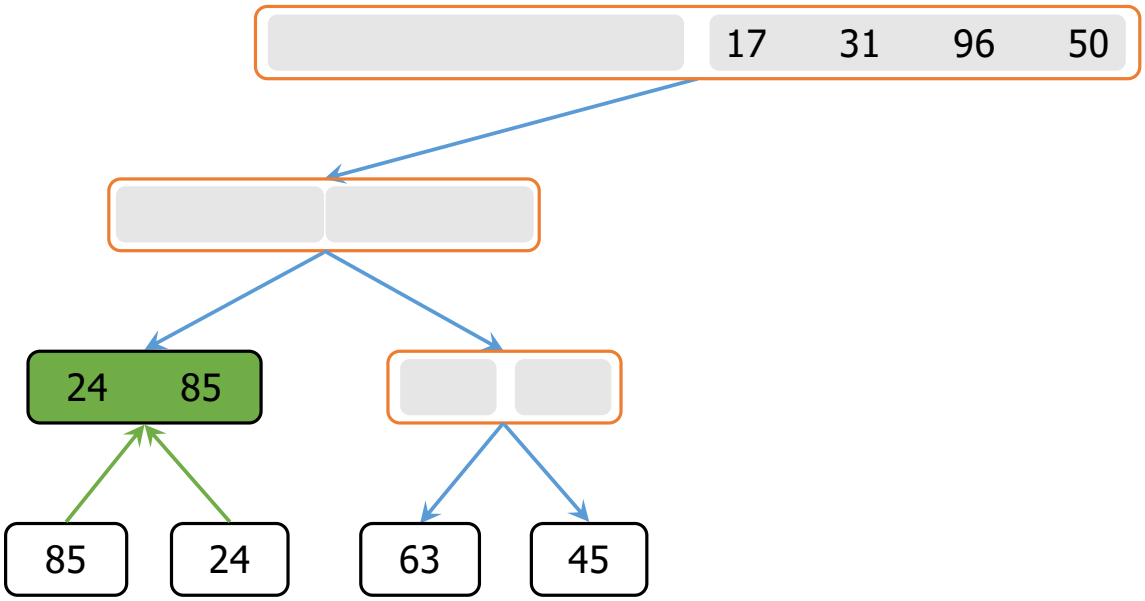



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Merge-Sort Algorithm

Base case

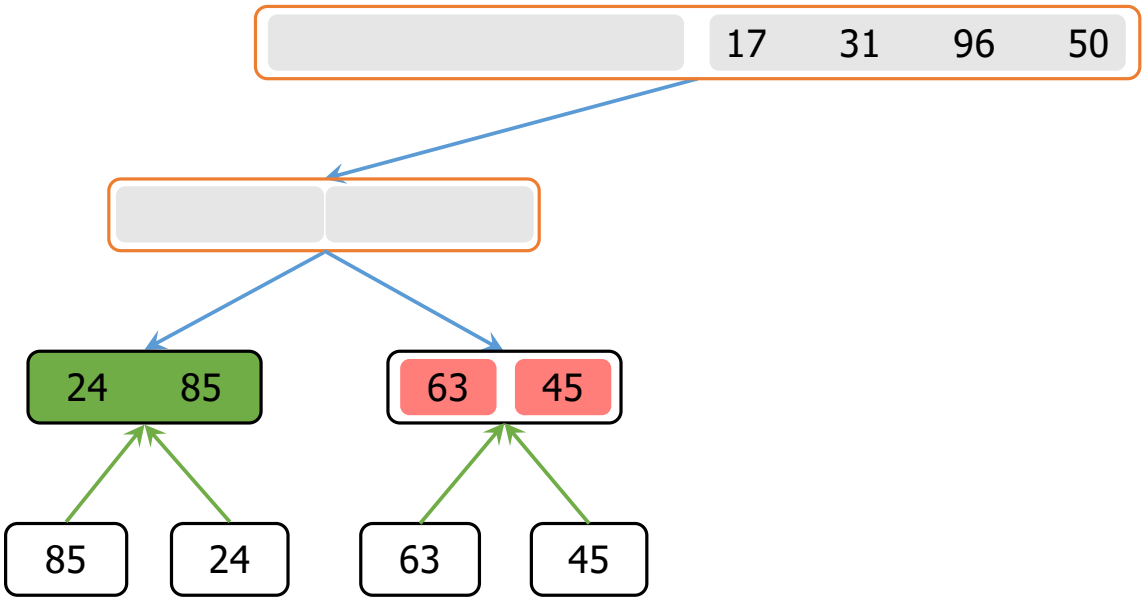



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

  if D.size() > 1
    (D1, D2) ← partition(D, n/2)
    mergeSort(D1)
    mergeSort(D2)
    D ← merge(D1, D2)
```

Merge-Sort Algorithm

Merge

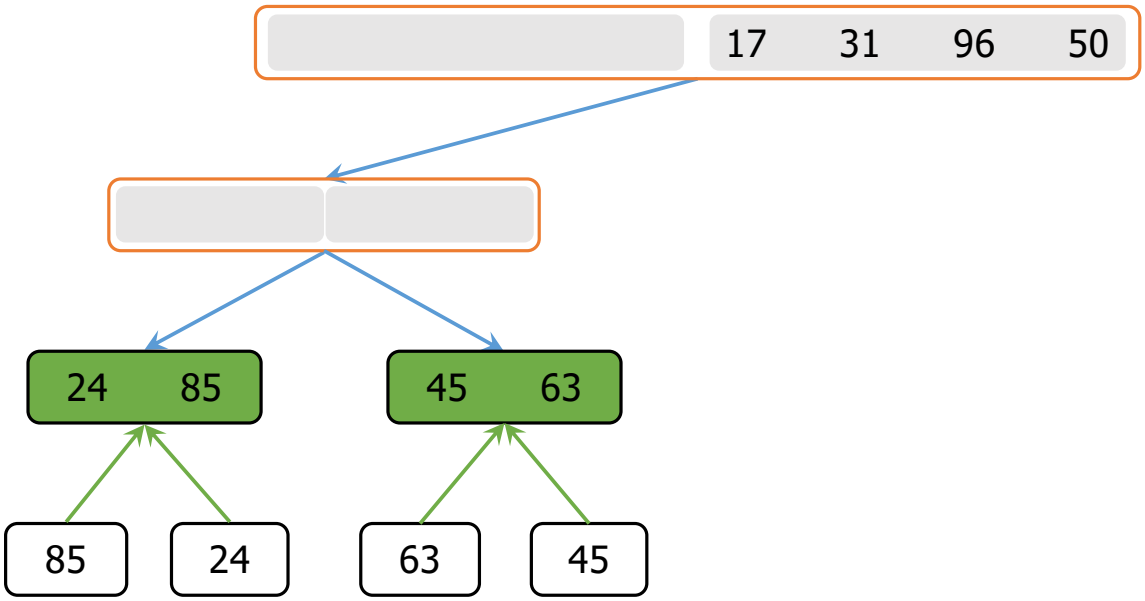



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Merge-Sort Algorithm

Merge

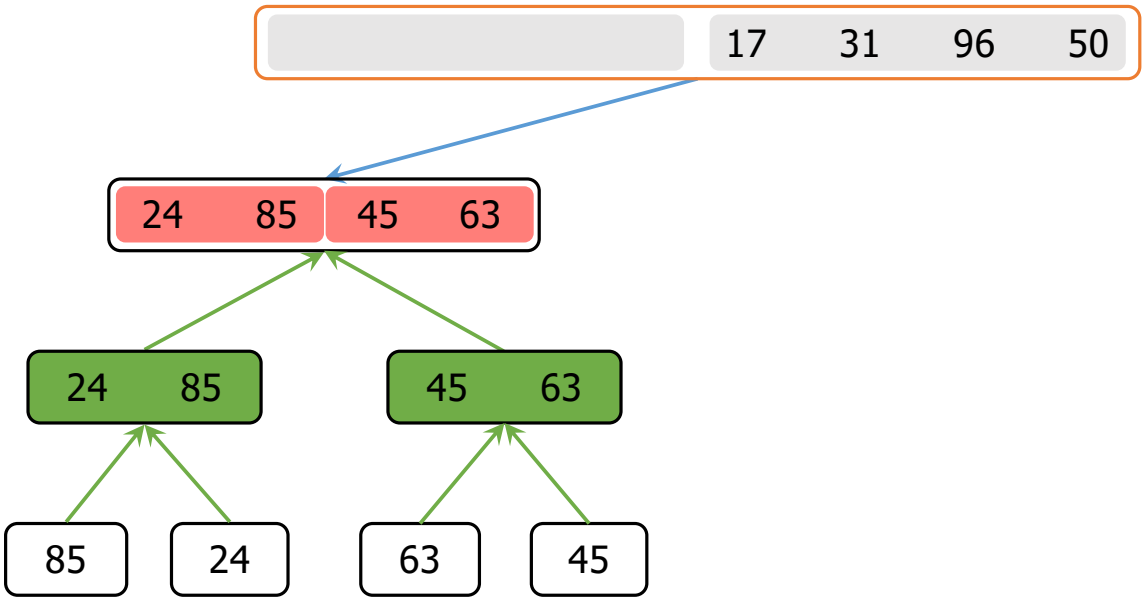



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Merge-Sort Algorithm

Merge

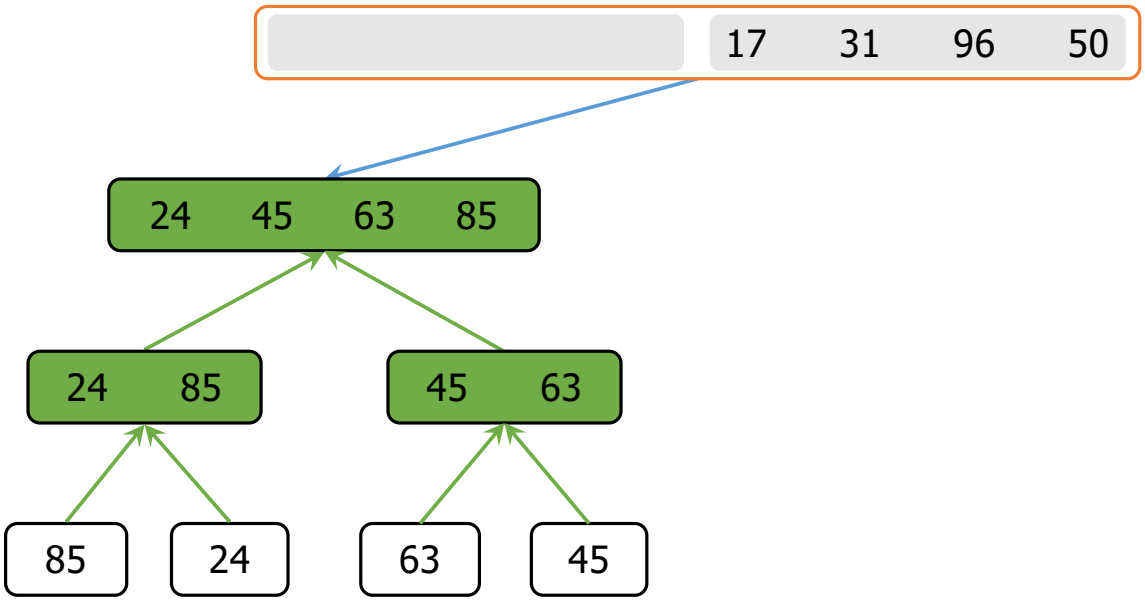



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Merge-Sort Algorithm

Merge

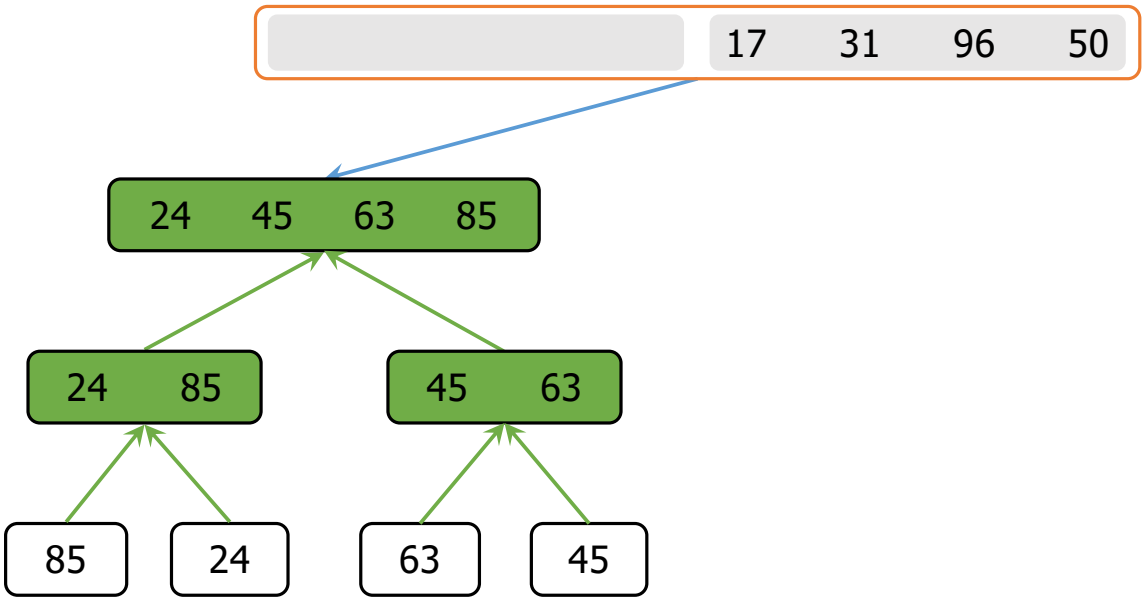



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Merge-Sort Algorithm

Conquer

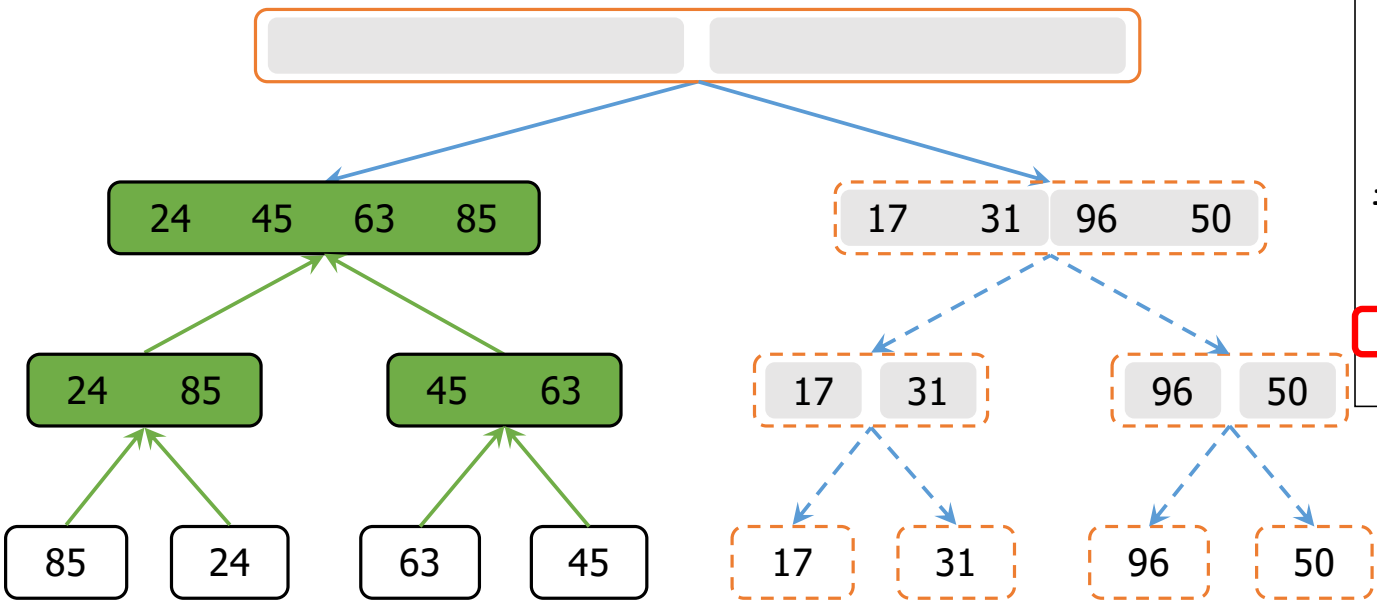



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Merge-Sort Algorithm

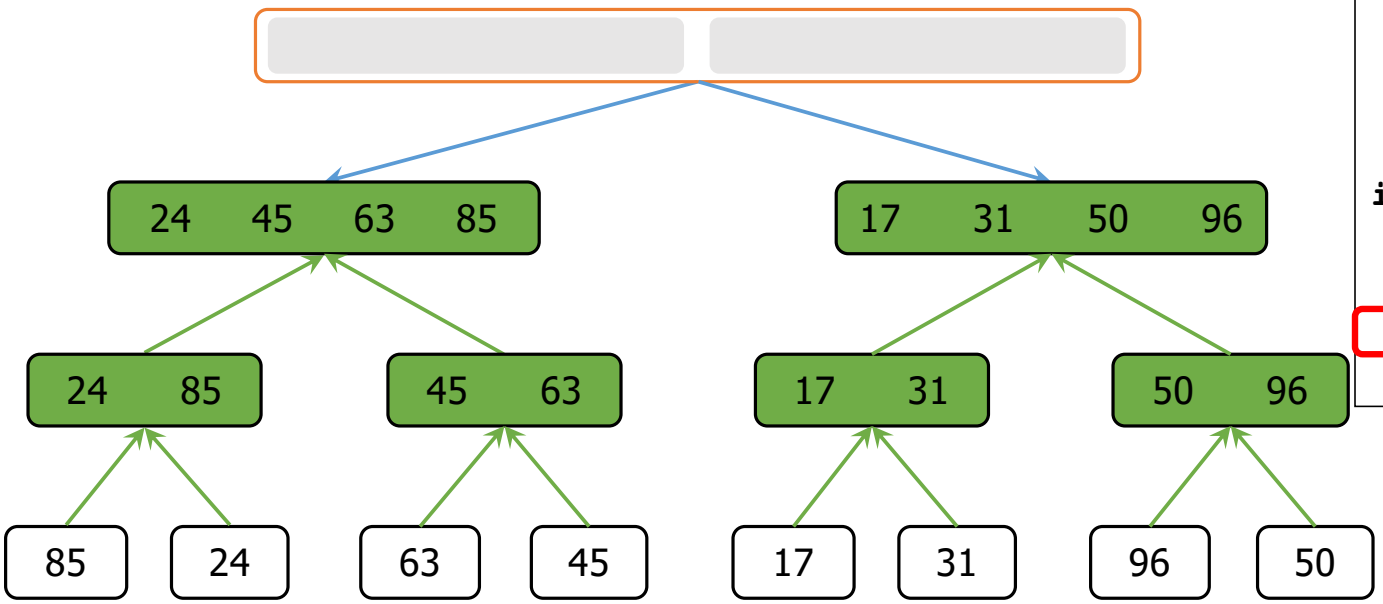
Conquer




```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Merge-Sort Algorithm

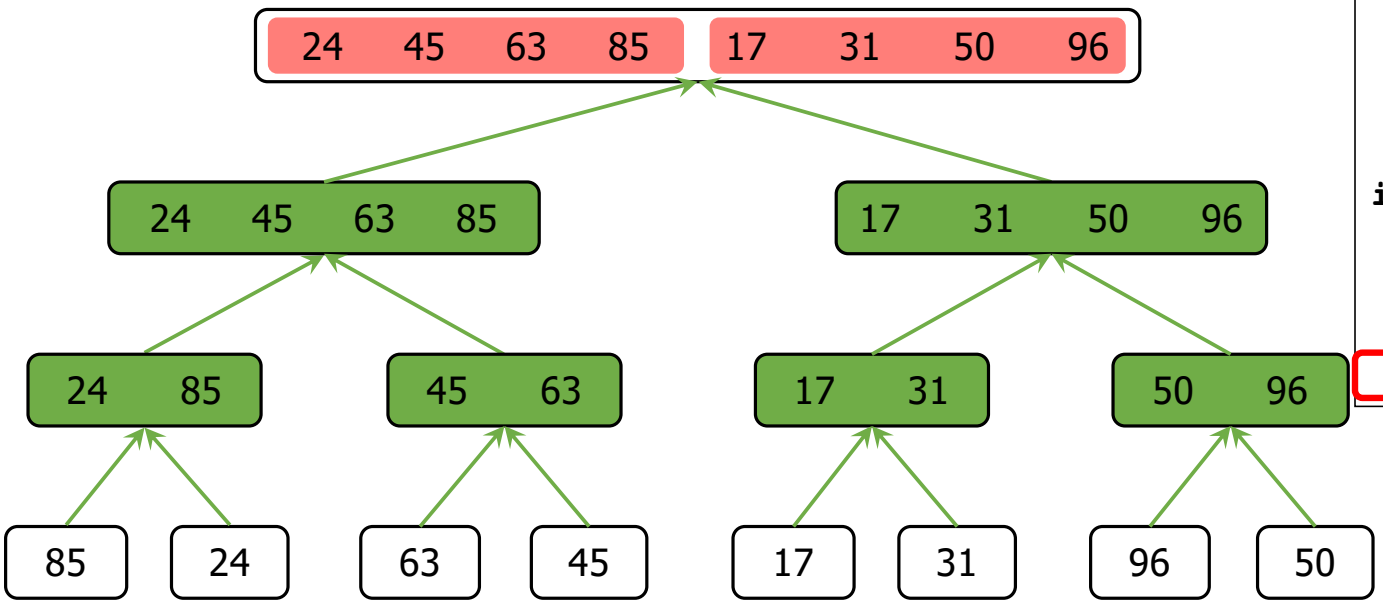



```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Merge-Sort Algorithm

Merge



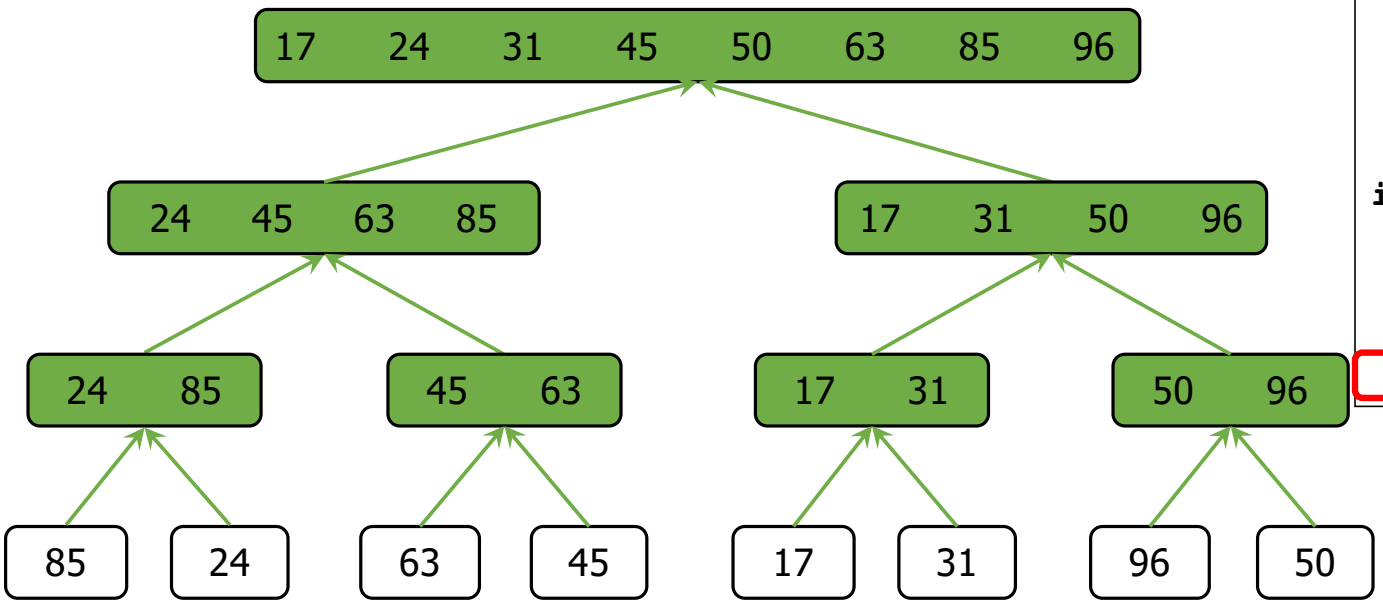

```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Merge-Sort Algorithm

Finished Sorting

Merge




```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Activity

- Write a python code for merge-sort algorithm

```
Algorithm mergeSort(D)
  Input sequence D with n
        elements
  Output sequence D sorted

if D.size() > 1
  (D1, D2) ← partition(D, n/2)
  mergeSort(D1)
  mergeSort(D2)
  D ← merge(D1, D2)
```

Activity Solution

- Write a python code for merge-sort algorithm

```
def mergesort(D):
    n = len(D)
    if n > 1:
        mid = n // 2
        #DIVIDING the data into half
        D1 = D[ : mid]
        D2 = D[mid : ]

        #CONQUERING through recursion
        mergesort(D1)
        mergesort(D2)

        #Combining/merging the data in order
        merge(D1, D2, D)
```

```
Algorithm mergeSort(D)
 Input sequence D with n
       elements
 Output sequence D sorted

if D.size() > 1
 (D1, D2) ← partition(D, n/2)
 mergeSort(D1)
 mergeSort(D2)
 D ← merge(D1, D2)
```

Merge-Sort Algorithm

`merge()` Algorithm

D1

24	45	63	85

i=0

D2

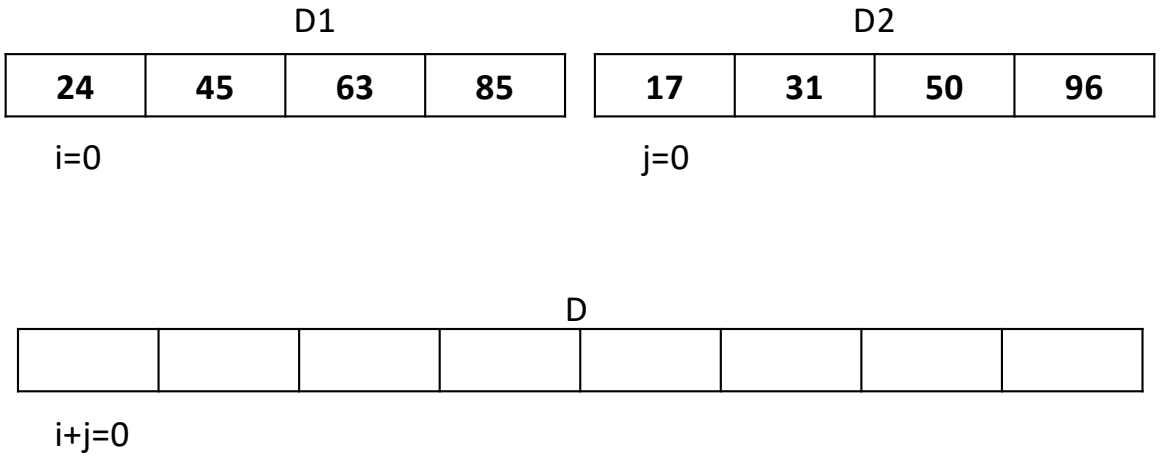
17	31	50	96

j=0

D

Merge-Sort Algorithm

`merge()` Algorithm

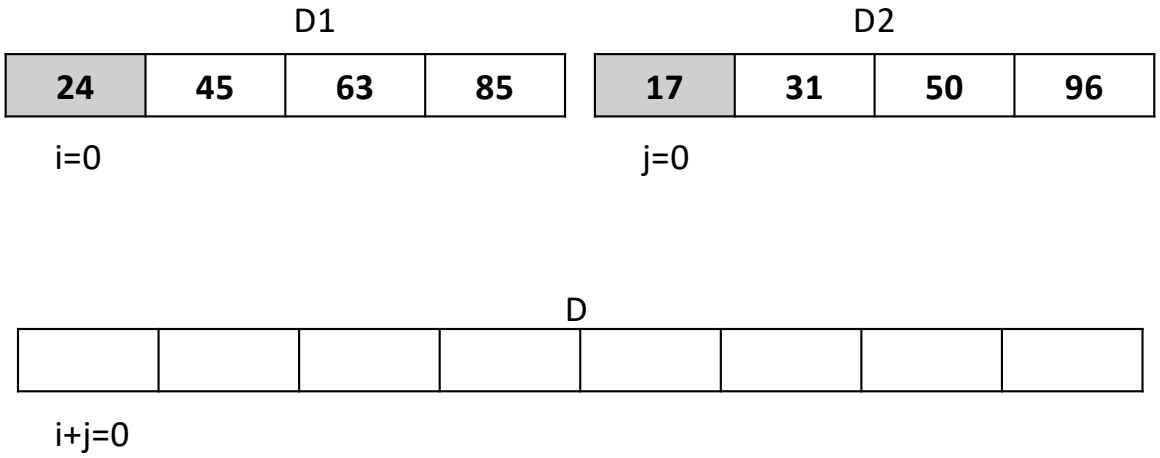



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

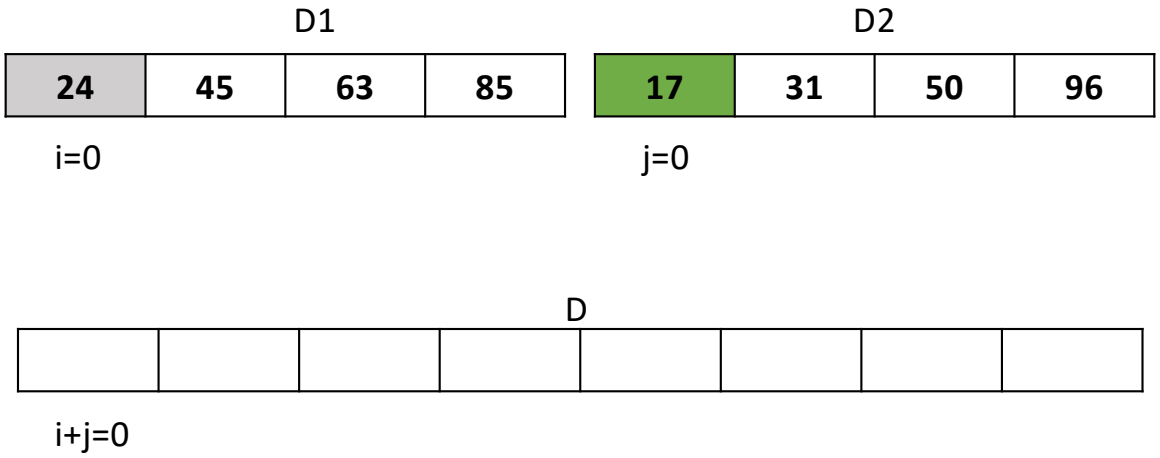



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

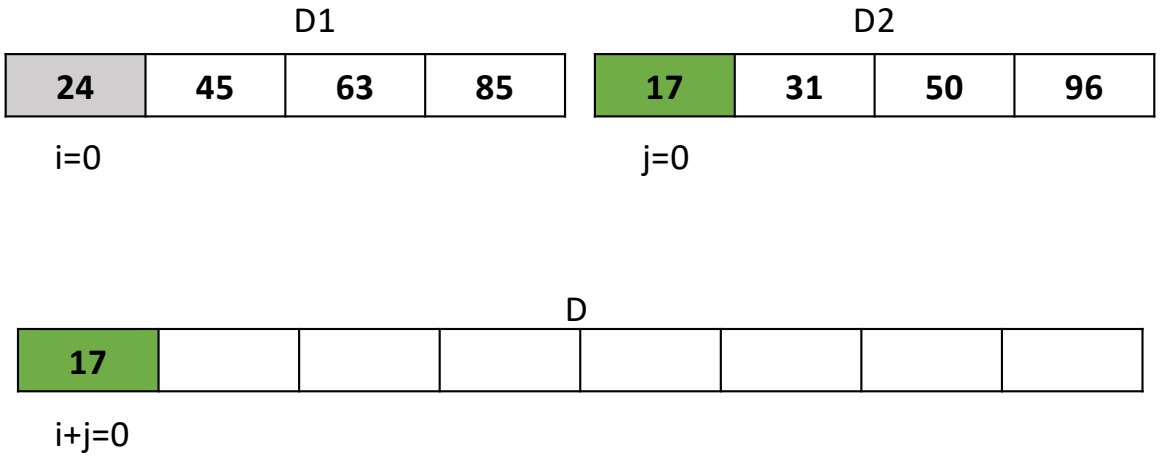



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

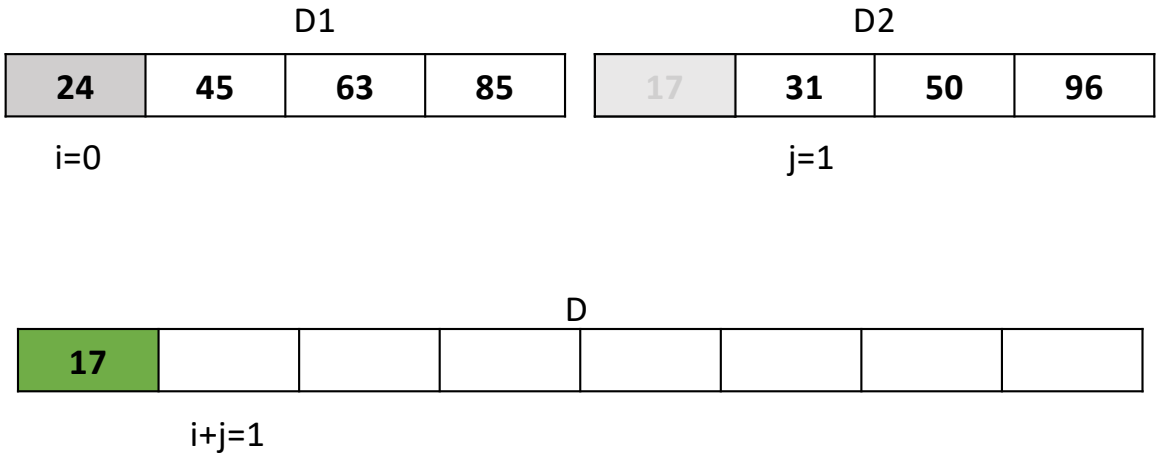



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

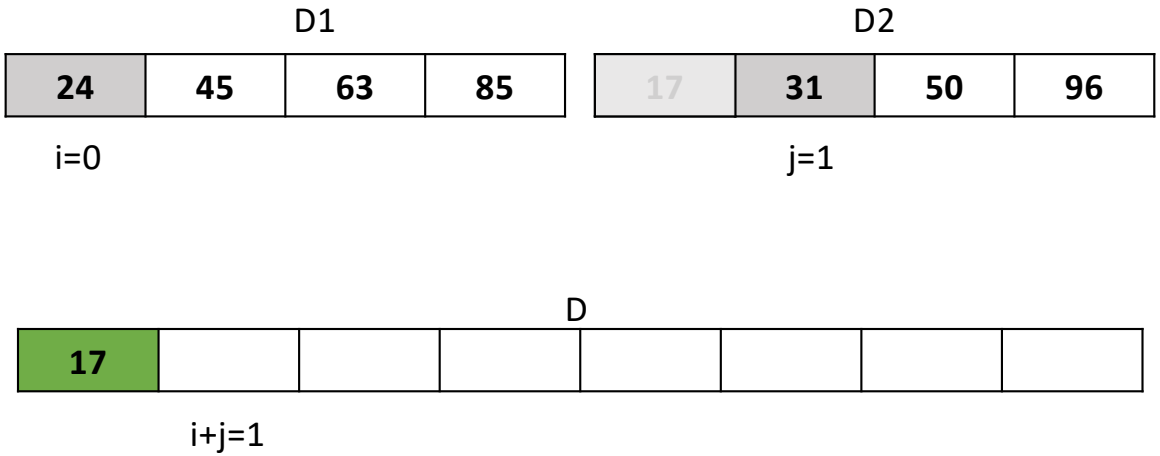



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

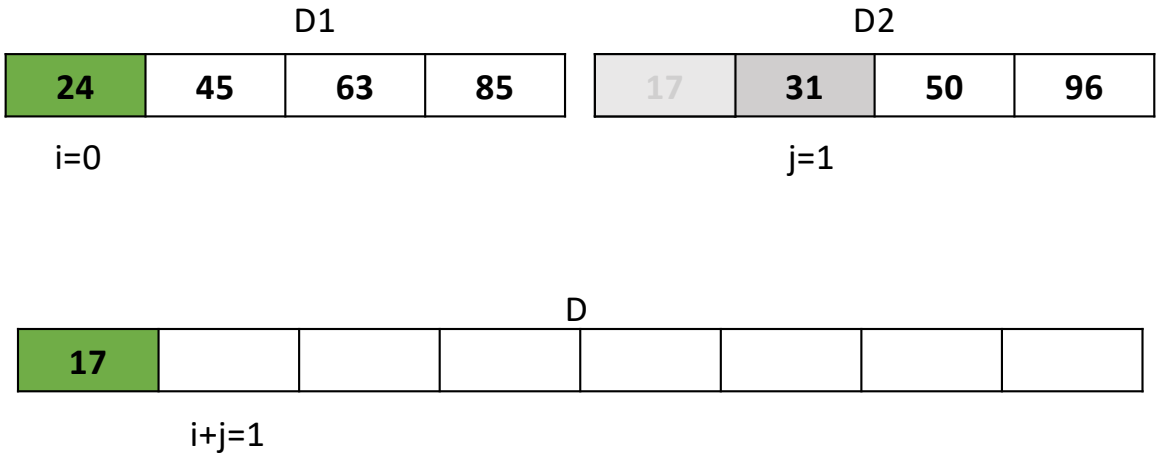



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

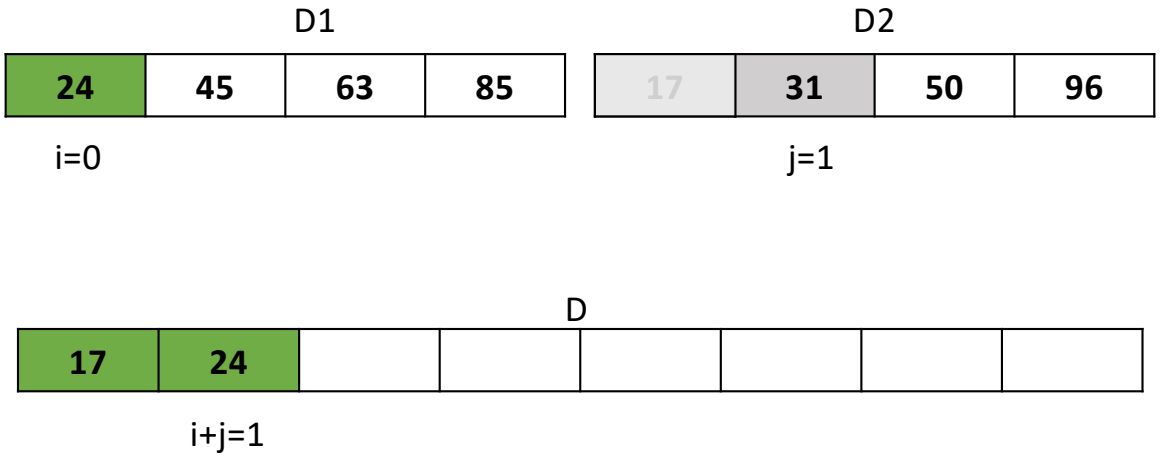



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

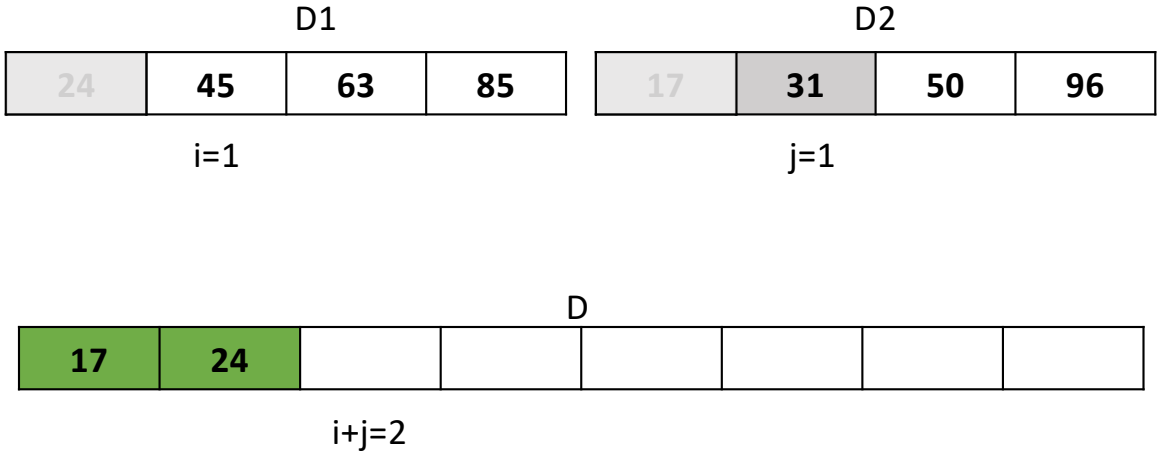



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

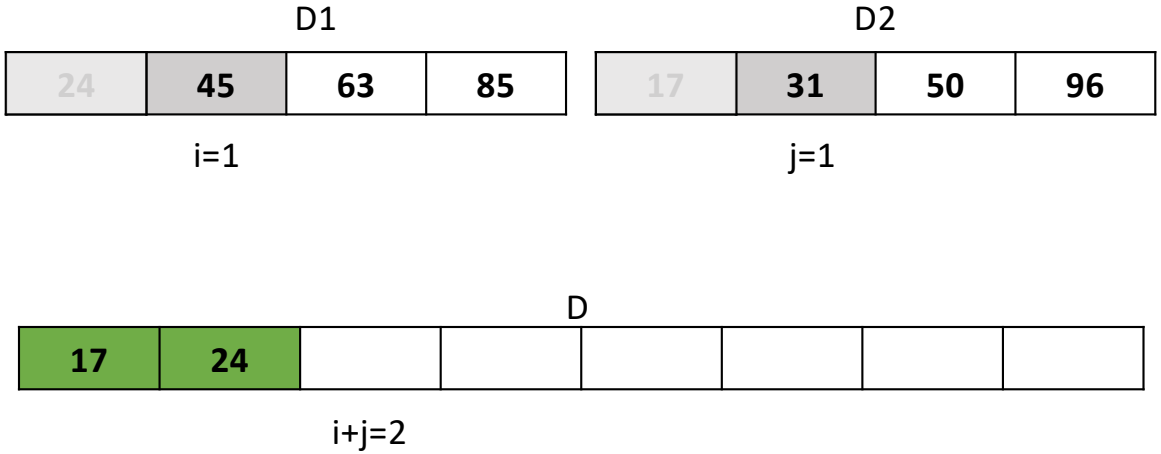



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

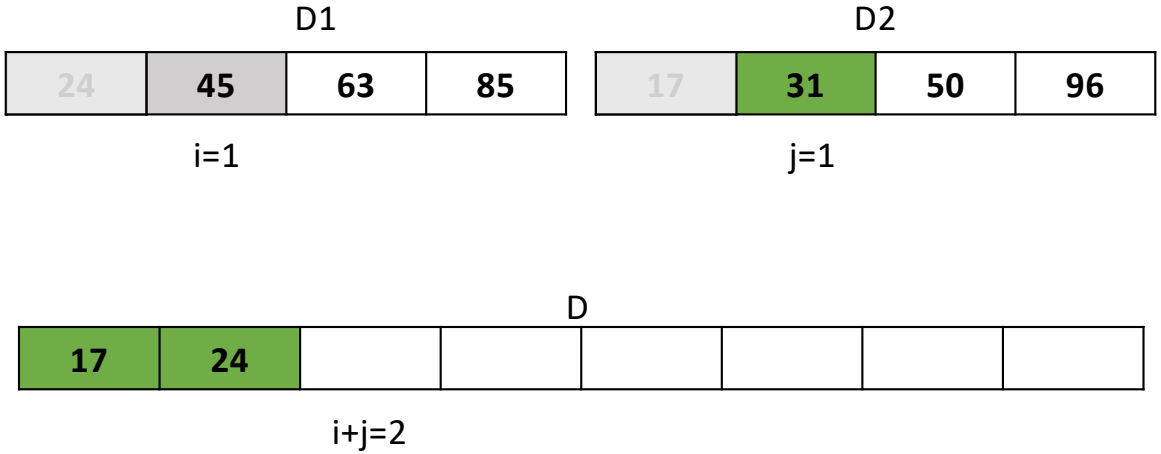



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

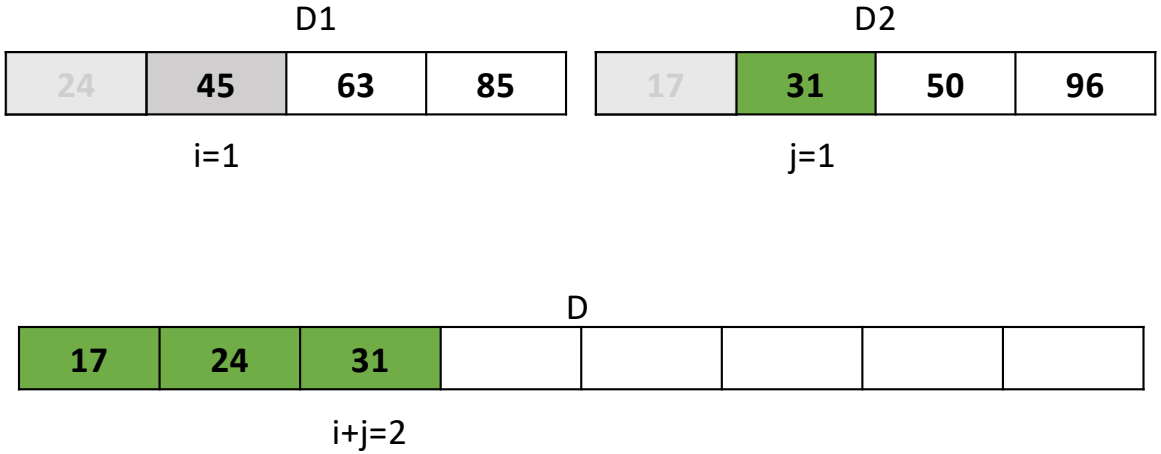



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

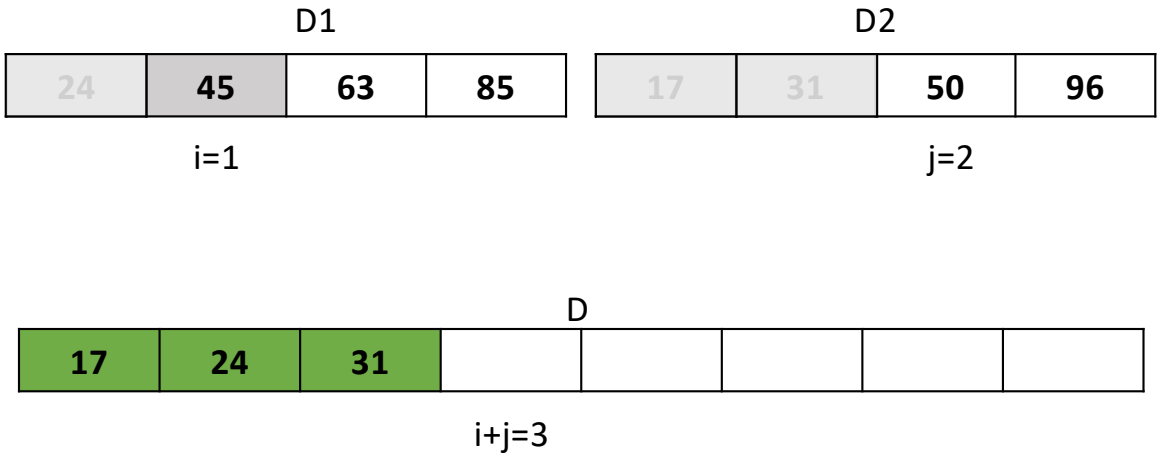



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

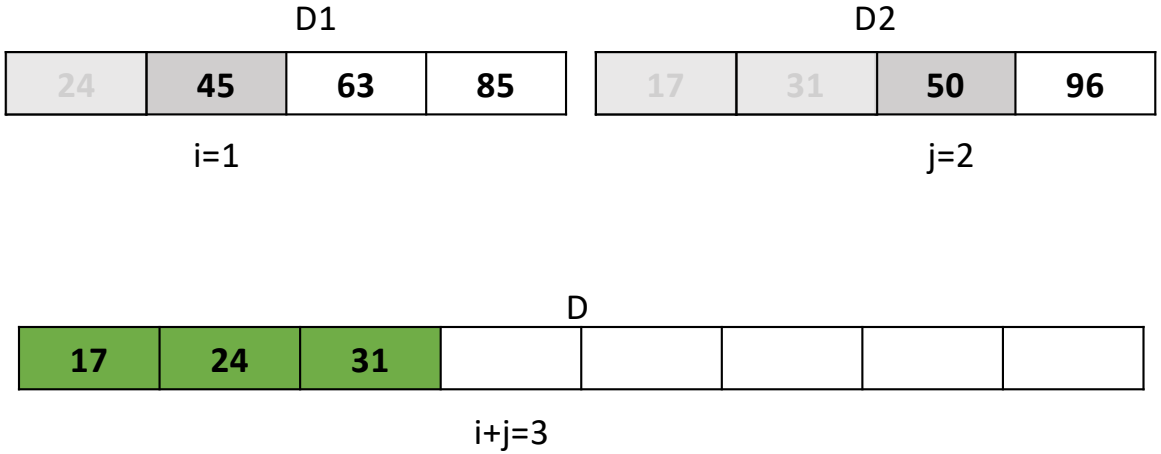



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

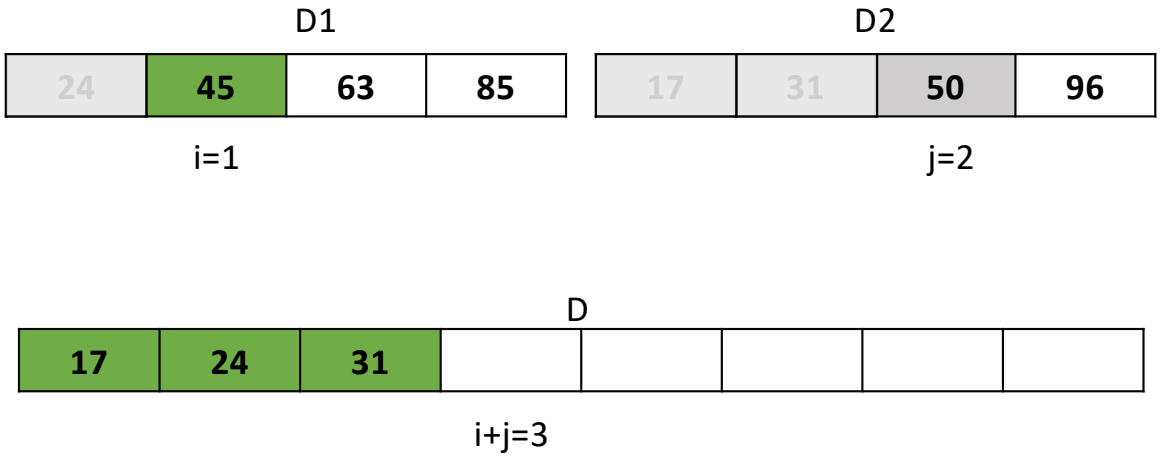



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

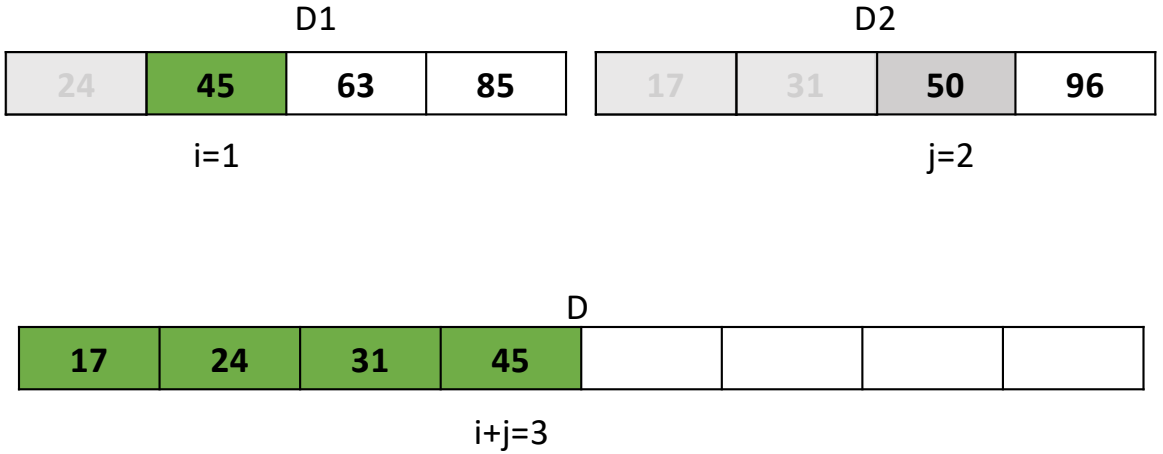



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

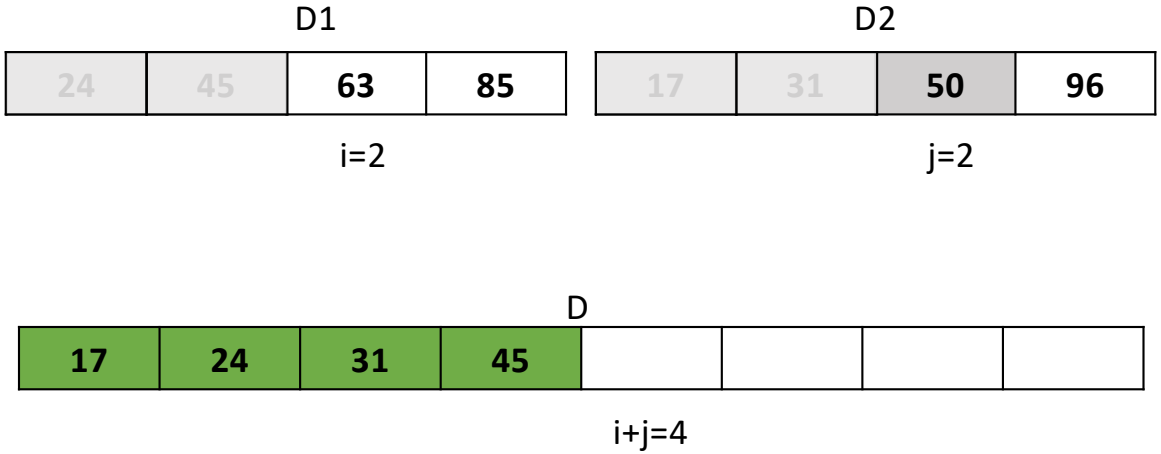



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

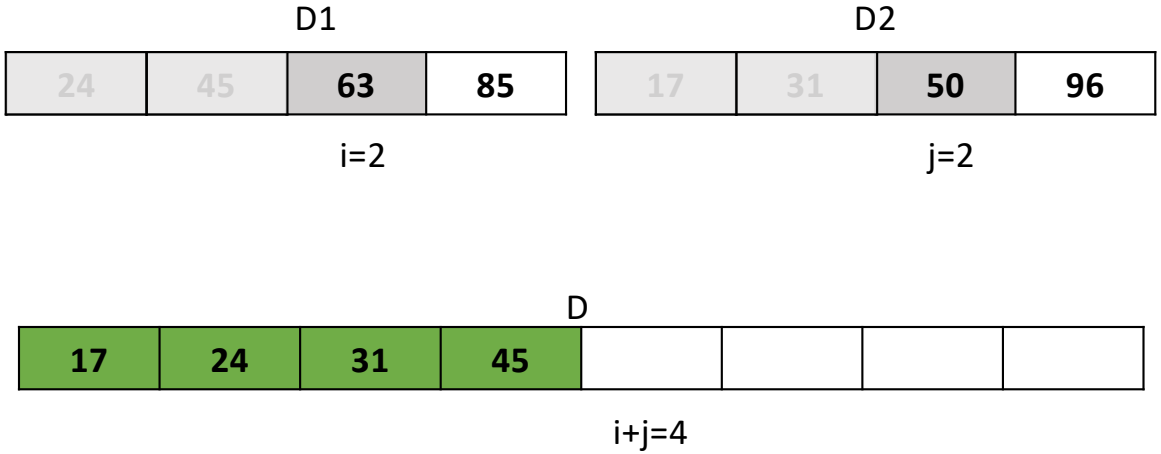



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

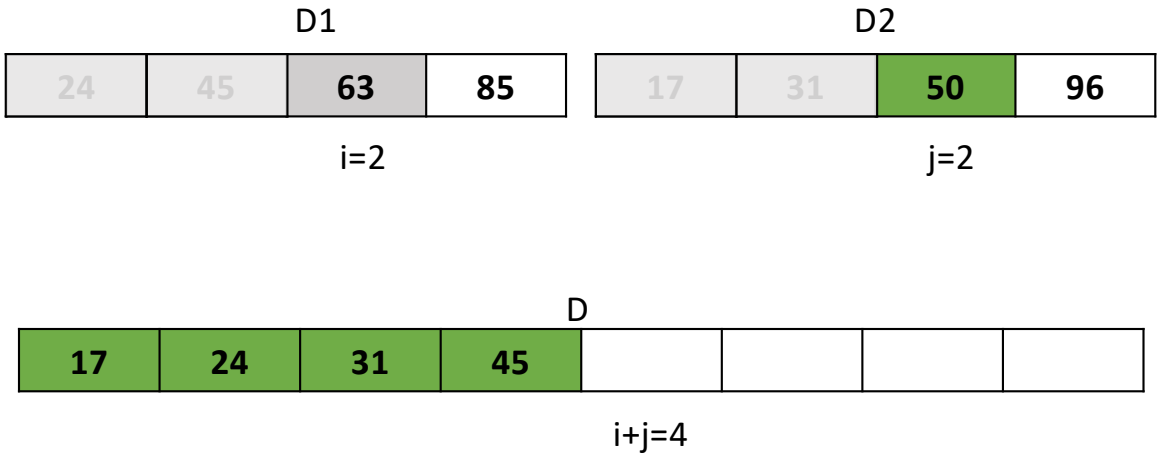



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

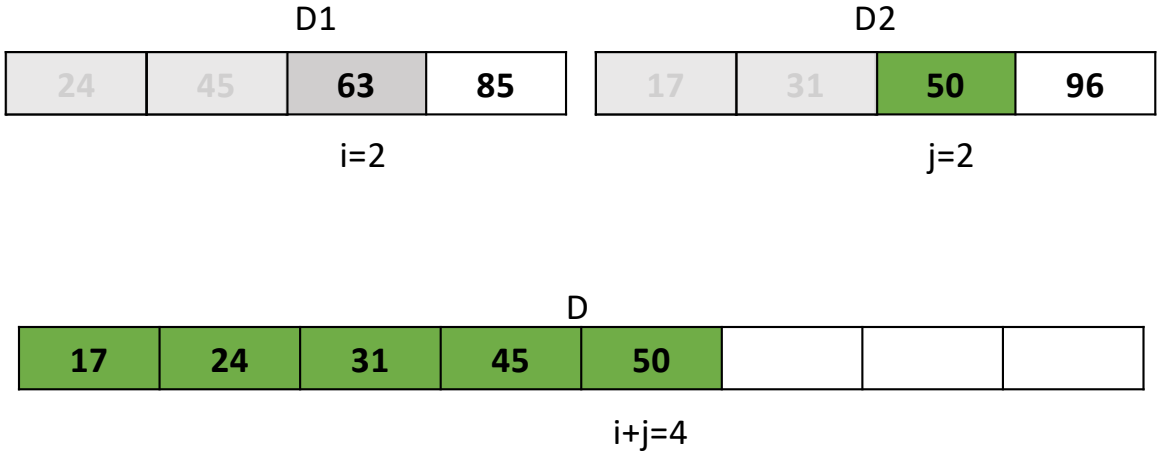



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

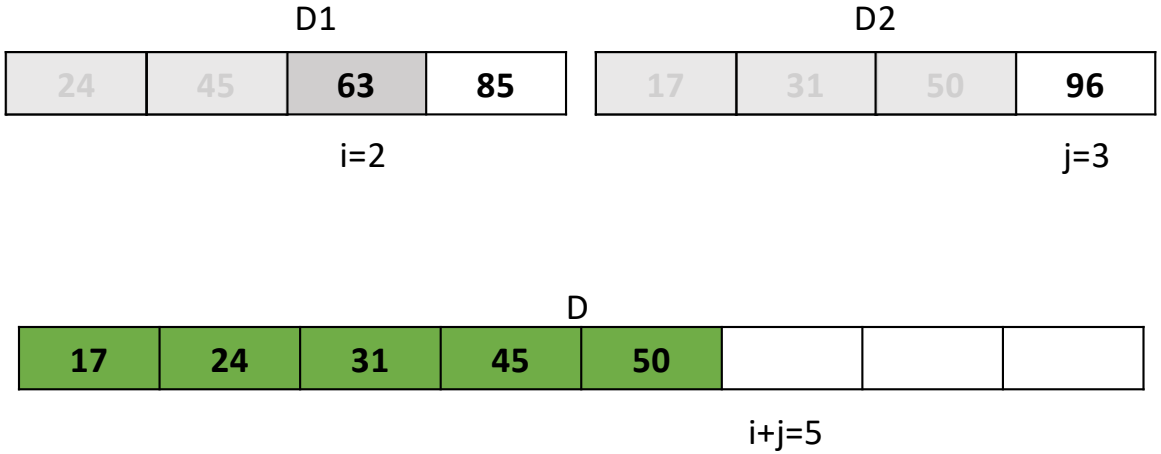



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

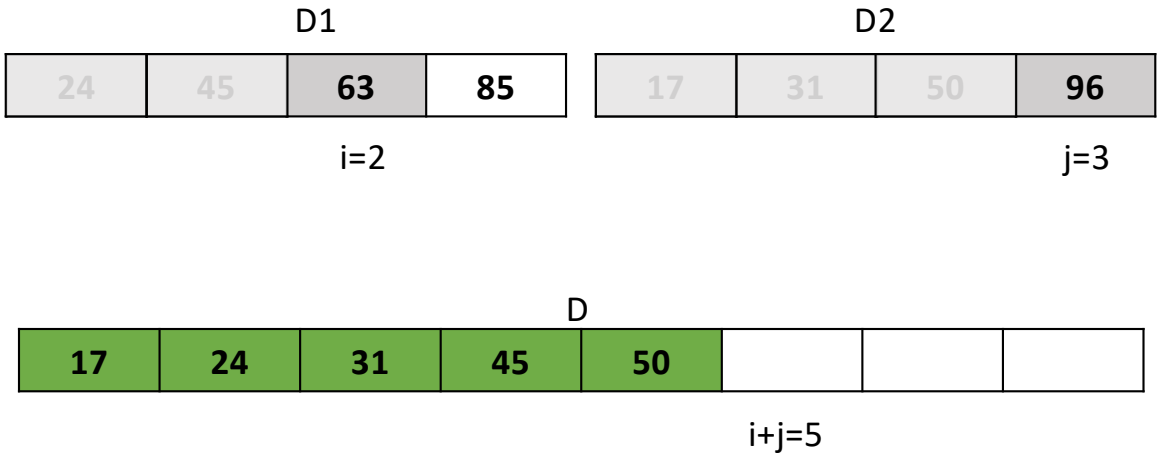



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

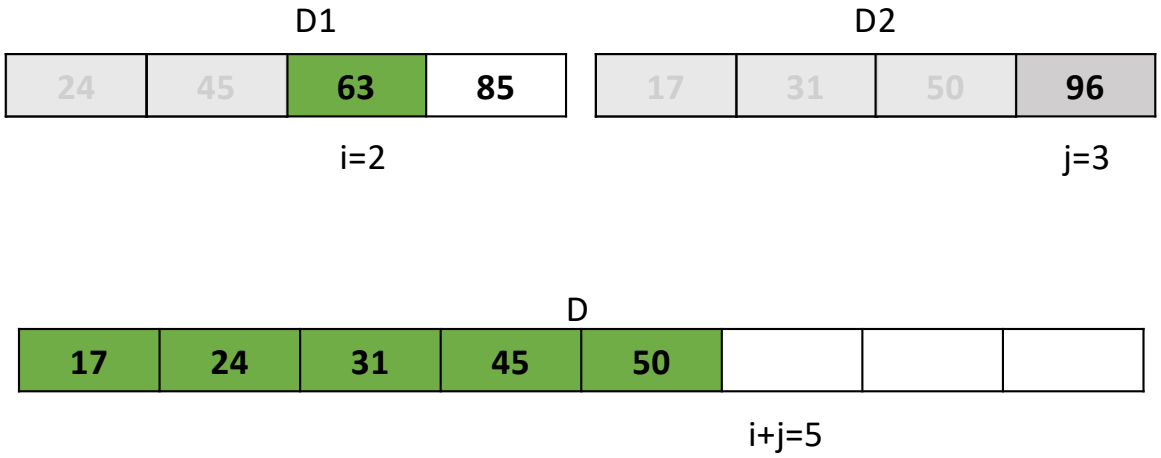



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

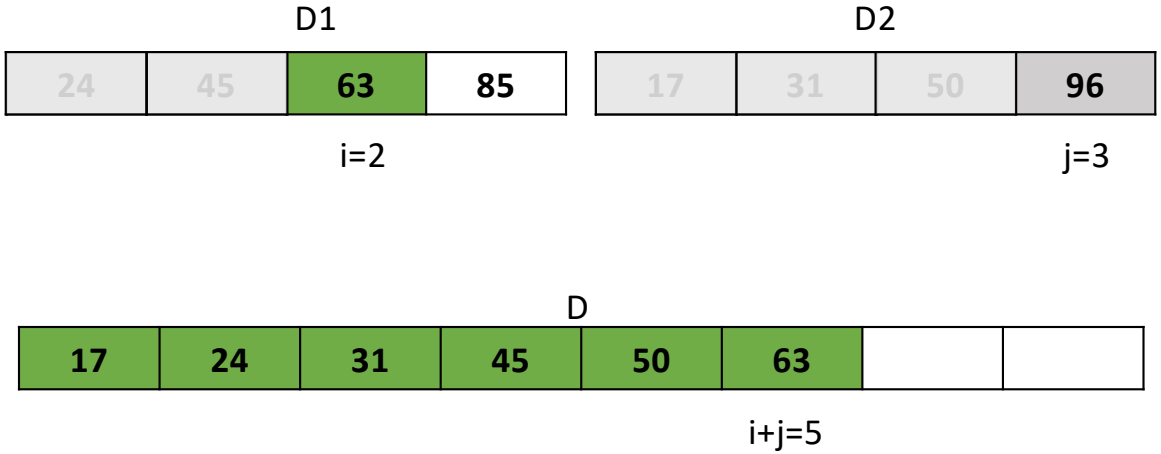



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

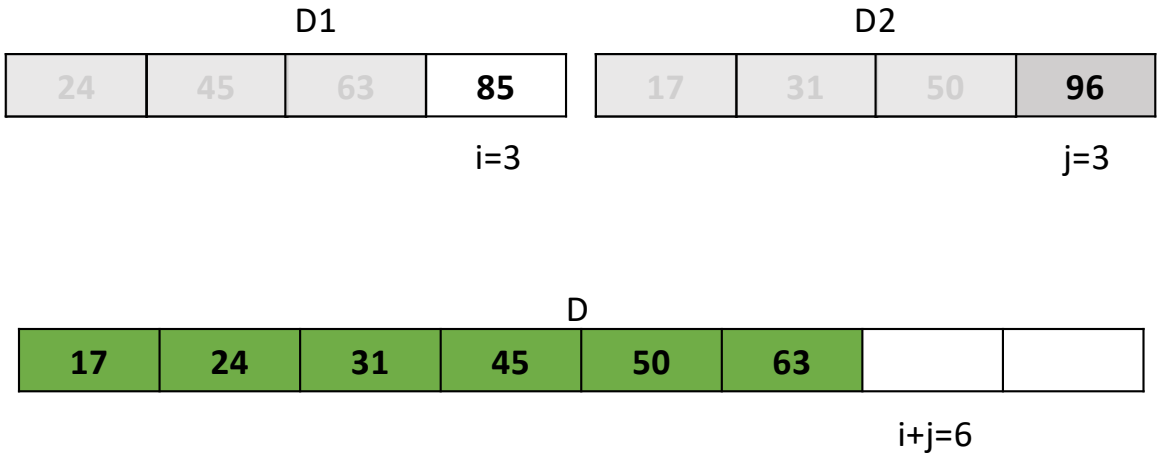



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

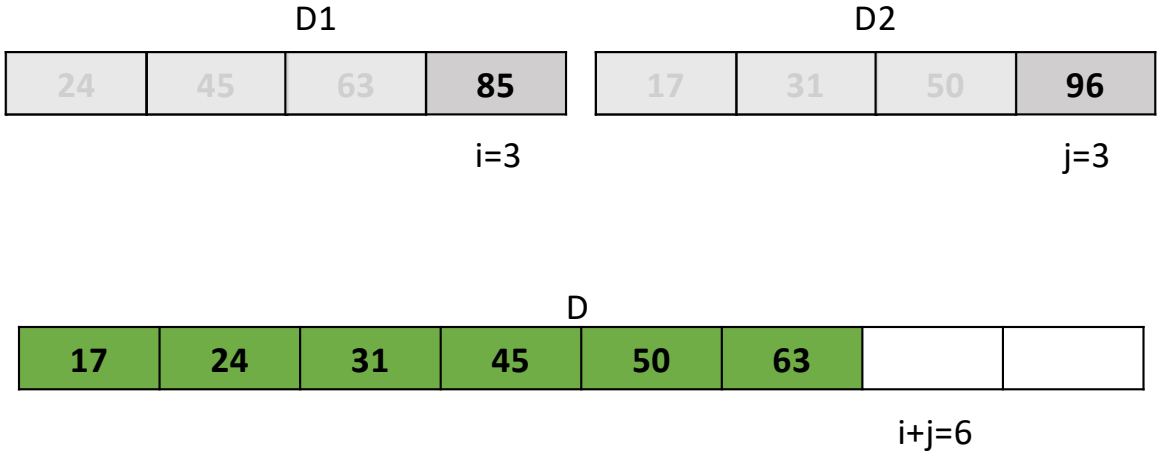



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

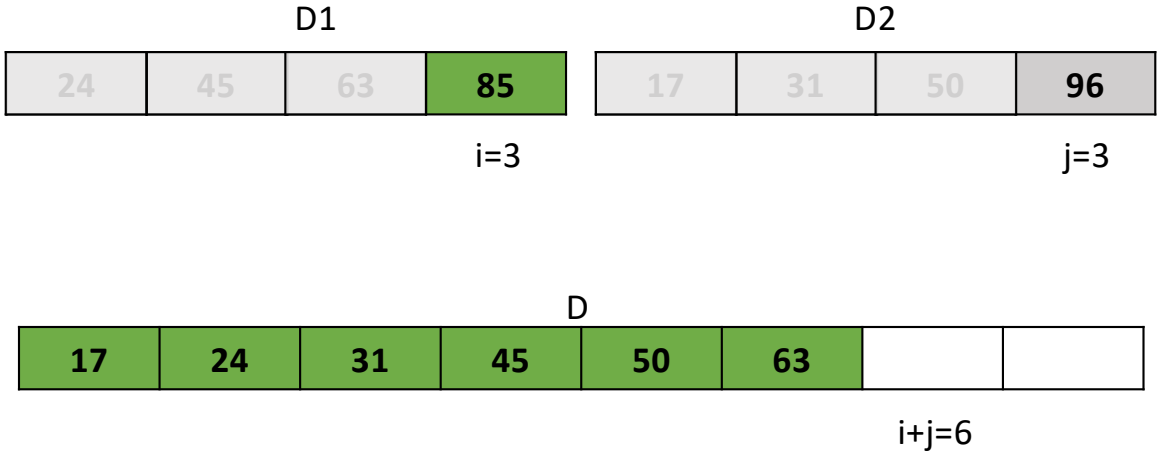



```
merge(D1, D2, D):
    i=j=0


    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

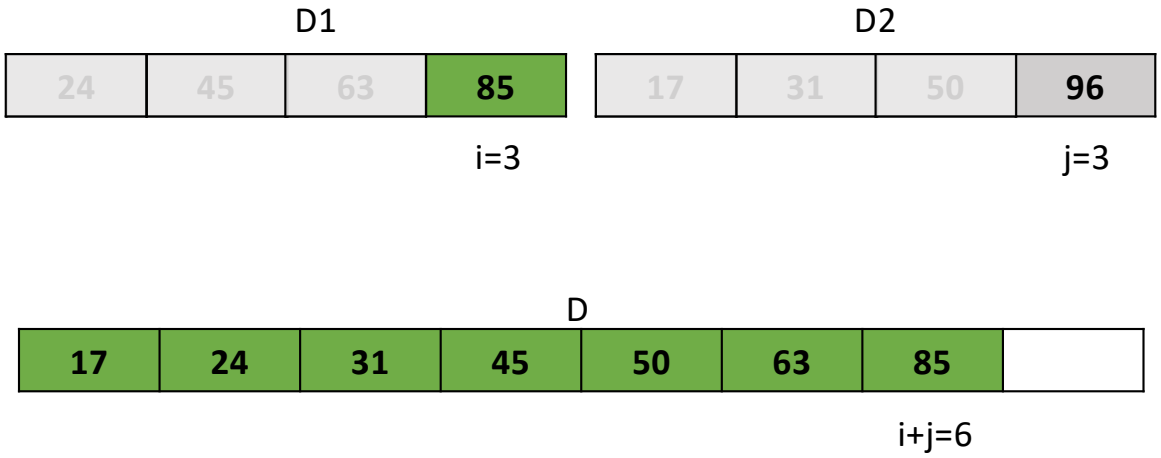



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

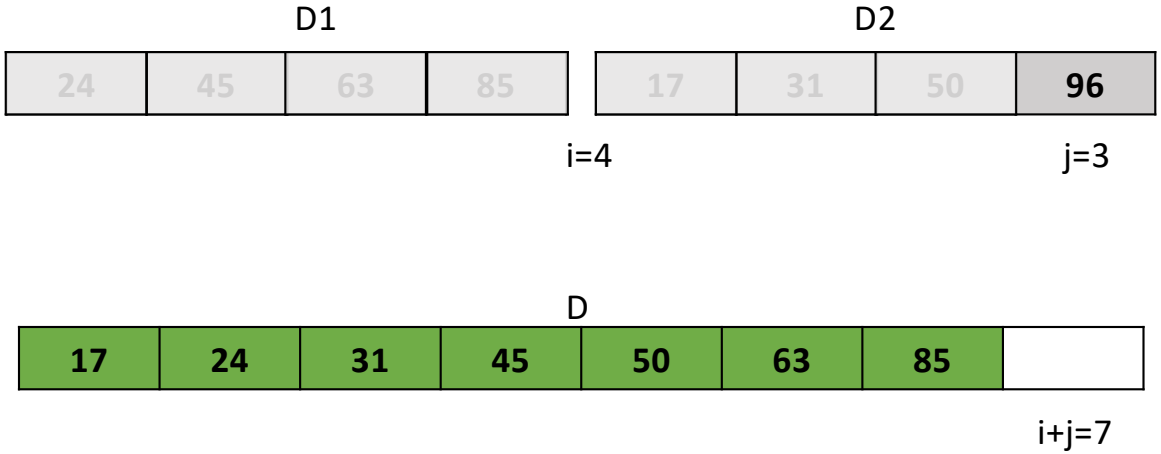



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

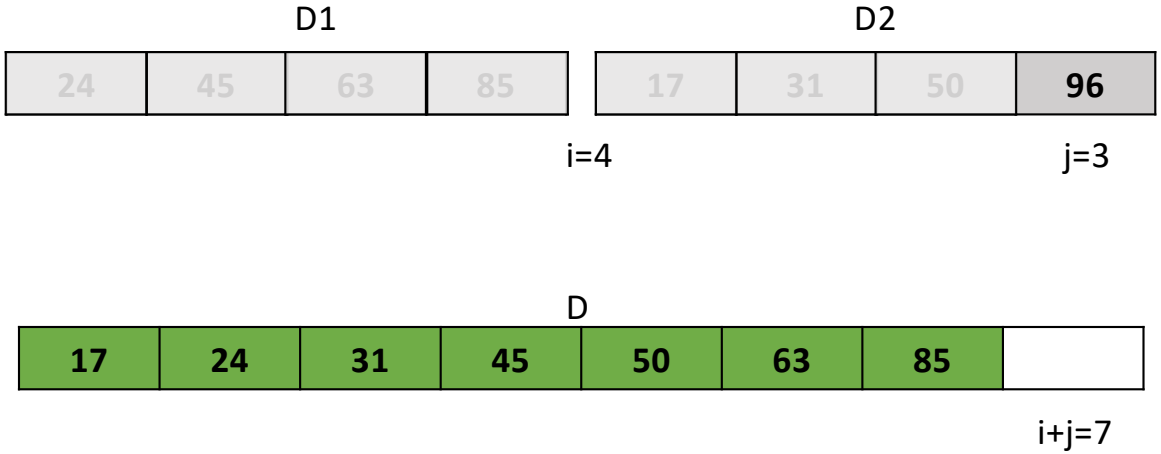



```
merge(D1, D2, D):
    i=j=0

    if D1[i] < D2[j]:
        D[i+j] = D1[i]
        i += 1
    else:
        D[i+j] = D2[j]
        j += 1
```

Merge-Sort Algorithm

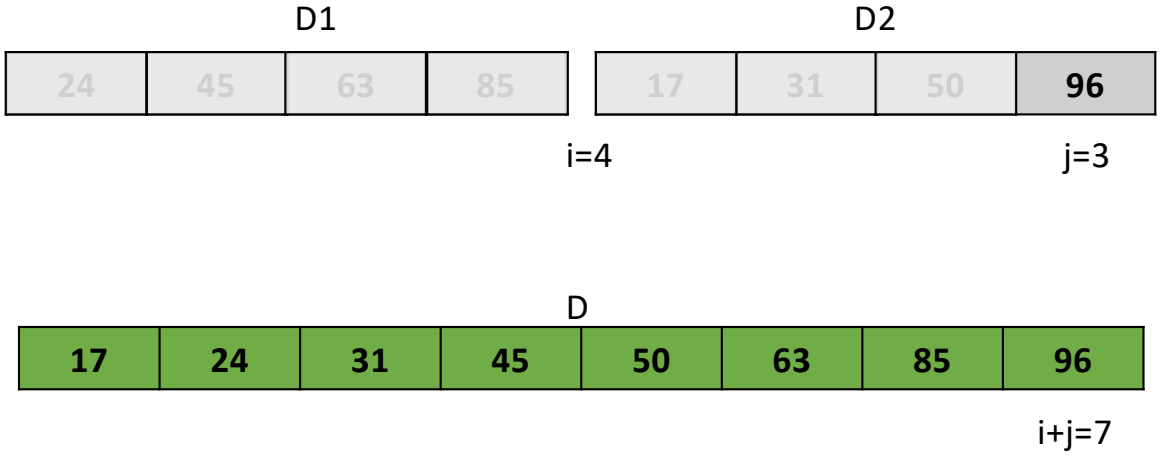
`merge()` Algorithm




```
merge(D1, D2, D):
    i=j=0
    while i < len(D1) and while j < len(D1)
        if D1[i] < D2[j]:
            D[i+j] = D1[i]
            i += 1
        else:
            D[i+j] = D2[j]
            j += 1
```

Merge-Sort Algorithm

`merge()` Algorithm

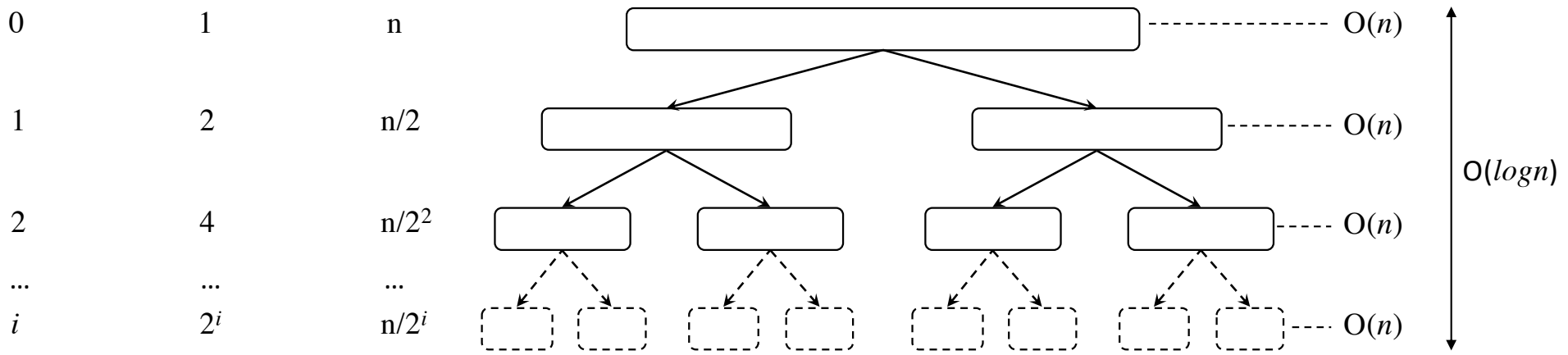



```
merge(D1, D2, D):
    i=j=0
    while i < len(D1) and while j < len(D1)
        if D1[i] < D2[j]:
            D[i+j] = D1[i]
            i += 1
        else:
            D[i+j] = D2[j]
            j += 1

    D[i+j:] = D1[i : ] + D2[j : ]
```

Merge-Sort Algorithm | Time Complexity

Depth | # nodes/sequences | size



- The amount of work done at each node is merge + partition
  - Total work done at depth i is: number of nodes x size of nodes = 2^i x $n/2^i$ → $O(n)$
- What is the stopping condition of recursion? → $O(logn)$

→ Total time complexity = **$O(nlogn)$**