Department of Computer Science and Engineering

Data Structures and Object-Oriented Design

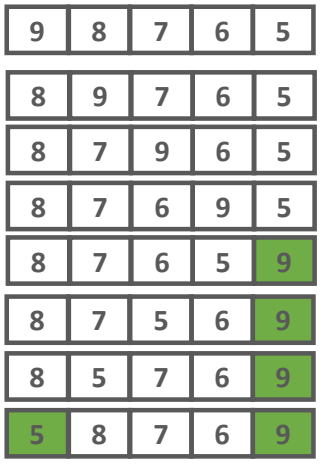
(CSE – 2050)

Hasan Baig

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

Quick Recap

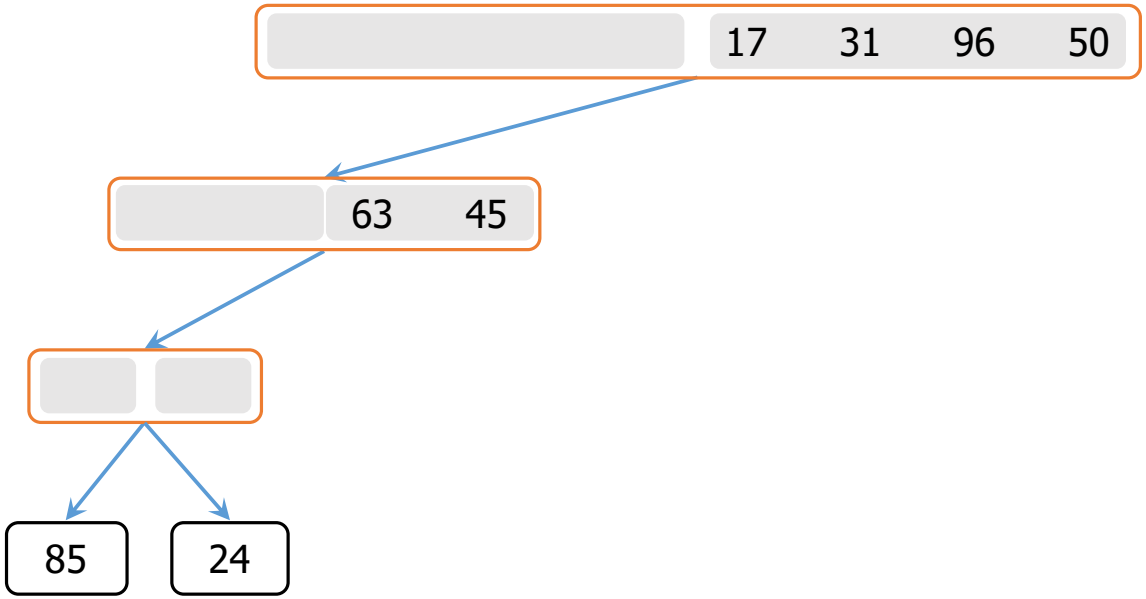
- Quadratic sorting algorithms: $O(n^2)$
  - Bubble sort (swap instantly)
  - Selection sort (keep on recording index and swap at the end of inner loop)
  - Insertion sort (check all preceding elements)
    - The swapped elements may not be at their right place until the end
  - Invariant of bubble sort – Cocktail sort
    - Sorts the largest and smallest elements at their right places during the same pass



Quick Recap

- Divide and Conquer:
  - Divide the data
  - Conquering using recursion
  - Combine

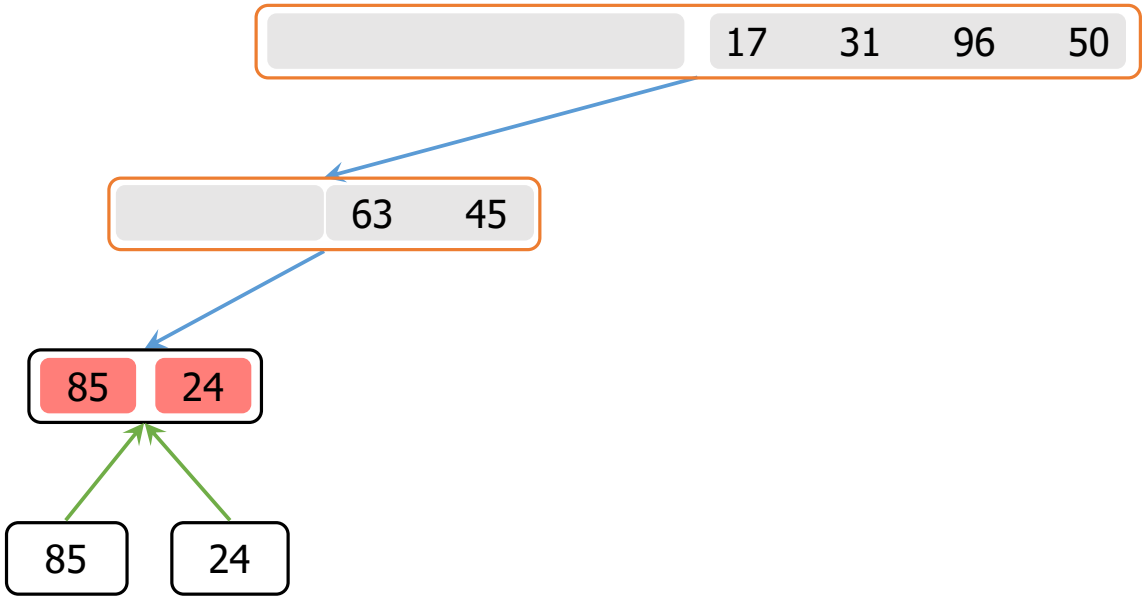
- Merge-sort algorithm



Quick Recap

- Divide and Conquer:
  - Divide the data
  - Conquering using recursion
  - Combine

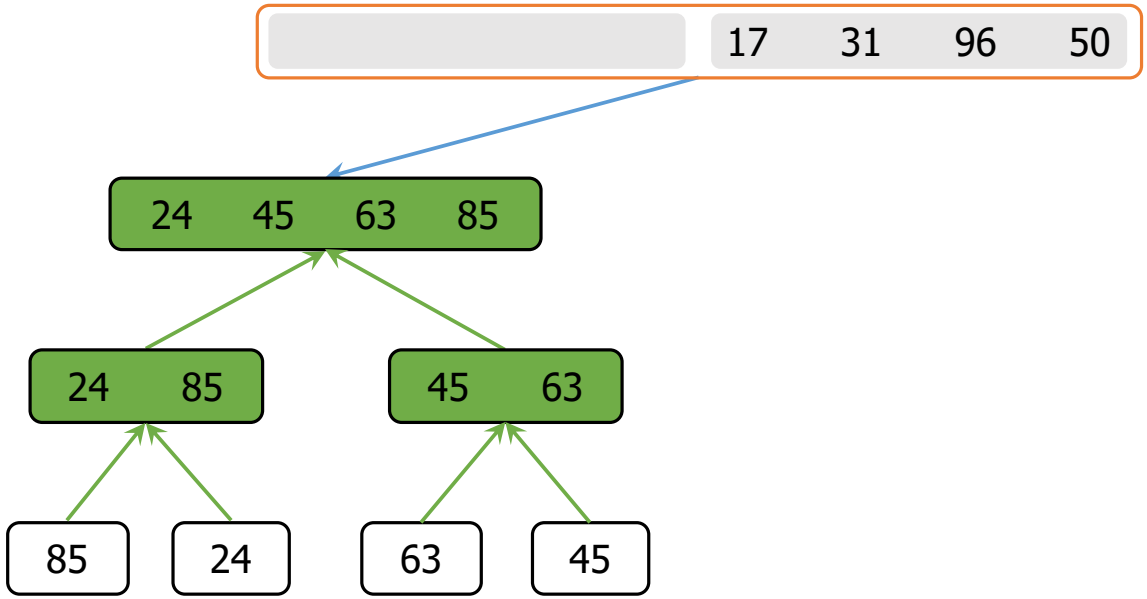
- Merge-sort algorithm



Quick Recap

- Divide and Conquer:
  - Divide the data
  - Conquering using recursion
  - Combine

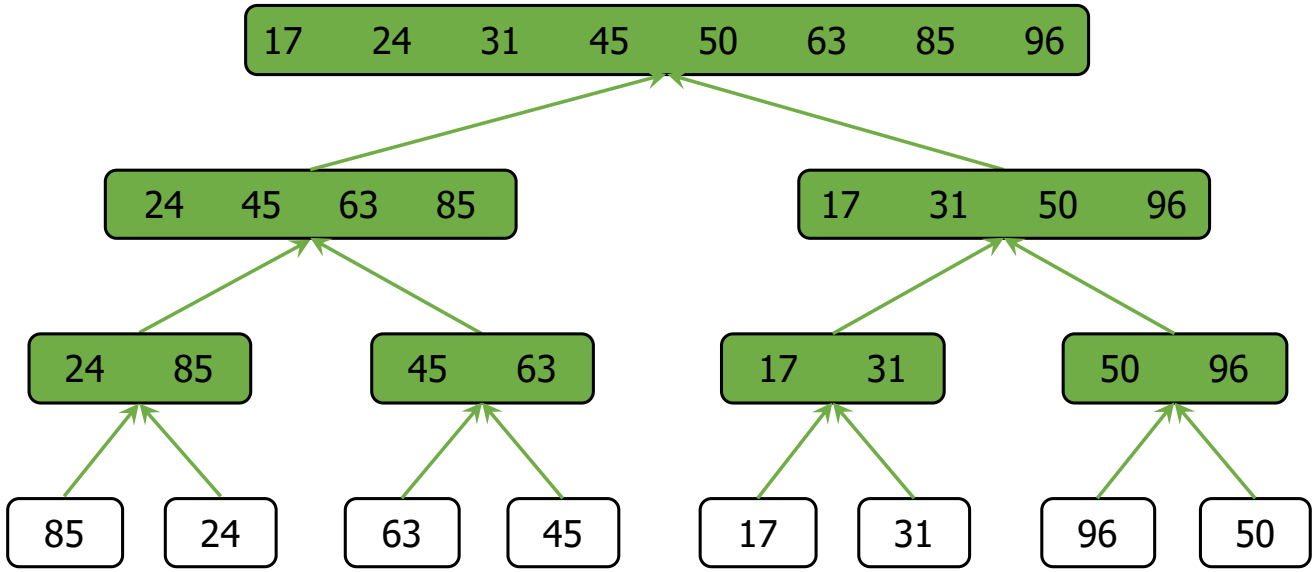
- Merge-sort algorithm



Quick Recap

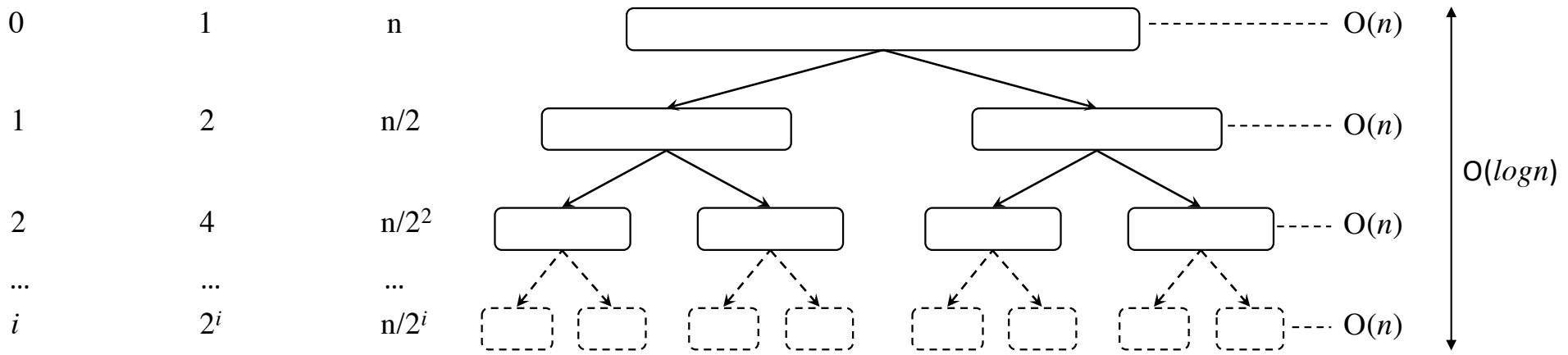
- Divide and Conquer:
  - Divide the data
  - Conquering using recursion
  - Combine

- Merge-sort algorithm



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 = **$O(nlogn)$**

Quick-Sort Algorithm

Algorithm Development

- Divide: Pick a random element x (called **pivot**) from D; then partition D into
  - L elements → less than x
  - E elements → equal to x
  - G elements → greater than x
  - *Nothing needs to be done if D has one or less element*
- Conquer: Recursively sort sequences *L* and *G*.
- Combine: Join L, E and G.

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

if D.size() > 1
 pivot ← pick x from D
 L ← elements less than x
 E ← element equal to x
 G ← elements greater than x
 L = quickSort(L)
 G = quickSort(G)
 return L + E + G
else:
 return D
```

Quick-Sort Algorithm

- Creating separate lists

85	24	63	45	17	31	96	50

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

if D.size() > 1
 pivot ← pick x from D
 L ← elements less than x
 E ← element equal to x
 G ← elements greater than x
 L = quickSort(L)
 G = quickSort(G)
 return L + E + G
else:
 return D
```

Quick-Sort Algorithm

- Creating separate lists
  - Picking last element in the list

85	24	63	45	17	31	96	**50**

Divide

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

if D.size() > 1
  pivot ← pick x from D
  L ← elements less than x
  E ← element equal to x
  G ← elements greater than x
  L = quickSort(L)
  G = quickSort(G)
  return L + E + G
else:
  return D
```

Quick-Sort Algorithm

- Creating separate lists
  - Picking last element in the list

24	45	17	31	**50**	85	63	96

Divide

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

if D.size() > 1
 pivot ← pick x from D
 L ← elements less than x
 E ← element equal to x
 G ← elements greater than x
 L = quickSort(L)
 G = quickSort(G)
 return L + E + G
else:
 return D
```

Quick-Sort Algorithm

- Creating separate lists
  - Picking last element in the list

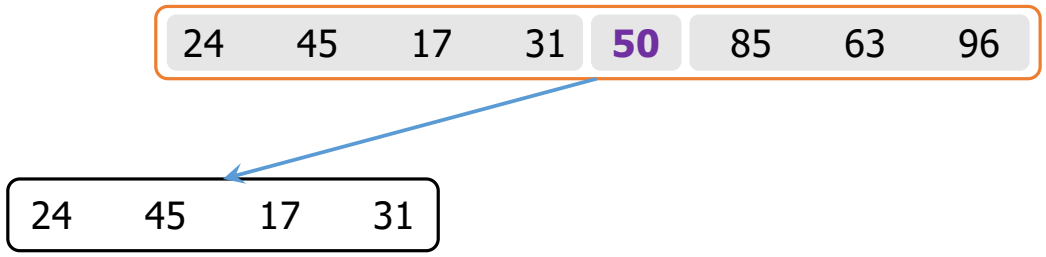
24	45	17	31	**50**	85	63	96

Divide

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

if D.size() > 1
 pivot ← pick x from D
 L ← elements less than x
 E ← element equal to x
 G ← elements greater than x
 L = quickSort(L)
 G = quickSort(G)
 return L + E + G
else:
 return D
```

Quick-Sort Algorithm

- Creating separate lists
  - Picking last element in the list

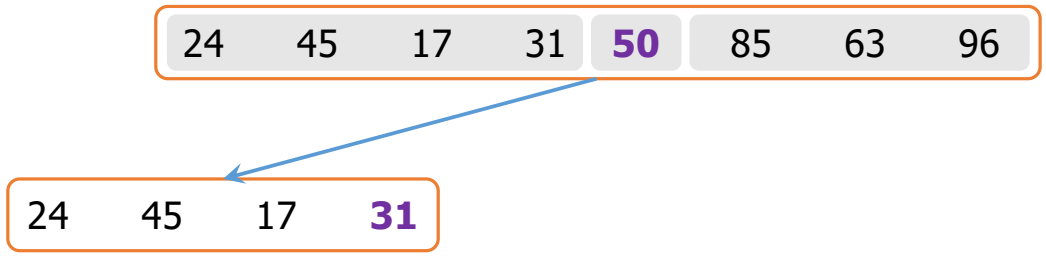


Conquer

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

if D.size() > 1
  pivot ← pick x from D
  L ← elements less than x
  E ← element equal to x
  G ← elements greater than x
  L = quickSort(L)
  G = quickSort(G)
  return L + E + G
else:
  return D
```

Quick-Sort Algorithm

- Creating separate lists
  - Picking last element in the list

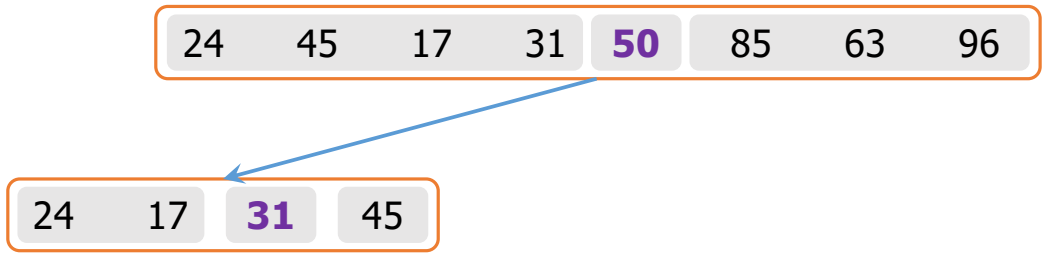


Divide

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

if D.size() > 1
  pivot ← pick x from D
  L ← elements less than x
  E ← element equal to x
  G ← elements greater than x
  L = quickSort(L)
  G = quickSort(G)
  return L + E + G
else:
  return D
```

Quick-Sort Algorithm

- Creating separate lists
  - Picking last element in the list

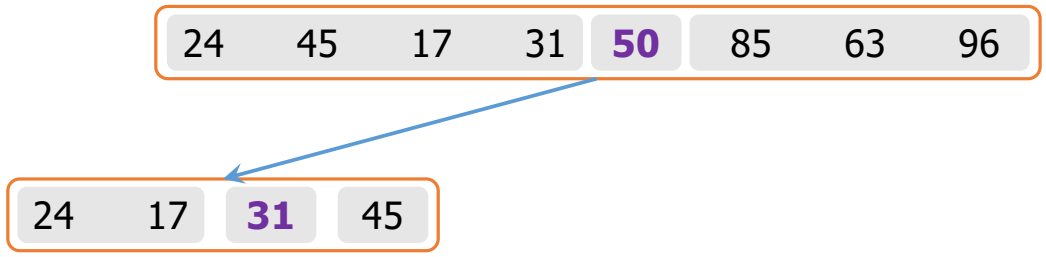


Divide

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

if D.size() > 1
  pivot ← pick x from D
  L ← elements less than x
  E ← element equal to x
  G ← elements greater than x
  L = quickSort(L)
  G = quickSort(G)
  return L + E + G
else:
  return D
```

Quick-Sort Algorithm

- Creating separate lists
  - Picking last element in the list

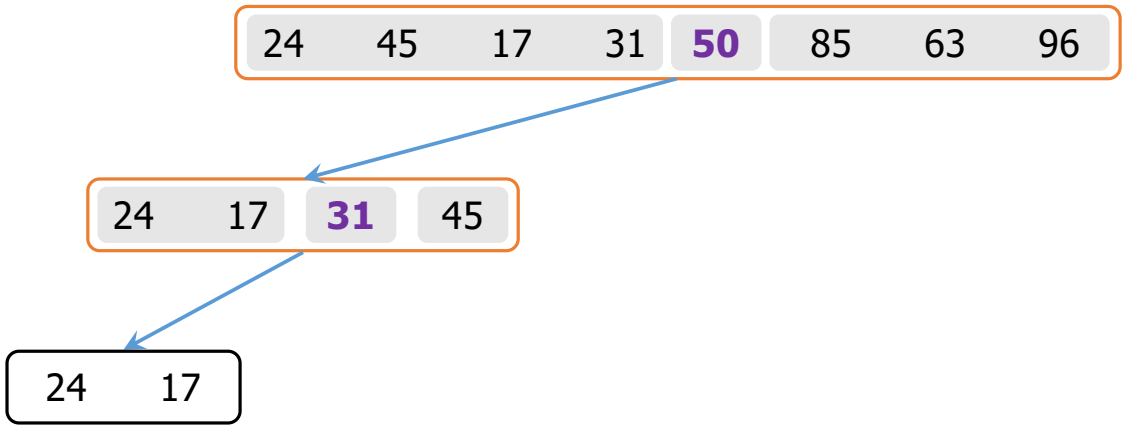


Divide

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

if D.size() > 1
  pivot ← pick x from D
  L ← elements less than x
  E ← element equal to x
  G ← elements greater than x
  L = quickSort(L)
  G = quickSort(G)
  return L + E + G
else:
  return D
```

Quick-Sort Algorithm

- Creating separate lists
  - Picking last element in the list

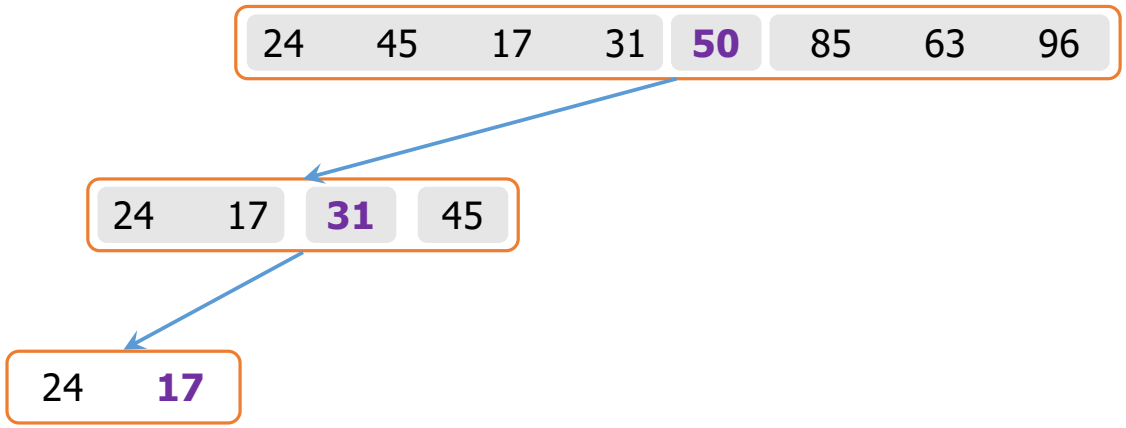


Conquer

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

  if D.size() > 1
    pivot ← pick x from D
    L ← elements less than x
    E ← element equal to x
    G ← elements greater than x
    L = quickSort(L)
    G = quickSort(G)
    return L + E + G
  else:
    return D
```

Quick-Sort Algorithm

- Creating separate lists
  - Picking last element in the list

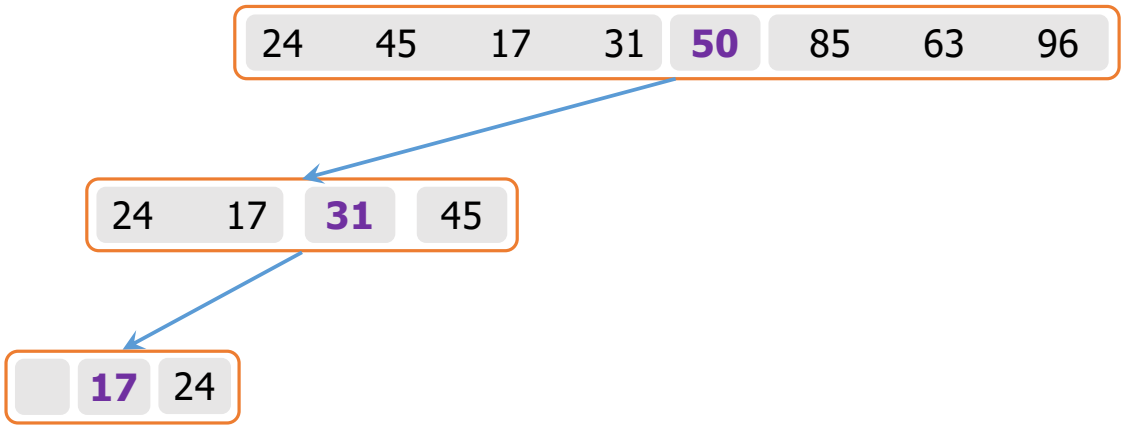


Divide

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

if D.size() > 1
  pivot ← pick x from D
  L ← elements less than x
  E ← element equal to x
  G ← elements greater than x
  L = quickSort(L)
  G = quickSort(G)
  return L + E + G
else:
  return D
```

Quick-Sort Algorithm

- Creating separate lists
  - Picking last element in the list

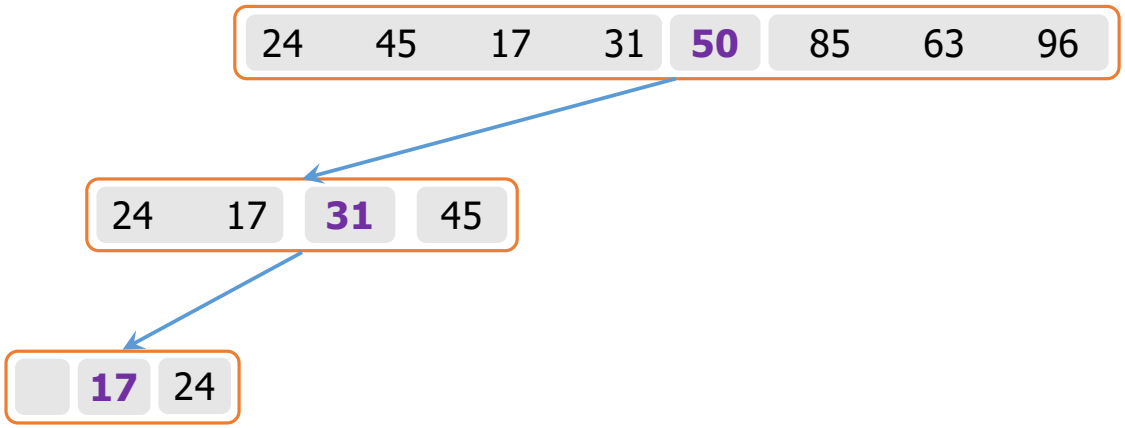


Divide

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

if D.size() > 1
  pivot ← pick x from D
  L ← elements less than x
  E ← element equal to x
  G ← elements greater than x
  L = quickSort(L)
  G = quickSort(G)
  return L + E + G
else:
  return D
```

Quick-Sort Algorithm

- Creating separate lists
  - Picking last element in the list

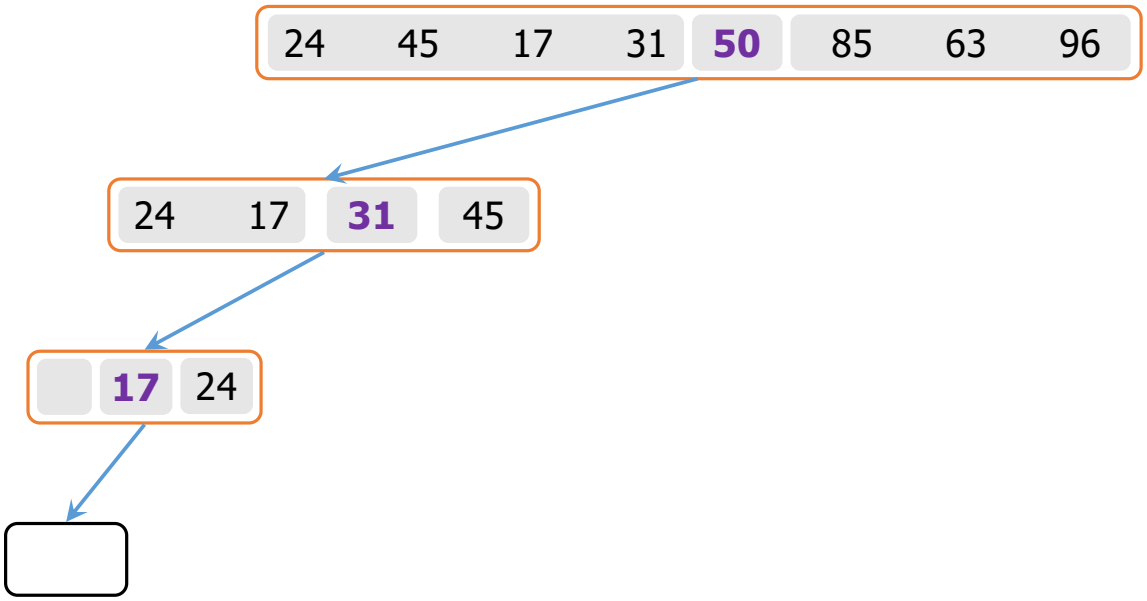


Divide

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

if D.size() > 1
 pivot ← pick x from D
 L ← elements less than x
 E ← element equal to x
 G ← elements greater than x
 L = quickSort(L)
 G = quickSort(G)
 return L + E + G
else:
 return D
```

Quick-Sort Algorithm

- Creating separate lists
  - Picking last element in the list

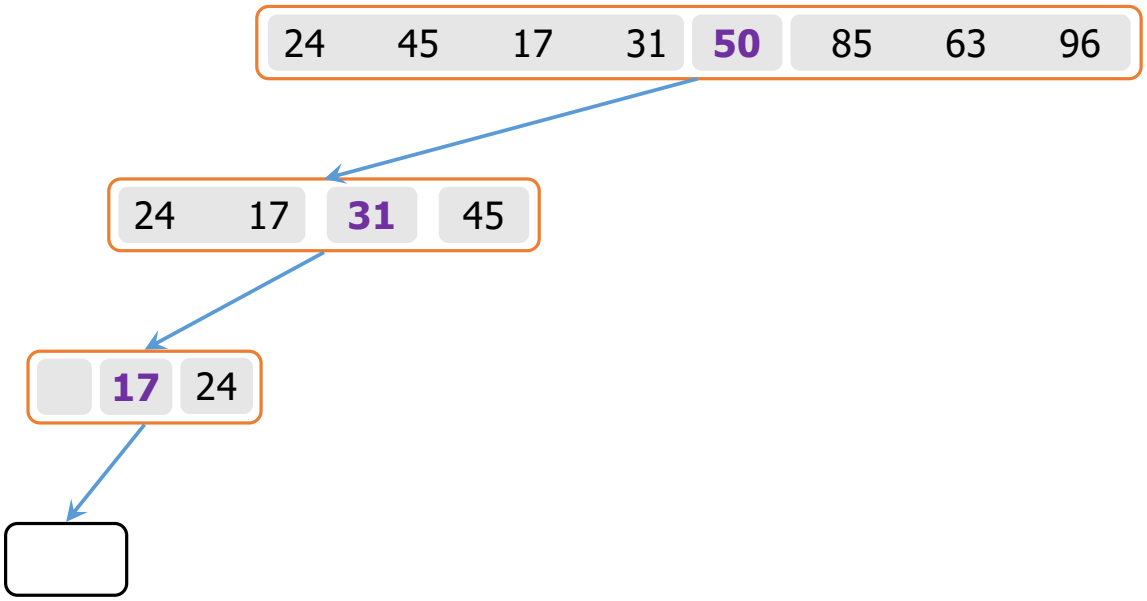


Conquer

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

  if D.size() > 1
    pivot ← pick x from D
    L ← elements less than x
    E ← element equal to x
    G ← elements greater than x
    L = quickSort(L)
    G = quickSort(G)
    return L + E + G
  else:
    return D
```

Quick-Sort Algorithm

- Creating separate lists
  - Picking last element in the list

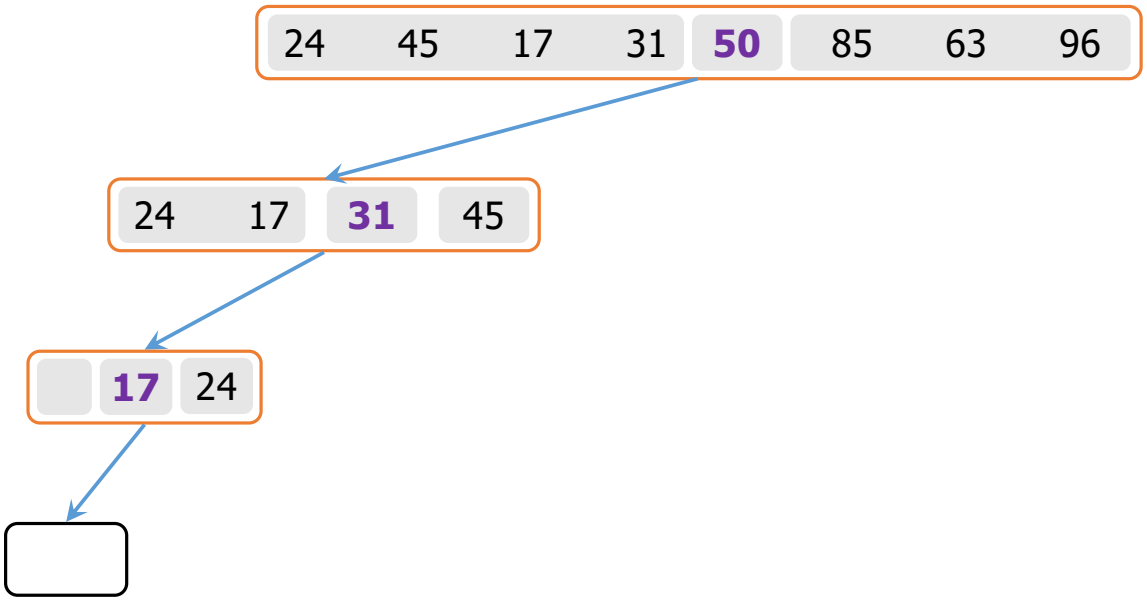


Base case

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

if D.size() > 1
  pivot ← pick x from D
  L ← elements less than x
  E ← element equal to x
  G ← elements greater than x
  L = quickSort(L)
  G = quickSort(G)
  return L + E + G
else:
  return D
```

Quick-Sort Algorithm

- Creating separate lists
  - Picking last element in the list

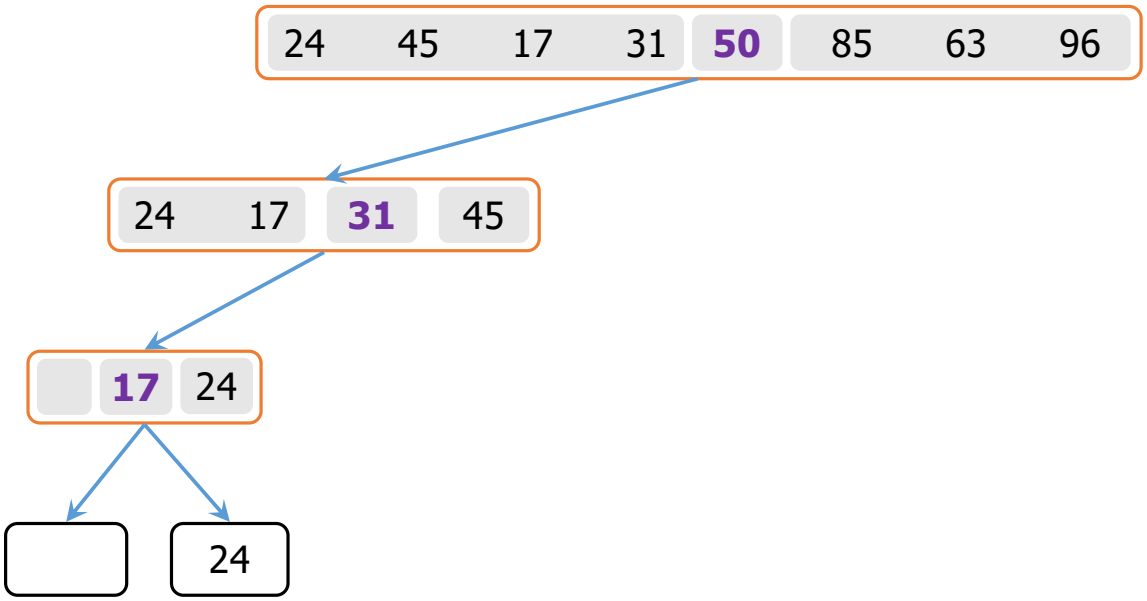


Conquer

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

if D.size() > 1
 pivot ← pick x from D
 L ← elements less than x
 E ← element equal to x
 G ← elements greater than x
 L = quickSort(L)
 G = quickSort(G)
 return L + E + G
else:
 return D
```

Quick-Sort Algorithm

- Creating separate lists
  - Picking last element in the list

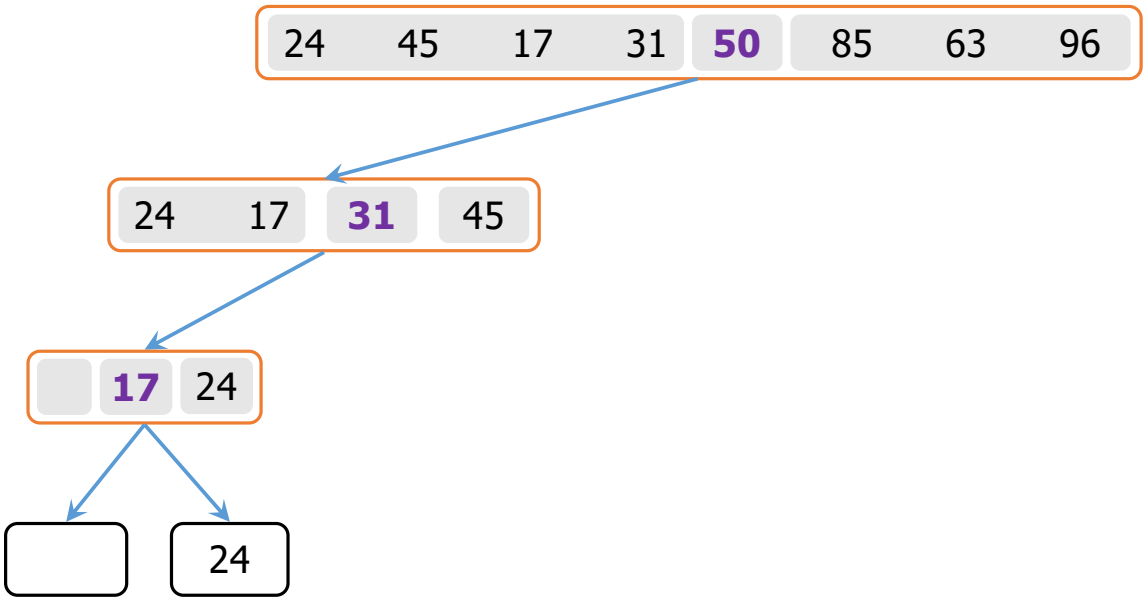


Conquer

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

if D.size() > 1
  pivot ← pick x from D
  L ← elements less than x
  E ← element equal to x
  G ← elements greater than x
  L = quickSort(L)
  G = quickSort(G)
  return L + E + G
else:
  return D
```

Quick-Sort Algorithm

- Creating separate lists
  - Picking last element in the list

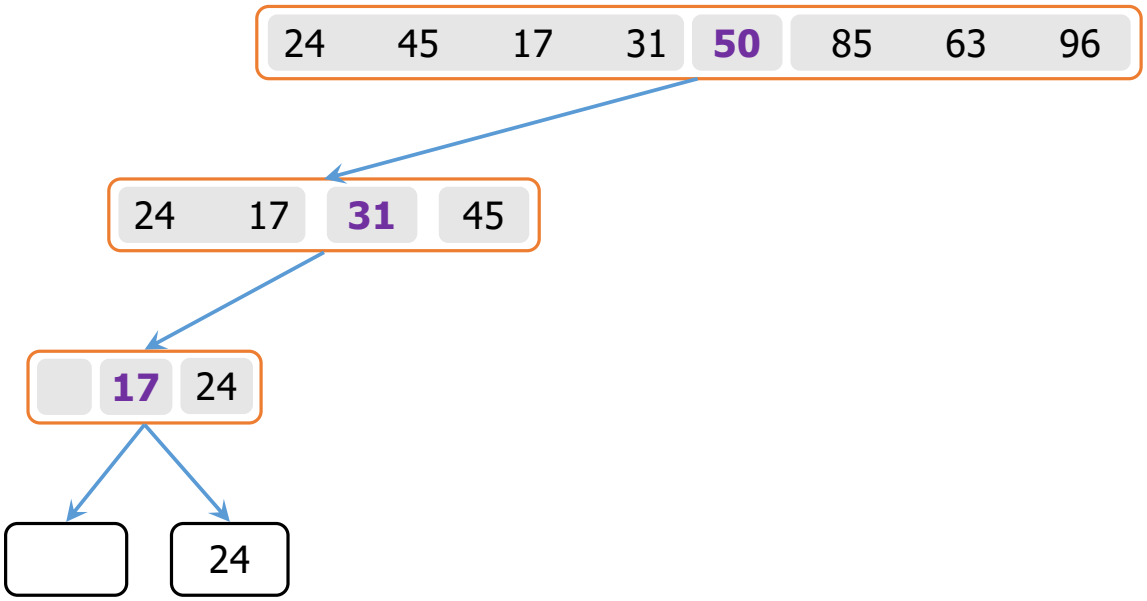


Base case

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

if D.size() > 1
  pivot ← pick x from D
  L ← elements less than x
  E ← element equal to x
  G ← elements greater than x
  L = quickSort(L)
  G = quickSort(G)
  return L + E + G
else:
  return D
```

Quick-Sort Algorithm

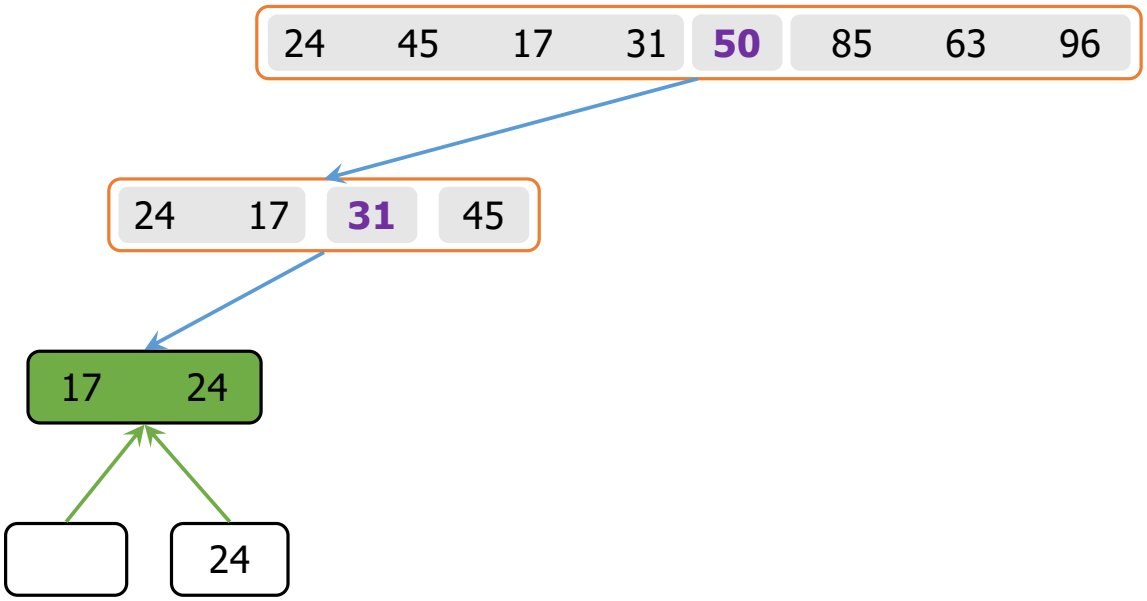
- Creating separate lists
  - Picking last element in the list




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

if D.size() > 1
 pivot ← pick x from D
 L ← elements less than x
 E ← element equal to x
 G ← elements greater than x
 L = quickSort(L)
 G = quickSort(G)
 return L + E + G
else:
 return D
```

Quick-Sort Algorithm

- Creating separate lists
  - Picking last element in the list

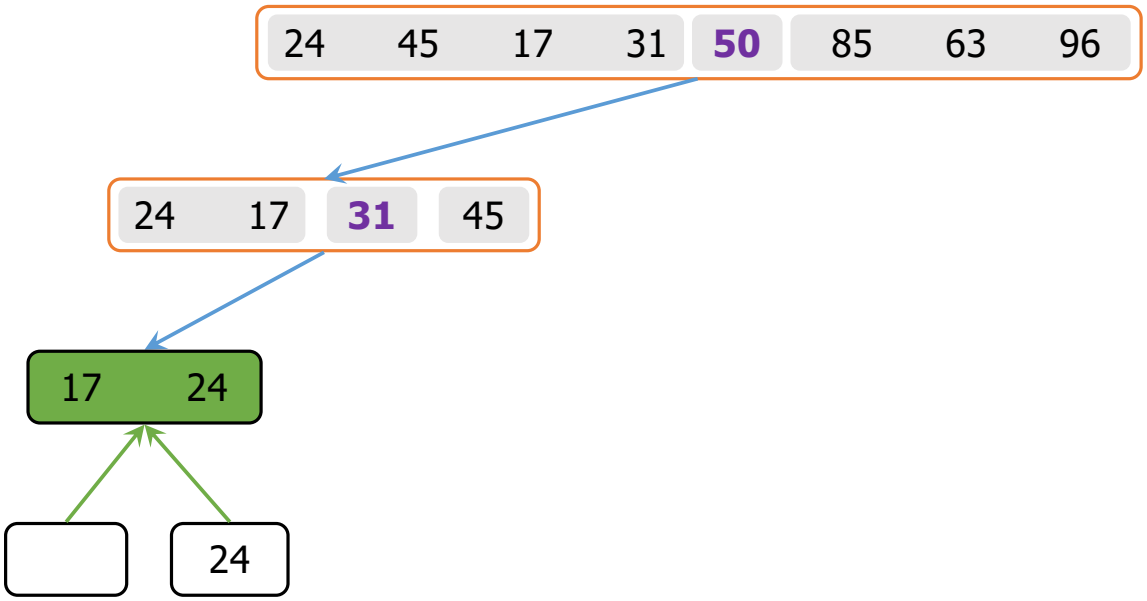


Combine

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

if D.size() > 1
 pivot ← pick x from D
 L ← elements less than x
 E ← element equal to x
 G ← elements greater than x
 L = quickSort(L)
 G = quickSort(G)
 return L + E + G
else:
 return D
```

Quick-Sort Algorithm

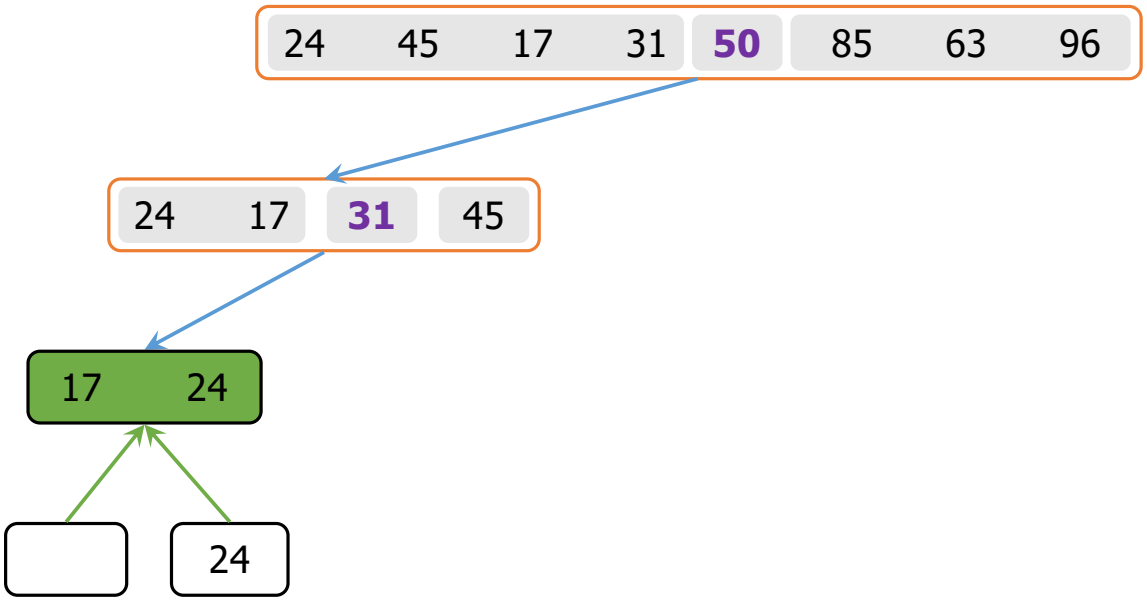
- Creating separate lists
  - Picking last element in the list




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

if D.size() > 1
 pivot ← pick x from D
 L ← elements less than x
 E ← element equal to x
 G ← elements greater than x
 L = quickSort(L)
 G = quickSort(G)
 return L + E + G
else:
 return D
```

Quick-Sort Algorithm

- Creating separate lists
  - Picking last element in the list

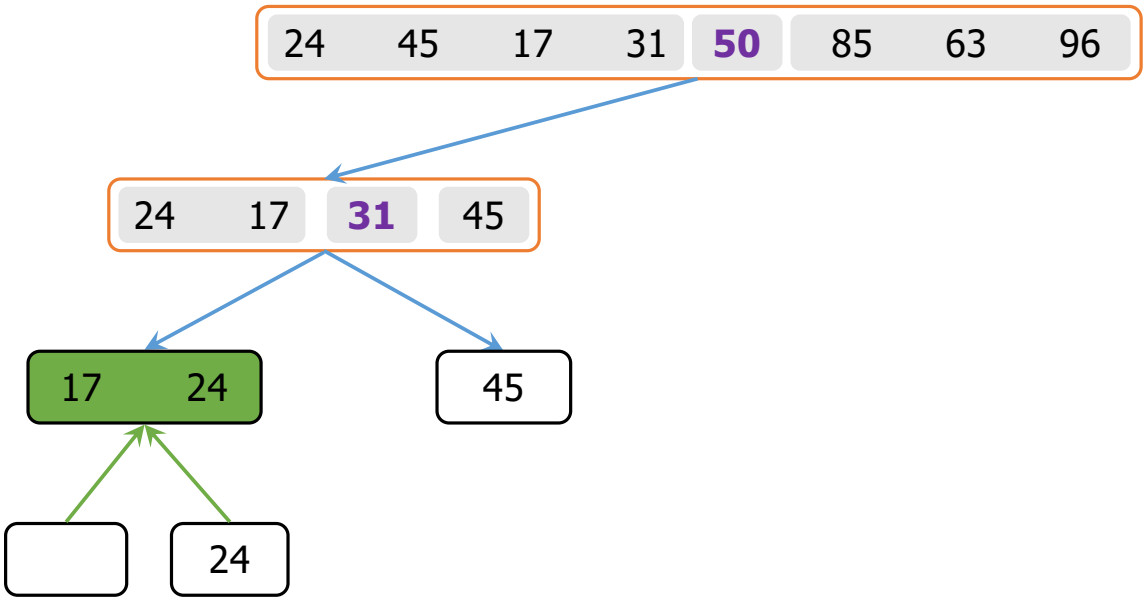


Conquer

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

if D.size() > 1
 pivot ← pick x from D
 L ← elements less than x
 E ← element equal to x
 G ← elements greater than x
 L = quickSort(L)
 G = quickSort(G)
 return L + E + G
else:
 return D
```

Quick-Sort Algorithm

- Creating separate lists
  - Picking last element in the list

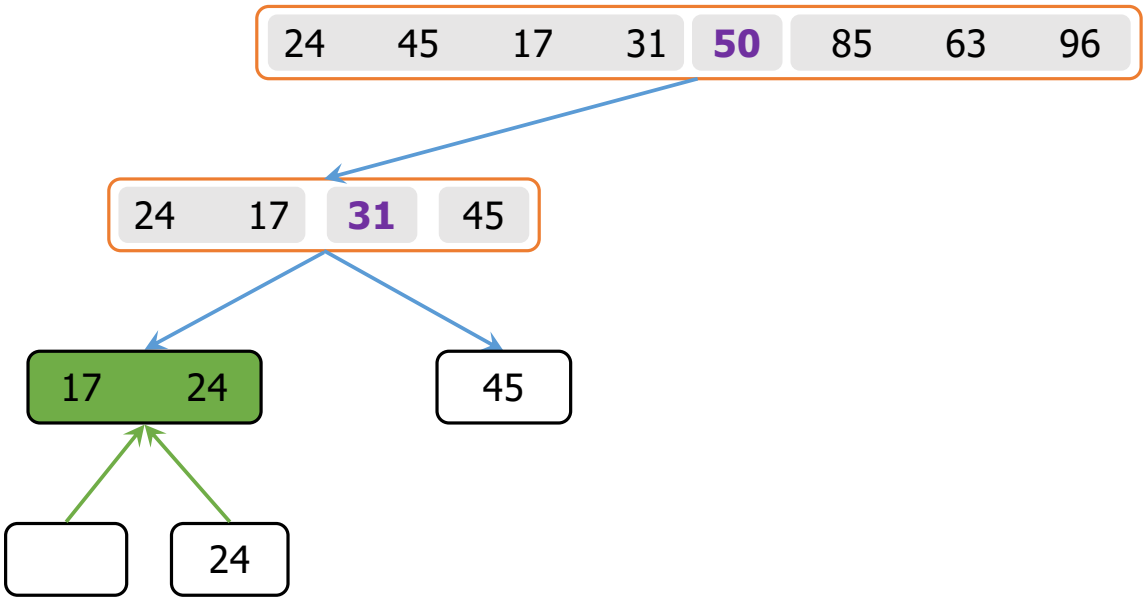


Conquer

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

if D.size() > 1
  pivot ← pick x from D
  L ← elements less than x
  E ← element equal to x
  G ← elements greater than x
  L = quickSort(L)
  G = quickSort(G)
  return L + E + G
else:
  return D
```

Quick-Sort Algorithm

- Creating separate lists
  - Picking last element in the list

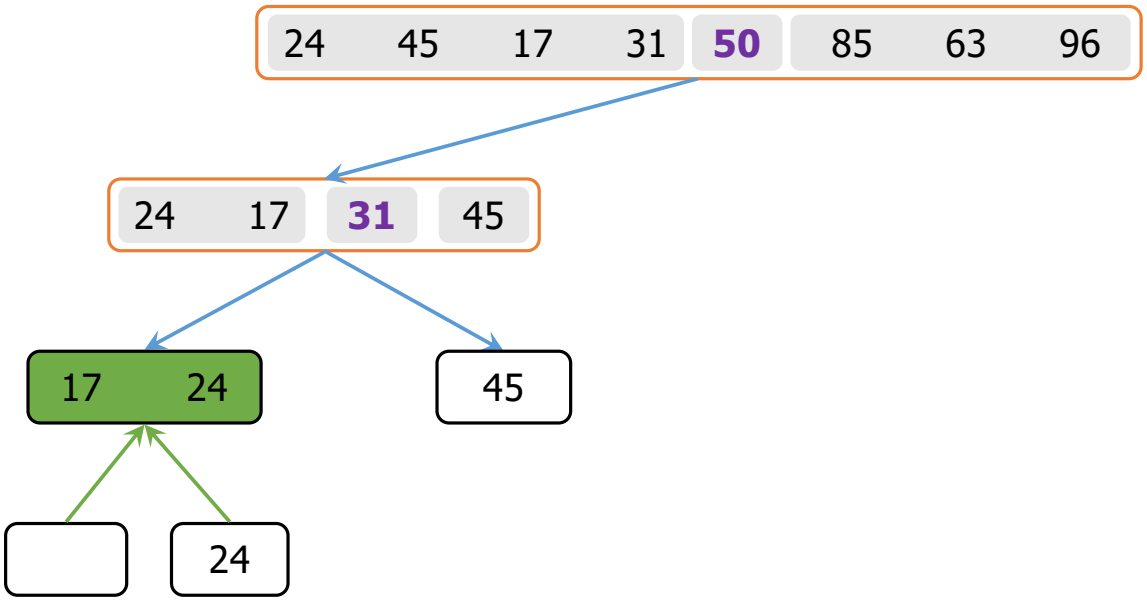


Base case

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

if D.size() > 1
  pivot ← pick x from D
  L ← elements less than x
  E ← element equal to x
  G ← elements greater than x
  L = quickSort(L)
  G = quickSort(G)
  return L + E + G
else:
  return D
```

Quick-Sort Algorithm

- Creating separate lists
  - Picking last element in the list

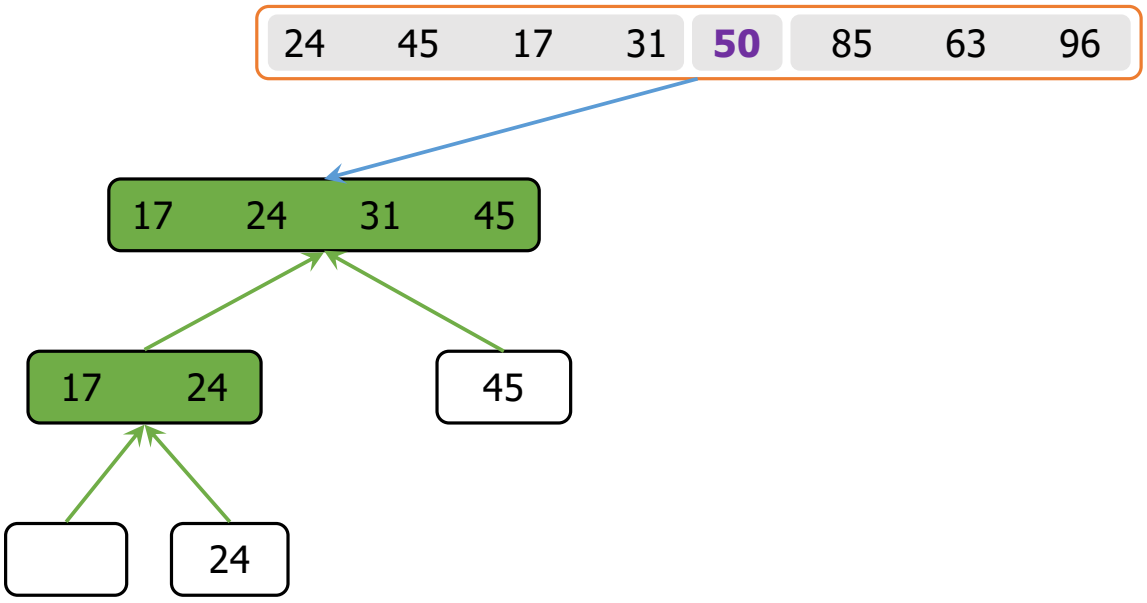


Combine

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

if D.size() > 1
 pivot ← pick x from D
 L ← elements less than x
 E ← element equal to x
 G ← elements greater than x
 L = quickSort(L)
 G = quickSort(G)
 return L + E + G
else:
 return D
```

Quick-Sort Algorithm

- Creating separate lists
  - Picking last element in the list

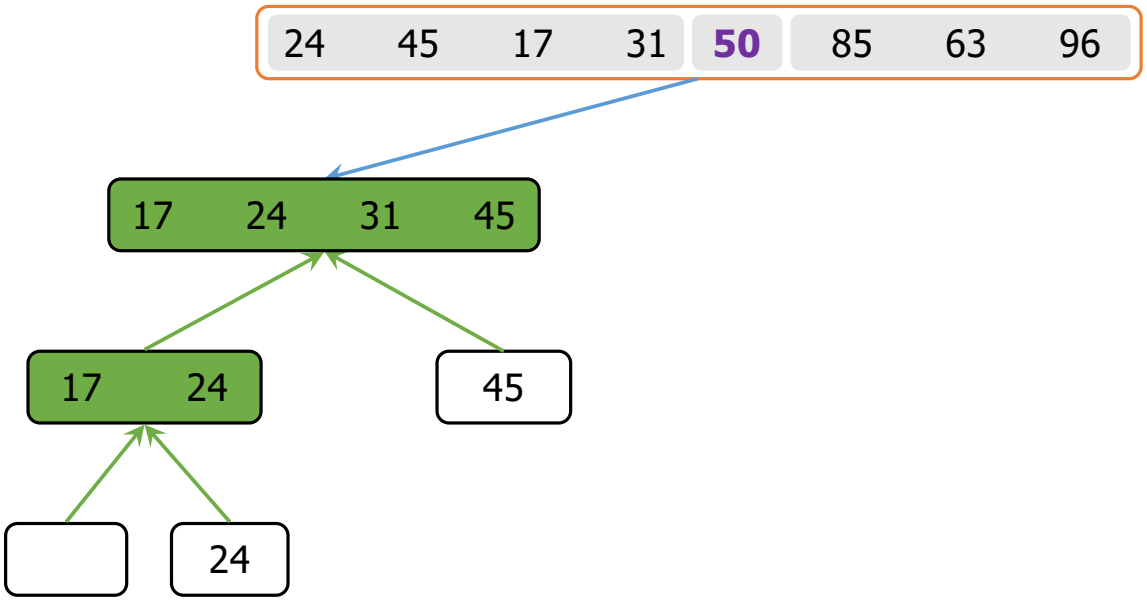


Combine

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

if D.size() > 1
 pivot ← pick x from D
 L ← elements less than x
 E ← element equal to x
 G ← elements greater than x
 L = quickSort(L)
 G = quickSort(G)
 return L + E + G
else:
 return D
```

Quick-Sort Algorithm

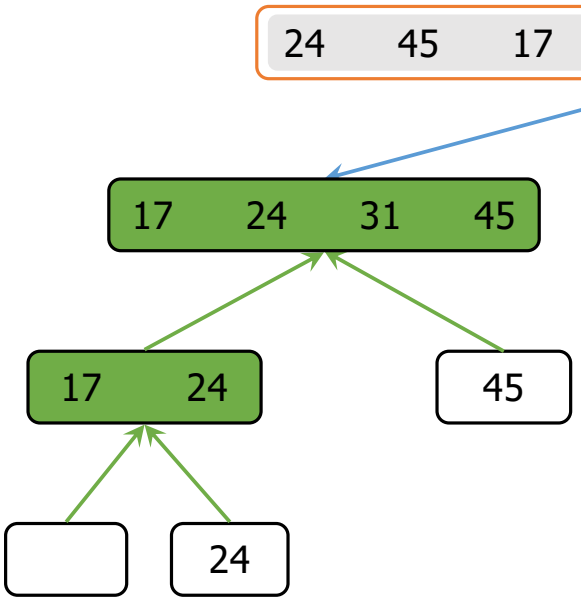
- Creating separate lists
  - Picking last element in the list




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

if D.size() > 1
 pivot ← pick x from D
 L ← elements less than x
 E ← element equal to x
 G ← elements greater than x
  L = quickSort(L)
  G = quickSort(G)
 return L + E + G
else:
 return D
```

Activity

- Draw a right-hand side tree
  - Picking last element in the list

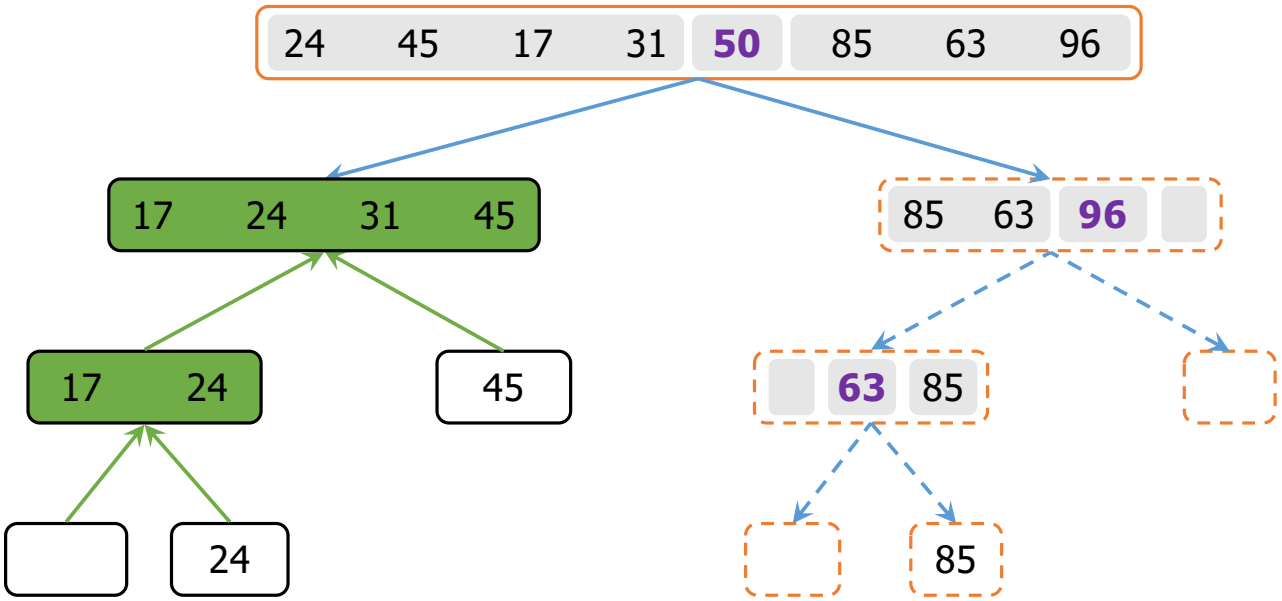


Draw a tree here

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

if D.size() > 1
  pivot ← pick x from D
  L ← elements less than x
  E ← element equal to x
  G ← elements greater than x
    L = quickSort(L)
    G = quickSort(G)
  return L + E + G
else:
  return D
```

Quick-Sort Algorithm

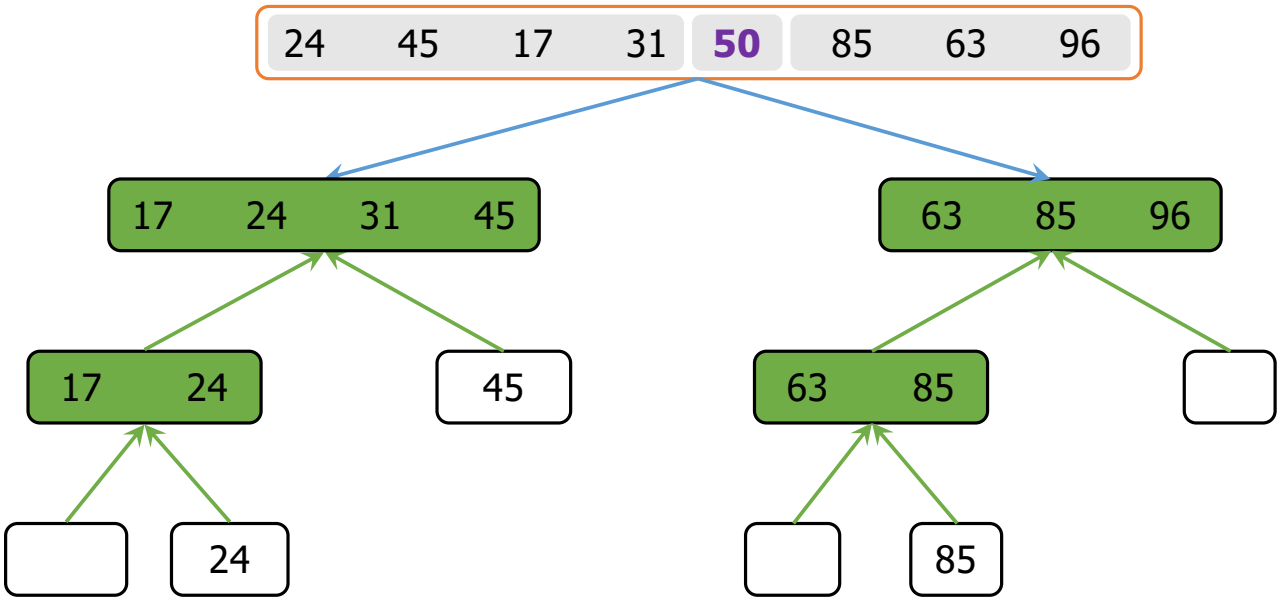
- Creating separate lists
  - Picking last element in the list




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

if D.size() > 1
 pivot ← pick x from D
 L ← elements less than x
 E ← element equal to x
 G ← elements greater than x
  L = quickSort(L)
  G = quickSort(G)
 return L + E + G
else:
 return D
```

Quick-Sort Algorithm

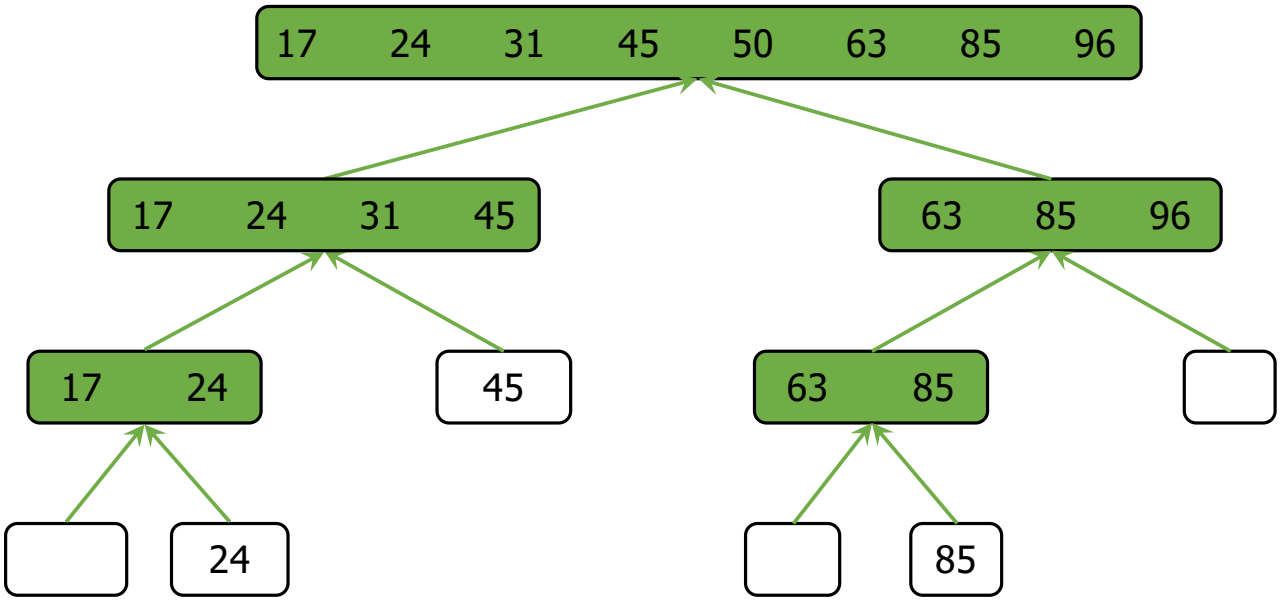
- Creating separate lists
  - Picking last element in the list




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

if D.size() > 1
 pivot ← pick x from D
 L ← elements less than x
 E ← element equal to x
 G ← elements greater than x
  L = quickSort(L)
  G = quickSort(G)
 return L + E + G
else:
 return D
```

Quick-Sort Algorithm

- Creating separate lists
  - Picking last element in the list




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

if D.size() > 1
  pivot ← pick x from D
  L ← elements less than x
  E ← element equal to x
  G ← elements greater than x
    L = quickSort(L)
    G = quickSort(G)
  return L + E + G
else:
  return D
```

Quick-Sort Algorithm

- Any element can be taken as pivot element – first, last or middle
- Pivot can also be chosen randomly using python's built-in `random` library

Quick-Sort Algorithm | In-place Implementation

- Algorithm is said to be *in-place* algo if it uses only a small amount of memory in addition to that needed for the original input data
  - It manipulates the data items within the same container to sort the data

- In previous implementation of quick-sort, we used additional container L, E, and G in each recursive call

Quick-Sort Algorithm | In-place Implementation

Algorithm Development

- Divide: Pick an element x (called **pivot**) between low and high indices of D; then rearrange D such that:
  - Values less than pivot comes before the pivot
  - Values greater than pivot comes after the pivot
  - Equal values can go either way
  - *Nothing needs to be done if D has one or less element*

- Conquer: Recursively apply above steps to rearrange item smaller and greater than pivot

- Base case → when low index becomes greater than or equal to high index

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:

        pivot = partition(D, L_Idx, H_Idx)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

Middle element as pivot

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

To compare all elements with pivot, place it at the right most place

Assume current smallest element is placed at “low” index

Placing all elements smaller than “pivot” towards left

After placing “pivot” at its right place, return pivot index

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:

        pivot = partition(D, L_Idx, H_Idx)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

23	6	4	-1	0	12	8	3	1

L_Idx=0 H_Idx=8

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:

        pivot = partition(D, L_Idx, H_Idx)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

23	6	4	-1	0	12	8	3	1

L_Idx=0 H_Idx=8

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:
        pivot = partition(D, L_Idx, H_Idx)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

23	6	4	-1	0	12	8	3	1

L_Idx=0 H_Idx=8

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:
        pivot = partition(D, 0, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

23	6	4	-1	0	12	8	3	1

L_Idx=0 H_Idx=8

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:
        pivot = partition(D, 0, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

23	6	4	-1	0	12	8	3	1

L_Idx=0 H_Idx=8

```
partition(D, 0, 8)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:
        pivot = partition(D, 0, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

23	6	4	-1	0	12	8	3	1

L_Idx=0 H_Idx=8

```
partition(D, 0, 8)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:
        pivot = partition(D, 0, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

23	6	4	-1	0	12	8	3	1

L_Idx=0 H_Idx=8

```
partition(D, 0, 8)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:
        pivot = partition(D, 0, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

23	6	4	-1	1	12	8	3	0

L_Idx=0 H_Idx=8

```
partition(D, 0, 8)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:
        pivot = partition(D, 0, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

23	6	4	-1	1	12	8	3	0

L_Idx=0 H_Idx=8

i = 0

```
partition(D, 0, 8)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:
        pivot = partition(D, 0, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

23	6	4	-1	1	12	8	3	0

L_Idx=0 H_Idx=8

j=0

i = 0

```
partition(D, 0, 8)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:
        pivot = partition(D, 0, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

```
partition(D, 0, 8)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

D

23	6	4	-1	1	12	8	3	0

L_Idx=0 H_Idx=8

j=1

i = 0

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:
        pivot = partition(D, 0, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

```
partition(D, 0, 8)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

D

23	6	4	-1	1	12	8	3	0

L_Idx=0 H_Idx=8

j=2

i = 0

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:
        pivot = partition(D, 0, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

```
partition(D, 0, 8)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

D

23	6	4	-1	1	12	8	3	0

L_Idx=0 H_Idx=8

j=3

i = 0

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:
        pivot = partition(D, 0, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

```
partition(D, 0, 8)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

D

23	6	4	-1	1	12	8	3	0

L_Idx=0 H_Idx=8

j=3

i = 0

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:
        pivot = partition(D, 0, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

23	6	4	-1	1	12	8	3	0

L_Idx=0 H_Idx=8

j=3

i = 0

```
partition(D, 0, 8)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:
        pivot = partition(D, 0, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

-1	6	4	23	1	12	8	3	0

L_Idx=0 H_Idx=8

j=3

i = 0

```
partition(D, 0, 8)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:
        pivot = partition(D, 0, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

-1	6	4	23	1	12	8	3	0

L_Idx=0 H_Idx=8

j=3

i = 1

```
partition(D, 0, 8)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:
        pivot = partition(D, 0, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

```
partition(D, 0, 8)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

D

-1	6	4	23	1	12	8	3	0

L_Idx=0 H_Idx=8

j=4

i = 1

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:
        pivot = partition(D, 0, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

```
partition(D, 0, 8)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

D

-1	6	4	23	1	12	8	3	0

L_Idx=0 H_Idx=8

j=5

i = 1

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:
        pivot = partition(D, 0, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

```
partition(D, 0, 8)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

D

-1	6	4	23	1	12	8	3	0

L_Idx=0 H_Idx=8

j=6

i = 1

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:
        pivot = partition(D, 0, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

```
partition(D, 0, 8)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

D

-1	6	4	23	1	12	8	3	0

L_Idx=0 H_Idx=8

j=7

i = 1

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:
        pivot = partition(D, 0, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

```
partition(D, 0, 8)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

D

-1	6	4	23	1	12	8	3	0

L_Idx=0 H_Idx=8

j=8

i = 1

Finally, comparing with itself
→ To put the pivot item at right place

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:
        pivot = partition(D, 0, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

-1	6	4	23	1	12	8	3	0

L_Idx=0

H_Idx=8

j=8

i = 1

```
partition(D, 0, 8)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:
        pivot = partition(D, 0, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

```
partition(D, 0, 8)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

D

-1	0	4	23	1	12	8	3	6

L_Idx=0

H_Idx=8

j=8

i = 1

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:
        pivot = partition(D, 0, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

-1	0	4	23	1	12	8	3	6

L_Idx=0

H_Idx=8

j=8

i = 2

```
partition(D, 0, 8)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:
        pivot = partition(D, 0, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

-1	0	4	23	1	12	8	3	6

L_Idx=0 H_Idx=8

i = 2

```
partition(D, 0, 8)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

pivot = 1

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:

        pivot = partition(D, 0, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

-1	0	4	23	1	12	8	3	6

L_Idx=0 H_Idx=8

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

pivot = 1

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:

        pivot = partition(D, 0, 8)
        quickSort(D, 0, 1-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

-1	0	4	23	1	12	8	3	6

L_Idx=0 H_Idx=8

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

pivot = 1

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 0, 0)
    if L_Idx < H_Idx:

        pivot = partition(D, L_Idx, H_Idx)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

-1	0	4	23	1	12	8	3	6

L_Idx=0 H_Idx=8

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 0, 0)
    if L_Idx < H_Idx:
        pivot = partition(D, L_Idx, H_Idx)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

-1	0	4	23	1	12	8	3	6

L_Idx=0 H_Idx=8

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:

        pivot = partition(D, 0, 8)
        quickSort(D, 0, 1-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

-1	0	4	23	1	12	8	3	6

L_Idx=0 H_Idx=8

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

pivot = 1

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, L_Idx, H_Idx)
    if L_Idx < H_Idx:

        pivot = partition(D, 0, 8)
        quickSort(D, 0, 1-1)
        quickSort(D, 1+1, 8)
end
```

D

-1	0	4	23	1	12	8	3	6

L_Idx=2 H_Idx=8

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

pivot = 1

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 2, 8)
    if L_Idx < H_Idx:
        pivot = partition(D, L_Idx, H_Idx)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

-1	0	4	23	1	12	8	3	6

L_Idx=2 H_Idx=8

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Group Activity

Instructions:

1. Spend 2 minutes individually to think and reflect about the activities
2. Turn your chairs around and split into 6 groups (A, B, C, D, E, F) having of 6~7 students in each
3. Discuss and try to come up with a solution within 10 ~ 15 mins
4. I will spin the wheel to randomly choose the first group to answer questions and earn bonus points.
    1. After you mutually agreed on the solution, only 1 member from the selected team will be responding to online quiz
    2. If the group give 1 correct answer, I will spin the wheel one time to select the next group. For 2 correct answer, I will spin the wheel twice to select 2 groups, and so on.
    3. For all correct answers, all other groups will be allowed to earn bonus points by answering the questions.
    4. Entire group will be awarded **0.4% bonus** that can be adjusted in final grade

Group Activity

1. Fill the contents of D according to the `partition` algorithm shown below.

```
quickSort(D, 2, 8)
    if L_Idx < H_Idx:
        pivot = partition(D, 2, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

D

-1	0	4	23	1	12	8	3	6

L_Idx=2 H_Idx=8

At the end of j^{th} iteration, contents of ***D*** and ***i*** are

j = 2	-1	0								i = 3
j = __	-1	0								i = __
j = __	-1	0								i = __
j = __	-1	0								i = __
j = __	-1	0								i = __
j = __	-1	0								i = __
j = __	-1	0								i = __
j = __	-1	0								i = __
j = __	-1	0								i = __

Group Activity

2. You are required to sort the following data with quicksort algorithm. Assume that the pivot item chosen during each successive recursive calls are 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10 respectively. (a) Draw its tree structure (b) Determine the time complexity

10	6	3	5	9	2	4	8	7	1

Group Activity

Fill the contents of D according to the `partition` algorithm shown below.

```
quickSort(D, 2, 8)
    if L_Idx < H_Idx:
        pivot = partition(D, 2, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

D

-1	0	4	23	1	12	8	3	6

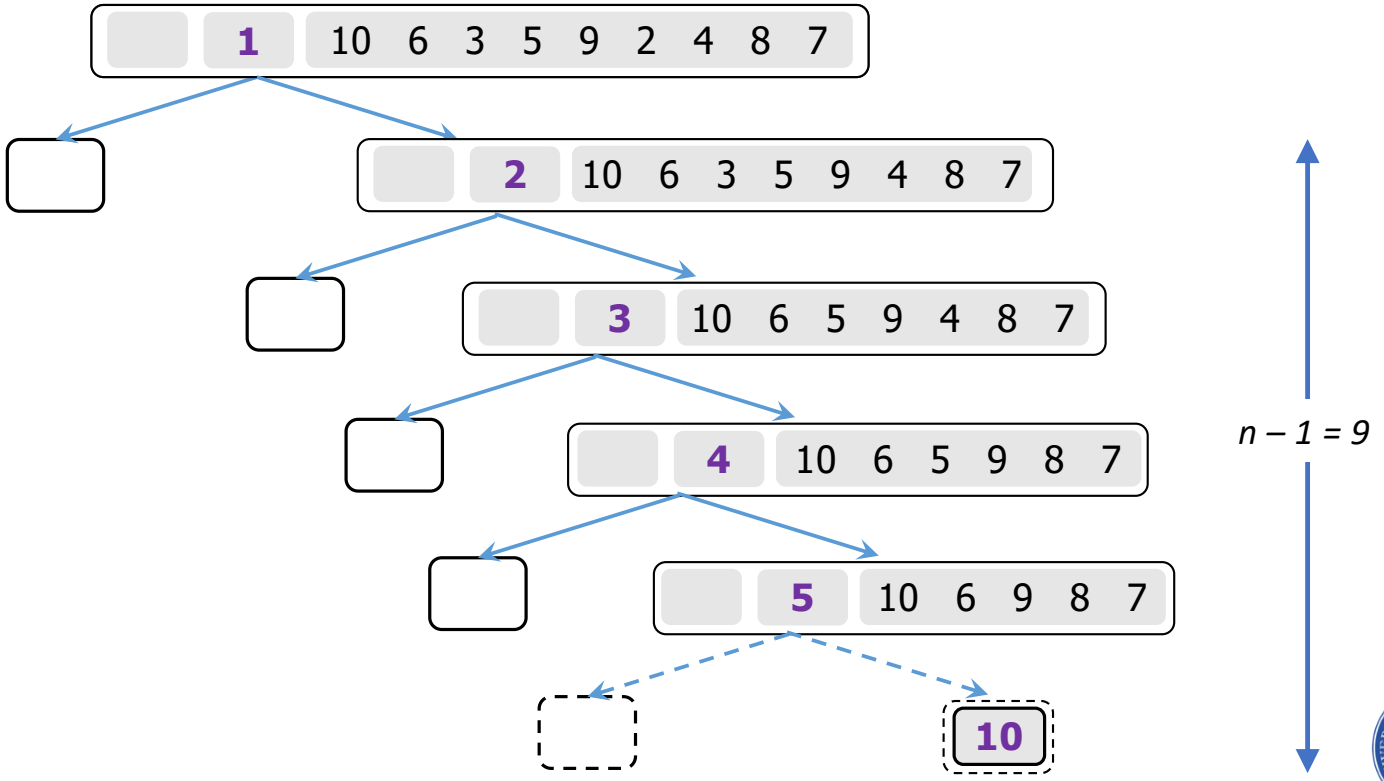
L_Idx=2 H_Idx=8

At the end of j^{th} iteration, contents of ***D*** and ***i*** are

	-1	0								
j = 2	-1	0	4	23	1	6	8	3	12	i = 3
j = 3	-1	0	4	23	1	6	8	3	12	i = 3
j = 4	-1	0	4	1	23	6	8	3	12	i = 4
j = 5	-1	0	4	1	6	23	8	3	12	i = 5
j = 6	-1	0	4	1	6	8	23	3	12	i = 6
j = 7	-1	0	4	1	6	8	3	23	12	i = 7
j = 8	-1	0	4	1	6	8	3	12	23	i = 8
j = __	-1	0								i = __
j = __	-1	0								i = __

Group Activity

2. You are required to sort the following data with quicksort algorithm. Assume that the pivot item chosen during each successive recursive calls are 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10 respectively. (a) Draw its tree structure (b) Indicate the time complexity



Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 2, 8)
    if L_Idx < H_Idx:

        pivot = partition(D, 2, 8)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

-1	0	4	1	6	8	3	12	23

L_Idx=2 H_Idx=8

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

pivot = 7

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 2, 8)
    if L_Idx < H_Idx:

        pivot = partition(D, 2, 8)
        quickSort(D, 2, 7-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

-1	0	4	1	6	8	3	12	23

L_Idx=2 H_Idx=8

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

pivot = 7

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 2, 6)
    if L_Idx < H_Idx:

        pivot = partition(D, L_Idx, H_Idx)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
end
```

D

-1	0	4	1	6	8	3	12	23

L_Idx=2 H_Idx=6

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 2, 6)
    if L_Idx < H_Idx:
        pivot = partition(D, 2, 6)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

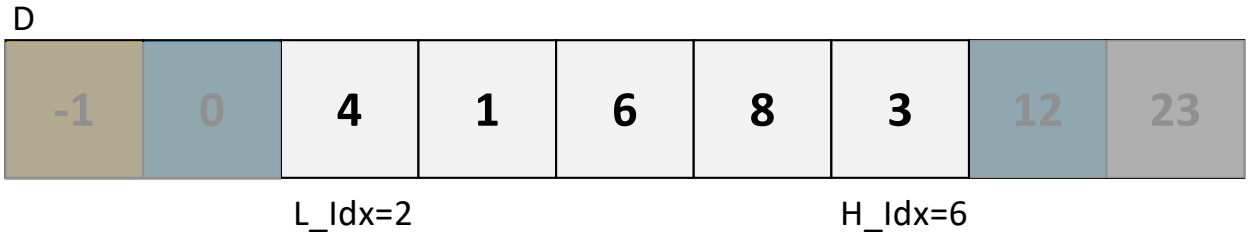
-1	0	4	1	6	8	3	12	23

L_Idx=2 H_Idx=6

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 2, 6)
    if L_Idx < H_Idx:
        pivot = partition(D, 2, 6)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

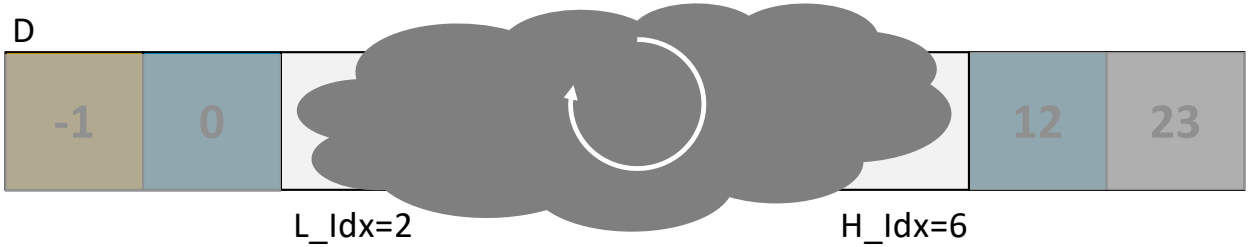


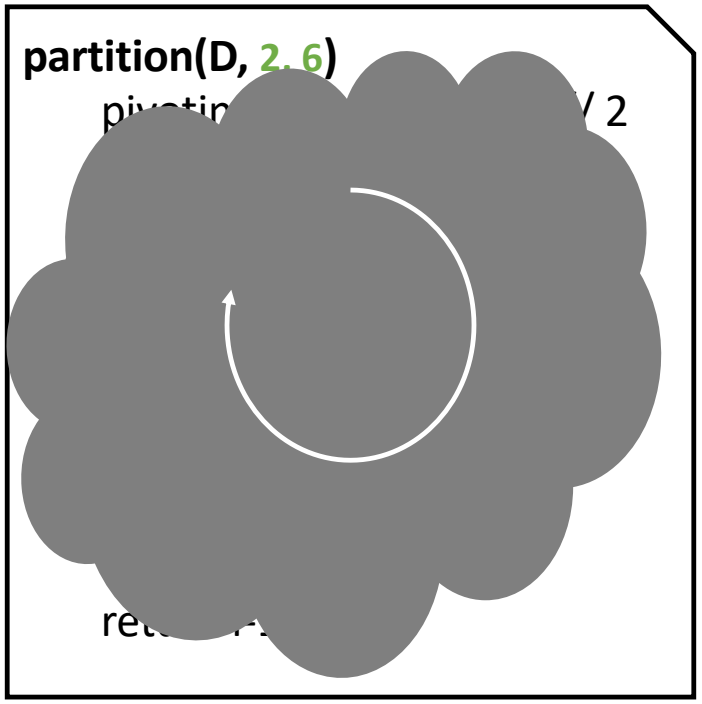

```
partition(D, 2, 6)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm

In-place Implementation

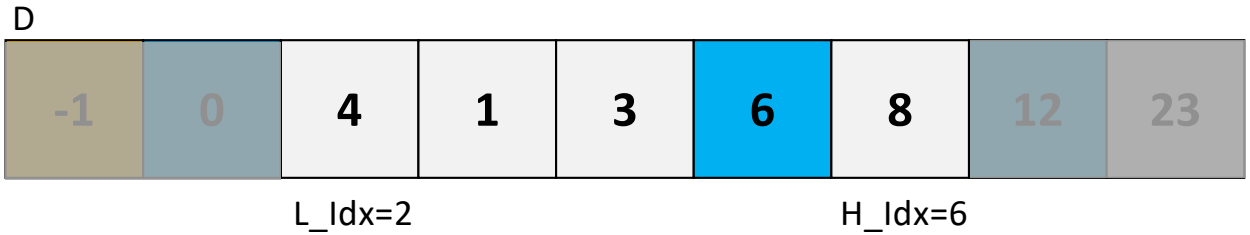
```
quickSort(D, 2, 6)
    if L_Idx < H_Idx:
        pivot = partition(D, 2, 6)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```





Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 2, 6)
    if L_Idx < H_Idx:
        pivot = partition(D, 2, 6)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```




```
partition(D, 2, 6)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

pivot = 5

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 2, 6)
    if L_Idx < H_Idx:

        pivot = partition(D, 2, 6)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

-1	0	4	1	3	6	8	12	23

L_Idx=2 H_Idx=6

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

pivot = 5

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 2, 6)
    if L_Idx < H_Idx:

        pivot = partition(D, 2, 6)
        quickSort(D, 2, 5-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

-1	0	4	1	3	6	8	12	23

L_Idx=2 H_Idx=6

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

pivot = 5

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 2, 4)
    if L_Idx < H_Idx:

        pivot = partition(D, L_Idx, H _Idx)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

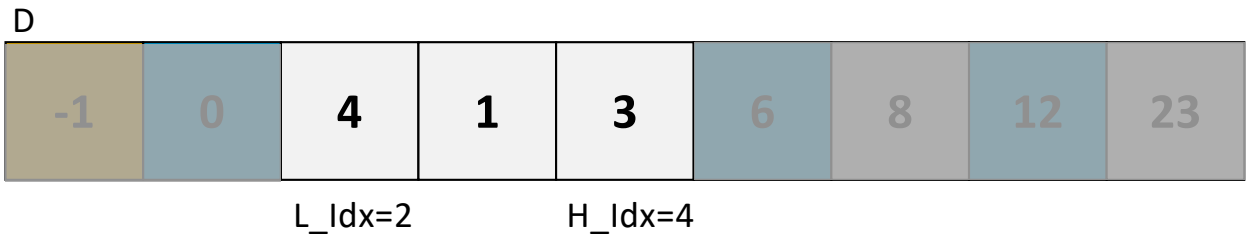
-1	0	4	1	3	6	8	12	23

L_Idx=2 H_Idx=4

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 2, 4)
    if L_Idx < H_Idx:
        pivot = partition(D, 2, 4)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

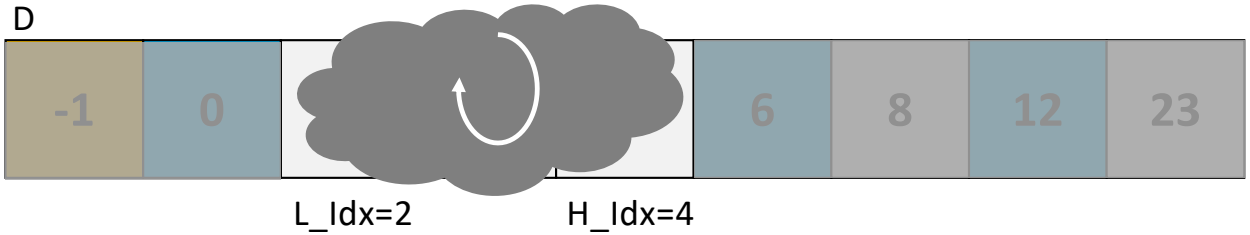



```
partition(D, 2, 4)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm

In-place Implementation

```
quickSort(D, 2, 4)
    if L_Idx < H_Idx:
        pivot = partition(D, 2, 4)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```




```
partition(D, 2, 4)
    pivotin                    / 2
    ret                        -1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 2, 4)
    if L_Idx < H_Idx:
        pivot = partition(D, 2, 4)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

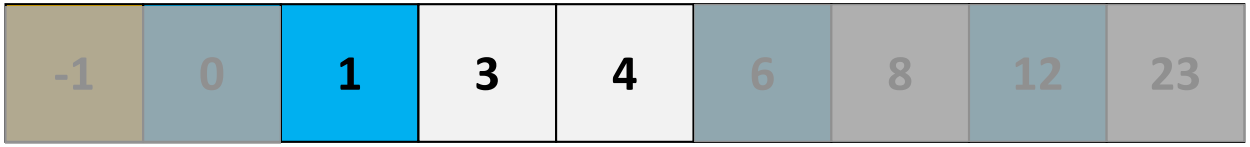

```
partition(D, 2, 4)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

pivot = 2

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 2, 4)
    if L_Idx < H_Idx:

        pivot = partition(D, 2, 4)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D




```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

pivot = 2

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 2, 4)
    if L_Idx < H_Idx:

        pivot = partition(D, 2, 4)
        quickSort(D, 2, 2-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

-1	0	1	3	4	6	8	12	23

L_Idx=2 H_Idx=4

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

pivot = 2

Quick-Sort Algorithm

In-place Implementation

```
quickSort(D, 2, 1)
    if L_Idx < H_Idx:

        pivot = partition(D, L_Idx, H_Idx)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```


```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 2, 4)
    if L_Idx < H_Idx:

        pivot = partition(D, 2, 4)
        quickSort(D, 2, 2-1)
        quickSort(D, pivot+1, H_Idx)
end
```


```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 2, 4)
    if L_Idx < H_Idx:
        pivot = partition(D, 2, 4)
        quickSort(D, 2, 2-1)
        quickSort(D, 2+1, 4)
end
```


```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 3, 4)
    if L_Idx < H_Idx:

        pivot = partition(D, L_Idx, H_Idx)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

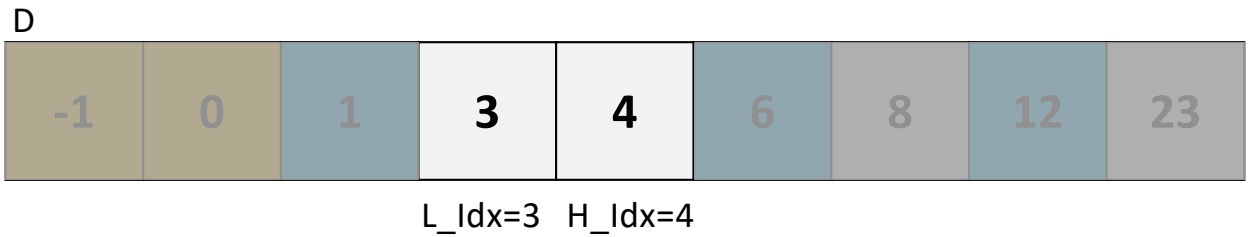
-1	0	1	**3**	**4**	6	8	12	23

L_Idx=3 H_Idx=4

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

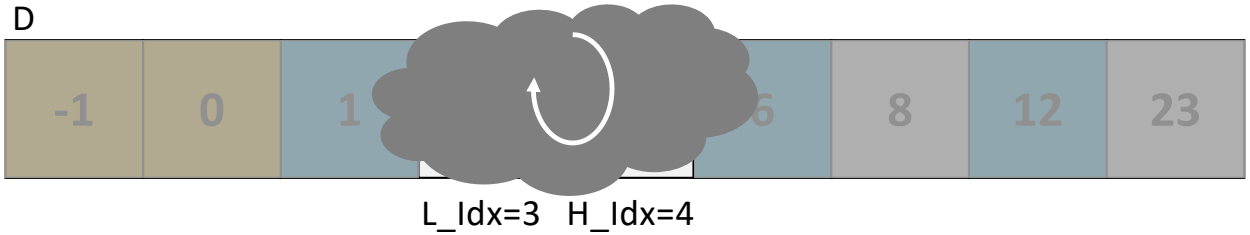
```
quickSort(D, 3, 4)
    if L_Idx < H_Idx:
        pivot = partition(D, L_Idx, H_Idx)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```




```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 3, 4)
    if L_Idx < H_Idx:
        pivot = partition(D, L_Idx, H_Idx)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```




```
partition(D, low, high)
    pivoting ... / 2
    ...
    return ...
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 3, 4)
    if L_Idx < H_Idx:
        pivot = partition(D, L_Idx, H_Idx)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

-1	0	1	**3**	**4**	6	8	12	23

L_Idx=3 H_Idx=4

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

pivot = 3

Quick-Sort Algorithm

In-place Implementation

```
quickSort(D, 3, 4)
    if L_Idx < H_Idx:

        pivot = partition(D, L_Idx, H_Idx)
        quickSort(D, 3, 3-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

-1	0	1	**3**	**4**	6	8	12	23

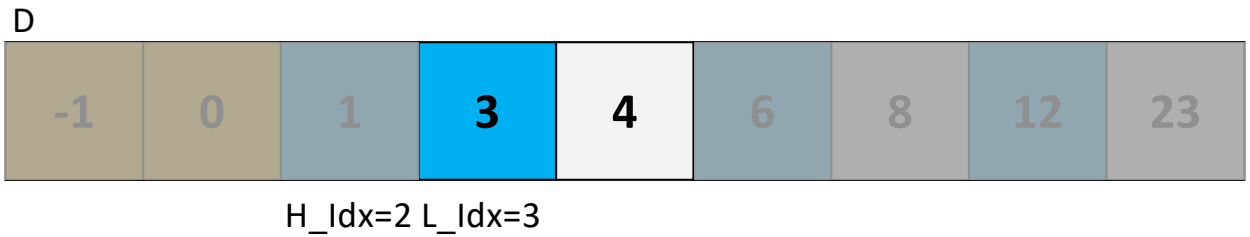
L_Idx=3 H_Idx=4

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

pivot = 3

Quick-Sort Algorithm — In-place Implementation

```
quickSort(D, 3, 2)
    if L_Idx < H_Idx:

        pivot = partition(D, L_Idx, H_Idx)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```




```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 3, 4)
    if L_Idx < H_Idx:

        pivot = partition(D, L_Idx, H_Idx)
        quickSort(D, 3, 3-1)
        quickSort(D, 3+1, 4)
end
```

D

-1	0	1	3	4	6	8	12	23

L_Idx=3 H_Idx=4

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

pivot = 3

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 4, 4)
    if L_Idx < H_Idx:

        pivot = partition(D, L_Idx, H_Idx)
        quickSort(D, L_Idx, pivot-1)
        quickSort(D, pivot+1, H_Idx)
end
```

D

-1	0	1	3	4	6	8	12	23

L_Idx=4
H_Idx=4

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

pivot = 3

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 3, 4)
    if L_Idx < H_Idx:

        pivot = partition(D, L_Idx, H_Idx)
        quickSort(D, 3, 3-1)
        quickSort(D, 3+1, 4)
end
```

D

-1	0	1	3	4	6	8	12	23

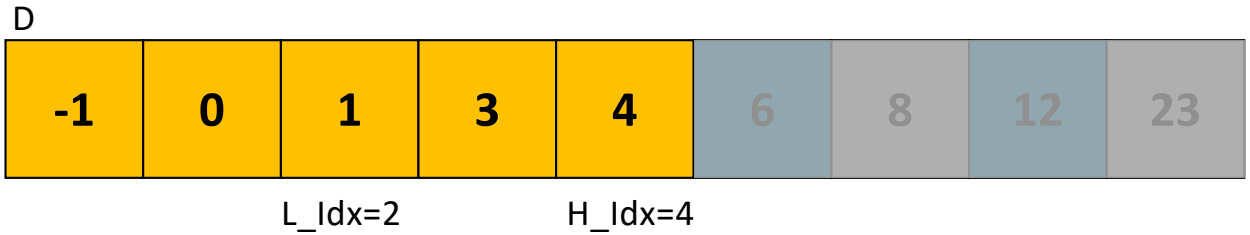
L_Idx=3 H_Idx=4

```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

pivot = 3

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 2, 4)
    if L_Idx < H_Idx:

        pivot = partition(D, 2, 4)
        quickSort(D, 2, 2-1)
        quickSort(D, 2+1, 4)
end
```




```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

pivot = 3

Quick-Sort Algorithm | In-place Implementation

```
quickSort(D, 2, 6)
    if L_Idx < H_Idx:

        pivot = partition(D, 2, 6)
        quickSort(D, 2, 5-1)
        quickSort(D, pivot+1, H_Idx)
end
end
```

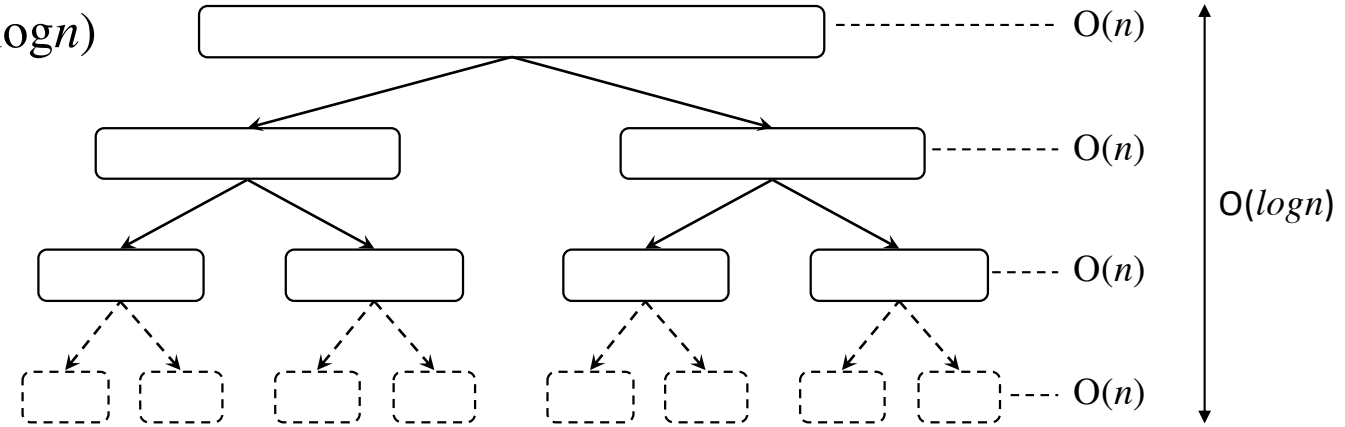
D

-1	0	1	3	4	6	8	12	23

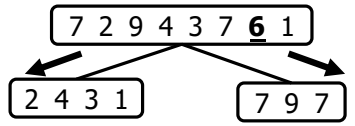
```
partition(D, low, high)
    pivotindex = (low+high) // 2
    swap(pivotindex, high)

    i = low

    for j in range (low, high+1)
        if D[j] <= D[high]
            swap(i, j)
            i += 1
    return i-1
```

Quick-Sort Algorithm — Time Complexity

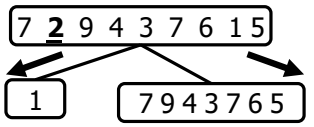
- Depends on pivot
- Best case runtime → $O(n \log n)$



- For a sequence of size D
  - Good Pivot → generates L and G each of size less than ¾D



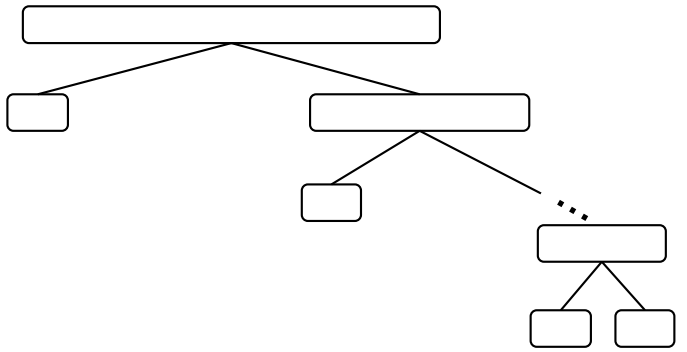
  - Bad Pivot → generates either L or G of size greater than ¾D



Quick-Sort Algorithm — Time Complexity

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element

depth	time
0	n
1	$n-1$
…	…
$n-1$	1



- Worst case runtime → $O(n^2)$