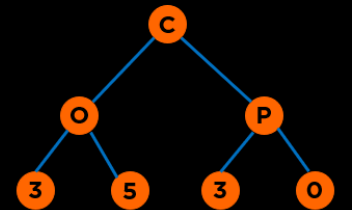


Sorting



Categories of Data Structures

Linear Ordered

Lists

Stacks

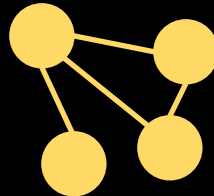
Queues



Non-linear Ordered

Trees

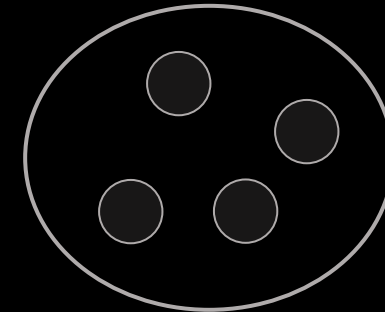
Graphs



Not Ordered

Sets

Tables/Maps



Problem (Sort)

Input: Unordered Collection of size n , $C_i [0.....n-1]$

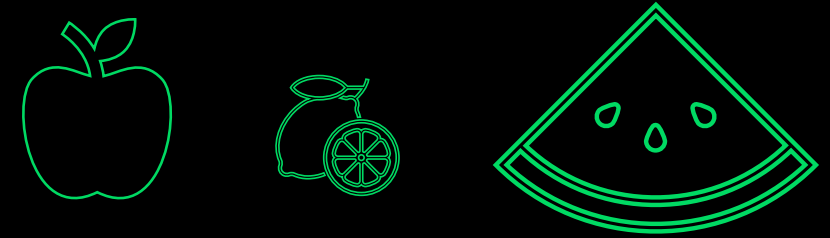
Output: Ordered Collection of size n , $C_o [0.....n-1]$

Example: C_o for ascending sort

$$C_o[0] \leq C_o[1] \leq \dots \leq C_o[n-1]$$

Selection Sort

Selection Sort



Premise:

- Find the smallest/largest element, e_1 in a Collection, C_i
- Move this element, e_1 to its correct position
- Find the next smallest/largest element, e_2 in C_i
- Move this element, e_2 to its correct position
- \vdots
- Repeat this entire process $C_i.size() - 1$ times

Selection Sort

Example:

Initial array

| | | | | |
|----|----|----|----|----|
| 29 | 10 | 14 | 37 | 13 |
|----|----|----|----|----|

 **Sorted**

Selection Sort

Example:

Initial array



1st pass



 **Sorted**

Selection Sort

Example:

Initial array



1st pass



2nd pass



 **Sorted**

Selection Sort

Example:

Initial array



1st pass



2nd pass



3rd pass



 **Sorted**

Selection Sort

Example:

Initial array



1st pass



2nd pass



3rd pass



4th pass



 **Sorted**

Selection Sort

Example:

Initial array



1st pass



2nd pass



3rd pass



4th pass



 **Sorted**

Selection Sort Code

```
01 void selectionSort(int array[], int size)
02 {
03     for (int i = 0; i < size - 1; i++)
04     {
05         int min_index = i;
06         for (int j = i + 1; j < size; j++)
07         {
08             if (array[j] < array[min_index])
09                 min_index = j;
10         }
11         // put min at the correct position
12         swap(&array[min_index], &array[i]);
13     }
14 }
```

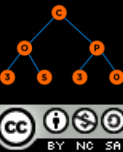
Selection Sort

Time Complexity

For very large n we can ignore all but the significant term in the expression, so the number of

- comparisons is $O(n^2)$
- exchanges is $O(n)$

An $O(n^2)$ sort is called a *quadratic sort*

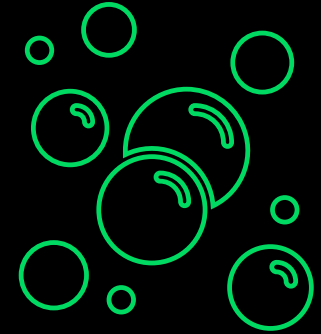


Selection Sort Complexity

| | Selection Sort | Bubble Sort | Insertion Sort |
|--------------|----------------|-------------|----------------|
| Worst Case | $O(n^2)$ | | |
| Average Case | $O(n^2)$ | | |
| Best Case | $O(n^2)$ | | |
| Space | $O(1)$ | | |

Bubble Sort

Bubble Sort



Premise:

- Swap adjacent elements, e_i and e_{i+1} in a Collection, C_i if they are out of order
- Repeat swapping till you reach the end of the Collection to bubble up the largest element after each iteration
- Repeat this entire process $C_i.size() - 1$ times stopping at $C_i.size() - i$ after i^{th} iteration

Bubble Sort

Example:

Initial array

29

10

14

37

13

 Sorted

Bubble Sort

Example:

Initial array

| | | | | |
|----|----|----|----|----|
| 29 | 10 | 14 | 37 | 13 |
|----|----|----|----|----|

1st pass

| | | | | |
|----|----|----|----|----|
| 10 | 14 | 29 | 13 | 37 |
|----|----|----|----|----|

 Sorted

Bubble Sort

Example:

Initial array

| | | | | |
|----|----|----|----|----|
| 29 | 10 | 14 | 37 | 13 |
|----|----|----|----|----|

1st pass

| | | | | |
|----|----|----|----|----|
| 10 | 14 | 29 | 13 | 37 |
|----|----|----|----|----|

2nd pass

| | | | | |
|----|----|----|----|----|
| 10 | 14 | 13 | 29 | 37 |
|----|----|----|----|----|

 Sorted

Bubble Sort

Example:

Initial array

| | | | | |
|----|----|----|----|----|
| 29 | 10 | 14 | 37 | 13 |
|----|----|----|----|----|

1st pass

| | | | | |
|----|----|----|----|----|
| 10 | 14 | 29 | 13 | 37 |
|----|----|----|----|----|

2nd pass

| | | | | |
|----|----|----|----|----|
| 10 | 14 | 13 | 29 | 37 |
|----|----|----|----|----|

3rd pass

| | | | | |
|----|----|----|----|----|
| 10 | 13 | 14 | 29 | 37 |
|----|----|----|----|----|

 Sorted

Bubble Sort

Example:

Initial array

| | | | | |
|----|----|----|----|----|
| 29 | 10 | 14 | 37 | 13 |
|----|----|----|----|----|

1st pass

| | | | | |
|----|----|----|----|----|
| 10 | 14 | 29 | 13 | 37 |
|----|----|----|----|----|

2nd pass

| | | | | |
|----|----|----|----|----|
| 10 | 14 | 13 | 29 | 37 |
|----|----|----|----|----|

3rd pass

| | | | | |
|----|----|----|----|----|
| 10 | 13 | 14 | 29 | 37 |
|----|----|----|----|----|

4th pass

| | | | | |
|----|----|----|----|----|
| 10 | 13 | 14 | 29 | 37 |
|----|----|----|----|----|

 Sorted

Bubble Sort

Example:

Initial array

| | | | | |
|----|----|----|----|----|
| 29 | 10 | 14 | 37 | 13 |
|----|----|----|----|----|

1st pass

| | | | | |
|----|----|----|----|----|
| 10 | 14 | 29 | 13 | 37 |
|----|----|----|----|----|

2nd pass

| | | | | |
|----|----|----|----|----|
| 10 | 14 | 13 | 29 | 37 |
|----|----|----|----|----|

3rd pass

| | | | | |
|----|----|----|----|----|
| 10 | 13 | 14 | 29 | 37 |
|----|----|----|----|----|

4th pass

| | | | | |
|----|----|----|----|----|
| 10 | 13 | 14 | 29 | 37 |
|----|----|----|----|----|

Final array

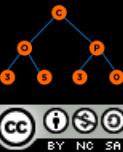
| | | | | |
|----|----|----|----|----|
| 10 | 13 | 14 | 29 | 37 |
|----|----|----|----|----|

 Sorted

Bubble Sort Code

```
01 void bubbleSort(int array[], int size)
02 {
03     for (int i = 0; i < size - 1; i++)
04     {
05         int swapped = 0;
06         for (int j = 0; j < size - i - 1; ++j)
07         {
08             if (array[j] > array[j + 1])
09             {
10                 int temp = array[j];
11                 array[j] = array[j + 1];
12                 array[j + 1] = temp;
13                 swapped = 1;
14             }
15         }
16         // If there is no swapping in the last swap, then the array is already sorted.
17         if (swapped == 0)
18             break;
19     }
20 }
```

<https://onlinegdb.com/H1thqETkv>



Bubble Sort

Time Complexity

In the worst case,

- comparisons is $O(n^2)$
- exchanges is $O(n^2)$

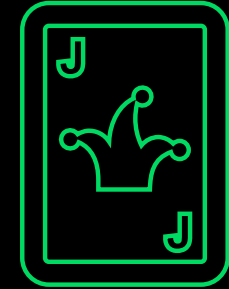
Compared to selection sort with its $O(n^2)$ comparisons and $O(n)$ exchanges, bubble sort usually performs worse

Bubble Sort Complexity

| | Selection Sort | Bubble Sort | Insertion Sort |
|---------------------|----------------|-------------|----------------|
| Worst Case | $O(n^2)$ | $O(n^2)$ | |
| Average Case | $O(n^2)$ | $O(n^2)$ | |
| Best Case | $O(n^2)$ | $O(n)$ | |
| Space | $O(1)$ | $O(1)$ | |

Insertion Sort

Insertion Sort



Premise:

- Keeps a track of two regions: **Sorted** and **Unsorted**
- Initially, the sorted region has one element
- Insert the first element in the unsorted region in the correct place in the sorted region
- \vdots
- Repeat this entire process till there are no more elements in unsorted region

Insertion Sort

Example:

Initial array

29

10

14

37

13

 Sorted

Insertion Sort

Example:

Initial array

| | | | | |
|----|----|----|----|----|
| 29 | 10 | 14 | 37 | 13 |
|----|----|----|----|----|

1st pass

| | | | | |
|----|----|----|----|----|
| 29 | 10 | 14 | 37 | 13 |
|----|----|----|----|----|

Sorted

Unsorted

 Sorted

Insertion Sort

Example:

Initial array

| | | | | |
|----|----|----|----|----|
| 29 | 10 | 14 | 37 | 13 |
|----|----|----|----|----|

1st pass

| | | | | |
|----|----|----|----|----|
| 29 | 10 | 14 | 37 | 13 |
|----|----|----|----|----|

2nd pass

| | | | | |
|----|----|----|----|----|
| 10 | 29 | 14 | 37 | 13 |
|----|----|----|----|----|

 Sorted

Insertion Sort

Example:

Initial array

| | | | | |
|----|----|----|----|----|
| 29 | 10 | 14 | 37 | 13 |
|----|----|----|----|----|

1st pass

| | | | | |
|----|----|----|----|----|
| 29 | 10 | 14 | 37 | 13 |
|----|----|----|----|----|

2nd pass

| | | | | |
|----|----|----|----|----|
| 10 | 29 | 14 | 37 | 13 |
|----|----|----|----|----|

3rd pass

| | | | | |
|----|----|----|----|----|
| 10 | 14 | 29 | 37 | 13 |
|----|----|----|----|----|

 Sorted

Insertion Sort

Example:

Initial array

| | | | | |
|----|----|----|----|----|
| 29 | 10 | 14 | 37 | 13 |
|----|----|----|----|----|

1st pass

| | | | | |
|----|----|----|----|----|
| 29 | 10 | 14 | 37 | 13 |
|----|----|----|----|----|

2nd pass

| | | | | |
|----|----|----|----|----|
| 10 | 29 | 14 | 37 | 13 |
|----|----|----|----|----|

3rd pass

| | | | | |
|----|----|----|----|----|
| 10 | 14 | 29 | 37 | 13 |
|----|----|----|----|----|

4th pass

| | | | | |
|----|----|----|----|----|
| 10 | 14 | 29 | 37 | 13 |
|----|----|----|----|----|

 Sorted

Insertion Sort

Example:

| | | | | | |
|-----------------------------------|----|----|----|----|----|
| Initial array | 29 | 10 | 14 | 37 | 13 |
| 1 st pass | 29 | 10 | 14 | 37 | 13 |
| 2 nd pass | 10 | 29 | 14 | 37 | 13 |
| 3 rd pass | 10 | 14 | 29 | 37 | 13 |
| 4 th pass | 10 | 14 | 29 | 37 | 13 |
| 5 th Pass/ Final array | 10 | 13 | 14 | 29 | 37 |

 Sorted

Insertion Sort Code

```
01 void insertionSort(int array[], int size)
02 {
03     for (int i = 1; i < size; i++)
04     {
05         int key = array[i];
06         int j = i-1;
07
08         // Compare key with each element in sorted till smaller value is found
09         while (key < array[j] && j >= 0)
10         {
11             array[j+1] = array[j];
12             j--;
13         }
14         array[j+1] = key;
15     }
16 }
```

Insertion Sort Pseudocode

Time Complexity

In the worst case,

- comparisons is $O(n^2)$
- exchanges is $O(n^2)$

Insertion Sort Complexity

| | Selection Sort | Bubble Sort | Insertion Sort |
|---------------------|----------------|-------------|----------------|
| Worst Case | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |
| Average Case | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |
| Best Case | $O(n^2)$ | $O(n)$ | $O(n)$ |
| Space | $O(1)$ | $O(1)$ | $O(1)$ |

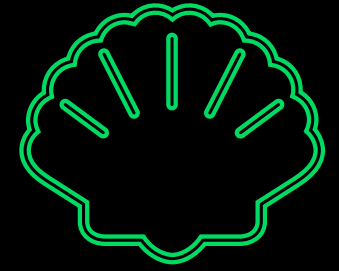
Resources

- <https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html>
- <https://www.programiz.com/dsa>
- <https://www.youtube.com/user/AlgoRythmics/videos>
- <https://www.toptal.com/developers/sorting-algorithms>

Questions

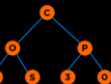
Shell Sort

Shell Sort



Premise:

- A Shell sort is a type of insertion sort, but with $O(n^{3/2})$ or better performance than the $O(n^2)$ sorts
- Instead of sorting the entire array, Shell sort sorts many smaller subarrays using insertion sort before sorting the entire array
- Named after Donald Shell



Shell Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/2$

Shell Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/2$



Shell Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/2$



Shell Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 4 | 6 | 3 | 7 | 8 | 9 | 5 |
|---|---|---|---|---|---|---|---|

Shell Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 4 | 6 | 3 | 7 | 8 | 9 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$

Shell Sort

Example:

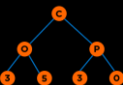
Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 4 | 6 | 3 | 7 | 8 | 9 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$



Shell Sort

Example:

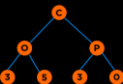
Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 4 | 6 | 3 | 7 | 8 | 9 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$



Shell Sort

Example:

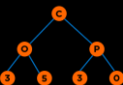
Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 8 | 9 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$



Shell Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 8 | 9 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$

Shell Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 8 | 9 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$

Shell Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 8 | 9 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$

Shell Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 8 | 9 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$

Shell Sort

Example:

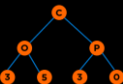
Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 8 | 9 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$



Shell Sort

Example:

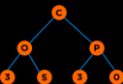
Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 5 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$



Shell Sort

Example:

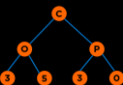
Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 5 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$



Shell Sort

Example:

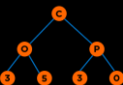
Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 5 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$



Shell Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 5 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 5 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/8$

Shell Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 5 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 5 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/8$

Shell Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

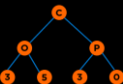
Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 5 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 5 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/8$



Shell Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 5 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 5 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/8$

Shell Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 5 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 6 | 7 | 5 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/8$

Shell Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 5 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 6 | 7 | 5 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/8$

Shell Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

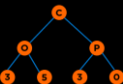
Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 5 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 6 | 7 | 5 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/8$



Shell Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

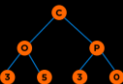
Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 5 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 6 | 5 | 7 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/8$



Shell Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 5 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 6 | 5 | 7 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/8$

Shell Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

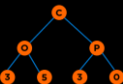
Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 5 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 6 | 7 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/8$



Shell Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 5 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 6 | 7 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/8$

Shell Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

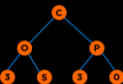
Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 5 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 6 | 7 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/8$



Shell Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

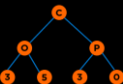
Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 5 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|

Gap = $n/8$



Shell Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Gap = $n/2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 6 | 4 | 7 | 5 | 9 | 8 |
|---|---|---|---|---|---|---|---|

Gap = $n/4$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|

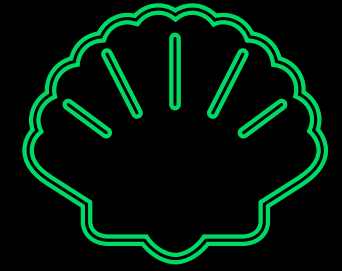
Gap = $n/8$

Shell Sort Pseudocode

Shell Sort Algorithm

1. Set the initial value of gap to $n / 2$
2. while gap > 0
3. for each array element from position gap to the last element
4. Insert this element where it belongs in its subarray.
5. if gap is 2, set it to 1
6. else gap = gap / 2.2. *// chosen by experimentation*

Shell Sort



Time Complexity:

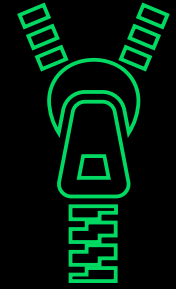
- A general analysis of Shell sort is an open research problem in computer science
- Performance depends on how the decreasing sequence of values for gap is chosen
- If successive powers of 2 are used for gap, performance is $O(n^2)$
- If successive values for gap are based on Hibbard's sequence,
 $2^k - 1$ (i.e. 31, 15, 7, 3, 1)
it can be proven that the performance is $O(n^{3/2})$
- Other sequences give similar or better performance

Shell Sort Time Complexity

| | Shell Sort |
|--------------|--------------|
| Worst Case | $O(n^2)$ |
| Average Case | $O(n^{5/4})$ |
| Best Case | $O(n^{7/6})$ |
| Space | $O(1)$ |

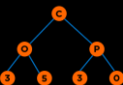
Merge Sort

Merge Sort



Premise:

- Merge sort splits the array in half, sorts the two smaller halves, then merges the two sorted halves together.
- **Divide** the array to be sorted into smaller subarrays till you reach a size of 1
- In the **Conquer** step, sort the two subarrays
- In the **Combine** step, combine two sorted arrays
 -
 -
 -
- Repeat this till you merge all elements in one array



Merge Sort

Example:

Initial array

| | | | | | |
|---|---|----|----|---|---|
| 6 | 5 | 22 | 10 | 9 | 1 |
|---|---|----|----|---|---|

☐ Sorted

Merge Sort

Example:

Divide

Initial array

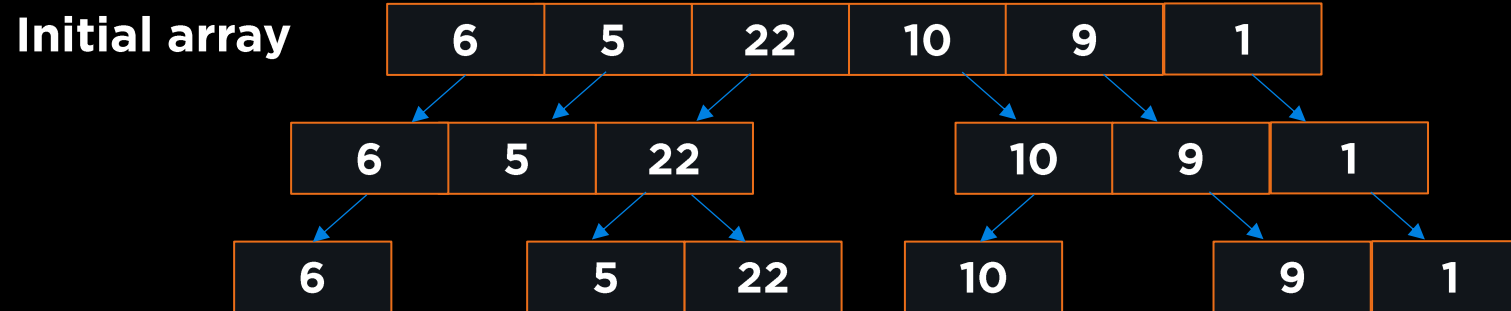


Sorted

Merge Sort

Example:

Divide

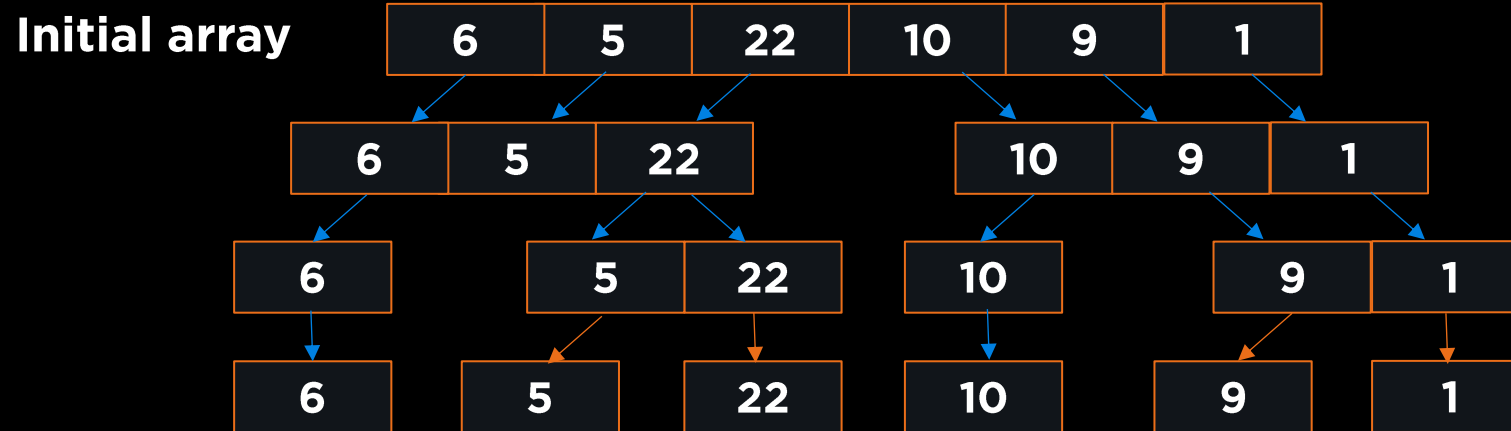


Sorted

Merge Sort

Example:

Divide

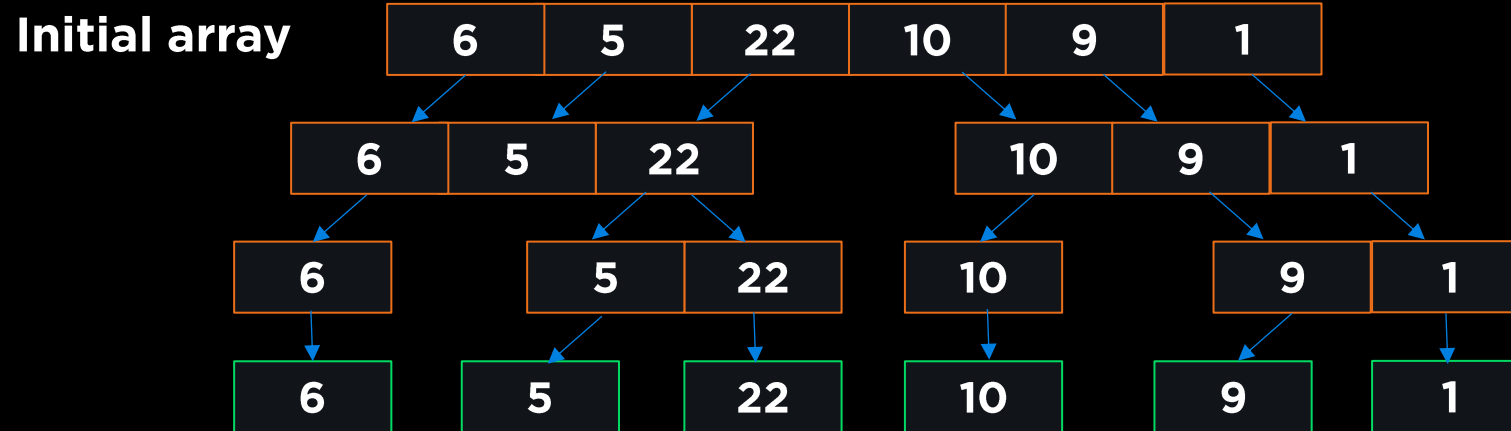


 Sorted

Merge Sort

Example:

Conquer

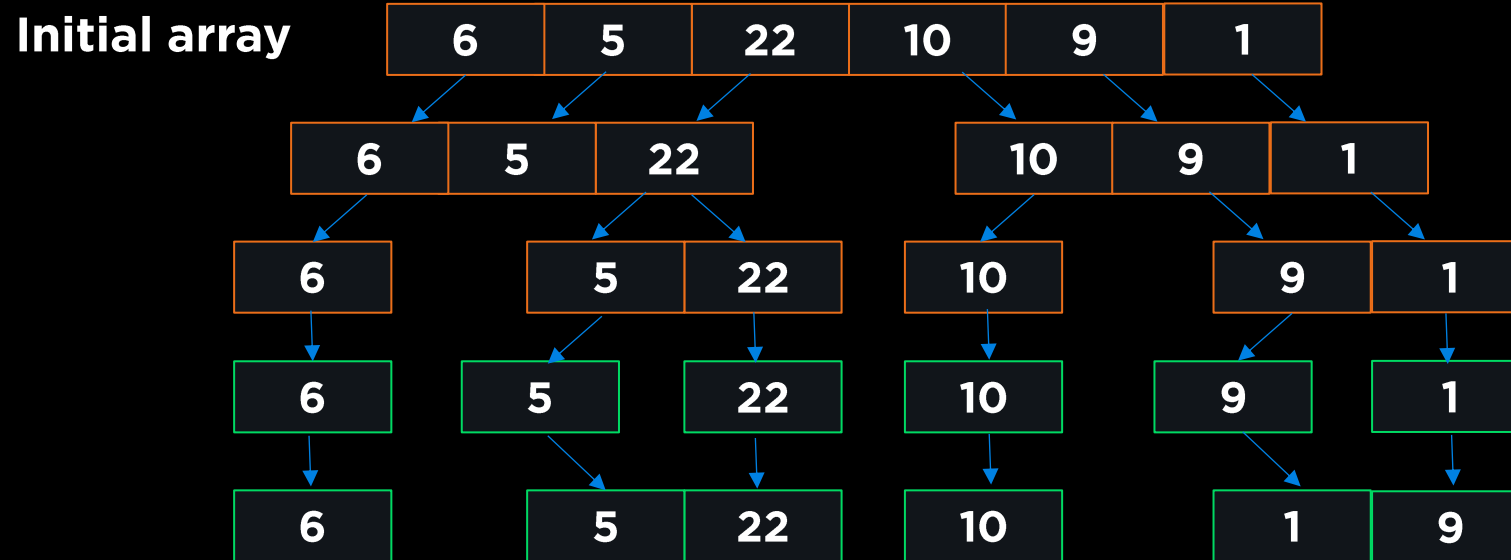


 Sorted

Merge Sort

Example:

Combine

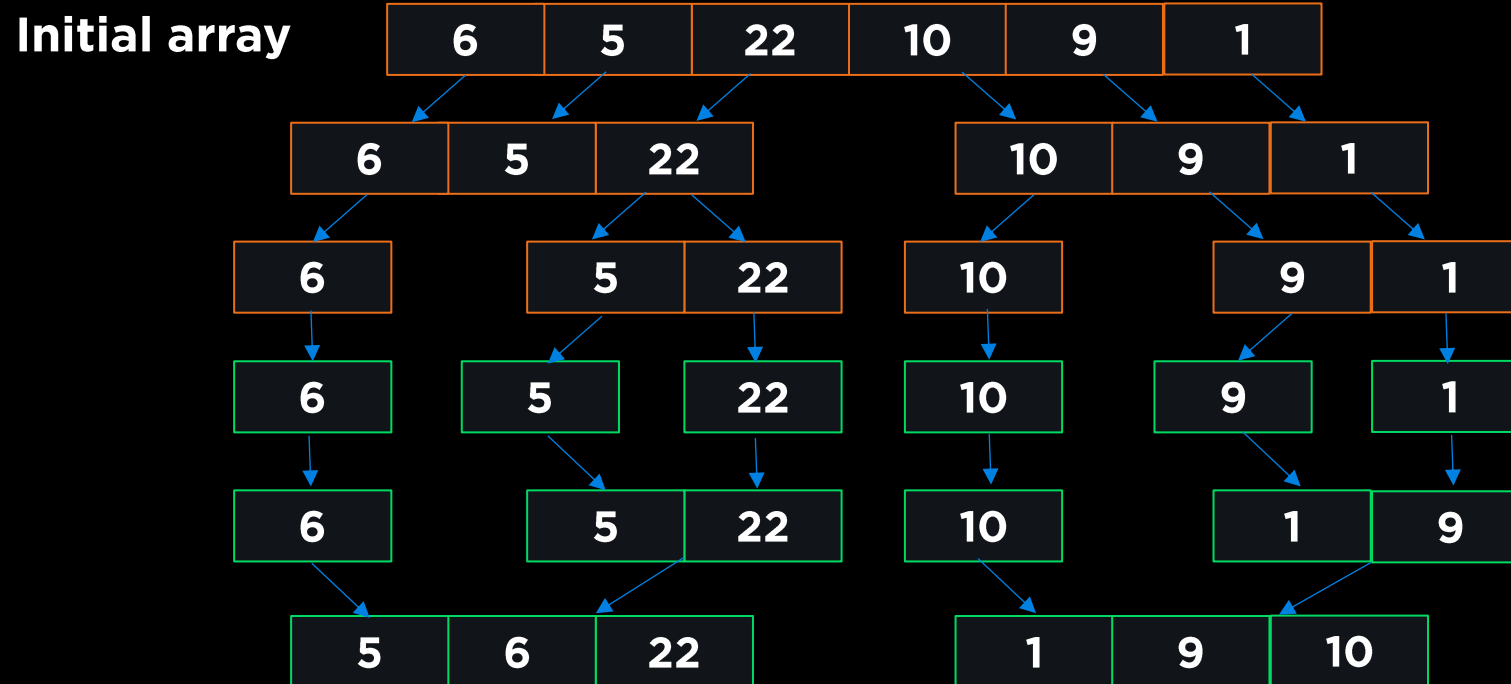


 Sorted

Merge Sort

Example:

Combine

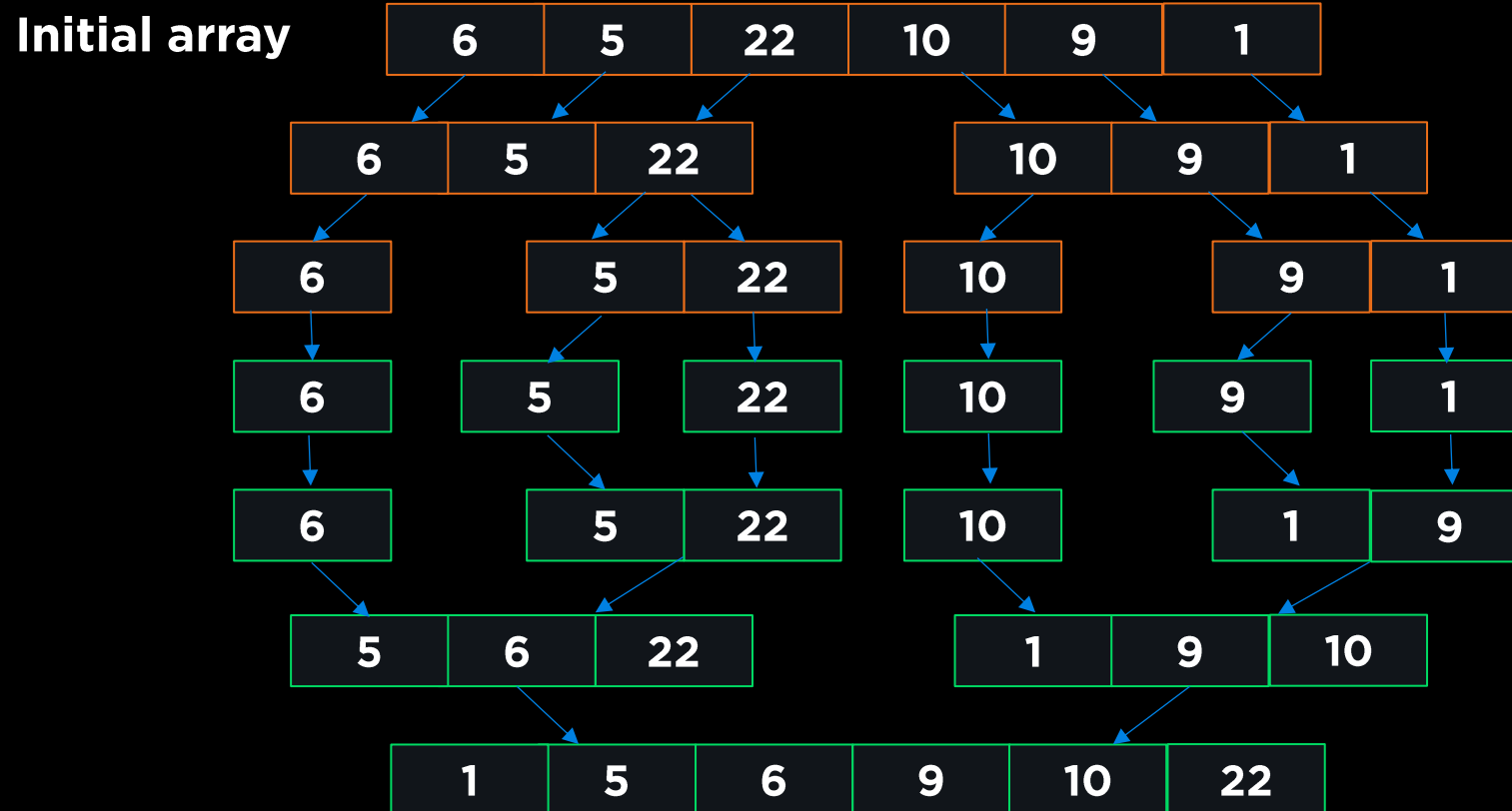


 Sorted

Merge Sort

Example:

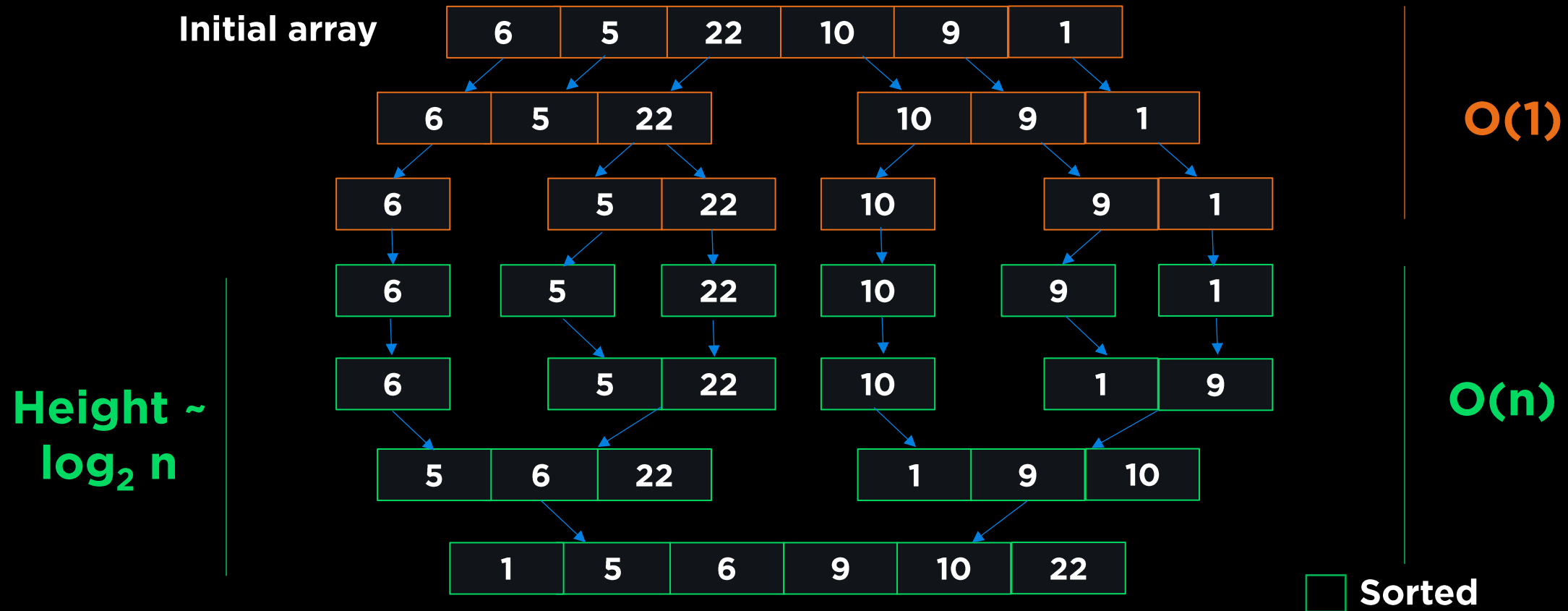
Combine



Sorted

Merge Sort

Example:



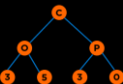
Merge Sort Pseudocode

```
mergeSort(int [] numbers, int start, int end)
{
    if (start < end) //base case is start = end and sorting an array of 1
    {
        middle = (start+end)/2;
        mergeSort(numbers, start, middle);
        mergeSort(numbers, middle+1, end);
        merge(numbers, start, middle, end);
    }
}
```

`mergeSort(A, 0, length(A)-1)`

Merge Algorithm

1. Access the first item from both sequences.
2. while not finished with either sequence
3. Compare the current items from the two sequences, copy the smaller current item to the output sequence and access the next item from the input sequence whose item was copied.
4. Copy any remaining items from the first sequence to the output sequence.
5. Copy any remaining items from the second sequence to the output sequence.



Merge Sort Code

```
1. void mergeSort(int arr[], int left, int right)
2. {
3.     if (left < right)
4.     {
5.         // m is the point where the array is divided into two subarrays
6.         int mid = left + (right - left) / 2;
7.         mergeSort(arr, left, mid);
8.         mergeSort(arr, mid + 1, right);
9.
10.        // Merge the sorted subarrays
11.        merge(arr, left, mid, right);
12.    }
13. }
```

mergeSort(A, 0, length(A)-1)

Merge Sort Code

```
14. // Merge two subarrays from arr
15. void merge(int arr[], int left, int mid, int right)
16. {
17.     // Create X ← arr[left..mid] & Y ← arr[mid+1..right]
18.     int n1 = mid - left + 1;
19.     int n2 = right - mid;
20.     int X[n1], Y[n2];
21.
22.     for (int i = 0; i < n1; i++)
23.         X[i] = arr[left + i];
24.     for (int j = 0; j < n2; j++)
25.         Y[j] = arr[mid + 1 + j];
26.
27. // Merge the arrays X and Y into arr
28.     int i, j, k;
29.     i = 0;
30.     j = 0;
31.     k = left;
```

```
32. while (i < n1 && j < n2)
33. {
34.     if (X[i] <= Y[j])
35.     {
36.         arr[k] = X[i];
37.         i++;
38.     }
39.     else
40.     {
41.         arr[k] = Y[j];
42.         j++;
43.     }
44.     k++;
45. }
```

```
46. // When we run out of elements
    // in either X or Y append the remaining elements
47. while (i < n1)
48. {
49.     arr[k] = X[i];
50.     i++;
51.     k++;
52. }
53. while (j < n2)
54. {
55.     arr[k] = Y[j];
56.     j++;
57.     k++;
58. }
59. }
```

Merge Sort Code

```
1. void mergeSort(int arr[], int left, int right)
2. {
3.     if (left < right)
4.     {
5.         int mid = left + (right - left) / 2;
6.         mergeSort(arr, left, mid);
7.         mergeSort(arr, mid + 1, right);
8.
9.         // Merge the sorted subarrays
10.        merge(arr, left, mid, right);
11.    }
12. }
```

arr

| | | | |
|----|---|---|---|
| 15 | 2 | 7 | 0 |
| 0 | 1 | 2 | 3 |

mergeSort(arr, 0, 3)

Merge Sort Code

```
1. void mergeSort(int arr[], int left, int right)
2. {
3.     if (left < right)
4.     {
5.         int mid = left + (right - left) / 2;
6.         mergeSort(arr, left, mid);
7.         mergeSort(arr, mid + 1, right);
8.
9.         // Merge the sorted subarrays
10.        merge(arr, left, mid, right);
11.    }
12. }
```

| arr | | | |
|-----|---|---|---|
| 15 | 2 | 7 | 0 |
| 0 | 1 | 2 | 3 |

mergeSort(arr, 0, 3)

mergeSort(arr, 0, 1)

Merge Sort Code

| arr | | | |
|-----|---|---|---|
| 15 | 2 | 7 | 0 |
| 0 | 1 | 2 | 3 |

```
1. void mergeSort(int arr[], int left, int right)
2. {
3.     if (left < right)
4.     {
5.         int mid = left + (right - left) / 2;
6.         mergeSort(arr, left, mid);
7.         mergeSort(arr, mid + 1, right);
8.
9.         // Merge the sorted subarrays
10.        merge(arr, left, mid, right);
11.    }
12. }
```

mergeSort(arr, 0, 3)

mergeSort(arr, 0, 1)

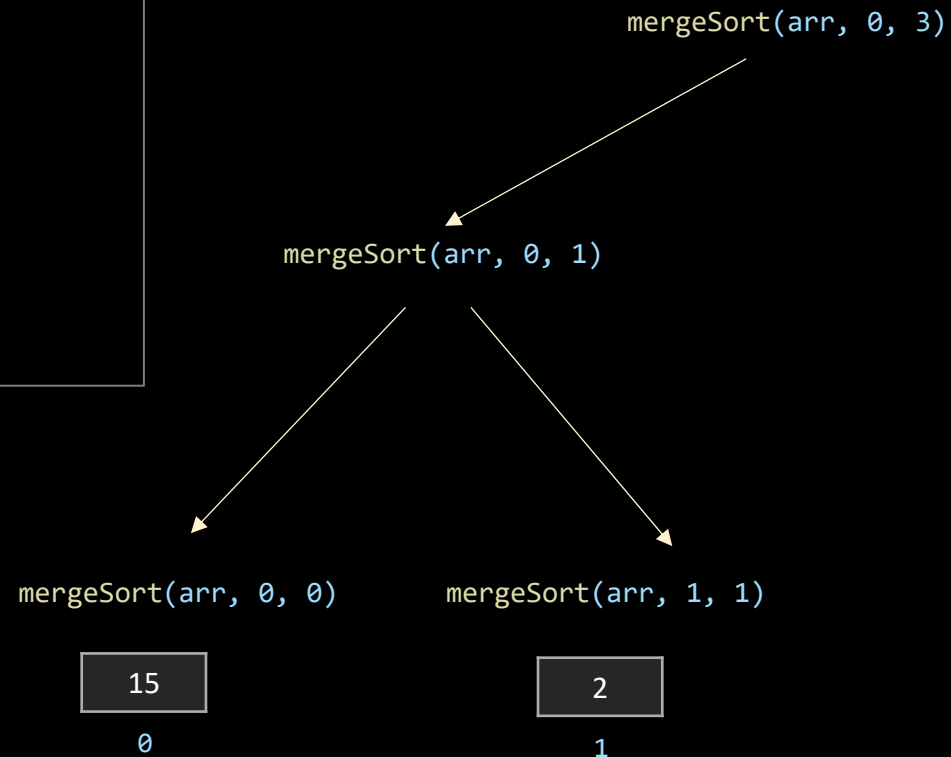
mergeSort(arr, 0, 0)

| |
|----|
| 15 |
| 0 |

Merge Sort Code

```
1. void mergeSort(int arr[], int left, int right)
2. {
3.     if (left < right)
4.     {
5.         int mid = left + (right - left) / 2;
6.         mergeSort(arr, left, mid);
7.         mergeSort(arr, mid + 1, right);
8.
9.         // Merge the sorted subarrays
10.        merge(arr, left, mid, right);
11.    }
12. }
```

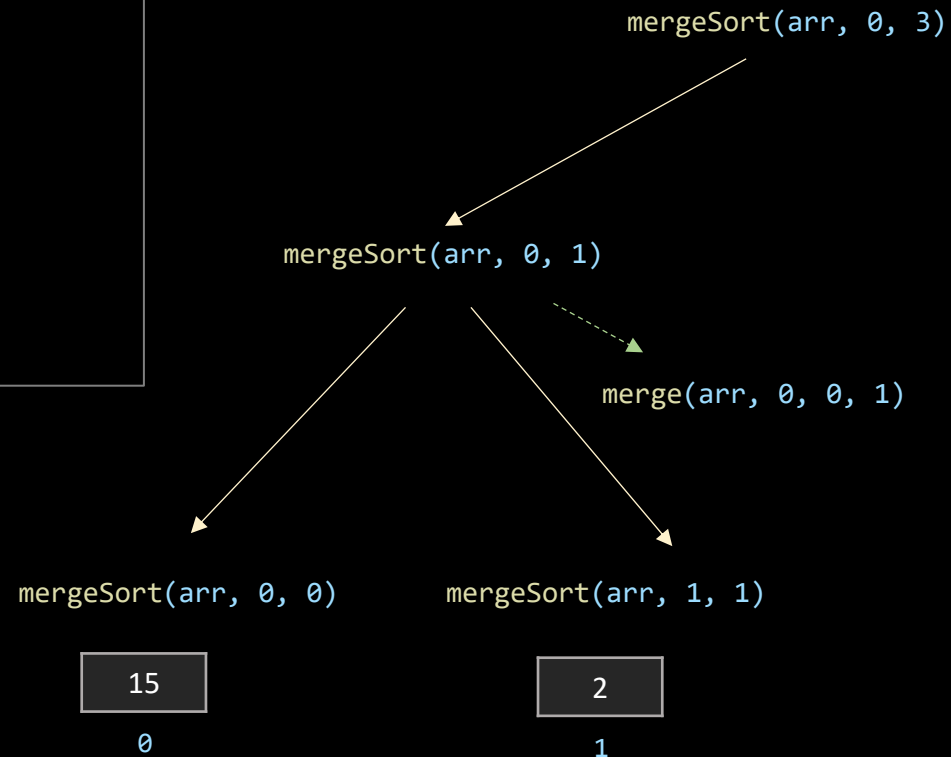
| arr | | | |
|-----|---|---|---|
| 15 | 2 | 7 | 0 |
| 0 | 1 | 2 | 3 |



Merge Sort Code

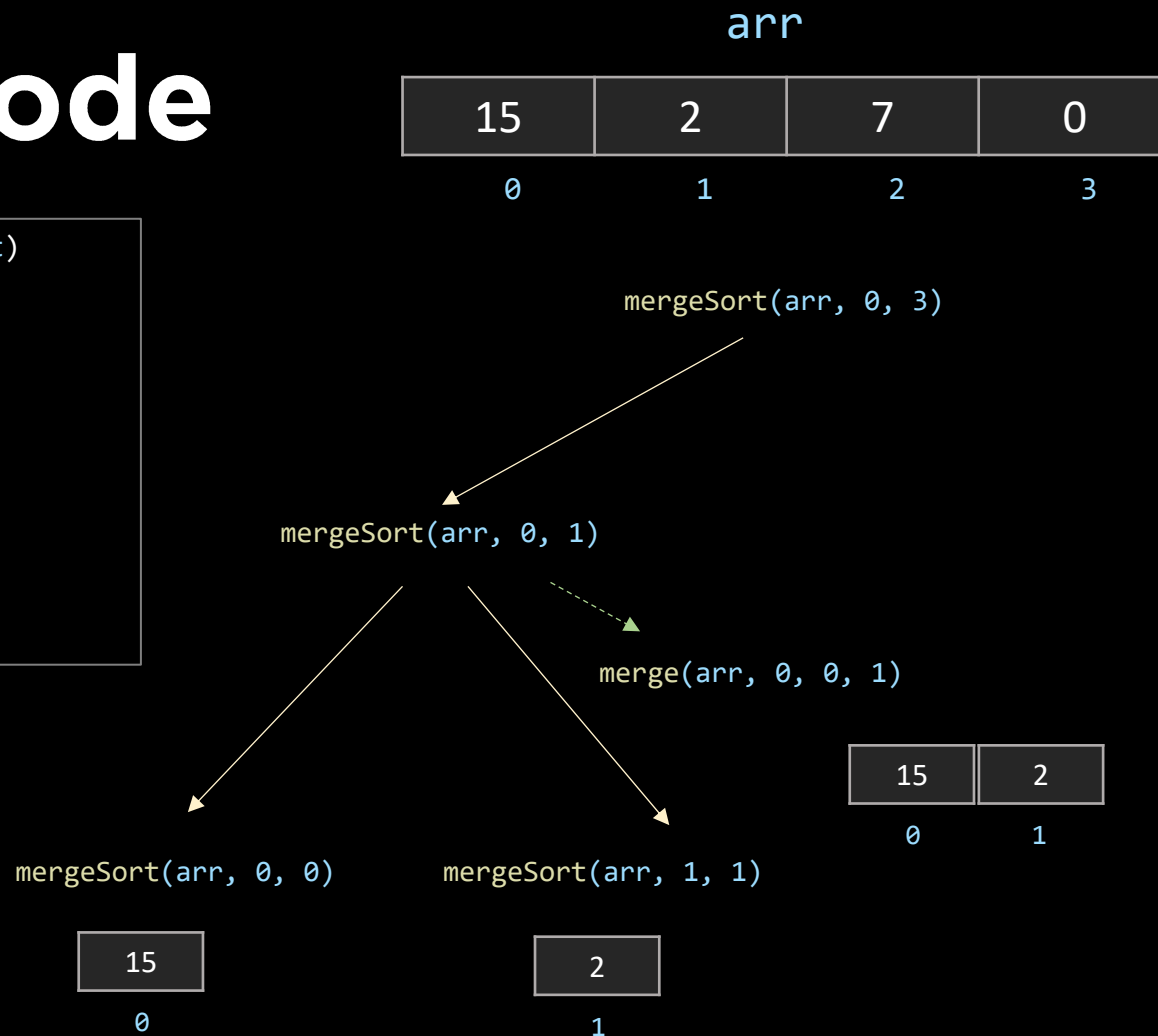
```
1. void mergeSort(int arr[], int left, int right)
2. {
3.     if (left < right)
4.     {
5.         int mid = left + (right - left) / 2;
6.         mergeSort(arr, left, mid);
7.         mergeSort(arr, mid + 1, right);
8.
9.         // Merge the sorted subarrays
10.        merge(arr, left, mid, right);
11.    }
12. }
```

| arr | | | |
|-----|---|---|---|
| 15 | 2 | 7 | 0 |
| 0 | 1 | 2 | 3 |



Merge Sort Code

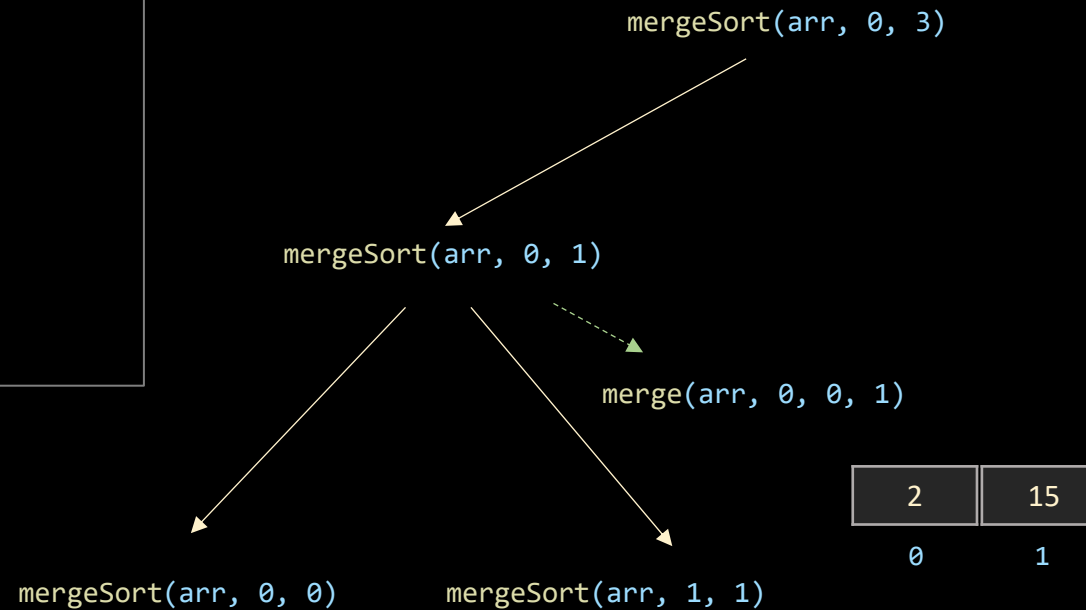
```
1. void mergeSort(int arr[], int left, int right)
2. {
3.     if (left < right)
4.     {
5.         int mid = left + (right - left) / 2;
6.         mergeSort(arr, left, mid);
7.         mergeSort(arr, mid + 1, right);
8.
9.         // Merge the sorted subarrays
10.        merge(arr, left, mid, right);
11.    }
12. }
```



Merge Sort Code

```
1. void mergeSort(int arr[], int left, int right)
2. {
3.     if (left < right)
4.     {
5.         int mid = left + (right - left) / 2;
6.         mergeSort(arr, left, mid);
7.         mergeSort(arr, mid + 1, right);
8.
9.         // Merge the sorted subarrays
10.        merge(arr, left, mid, right);
11.    }
12. }
```

| arr | | | |
|-----|---|---|---|
| 15 | 2 | 7 | 0 |
| 0 | 1 | 2 | 3 |



Merge Sort Code

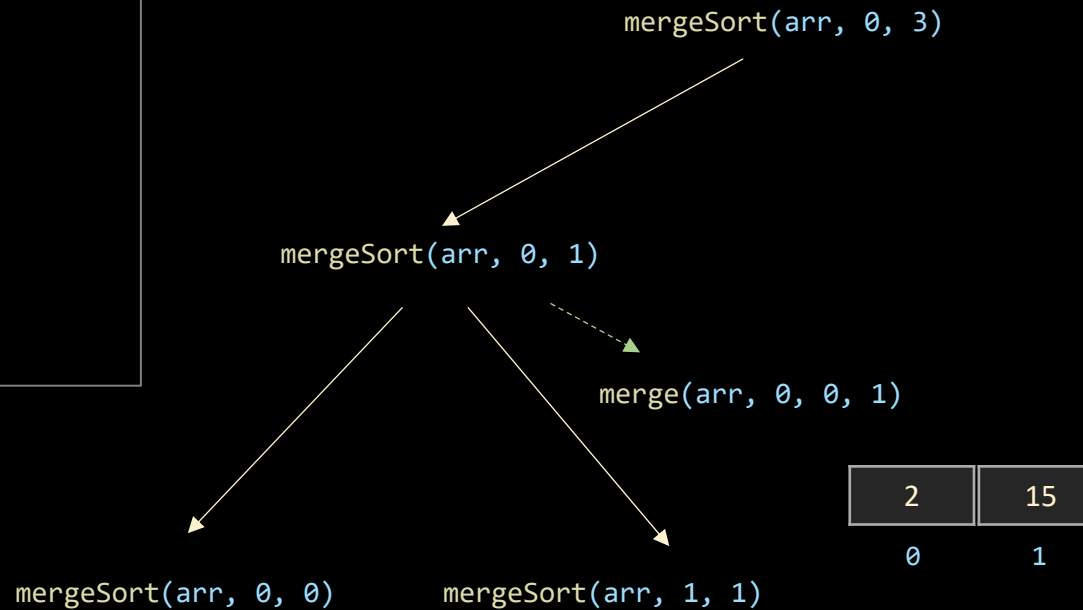
```
1. void mergeSort(int arr[], int left, int right)
2. {
3.     if (left < right)
4.     {
5.         int mid = left + (right - left) / 2;
6.         mergeSort(arr, left, mid);
7.         mergeSort(arr, mid + 1, right);
8.
9.         // Merge the sorted subarrays
10.        merge(arr, left, mid, right);
11.    }
12. }
```

arr

| | | | |
|---|----|---|---|
| 2 | 15 | 7 | 0 |
| 0 | 1 | 2 | 3 |

arr

| | | | |
|----|---|---|---|
| 15 | 2 | 7 | 0 |
| 0 | 1 | 2 | 3 |

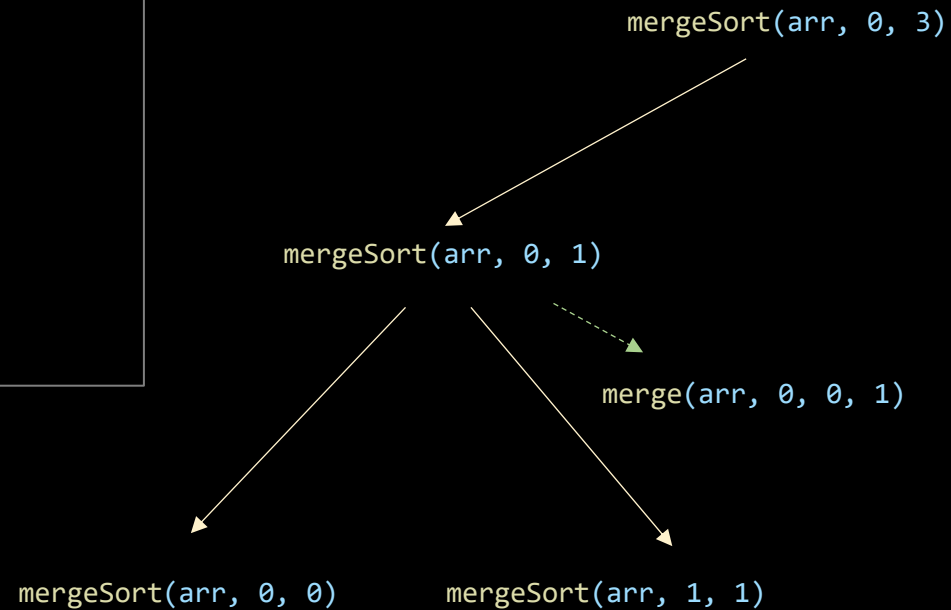


Merge Sort Code

```
1. void mergeSort(int arr[], int left, int right)
2. {
3.     if (left < right)
4.     {
5.         int mid = left + (right - left) / 2;
6.         mergeSort(arr, left, mid);
7.         mergeSort(arr, mid + 1, right);
8.
9.         // Merge the sorted subarrays
10.        merge(arr, left, mid, right);
11.    }
12. }
```

| arr | | | |
|-----|---|---|---|
| 15 | 2 | 7 | 0 |
| 0 | 1 | 2 | 3 |

| arr | | | |
|-----|----|---|---|
| 2 | 15 | 7 | 0 |
| 0 | 1 | 2 | 3 |

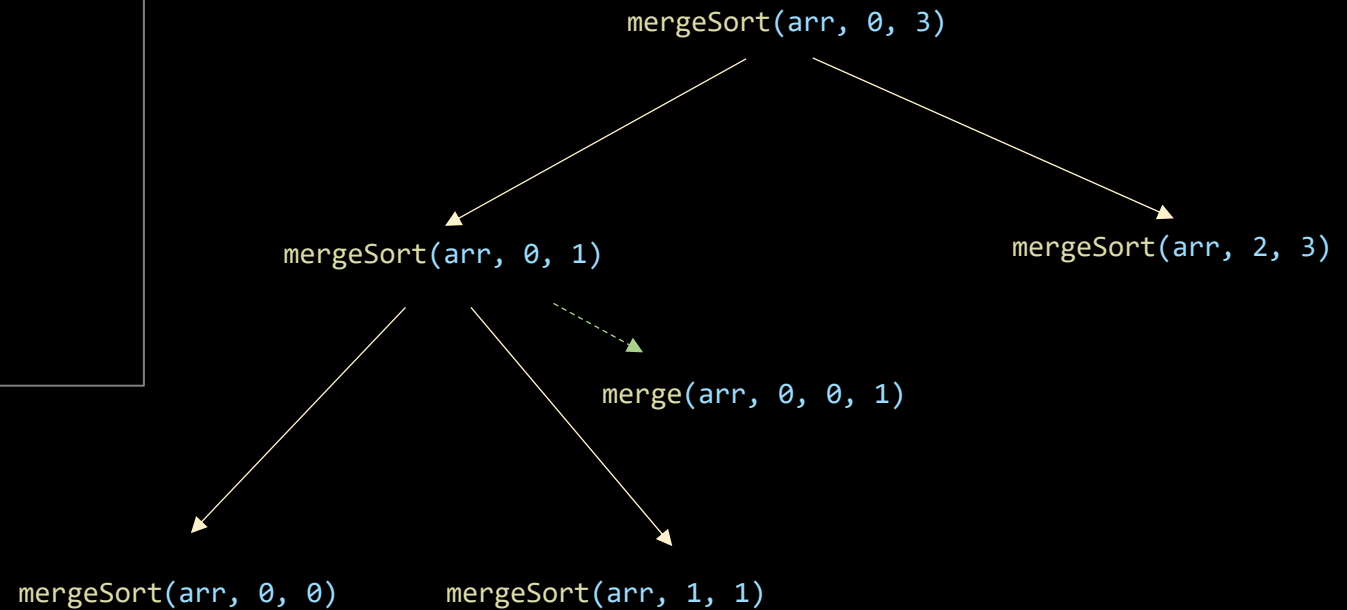


Merge Sort Code

```
1. void mergeSort(int arr[], int left, int right)
2. {
3.     if (left < right)
4.     {
5.         int mid = left + (right - left) / 2;
6.         mergeSort(arr, left, mid);
7.         mergeSort(arr, mid + 1, right);
8.
9.         // Merge the sorted subarrays
10.        merge(arr, left, mid, right);
11.    }
12. }
```

| arr | | | |
|-----|---|---|---|
| 15 | 2 | 7 | 0 |
| 0 | 1 | 2 | 3 |

| arr | | | |
|-----|----|---|---|
| 2 | 15 | 7 | 0 |
| 0 | 1 | 2 | 3 |



Merge Sort Code

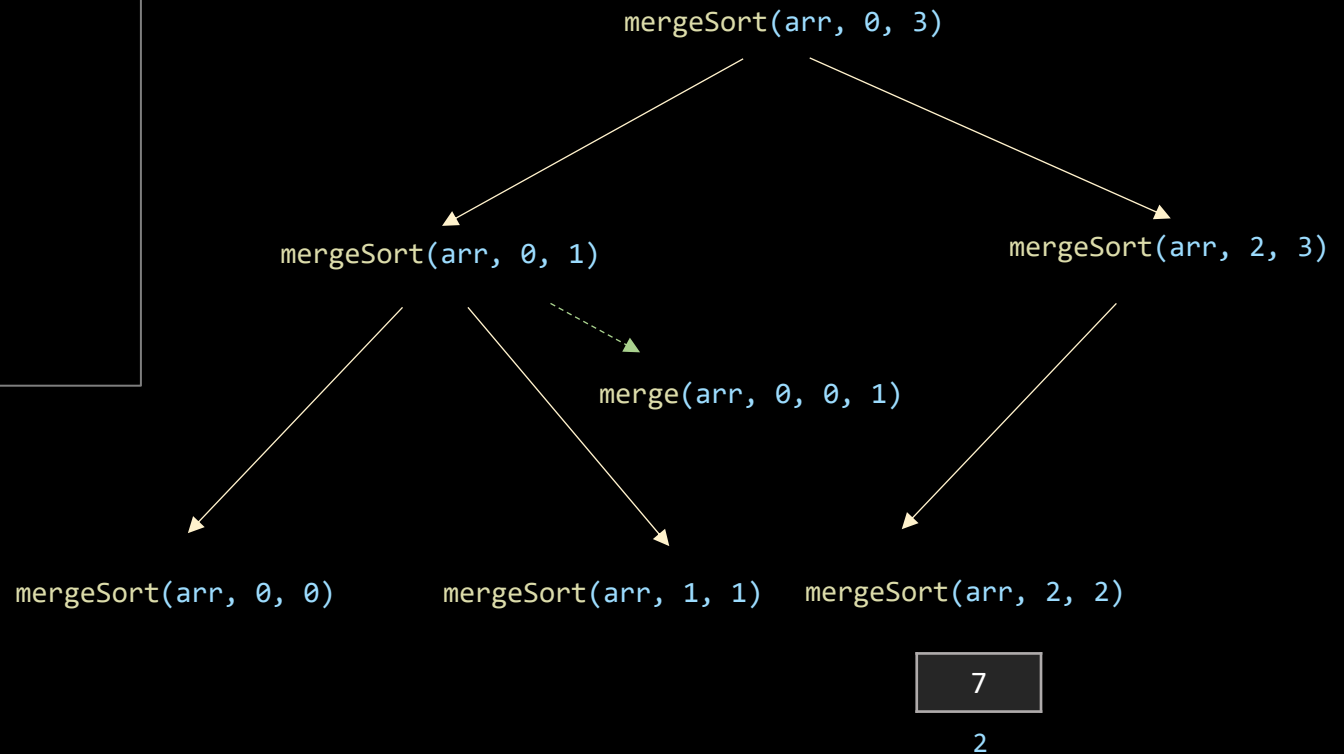
```
1. void mergeSort(int arr[], int left, int right)
2. {
3.     if (left < right)
4.     {
5.         int mid = left + (right - left) / 2;
6.         mergeSort(arr, left, mid);
7.         mergeSort(arr, mid + 1, right);
8.
9.         // Merge the sorted subarrays
10.        merge(arr, left, mid, right);
11.    }
12. }
```

arr

| | | | |
|---|----|---|---|
| 2 | 15 | 7 | 0 |
| 0 | 1 | 2 | 3 |

arr

| | | | |
|----|---|---|---|
| 15 | 2 | 7 | 0 |
| 0 | 1 | 2 | 3 |



Merge Sort Code

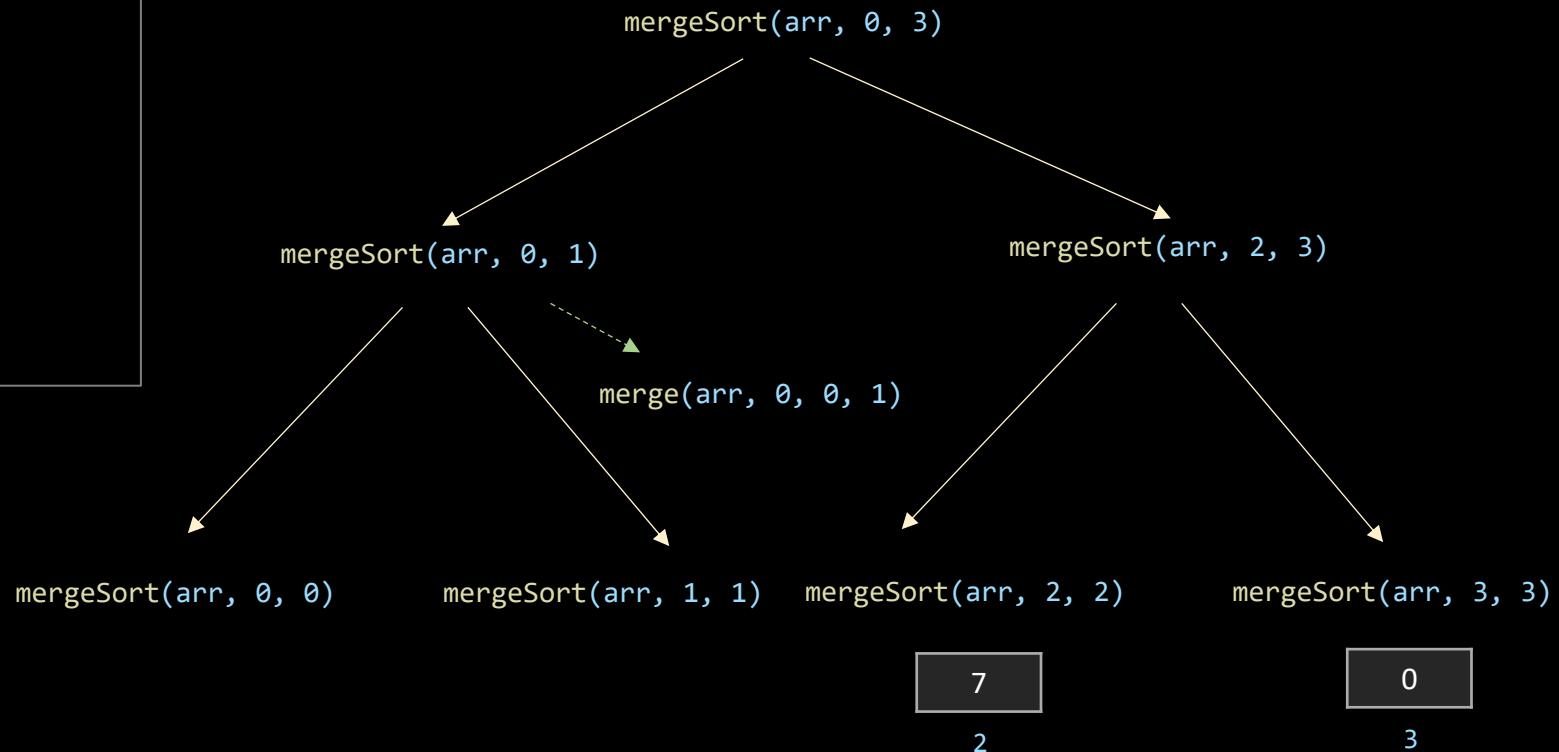
```
1. void mergeSort(int arr[], int left, int right)
2. {
3.     if (left < right)
4.     {
5.         int mid = left + (right - left) / 2;
6.         mergeSort(arr, left, mid);
7.         mergeSort(arr, mid + 1, right);
8.
9.         // Merge the sorted subarrays
10.        merge(arr, left, mid, right);
11.    }
12. }
```

arr

| | | | |
|---|----|---|---|
| 2 | 15 | 7 | 0 |
| 0 | 1 | 2 | 3 |

arr

| | | | |
|----|---|---|---|
| 15 | 2 | 7 | 0 |
| 0 | 1 | 2 | 3 |



Merge Sort Code

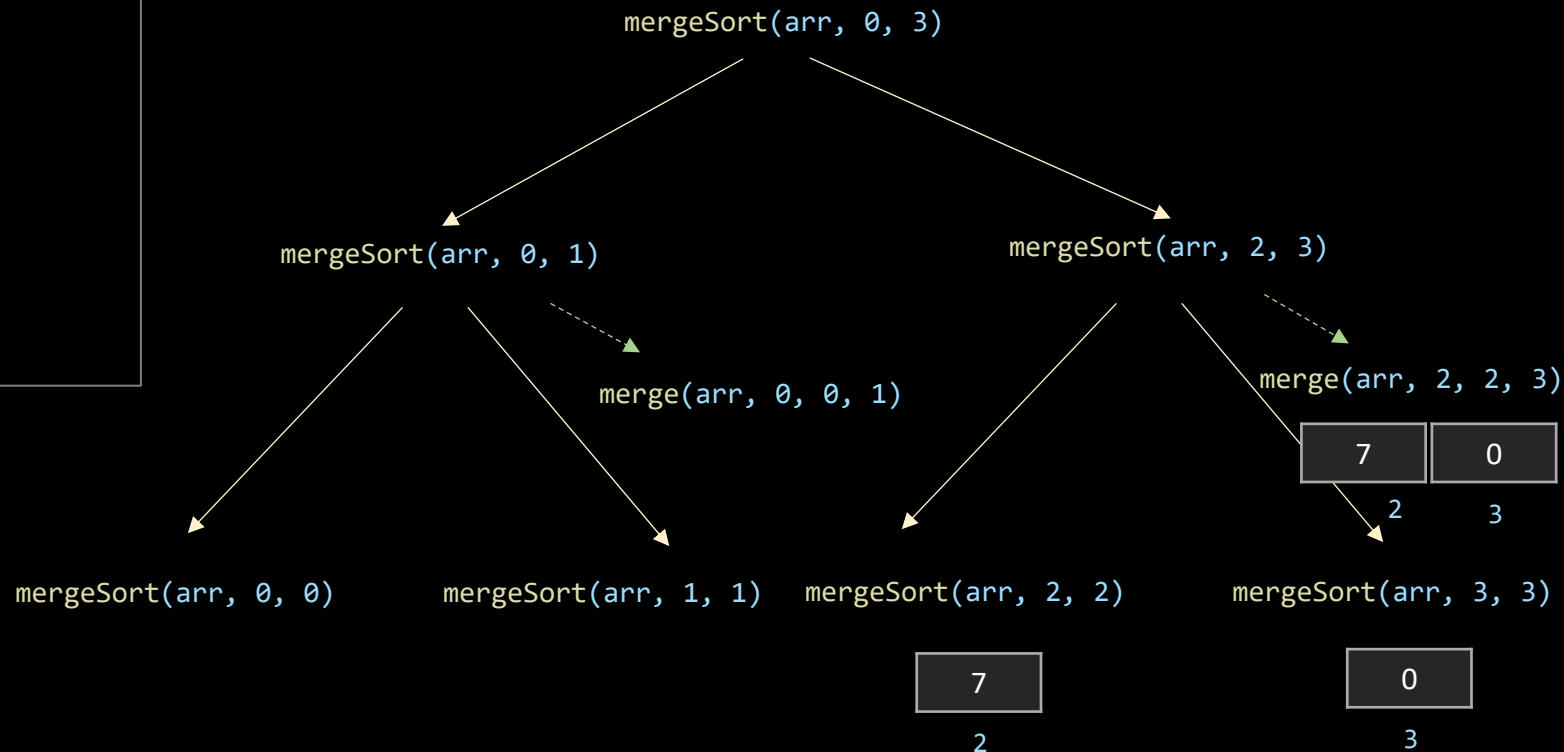
```
1. void mergeSort(int arr[], int left, int right)
2. {
3.     if (left < right)
4.     {
5.         int mid = left + (right - left) / 2;
6.         mergeSort(arr, left, mid);
7.         mergeSort(arr, mid + 1, right);
8.
9.         // Merge the sorted subarrays
10.        merge(arr, left, mid, right);
11.    }
12. }
```

arr

| | | | |
|---|----|---|---|
| 2 | 15 | 7 | 0 |
| 0 | 1 | 2 | 3 |

arr

| | | | |
|----|---|---|---|
| 15 | 2 | 7 | 0 |
| 0 | 1 | 2 | 3 |



Merge Sort Code

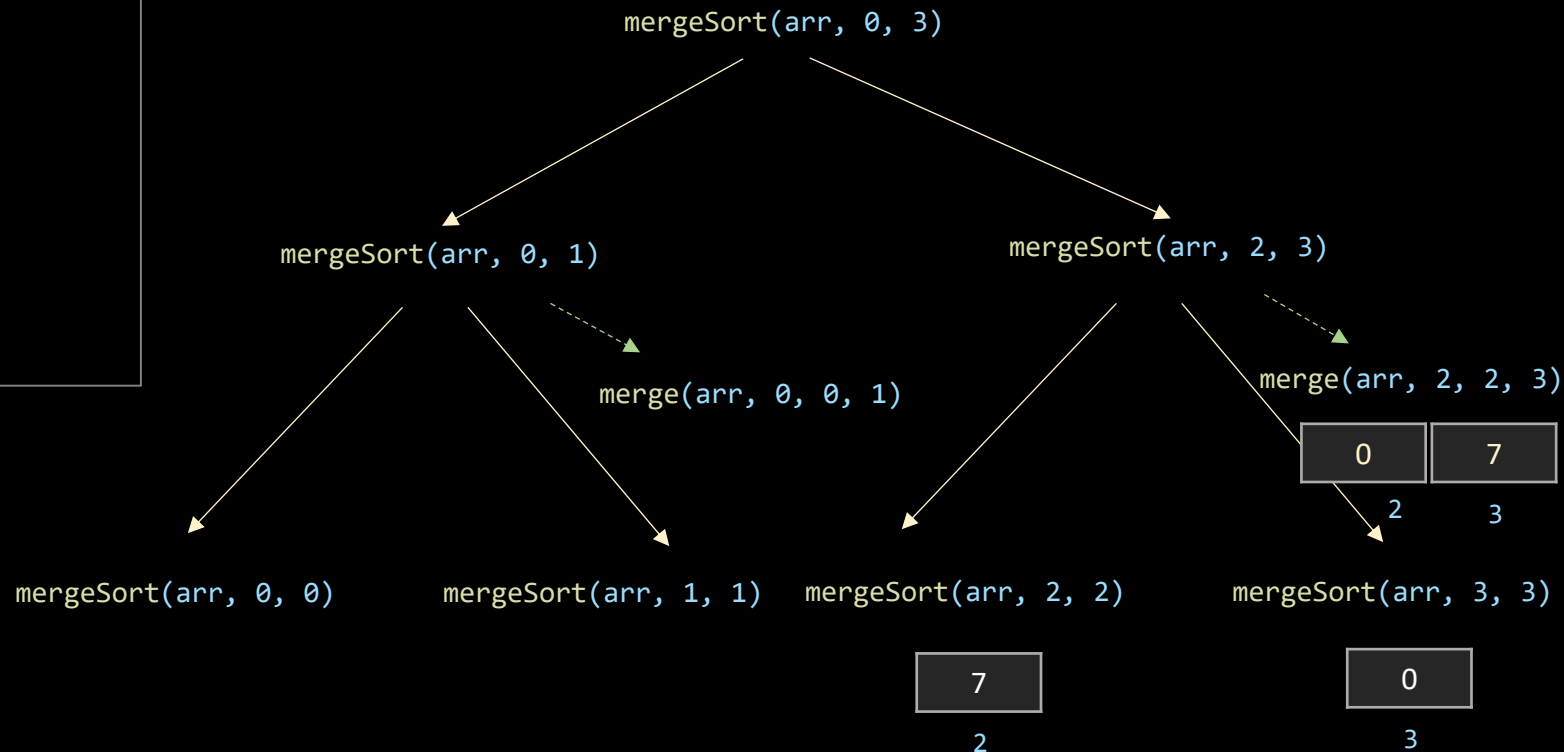
```
1. void mergeSort(int arr[], int left, int right)
2. {
3.     if (left < right)
4.     {
5.         int mid = left + (right - left) / 2;
6.         mergeSort(arr, left, mid);
7.         mergeSort(arr, mid + 1, right);
8.
9.         // Merge the sorted subarrays
10.        merge(arr, left, mid, right);
11.    }
12. }
```

arr

| | | | |
|---|----|---|---|
| 2 | 15 | 0 | 7 |
| 0 | 1 | 2 | 3 |

arr

| | | | |
|----|---|---|---|
| 15 | 2 | 7 | 0 |
| 0 | 1 | 2 | 3 |



Merge Sort Code

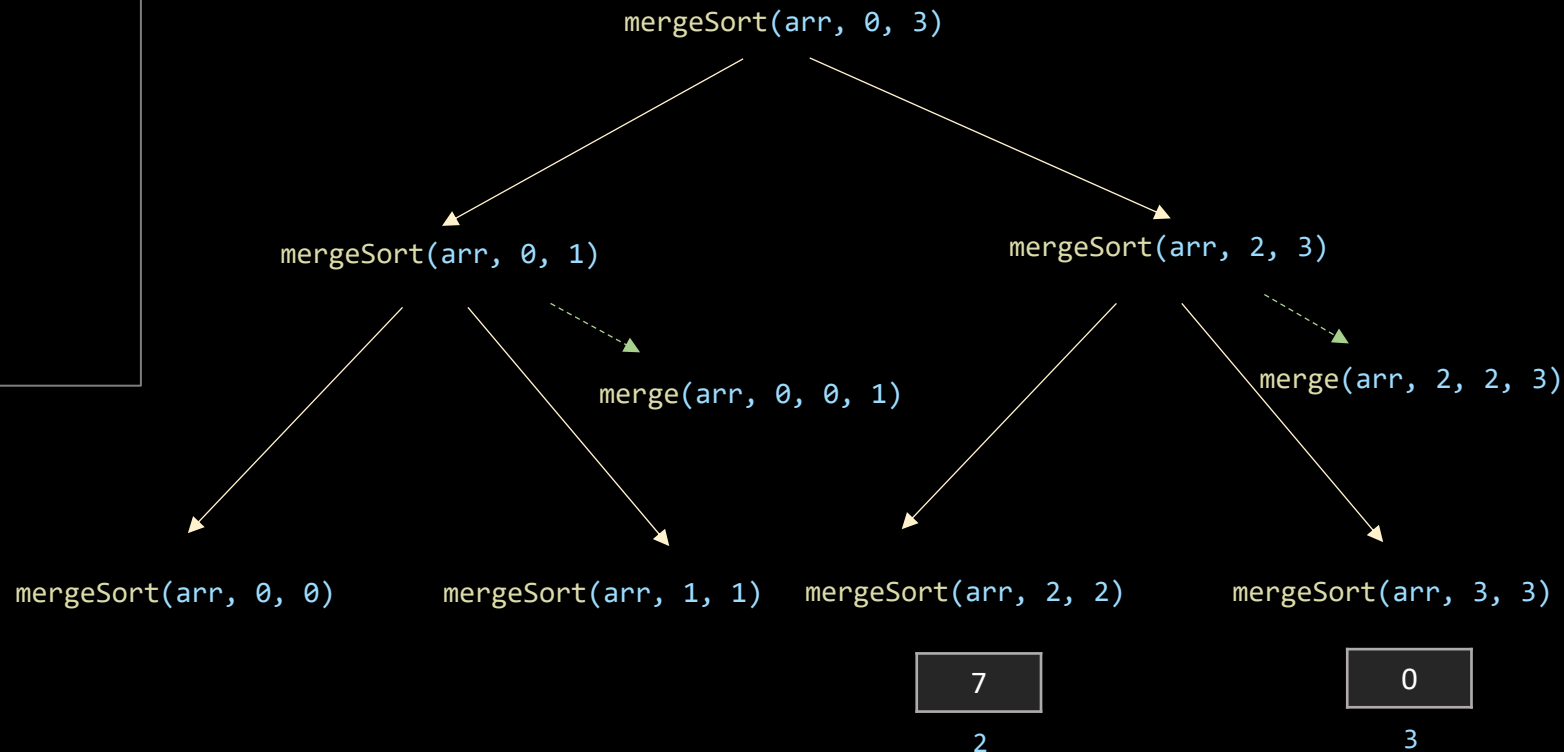
```
1. void mergeSort(int arr[], int left, int right)
2. {
3.     if (left < right)
4.     {
5.         int mid = left + (right - left) / 2;
6.         mergeSort(arr, left, mid);
7.         mergeSort(arr, mid + 1, right);
8.
9.         // Merge the sorted subarrays
10.        merge(arr, left, mid, right);
11.    }
12. }
```

arr

| | | | |
|---|----|---|---|
| 2 | 15 | 0 | 7 |
| 0 | 1 | 2 | 3 |

arr

| | | | |
|----|---|---|---|
| 15 | 2 | 7 | 0 |
| 0 | 1 | 2 | 3 |



Merge Sort Code

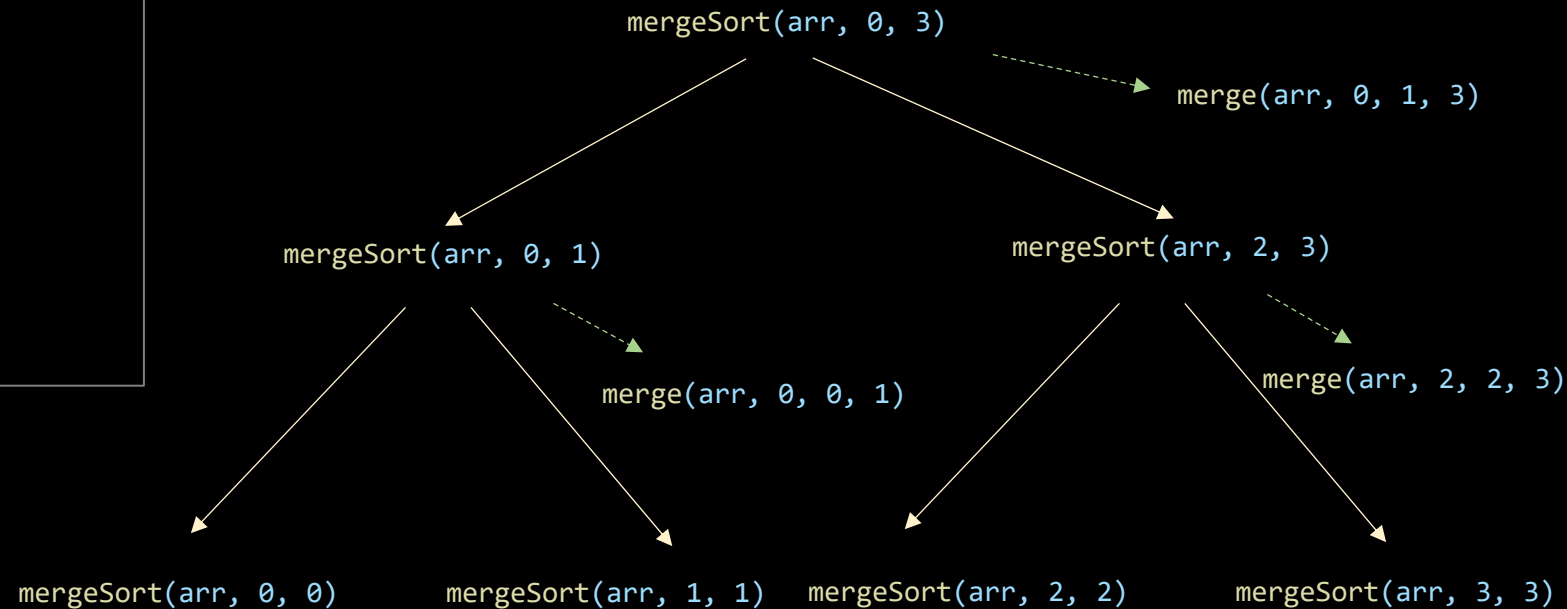
```
1. void mergeSort(int arr[], int left, int right)
2. {
3.     if (left < right)
4.     {
5.         int mid = left + (right - left) / 2;
6.         mergeSort(arr, left, mid);
7.         mergeSort(arr, mid + 1, right);
8.
9.         // Merge the sorted subarrays
10.        merge(arr, left, mid, right);
11.    }
12. }
```

arr

| | | | |
|---|----|---|---|
| 2 | 15 | 0 | 7 |
| 0 | 1 | 2 | 3 |

arr

| | | | |
|----|---|---|---|
| 15 | 2 | 7 | 0 |
| 0 | 1 | 2 | 3 |



Merge Sort Code

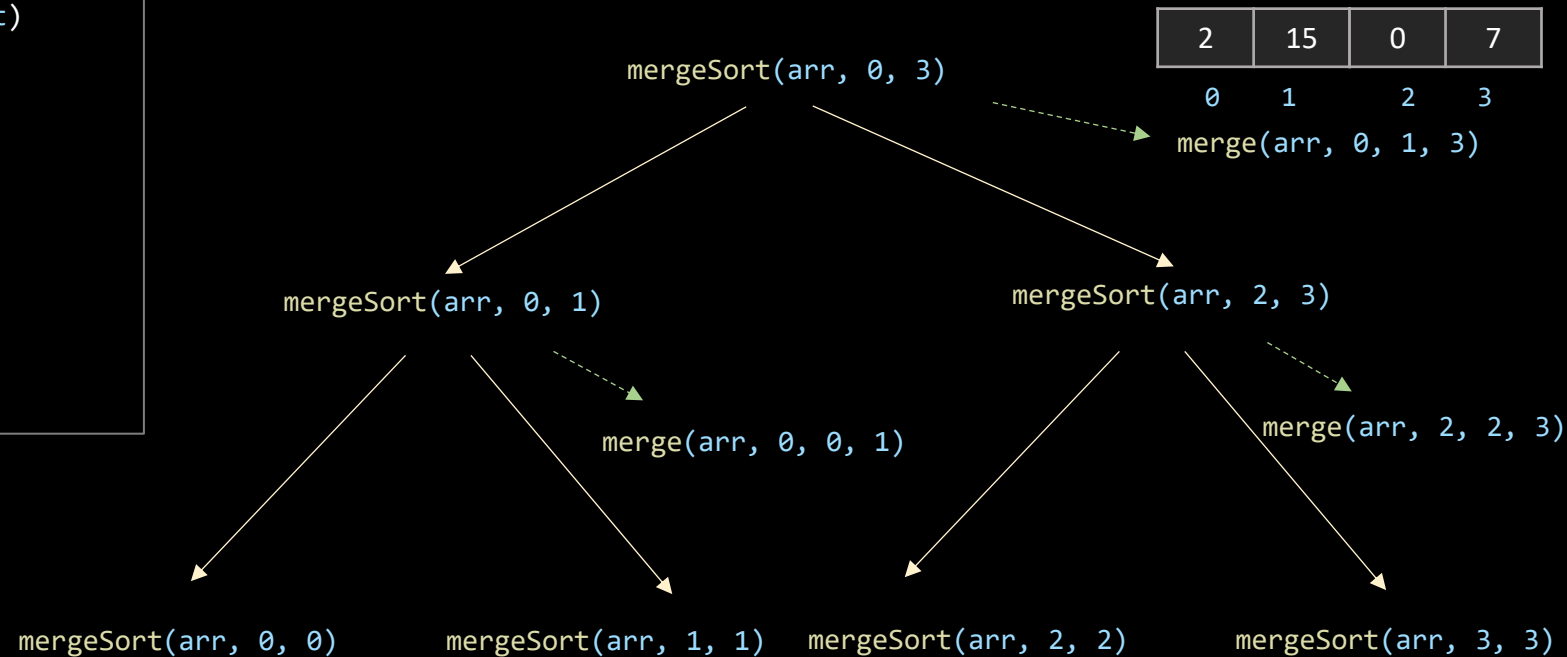
```
1. void mergeSort(int arr[], int left, int right)
2. {
3.     if (left < right)
4.     {
5.         int mid = left + (right - left) / 2;
6.         mergeSort(arr, left, mid);
7.         mergeSort(arr, mid + 1, right);
8.
9.         // Merge the sorted subarrays
10.        merge(arr, left, mid, right);
11.    }
12. }
```

arr

| | | | |
|---|----|---|---|
| 2 | 15 | 0 | 7 |
| 0 | 1 | 2 | 3 |

arr

| | | | |
|----|---|---|---|
| 15 | 2 | 7 | 0 |
| 0 | 1 | 2 | 3 |



Merge Sort Code

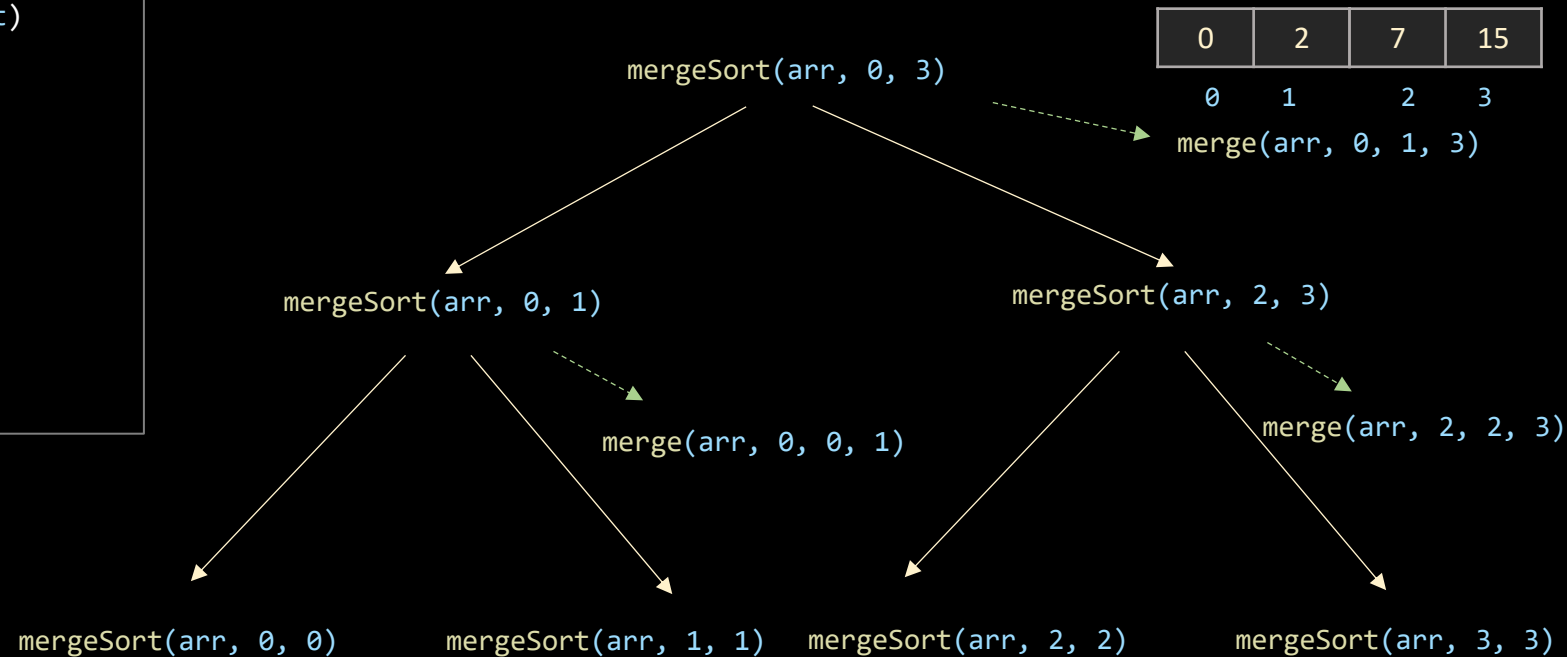
```
1. void mergeSort(int arr[], int left, int right)
2. {
3.     if (left < right)
4.     {
5.         int mid = left + (right - left) / 2;
6.         mergeSort(arr, left, mid);
7.         mergeSort(arr, mid + 1, right);
8.
9.         // Merge the sorted subarrays
10.        merge(arr, left, mid, right);
11.    }
12. }
```

arr

| | | | |
|---|---|---|----|
| 0 | 2 | 7 | 15 |
| 0 | 1 | 2 | 3 |

arr

| | | | |
|----|---|---|---|
| 15 | 2 | 7 | 0 |
| 0 | 1 | 2 | 3 |

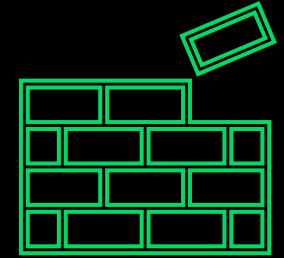


Merge Sort Time Complexity

| | Merge Sort |
|--------------|---------------------|
| Worst Case | $O(n \cdot \log n)$ |
| Average Case | $O(n \cdot \log n)$ |
| Best Case | $O(n \cdot \log n)$ |
| Space | $O(n)$ |

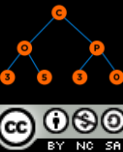
Quick Sort

Quick Sort

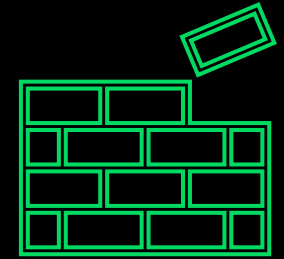


Premise:

- Quicksort rearranges the array into two parts – called **partitioning**
- A pivot is selected, and the following is executed:
 - All the elements in the left subarray are less than or equal to the pivot
 - All the elements in the right subarray are larger than the pivot
 -
 -
- The process is repeated until the array is sorted



Quick Sort

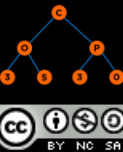


Algorithm for Quicksort

if first < last then

- Partition the elements in the subarray first . . . last so that the pivot value is in its correct place (subscript pivotIndex)
- Recursively apply quicksort to the subarray first . . . pivotIndex - 1
- Recursively apply quicksort to the subarray pivotIndex + 1 . . . last

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|



Quick Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Algorithm for partition Method

1. Define the pivot value as the contents of `table[first]`.
2. Initialize `up` to `first` and `down` to `last`.
3. **do**
4. Increment `up` until `up` selects the first element greater than the pivot value or `up` has reached `last`.
5. Decrement `down` until `down` selects the first element less than or equal to the pivot value or `down` has reached `first`.
6. **if** `up < down` **then**
7. Exchange `table[up]` and `table[down]`.
8. **while** `up` is to the left of `down`
9. Exchange `table[first]` and `table[down]`.
10. Return the value of `down` to `pivIndex`.

Quick Sort

Example:

Initial array

| | | | | | | | |
|----|---|---|---|------|---|---|---|
| Up | | | | Down | | | |
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |

Pivot

Algorithm for partition Method

1. Define the pivot value as the contents of `table[first]`.
2. Initialize `up` to `first` and `down` to `last`.
3. **do**
4. Increment `up` until `up` selects the first element greater than the pivot value or `up` has reached `last`.
5. Decrement `down` until `down` selects the first element less than or equal to the pivot value or `down` has reached `first`.
6. **if** `up < down` **then**
7. Exchange `table[up]` and `table[down]`.
8. **while** `up` is to the left of `down`
9. Exchange `table[first]` and `table[down]`.
10. Return the value of `down` to `pivIndex`.

Quick Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 9 | 3 | 2 | 8 | 6 | 5 |
|---|---|---|---|---|---|---|---|

Up

Down

Pivot

Algorithm for partition Method

1. Define the pivot value as the contents of `table[first]`.
2. Initialize `up` to `first` and `down` to `last`.
3. **do**
4. Increment `up` until `up` selects the first element greater than the pivot value or `up` has reached `last`.
5. Decrement `down` until `down` selects the first element less than or equal to the pivot value or `down` has reached `first`.
6. **if** `up < down` **then**
7. Exchange `table[up]` and `table[down]`.
8. **while** `up` is to the left of `down`
9. Exchange `table[first]` and `table[down]`.
10. Return the value of `down` to `pivIndex`.

Quick Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 5 | 3 | 2 | 8 | 6 | 9 |
|---|---|---|---|---|---|---|---|

Up

Down

Pivot

Algorithm for partition Method

1. Define the pivot value as the contents of `table[first]`.
2. Initialize `up` to `first` and `down` to `last`.
3. **do**
4. Increment `up` until `up` selects the first element greater than the pivot value or `up` has reached `last`.
5. Decrement `down` until `down` selects the first element less than or equal to the pivot value or `down` has reached `first`.
6. **if** `up < down` **then**
7. Exchange `table[up]` and `table[down]`.
8. **while** `up` is to the left of `down`
9. Exchange `table[first]` and `table[down]`.
10. Return the value of `down` to `pivIndex`.

Quick Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 5 | 3 | 2 | 8 | 6 | 9 |
|---|---|---|---|---|---|---|---|

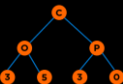
Pivot

Up

Down

Algorithm for partition Method

1. Define the pivot value as the contents of `table[first]`.
2. Initialize `up` to `first` and `down` to `last`.
3. **do**
4. Increment `up` until `up` selects the first element greater than the pivot value or `up` has reached `last`.
5. Decrement `down` until `down` selects the first element less than or equal to the pivot value or `down` has reached `first`.
6. **if** `up < down` **then**
7. Exchange `table[up]` and `table[down]`.
8. **while** `up` is to the left of `down`
9. Exchange `table[first]` and `table[down]`.
10. Return the value of `down` to `pivIndex`.



Quick Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 5 | 3 | 2 | 6 | 8 | 9 |
|---|---|---|---|---|---|---|---|

Pivot

Up

Down

Algorithm for partition Method

1. Define the pivot value as the contents of `table[first]`.
2. Initialize `up` to `first` and `down` to `last`.
3. **do**
4. Increment `up` until `up` selects the first element greater than the pivot value or `up` has reached `last`.
5. Decrement `down` until `down` selects the first element less than or equal to the pivot value or `down` has reached `first`.
6. **if** `up < down` **then**
7. Exchange `table[up]` and `table[down]`.
8. **while** `up` is to the left of `down`
9. Exchange `table[first]` and `table[down]`.
10. Return the value of `down` to `pivIndex`.

Quick Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 4 | 5 | 3 | 2 | 6 | 8 | 9 |
|---|---|---|---|---|---|---|---|

Pivot

Down

Up

Algorithm for partition Method

1. Define the pivot value as the contents of `table[first]`.
2. Initialize `up` to `first` and `down` to `last`.
3. **do**
4. Increment `up` until `up` selects the first element greater than the pivot value or `up` has reached `last`.
5. Decrement `down` until `down` selects the first element less than or equal to the pivot value or `down` has reached `first`.
6. **if** `up < down` **then**
7. Exchange `table[up]` and `table[down]`.
8. **while** `up` is to the left of `down`
9. Exchange `table[first]` and `table[down]`.
10. Return the value of `down` to `pivIndex`.

Quick Sort

Example:

Initial array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 4 | 5 | 3 | 2 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|

Pivot

Down

Up

Algorithm for partition Method

1. Define the pivot value as the contents of `table[first]`.
2. Initialize `up` to `first` and `down` to `last`.
3. **do**
4. Increment `up` until `up` selects the first element greater than the pivot value or `up` has reached `last`.
5. Decrement `down` until `down` selects the first element less than or equal to the pivot value or `down` has reached `first`.
6. **if** `up < down` **then**
7. Exchange `table[up]` and `table[down]`.
8. **while** `up` is to the left of `down`
9. Exchange `table[first]` and `table[down]`.
10. Return the value of `down` to `pivIndex`.

Quick Sort

Example:

Initial array

| | | | | | | | |
|------|---|---|----|---|---|---|---|
| Down | | | Up | | | | |
| 6 | 4 | 5 | 3 | 2 | 7 | 8 | 9 |

Pivot

Algorithm for partition Method

1. Define the pivot value as the contents of `table[first]`.
2. Initialize `up` to `first` and `down` to `last`.
3. **do**
4. Increment `up` until `up` selects the first element greater than the pivot value or `up` has reached `last`.
5. Decrement `down` until `down` selects the first element less than or equal to the pivot value or `down` has reached `first`.
6. **if** `up < down` **then**
7. Exchange `table[up]` and `table[down]`.
8. **while** `up` is to the left of `down`
9. Exchange `table[first]` and `table[down]`.
10. Return the value of `down` to `pivIndex`.

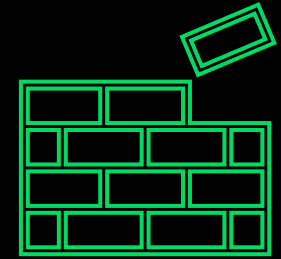
Quick Sort Code

```
1. void quickSort(int array[], int low, int high)
2. {
3.     if (low < high)
4.     {
5.         int pivot = partition(array, low, high);
6.         quickSort(array, low, pivot - 1);
7.         quickSort(array, pivot + 1, high);
8.     }
9. }
```

```
36. int main()
37. {
38.     int data[] = {15, 0, 5, 6};
39.     int n = sizeof(data) / sizeof(data[0]);
40.     quickSort(data, 0, n - 1);
41.     return 0;
42. }
```

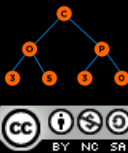
```
10. int partition(int array[], int low, int high)
11. {
12.     // Select the pivot element
13.     int pivot = array[low];
14.     int up = low, down = high;
15.
16.     while(up < down)
17.     {
18.         for (int j = up; j < high; j++)
19.         {
20.             if(array[j] > pivot)
21.                 break;
22.             up++;
23.         }
24.         for (int j = high; j > low; j--)
25.         {
26.             if(array[j] < pivot)
27.                 break;
28.             down--;
29.         }
30.         if(up < down)
31.             swap(&array[up], &array[down]);
32.     }
33.     swap(&array[low], &array[down]);
34.     return down;
35. }
```

Quick Sort



Time Complexity:

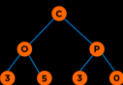
- If the pivot value is a random value selected from the current subarray,
 - then statistically half of the items in the subarray will be less than the pivot and half will be greater
 - thus there will be $\log n$ levels of recursion
- Partitioning requires n moves
- Total time: $O(n \log n)$ on average
- A quicksort will give very poor behavior if, each time the array is partitioned, a subarray is empty. In that case, the sort will be $O(n^2)$. Under these circumstances, the overhead of recursive calls and the extra run-time stack storage required by these calls makes this version of quicksort a poor performer relative to the quadratic sorts



Quick Sort Time Complexity

| | Merge Sort | Quick Sort |
|---------------------|---------------------|---------------------|
| Worst Case | $O(n \cdot \log n)$ | $O(n^2)$ |
| Average Case | $O(n \cdot \log n)$ | $O(n \cdot \log n)$ |
| Best Case | $O(n \cdot \log n)$ | $O(n \cdot \log n)$ |
| Space | $O(n)$ | $O(\log n)^*$ |

* Recursion Stack



Other Sorts

- **Sleep sort**
- **Counting sort**
- **Tim Sort**
- **Radix Sort**
- **Bucket Sort**

Resources

- <https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html>
- <https://www.programiz.com/dsa>
- <https://www.youtube.com/user/AlgoRythmics/videos>
- <https://www.toptal.com/developers/sorting-algorithms>

Questions

Mentimeter

Menti.com

2728 5922

