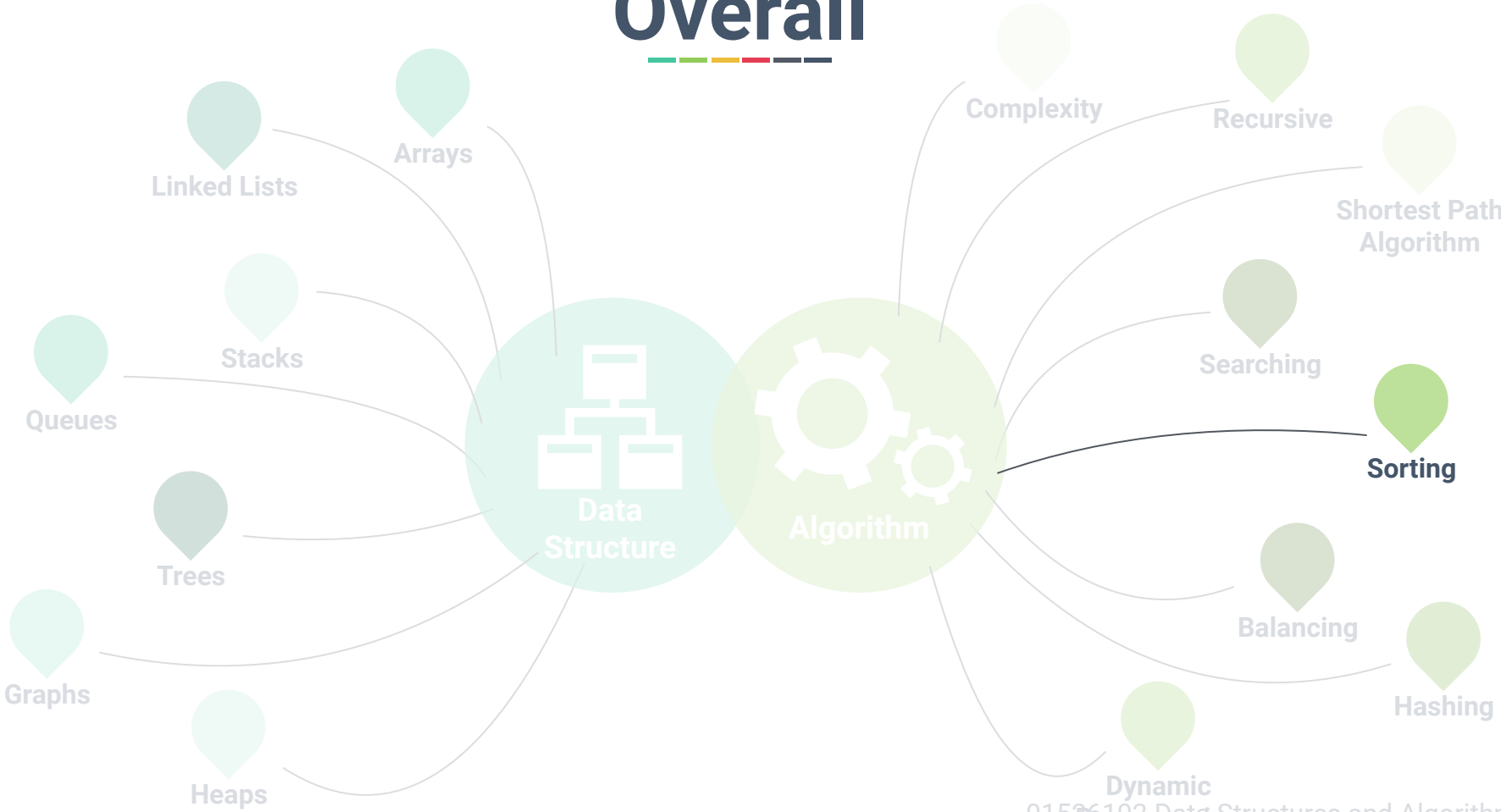
Chapter 10: Sorting

Dr. Sirasit Lochanachit

Overall

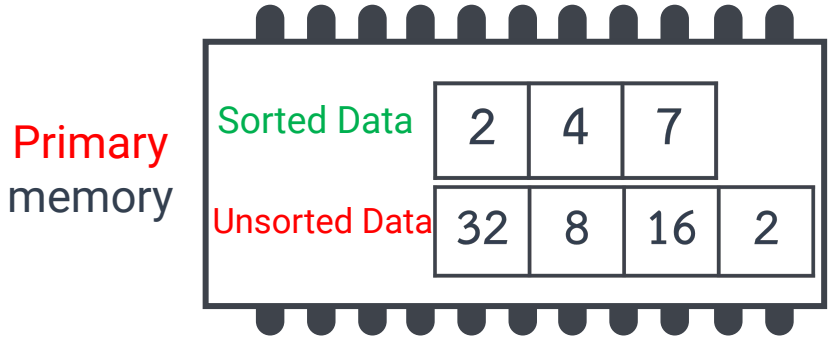

Linked Lists
Arrays
Stacks
Queues
Trees
Graphs
Heaps
Data Structure
Algorithm
Complexity
Recursive
Shortest Path Algorithm
Searching
Sorting
Balancing
Hashing
Dynamic Programming

Outline

Sorting

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- [Optional] Quick Sort

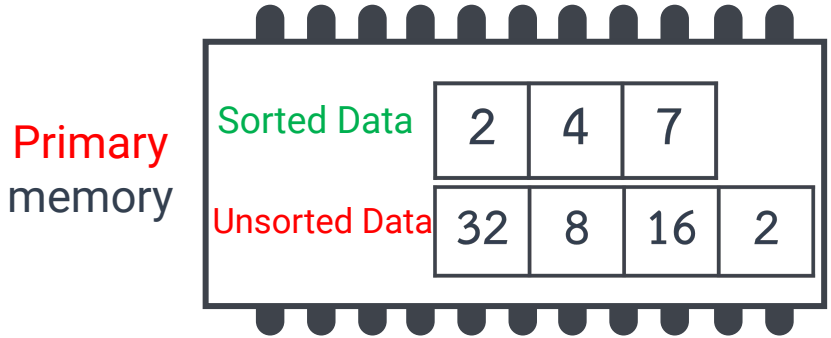
What is Sorting?



Sorting is the process of placing elements from a collection in some kind of order.

For instance, a list of words could be sorted by alphabet from a-z or z-a.

Sorting Operations

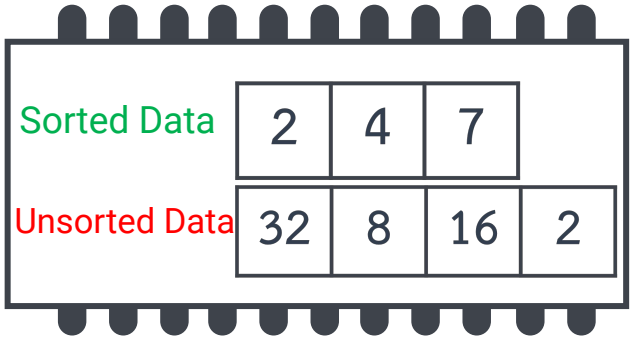


Generally, sorting has two operations:

1. **Compare** between two values to determine which is smaller (or larger).
    - The **total number of comparisons** is crucial to measure sorting efficiency.

Sorting Operations

Primary memory

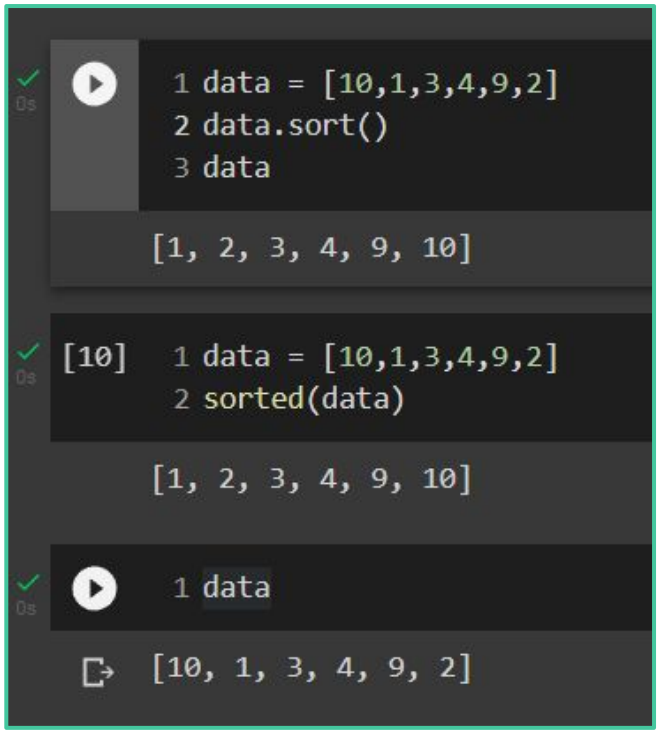



```
temp = alist[i]
alist[i] = alist[j]
alist[j] = temp
```

Generally, sorting has two operations:

2. **Exchange** two values.
   - Exchange is a costly operation.
   - The **total number of exchanges** is also important for evaluating efficiency of the algorithm.

Python Built-in Sort Method



Bubble Sort

Bubble sort is a sorting algorithm which makes multiple iterations through a given list of unsorted elements.

- It compares each pair of adjacent elements and exchanges those that are out of order.

Ref : https://www.faceprep.in/bubble-sort-in-c/

Bubble Sort

78	56	32	45	8	23	19

- If there are *n* items in the list, then there are *n* - 1 pairs that needs to be compared on the first round.
- At the start of the 2nd round, there are *n* - 1 items left to sort which means there will be *n* - 2 pairs.
- Therefore, the total number of rounds is *n* - 1.

Great example: https://youtu.be/lyZQPjUT5B4?t=54

Bubble Sort Example

Round 1

Right-to-left

n - 1 pairs

78	56	32	45	8	23	19	Exchange
78	56	32	45	8	19	23	No Exchange
78	56	32	45	8	19	23	Exchange
78	56	32	8	45	19	23	Exchange
78	56	8	32	45	19	23	Exchange
78	8	56	32	45	19	23	Exchange
8	78	56	32	45	19	23	

Bubble Sort Example

Round 2

Right-to-left

n - 2 pairs

8	78	56	32	45	19	23	No Exchange
8	78	56	32	45	19	23	Exchange
8	78	56	32	19	45	23	Exchange
8	78	56	19	32	45	23	Exchange
8	78	19	56	32	45	23	Exchange

8	19	78	56	32	45	23

Bubble Sort Example

Round 3

Right-to-left

n - 3 pairs

8	19	78	56	32	45	23	Exchange
8	19	78	56	32	23	45	Exchange
8	19	78	56	23	32	45	Exchange
8	19	78	23	56	32	45	Exchange

8	19	23	78	56	32	45

Bubble Sort Example

Round 4

Right-to-left

n - 4 pairs

8	19	23	78	56	32	45	No Exchange
8	19	23	78	56	32	45	Exchange
8	19	23	78	32	56	45	Exchange

8	19	23	32	78	56	45

Bubble Sort Example

Round 5

Right-to-left

n - 5 pairs

8	19	23	32	78	56	45	Exchange
8	19	23	32	78	45	56	Exchange
8	19	23	32	45	78	56	

Bubble Sort Example

Round 6

Right-to-left

n - 6 pairs

8	19	23	32	45	78	56

Exchange

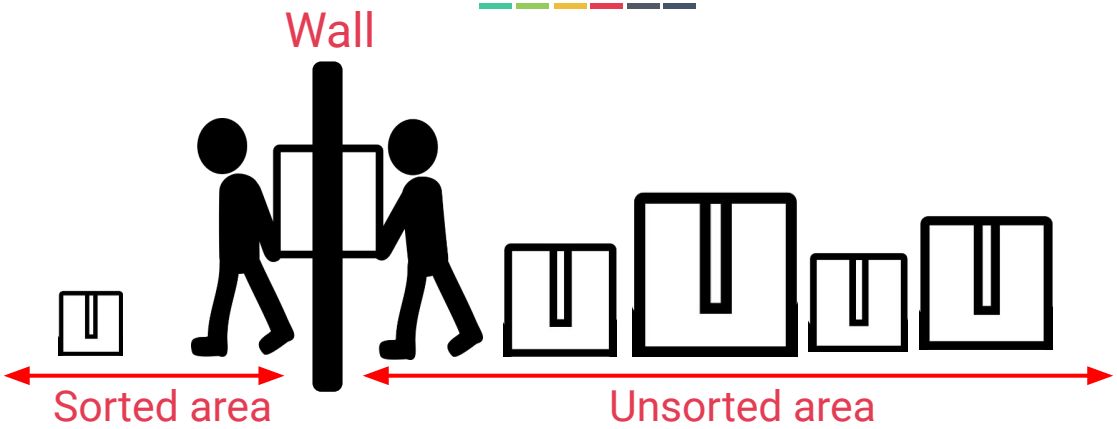
8	19	23	32	45	56	78

Bubble Sort Performance

Round	Number of comparisons to find the smallest/largest value
1	n - 1
2	n - 2
3	n - 3
...	...
n - 1	1

```
for round in range(0, n-1):
    for i in range(n-1, round, -1):
        if data[i-1] > data[i]:
            swap(data[i-1], data[i])
```

Selection Sort



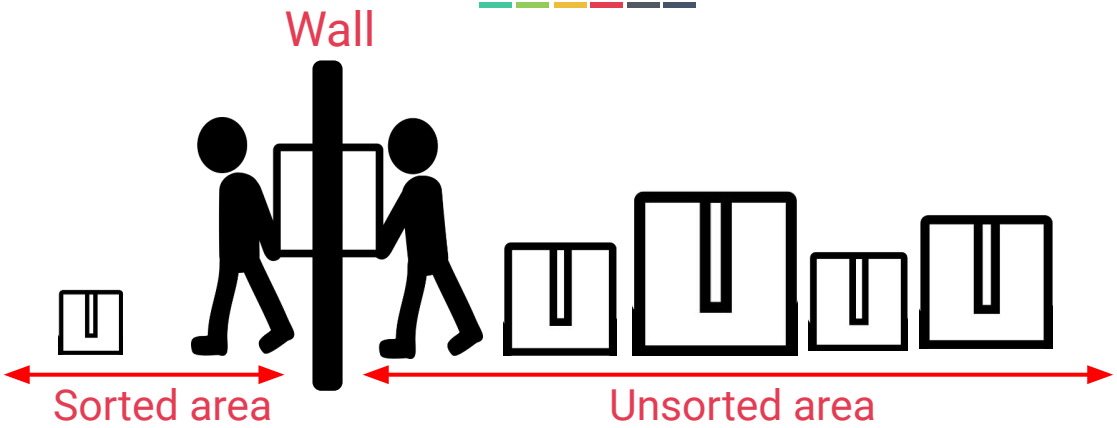
How to selection sort:

The list is divided into two sub-lists, **sorted** and **unsorted**, which are divided by an imaginary wall.

Given a list of unsorted data, it selects the smallest value and place it in a sorted list.

These steps are then repeated until all of the data are sorted.

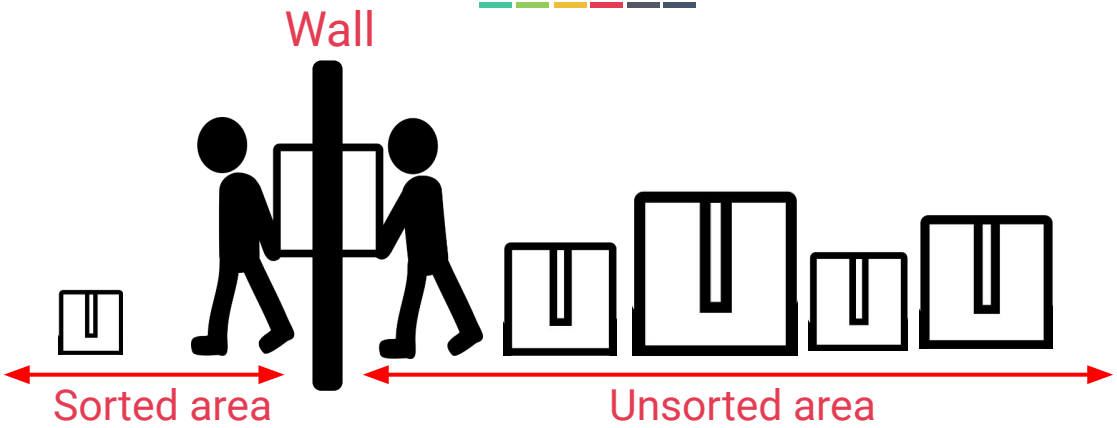
Selection Sort



In other words,

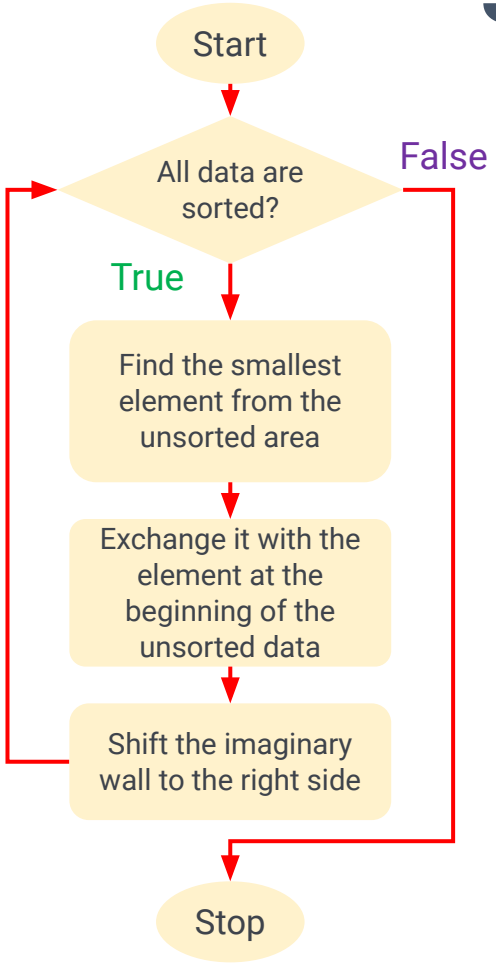
the smallest element from the unsorted sub-list are selected and exchange it with the element at the beginning of the unsorted data.

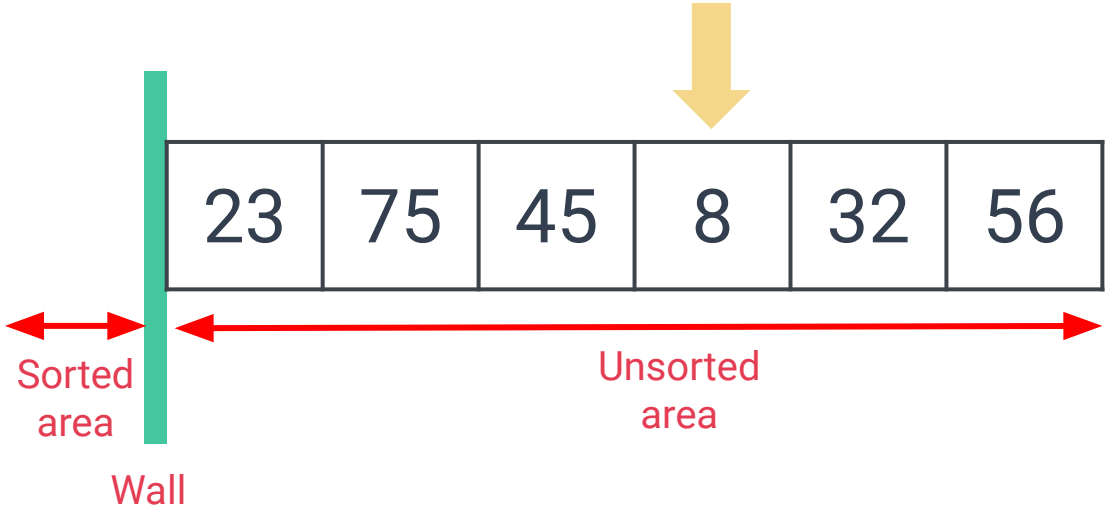
Selection Sort Real-life Demo



Select-sort with Gypsy folk dance

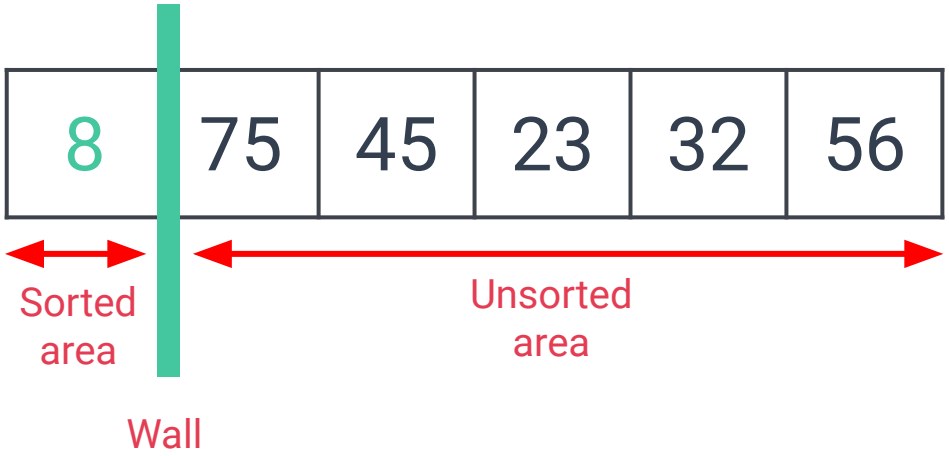
Selection Sort Example



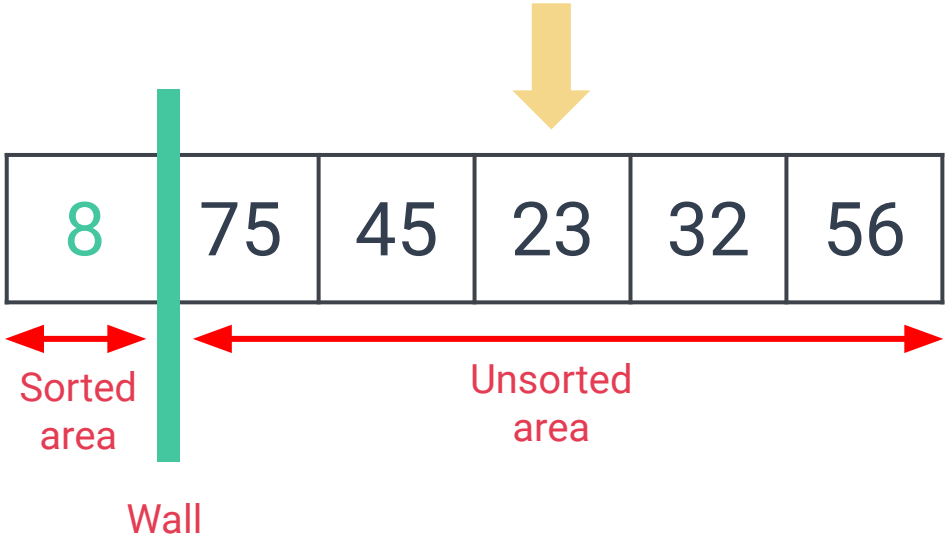


Selection Sort Example

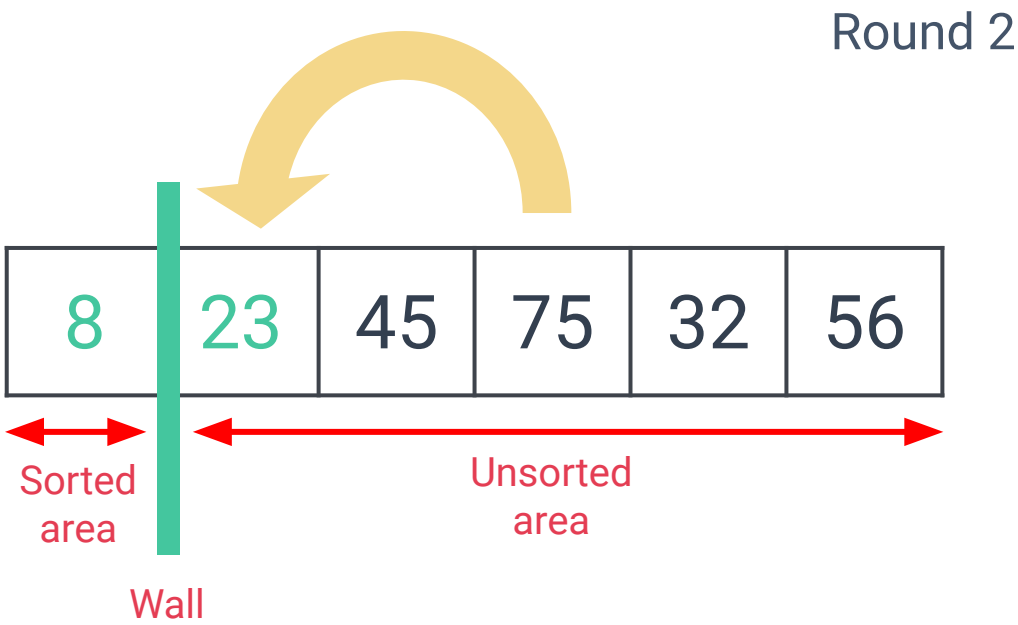
Selection Sort Example



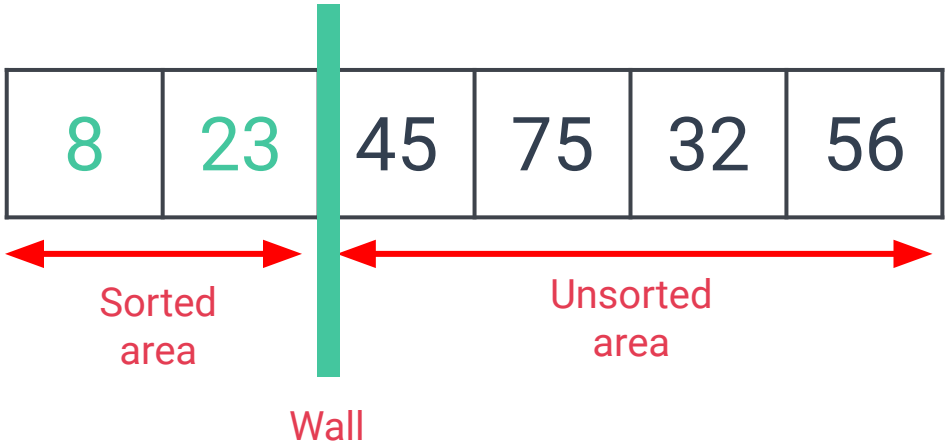
Selection Sort Example



Selection Sort Example

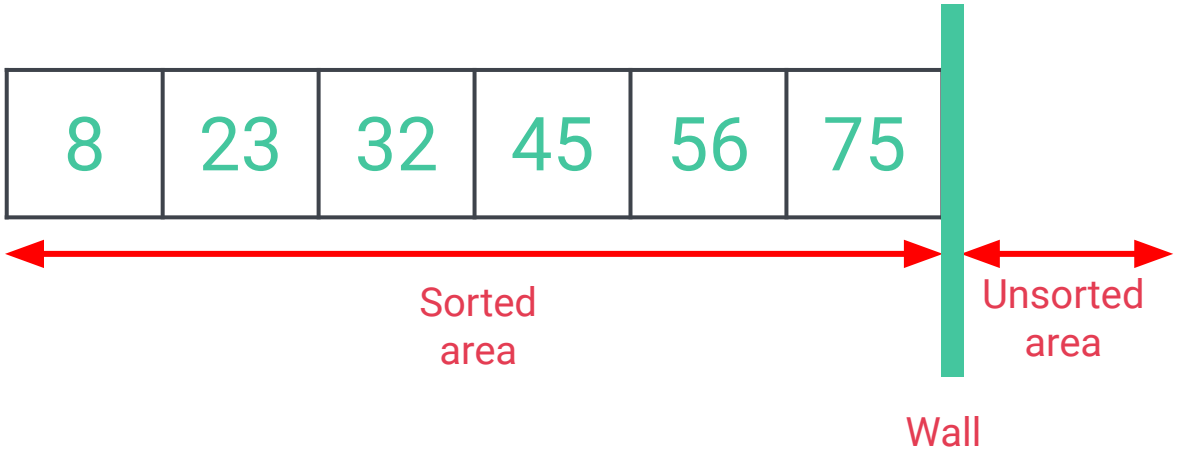


Selection Sort Example



Selection Sort Example

After Round 5



Selection Sort Performance

Round	Number of comparisons to find the smallest/largest value
1	n - 1
2	n - 2
3	n - 3
...	...
n - 1	1

```
for ??:
    minIndex = ?
    for ??:
        if ??
            Update minIndex
    Swap between the smallest item and the first item in unsorted area.
```

Insertion Sort

Similar to selection sort, a list is divided to two parts: **sorted** and **unsorted**.

In each round, the first element of the unsorted sublist is transferred to the sorted sublist by inserting it at the appropriate place.

A real example is sorting cards by card players. As they pick up each card, they insert it into the proper sequence in their hand.

Insertion Sort

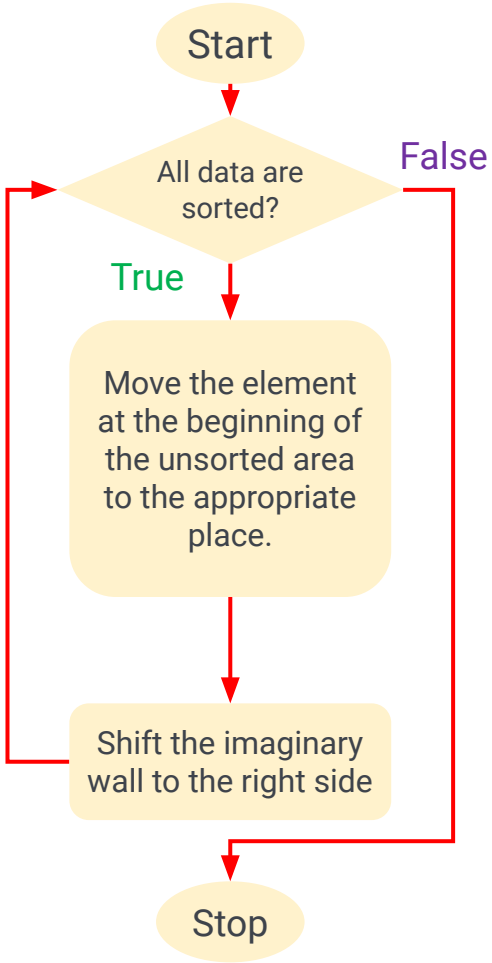
Where to insert?

- The selected item is checked against items in the sorted sublist.
- The items in the sorted sublist that have greater value are shifted to the right.
- When reach a smaller item or the start of the sublist, the selected item can be inserted.

Insertion Sort Real-life Demo

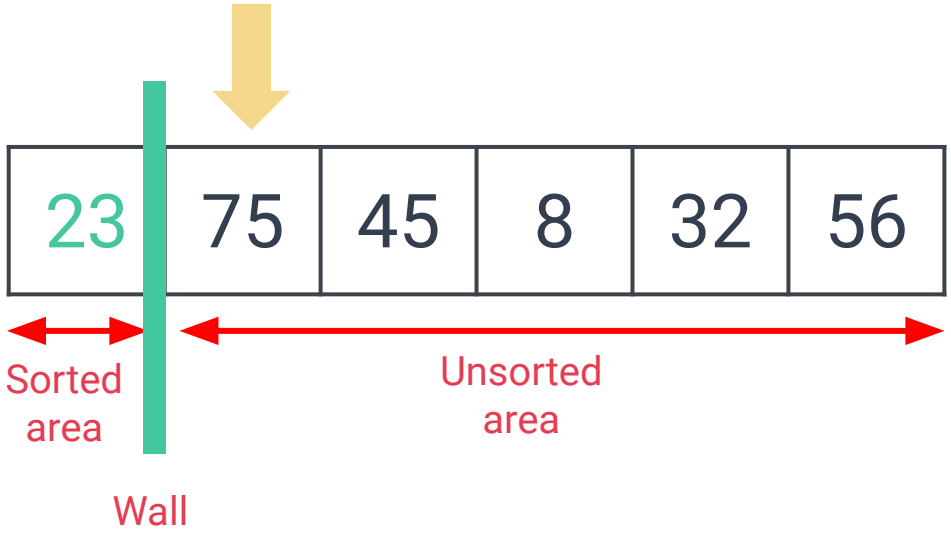
Insert-sort with Romanian folk dance

Insertion Sort Example



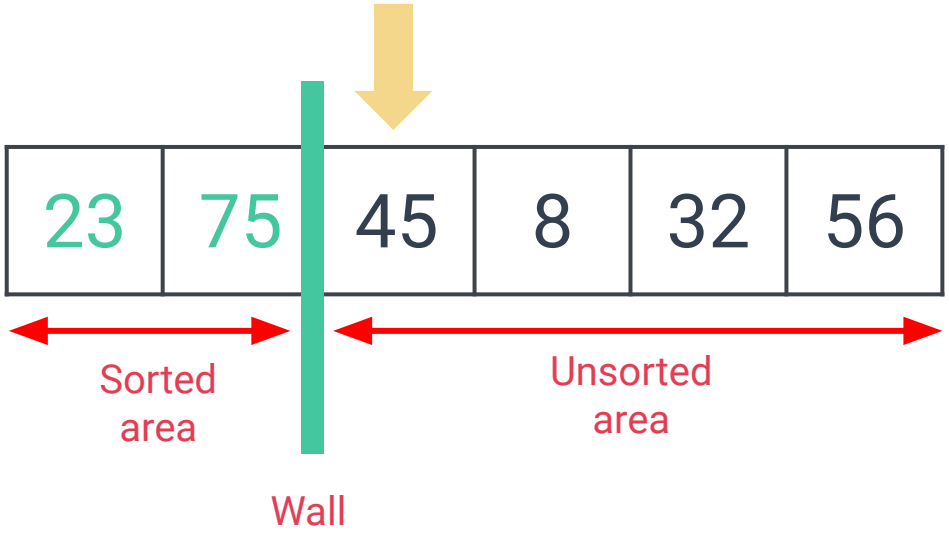
Insertion Sort Example

Round 1



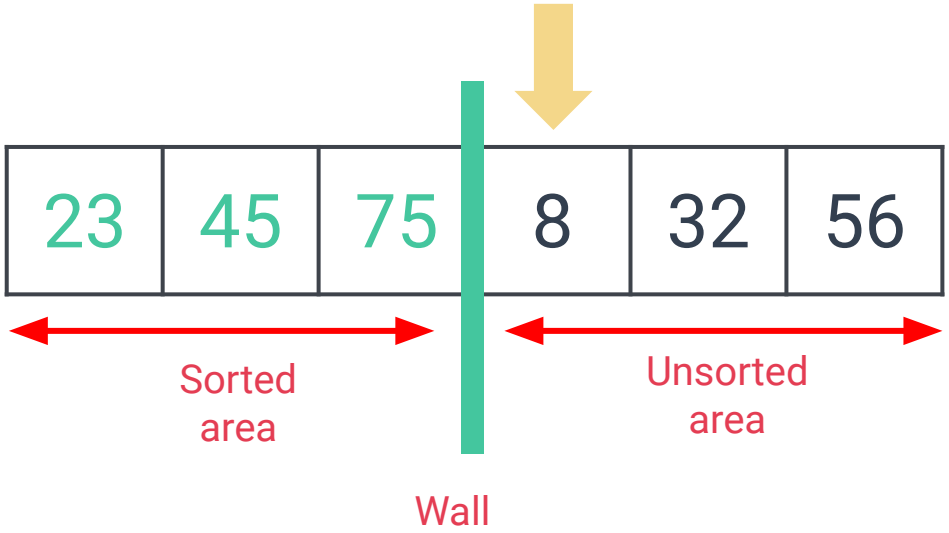
Insertion Sort Example

Round 2



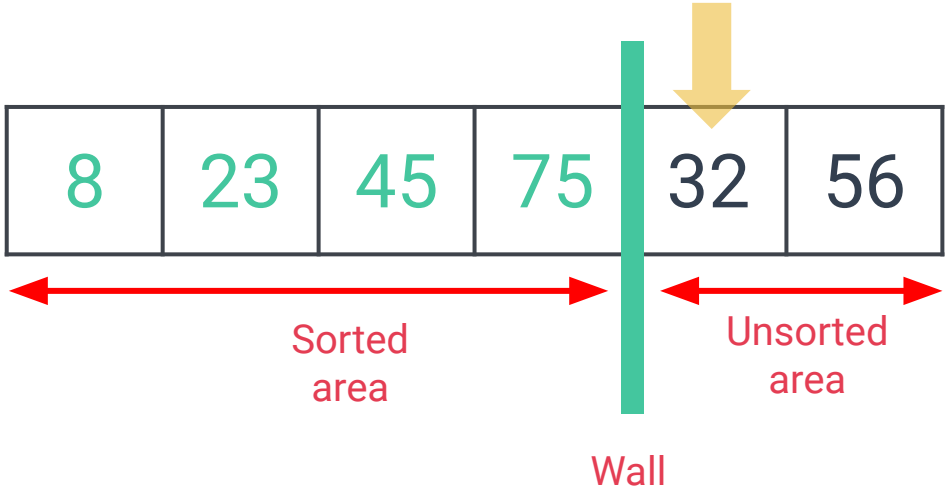
Insertion Sort Example

Round 3

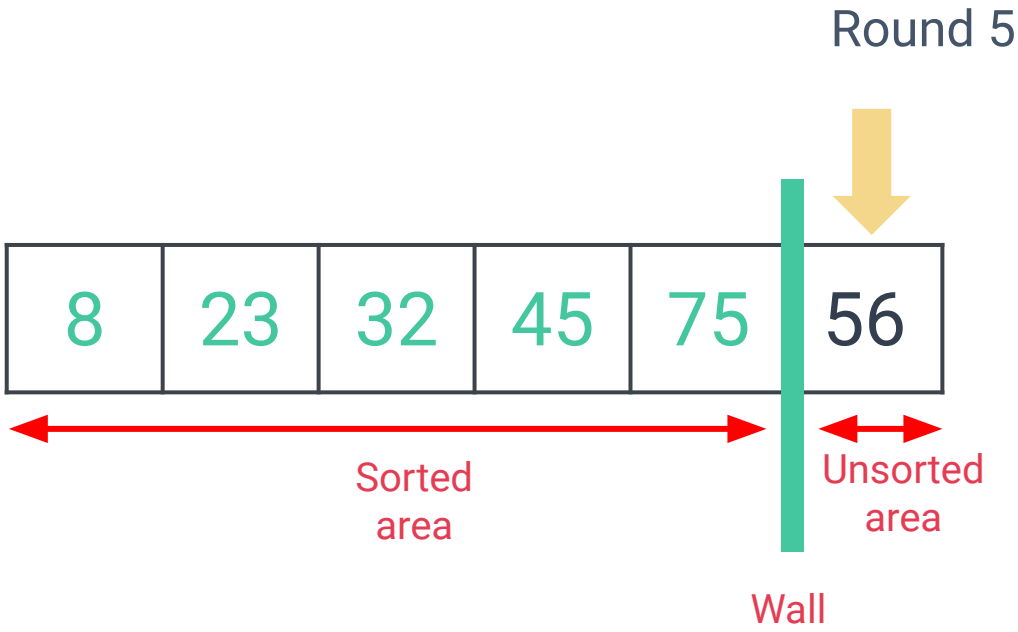


Insertion Sort Example

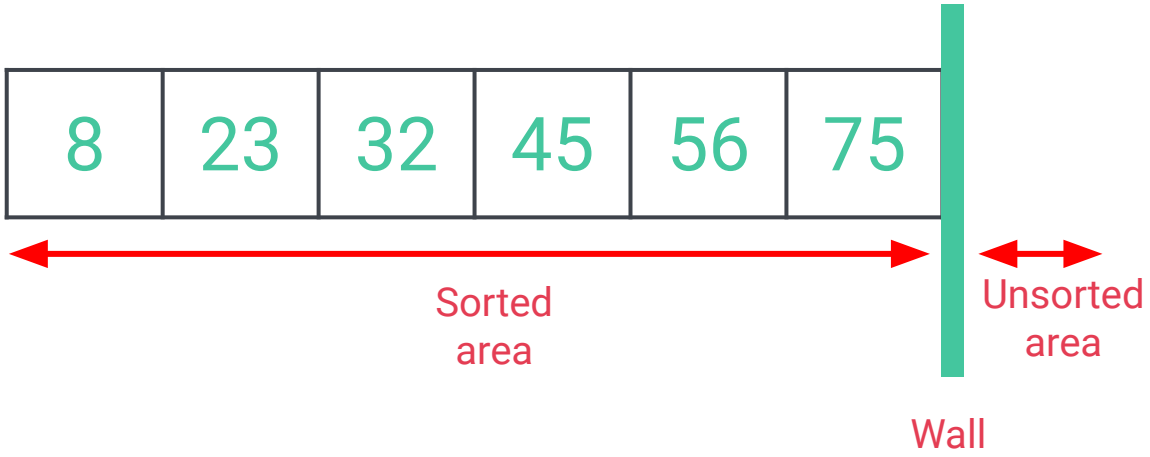
Round 4



Insertion Sort Example



Insertion Sort Example



Insertion Sort Performance

Round	Number of comparisons
1	1
2	2
3	3
...	...
n - 1	*n* - 1

for ??:

currentValue = ?

position = ?

while ??:

Swap position if the item to insert is smaller than the item in sorted sublist.

Insert the selected item at the proper place.

Divide and Conquer Strategy

3 Steps[2]:

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

[2] Michael T. Goodrich et al., Data Structures and Algorithms in Python, 2013

Merge Sort

Merge sort is a recursive sorting technique based on divide and conquer technique by continually splits a list into equal halves until it[1].

Base case: If the list is empty or has only one item, it is sorted.

Recursive case: If the list has > 1 item, split the list and recursively merge sort on both halves.

After the two halves are sorted, then a merge is invoked, combining them together into a single sorted new list.

[1] Bradley N. Miller, David L. Ranum, Problem Solving with Algorithms and Data Structures using Python, 2011

Merge Sort Real-life Demo

Merge-sort with Transylvanian-saxon (German) folk dance

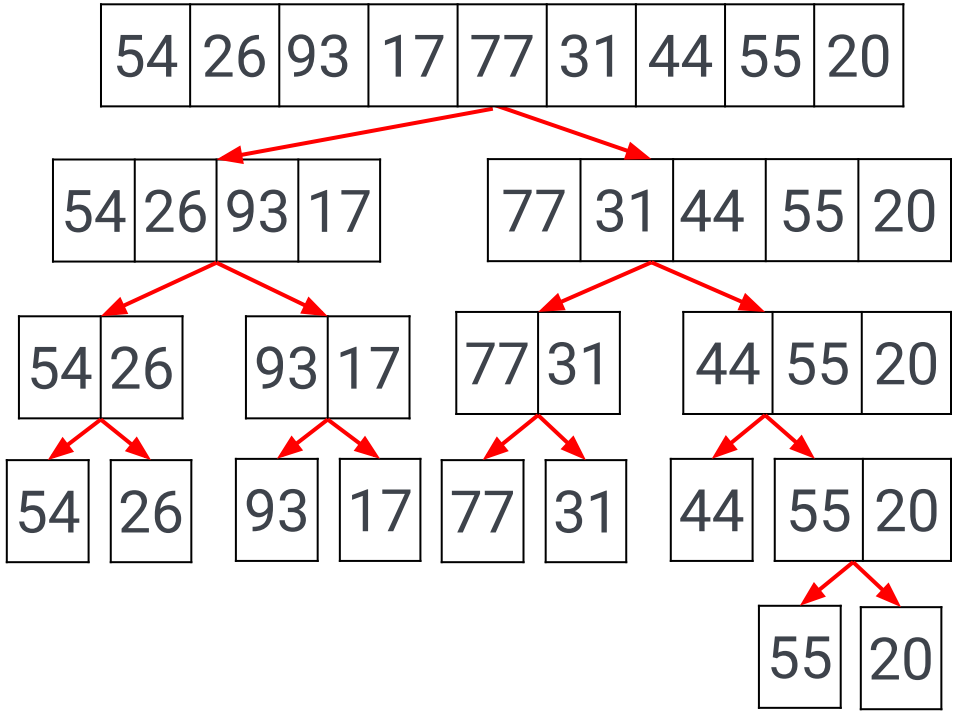
Merge Sort Example

2	45	6	9	15	12	7	8

Merge Sort Example

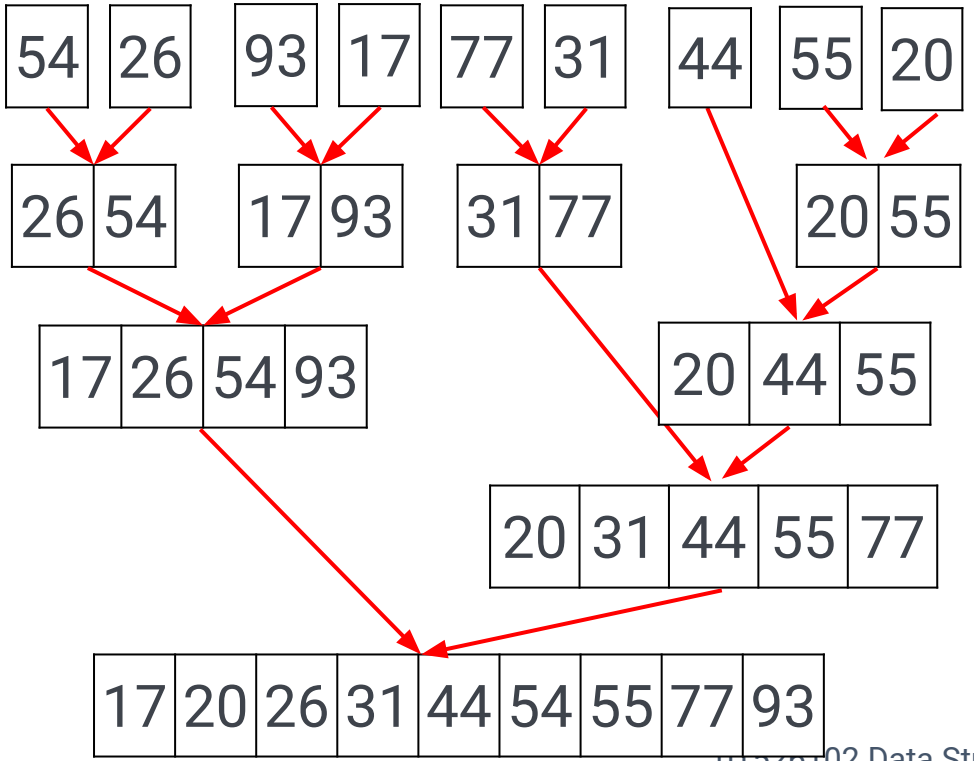
Merge Sort Example

Even elements



Merge Sort Example

Even elements



Merge Sort Performance

2 Processes[1]:

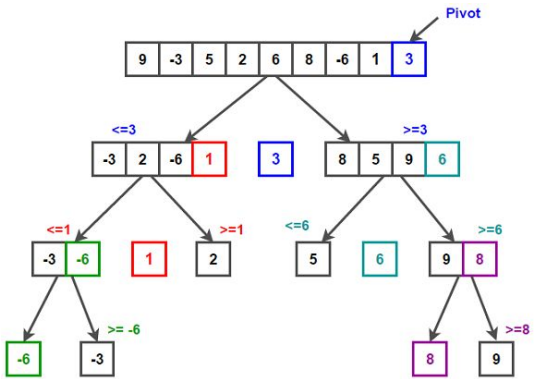
1. Splitting the list into halves
2. Merge operation and sorting

Merge sort requires extra memory space to hold two halves as they are continually splitting.

- This can be a critical factor if the list is large.

[1] Bradley N. Miller, David L. Ranum, Problem Solving with Algorithms and Data Structures using Python, 2011

Quick Sort



Quick Sort is one of the most efficient sorting algorithms and is based on the splitting of a list into smaller ones without using additional storage[1].

However, it is possible that the list may not be divided in half.

[1] Bradley N. Miller, David L. Ranum, Problem Solving with Algorithms and Data Structures using Python, 2011
Ref: https://www.interviewbit.com/tutorial/quicksort-algorithm/

Sorting Paper Example

Example: Sort the papers containing the names of the students, by name from A-Z.

Quick Sort

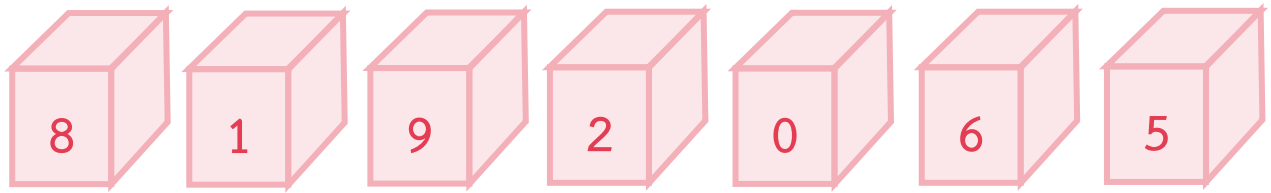
Technically, quick sort follows the below steps:

1. Make any element as **pivot**.
2. Partition the list on the basis of pivot.
3. Apply quick sort on left partition recursively.
4. Apply quick sort on right partition recursively.

Choosing a Pivot

To determine the pivot element, there are 3 general approaches[3]:

1) Choosing the first or last element of the unsorted list.
2) Random method.
3) Median-of-three method. It considers three elements including the first, last, and middle element in the list.



[3] Robert Sedgewick, Algorithms in C: Fundamentals, Data Structures, Sorting, Searching, Parts 1–4, 1998

Quick Sort Real-life Demo

Quick-sort with Hungarian (Küküllőmenti legényes) folk dance

Quick Sort Example

Example: Sort this list with divide and conquer strategy without using extra space.

50	23	9	18	61	32

Step 1: Make any element as **pivot**.

- For convenience, the rightmost index is selected as pivot.
- 'Low' and 'High' pointers corresponds to the first index and last index respectively.
- In this example, the low is at index 0 and high is at index 5.
- The value at pivot is 32.

Quick Sort Example

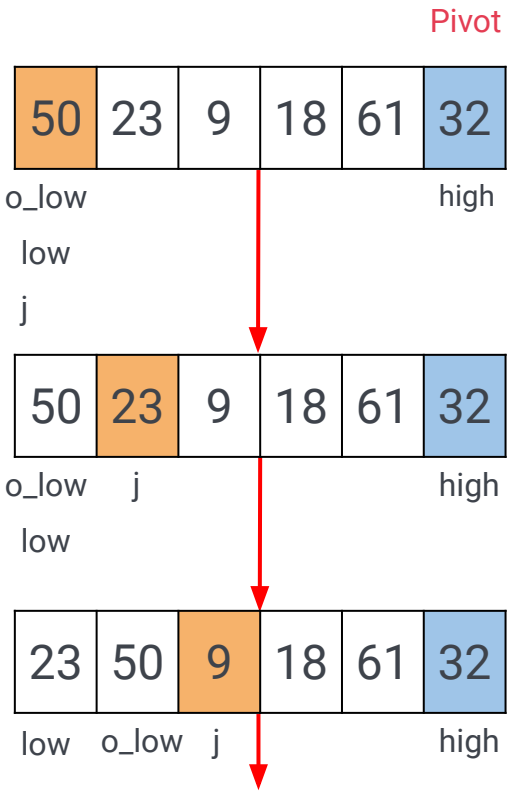
Example: Sort this list with divide and conquer strategy without using extra space.

50	23	9	18	61	32

Step 2: Partition the list on the basis of pivot.

- Rearranges the list in such a way that pivot(32) is at its proper position.
  - To the left of the pivot, the value of all elements is less than or equal to it.
  - To the right of the pivot, the value of all elements is higher than it.

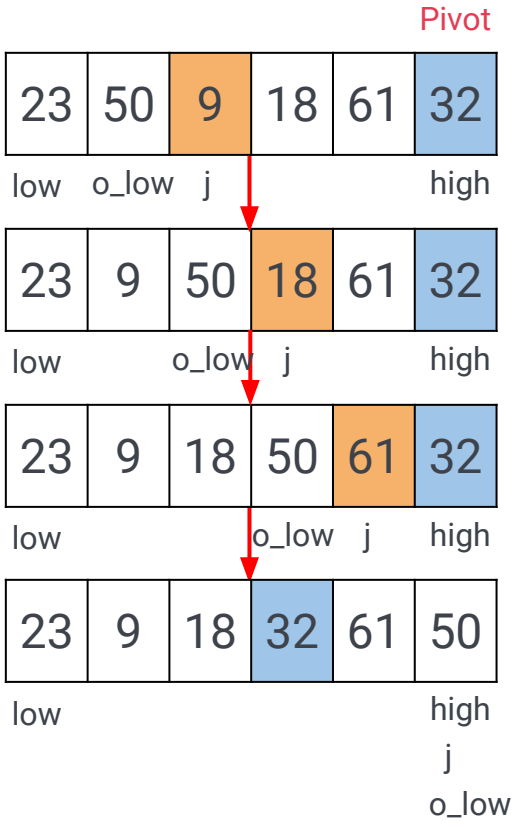
Quick Sort Example




```
original_low = low
pivot = alist[high]

for j in range(low, high):
    if alist[j] <= pivot:
        swap(alist[original_low], alist[j])
        original_low += 1
    else:
        continue
```

Quick Sort Example




```
original_low = low
pivot = alist[high]


for j in range(low, high):
        if alist[j] <= pivot:
                swap(alist[original_low], alist[j])
                original_low += 1
        else:
                continue
swap(alist[original_low], alist[high])
```

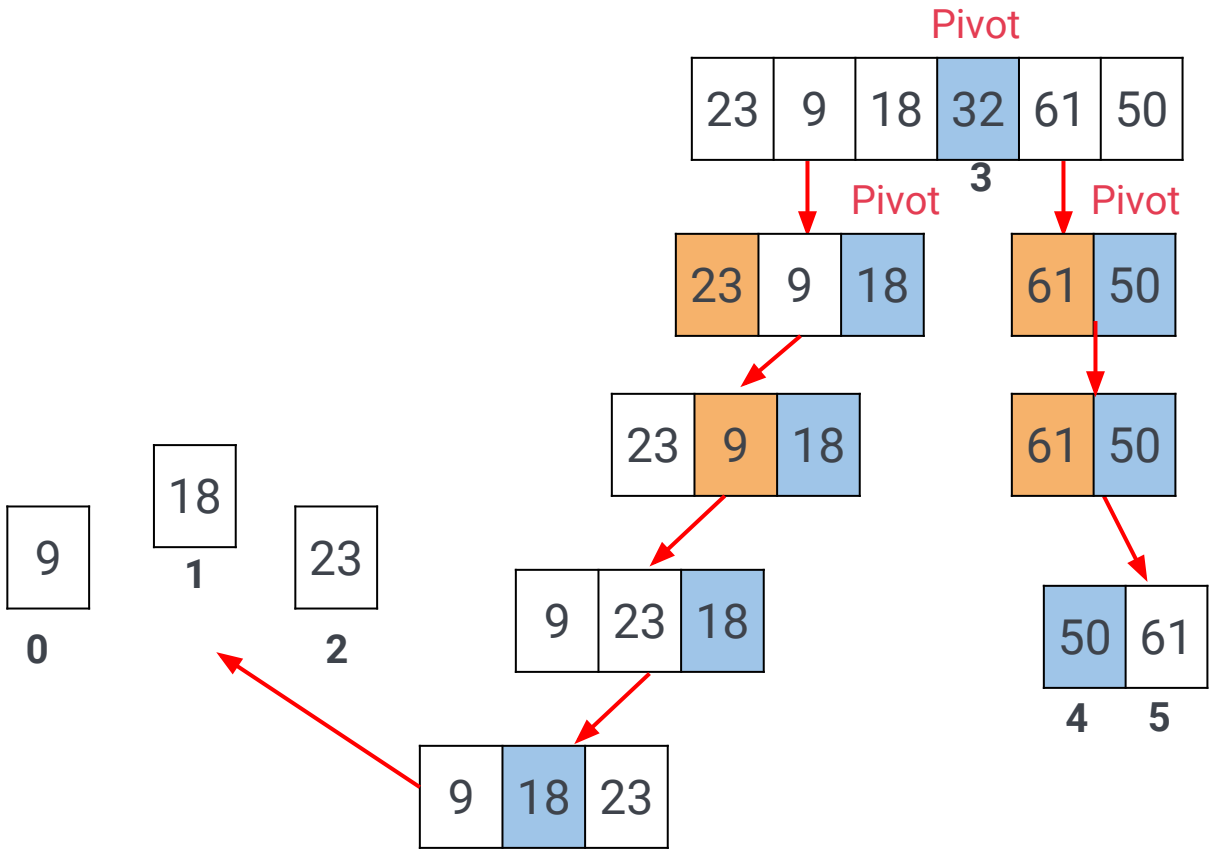
Quick Sort Example

50	23	9	18	61	32

Step 3: Divide the list into two parts - left and right sublists.

Step 4: Repeat the steps for the left and right sublists recursively.

Quick Sort Example



Quick Sort Performance

No need for additional memory as in the merge sort process.

If partition occurs in the middle of the list: log *n* times to split[1]

Each item checked against the pivot value: *n* times[1]

However, the split points may not be in the middle and can be very skewed to the left or the right, leaving a very uneven division.

For example, sorting a list of *n* items divides into sorting a list of 0 items and a list of n−1 items.

[1] Bradley N. Miller, David L. Ranum, Problem Solving with Algorithms and Data Structures using Python, 2011

Comparing Sorting Algorithms

Sorting Algorithms Animations