SAMSUNG

Samsung Innovation Campus

Coding and Programming

Together for Tomorrow!
Enabling People
Education for Future Generations

Chapter 5.

Algorithm 2
- Sorting Algorithms

Coding and Programming

Chapter Description

Chapter objectives

- ✓ Learners can understand sorting problems and solve sorting problems using bubble sort, selection sort, insertion sort, merge sort, and quick sort methods. In addition, the sorting algorithm can be implemented in Python, and learner can understand and compare the time complexity of each sorting method.

Chapter contents

- ✓ Unit 27. Bubble, Selection, and Insertion Sort
- ✓ Unit 28. Merge Sort
- ✓ Unit 29. Quick Sort

Unit 27.

Bubble, Selection, and Insertion Sort

Unit learning objective (1/3)

Learning objectives

- ✓ Understand the sorting problem and be able to solve the sorting problem using various sorting techniques.
- ✓ Understand and be able to explain the differences between bubble sort, selection sort, and insertion sort.
- ✓ Understand and be able to describe the time complexity of bubble sort, selection sort, and insertion sort.

Unit learning objective (2/3)

Learning overview

- ✓ Implement bubble sort, which sorts two adjacent elements by swapping them.
- ✓ Implement selection sort, which finds the smallest element and creates a sorted list.
- ✓ Implement insertion sort, which inserts a new element into a sorted list.

Concepts you will need to know from previous units

- ✓ Using comparison operators to compare two numbers.
- ✓ Using the swap function to swap the positions of two numbers.
- ✓ Applying Big O notation to analyze the time complexity of an algorithm.

Keywords

- Sorting Problem
- Bubble Sort
- Selection Sort
- Insertion Sort
- Quadratic Time

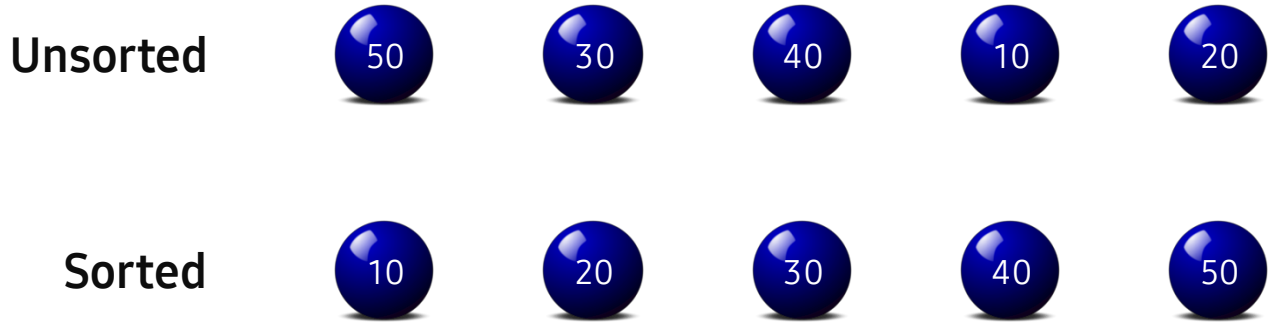
Mission

1. Real world problem

1.1. The problem of sorting

In our daily life, we often encounter alignment problems.

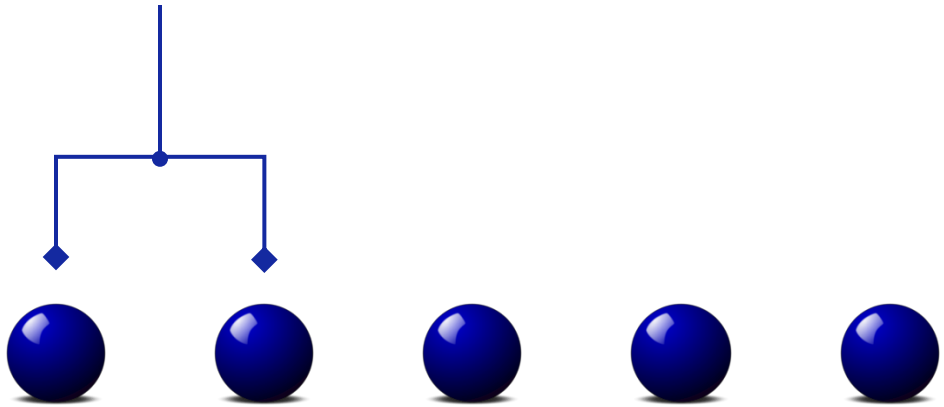
Ex Suppose you are given 5 blue iron marbles as shown below. These iron marbles are the same shape and size but have different weights. What method should be used to sort these iron marbles in ascending order of weight as shown below?



2. Mission

2.1. Bubble sort: when you can swap only two adjacent elements

- Imagine that a tool we can use is a device that can do the following:
  - You can pick up two marbles adjacent to each other and compare their weights.
  - The two marbles can be put down by swapping positions.
- Since the device cannot remember the weight of the marbles, only two adjacent marbles at a time can change their positions. It can use a strategy to send the heavier marble to the right for each comparison.
- With this strategy, how many comparisons do all the marbles need to be placed in a sorted order by weight?



2. Mission

2.2. Selection sort: when you can use a balance scale

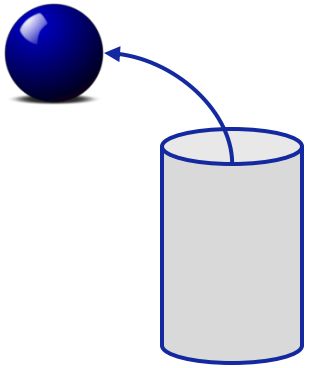
- Use a balance scale. The balance scale can do the following operations:
  - You can pick up two marbles that are not adjacent to each other and compare their weights.
  - The two marbles can be returned by swapping their positions.
- The balance scale also cannot remember the weight of the marble, but it can swap the positions of two marbles that are not adjacent to each other at once. You can choose a strategy to compare the first marble with the least weighted marble so that it can be swapped.
- With this strategy, how many comparisons would it take to place all the marbles in sorted order by weight?

2. Mission

2.3. Insertion sort: when you can get only a ball at a time

- Use the balance scale. This time there are limitations:
  - The iron marbles are in the barrel, and they can only be taken out one at a time.
  - Whenever a marble is taken out, it should always be placed in a sorted position.
- In this case, it is not possible to compare all the marbles at once, but since there are already sorted marbles each time they are taken out, the new marble can be searched for in sequence.
- With this strategy, how many comparisons would it take to place all the marbles in sorted order by weight?

Already Sorted



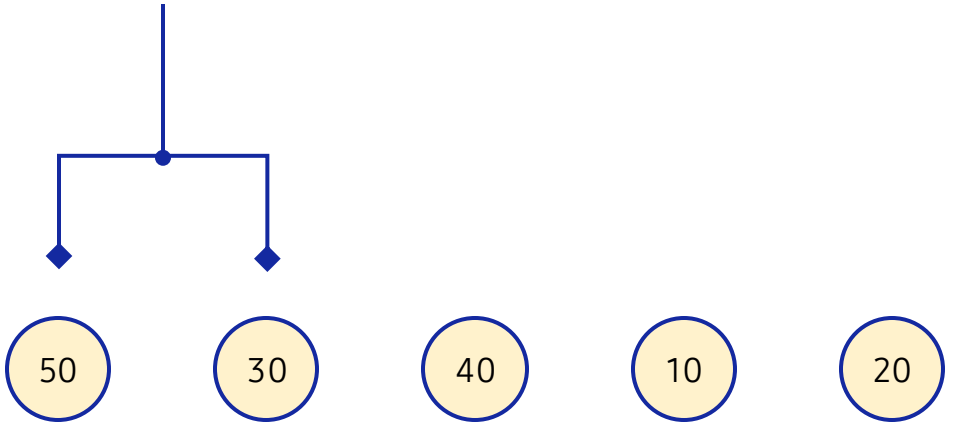
Key concept

1. Bubble Sort

1.1. An example of bubble sort

Bubble sort is a sorting algorithm that compares two adjacent elements and swaps their positions if the order is reversed.

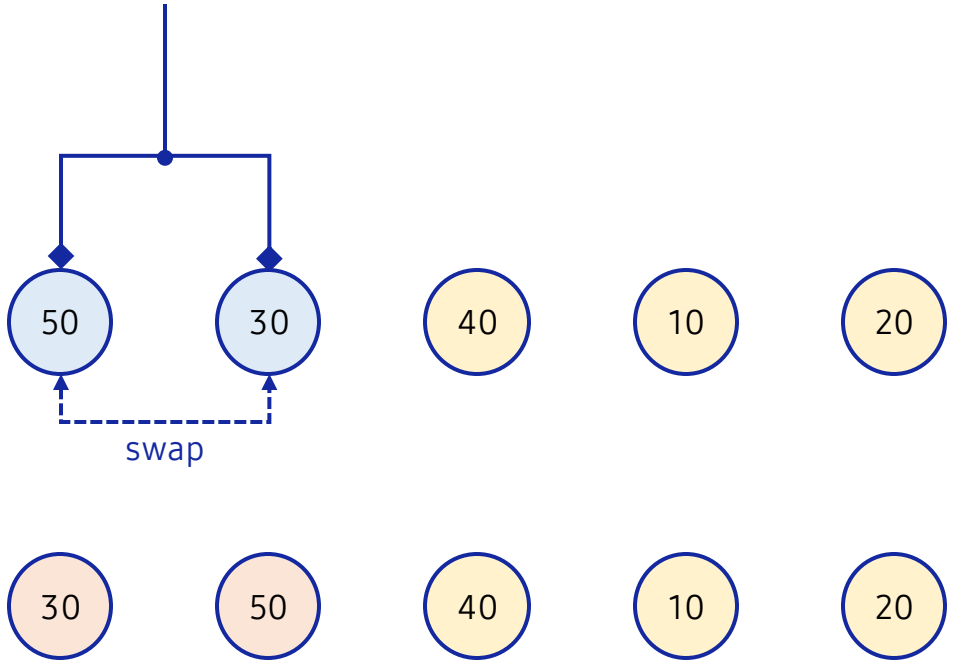
Ex Suppose we want to sort a list arranged in the order of [50, 30, 40, 10, 20] as follows.



1. Bubble Sort

1.2. The process of bubble sort

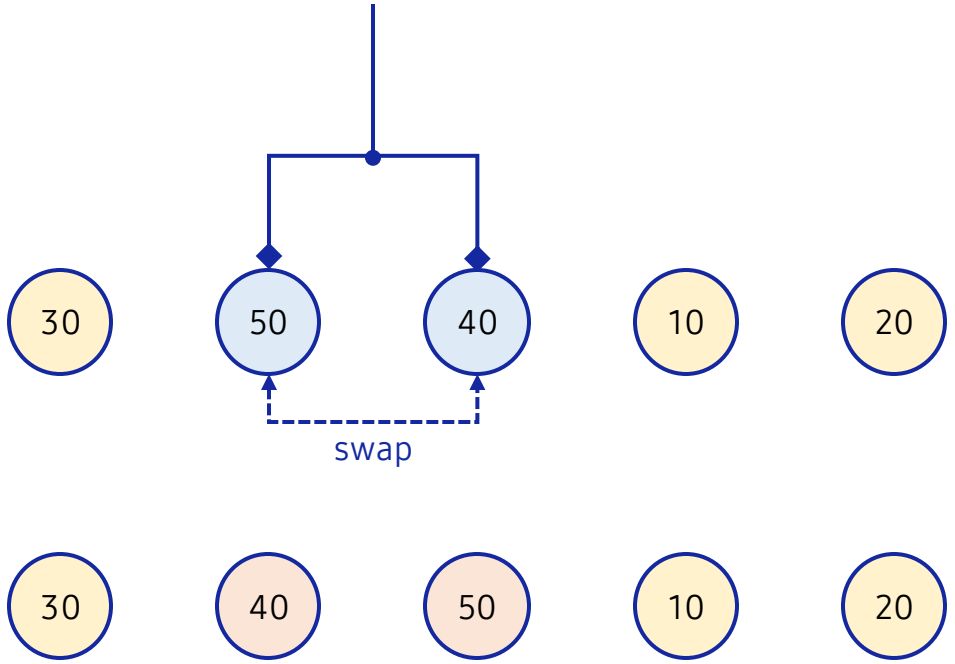
- First, compare the first element 50 with the second element 30. Since 50 is greater than 30, the places of the two numbers must be swapped.



1. Bubble Sort

1.2. The process of bubble sort

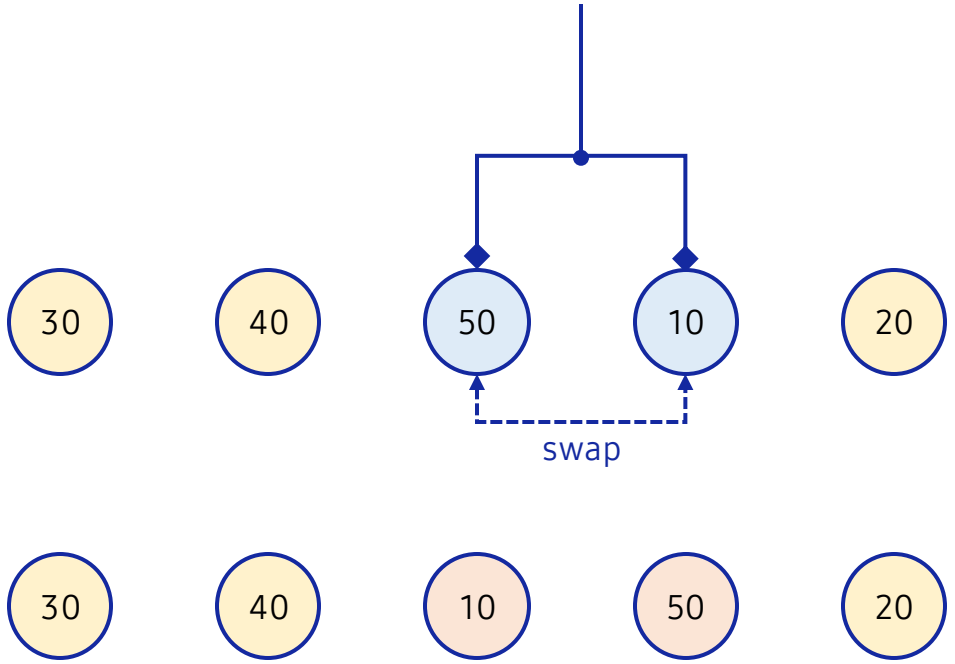
Next, compare the second element 50 with the third element 40. Since 50 is greater than 40, the places of the two numbers must be swapped.



1. Bubble Sort

1.2. The process of bubble sort

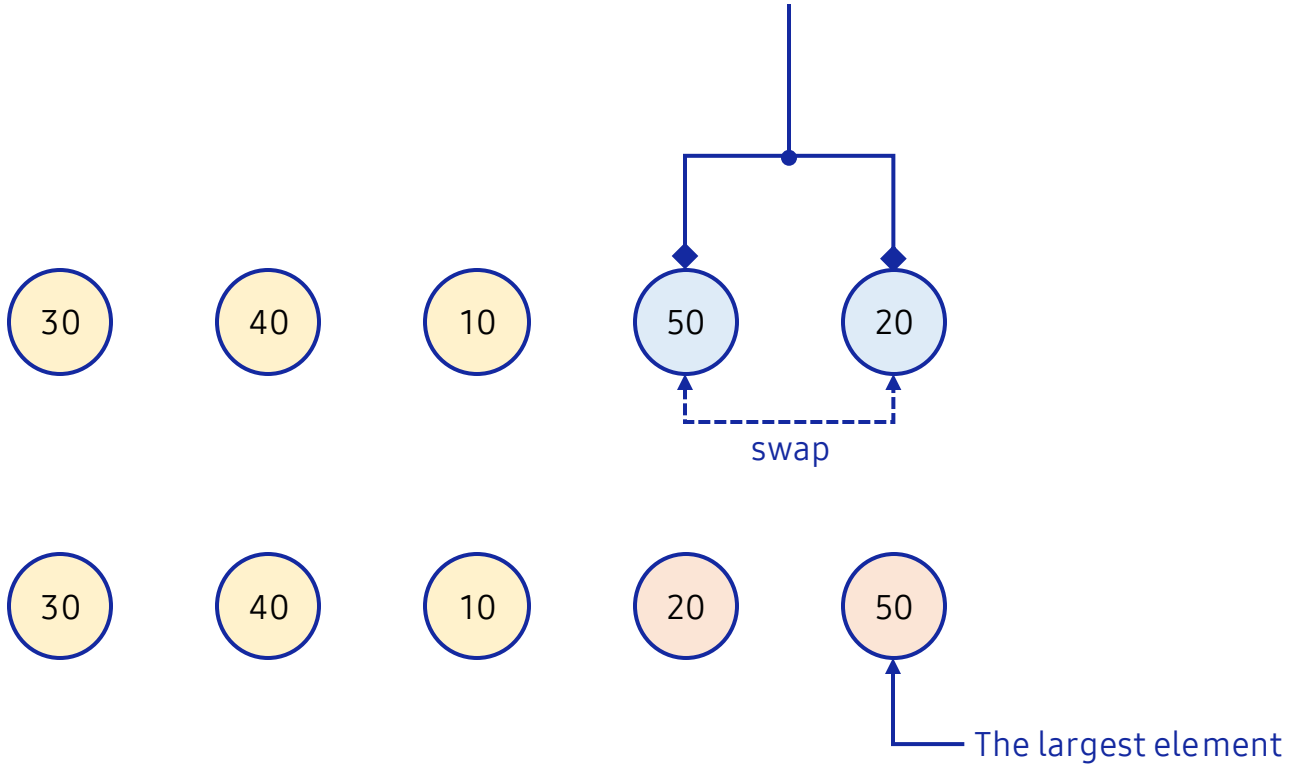
Next, compare the third element 50 with the fourth element 10. Since 50 is greater than 10, the places of the two numbers must be swapped.



1. Bubble Sort

1.2. The process of bubble sort

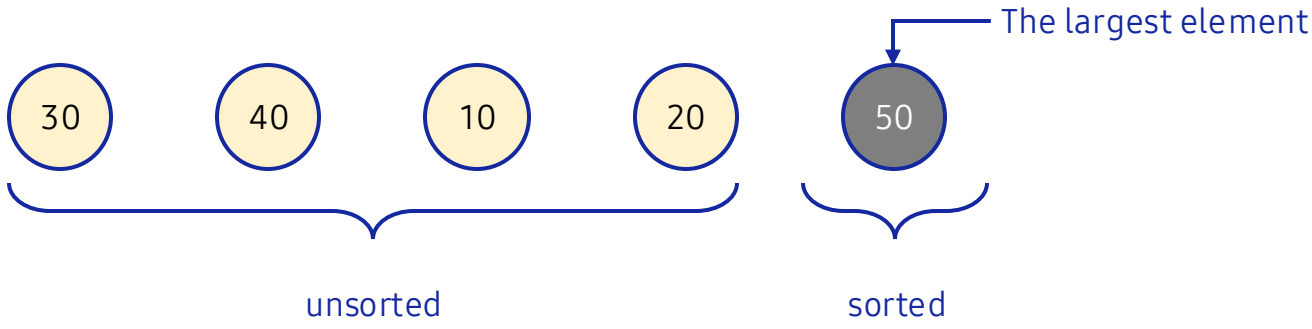
Next, compare the fourth element 50 with the last element 20. Since 50 is greater than 20, the places of the two numbers must be swapped.



1. Bubble Sort

1.2. The process of bubble sort

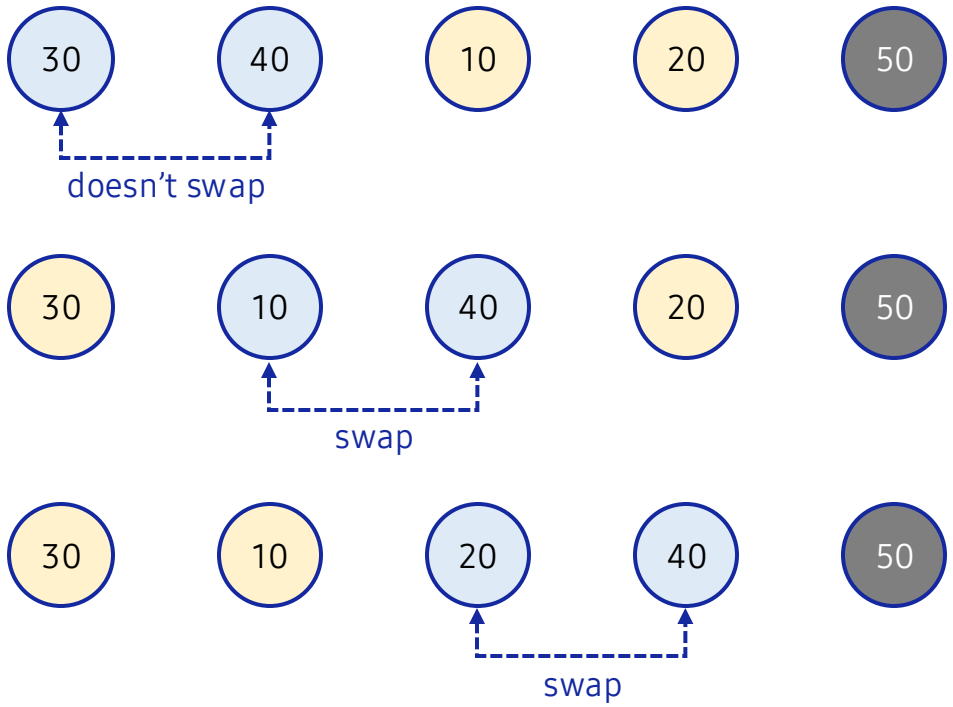
- Note that after the first pass, the largest element in the list 50 is placed on the far right. The last element from sorting in the second pass can be excluded.



1. Bubble Sort

1.2. The process of bubble sort

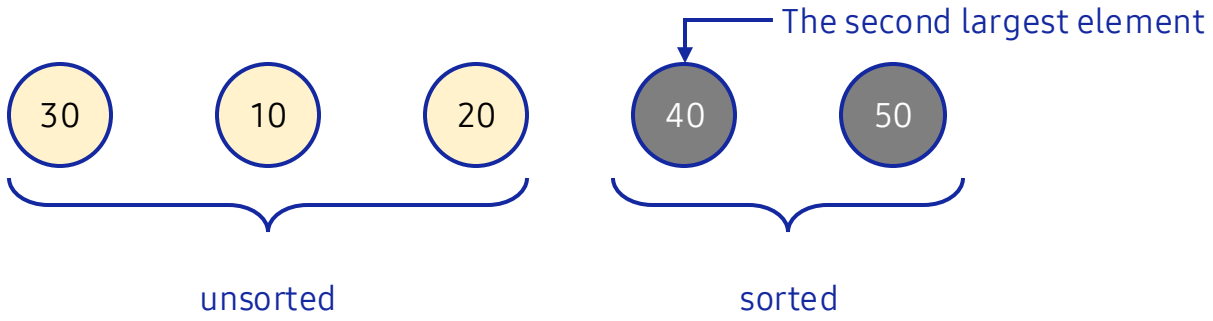
Continue bubble sorting on an unsorted list except for the last element.



1. Bubble Sort

1.2. The process of bubble sort

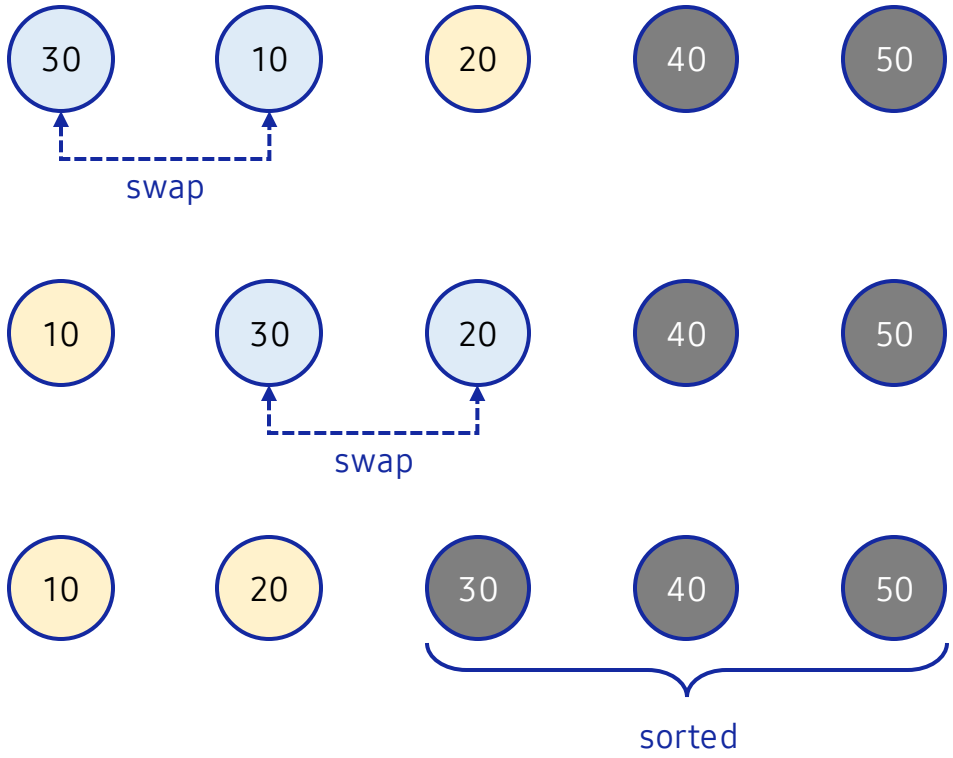
Note that the second pass found the second largest element. Now this second element can also be excluded from sorting.



1. Bubble Sort

1.2. The process of bubble sort

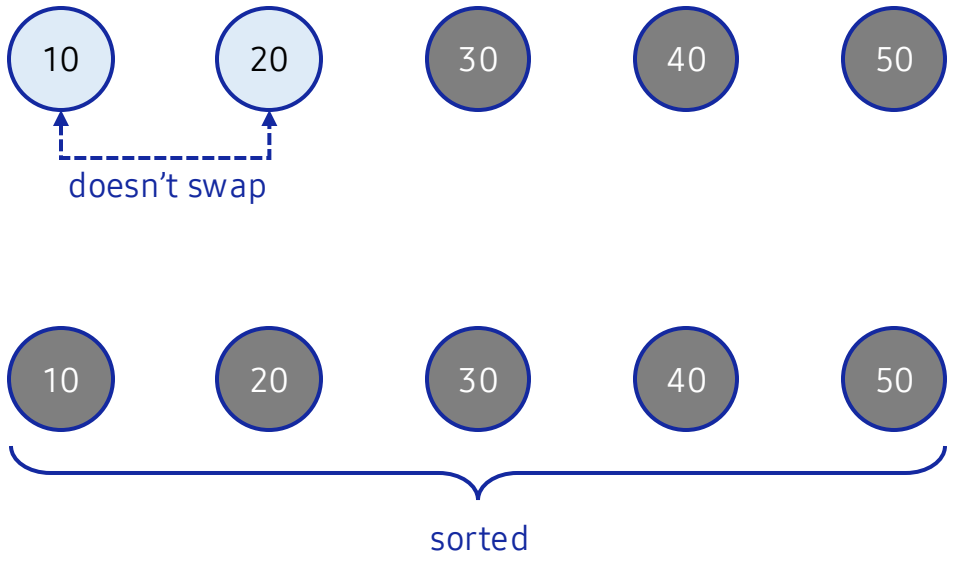
In the same way, bubble sorting can be continued in the third pass.



1. Bubble Sort

1.2. The process of bubble sort

In the same way, bubble sort can be continued until all elements are sorted.



2. Selection Sort

2.1. An example of selection sort

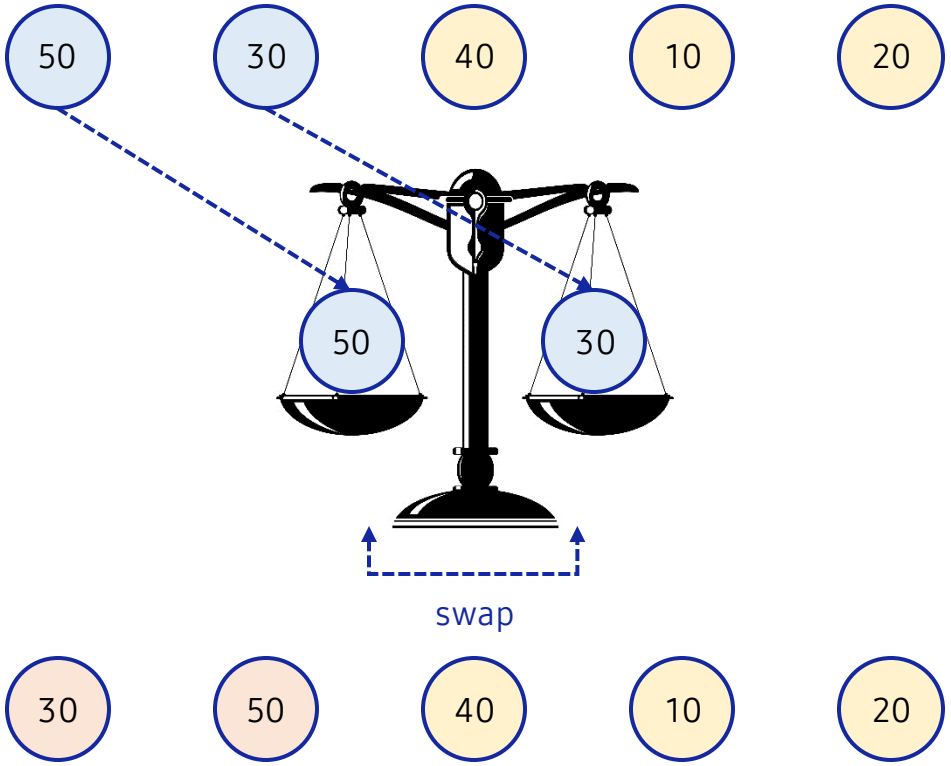
Selection sort is a sorting algorithm that finds the smallest element in a list and place it on the leftmost side. To do this, it uses a strategy to compare the first element of an unsorted list with the rest of the list and to swap positions with the smaller element.

Ex Suppose we want to sort a list arranged in the order of [50, 30, 40, 10, 20] as follows.

2. Selection Sort

2.2. The process of selection sort

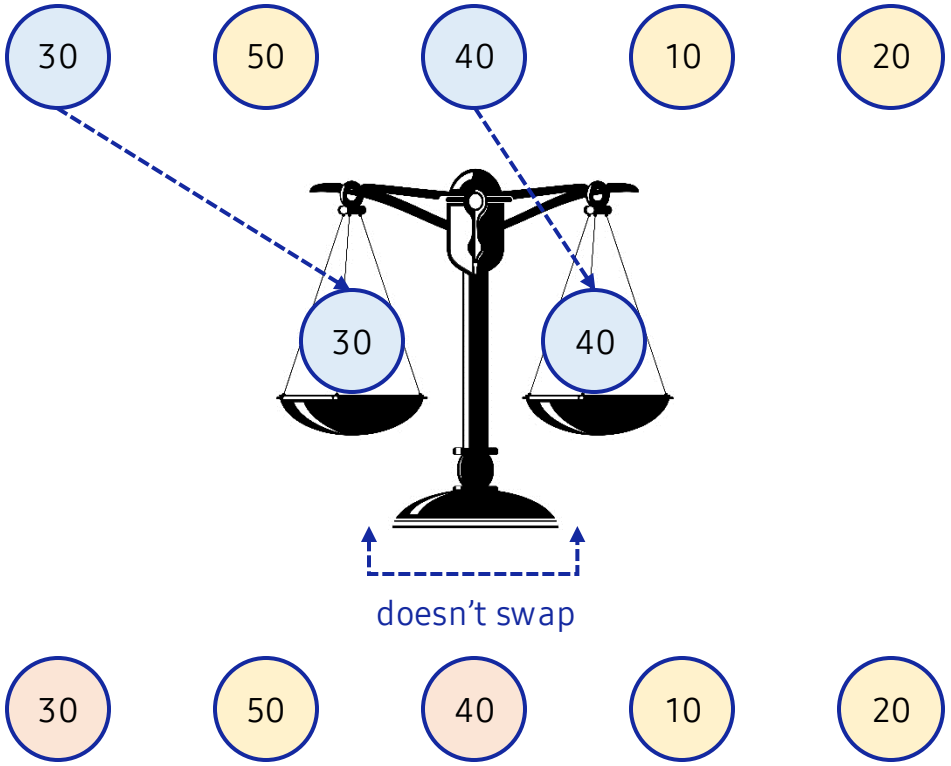
First, compare the first element 50 with the second element 30. Since 50 is greater than 30, the places of each other must be changed.



2. Selection Sort

2.2. The process of selection sort

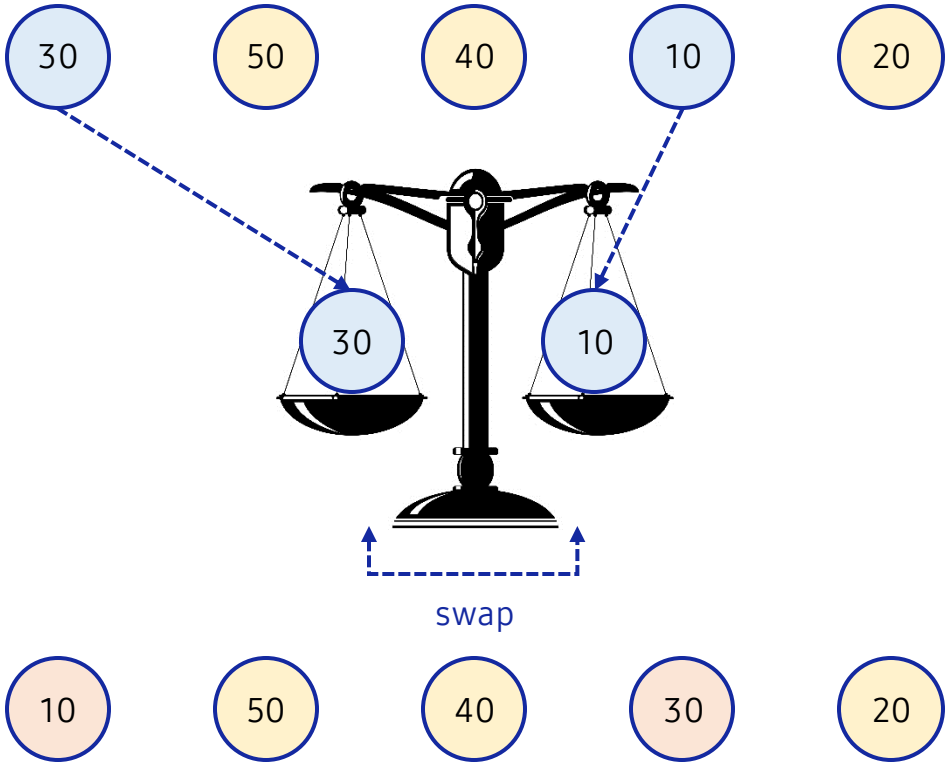
Next, compare the first element 30 with the third element 40. Since 30 is less than 40, they do not change their positions.



2. Selection Sort

2.2. The process of selection sort

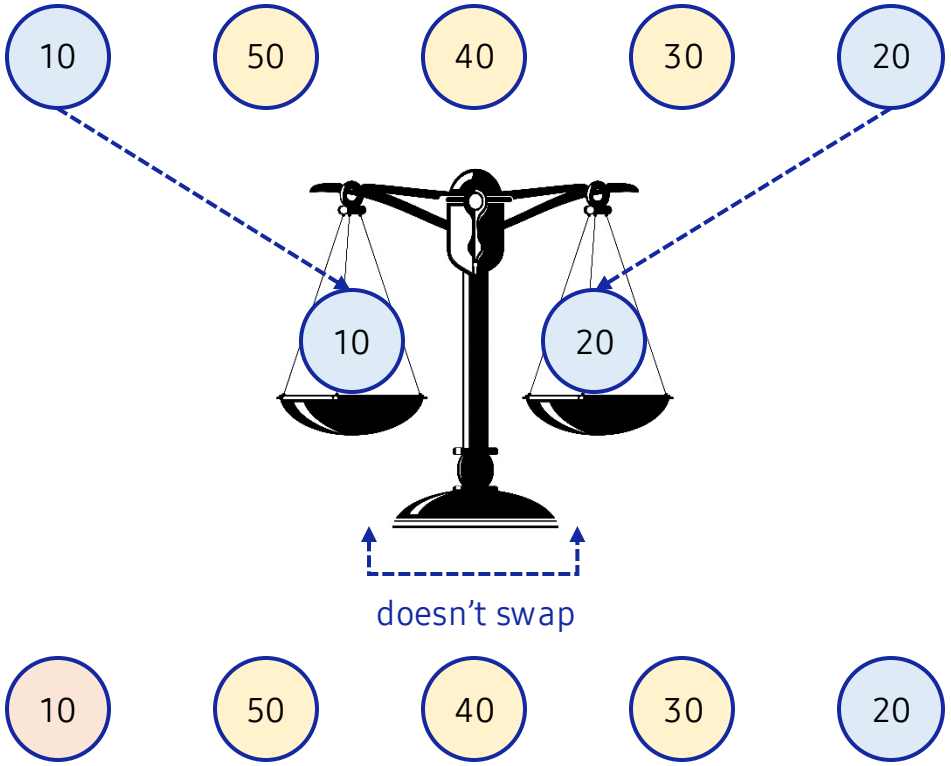
Next, compare the first element 30 with the fourth element 10. Since 10 is less than 30, the places of each other must be changed.



2. Selection Sort

2.2. The process of selection sort

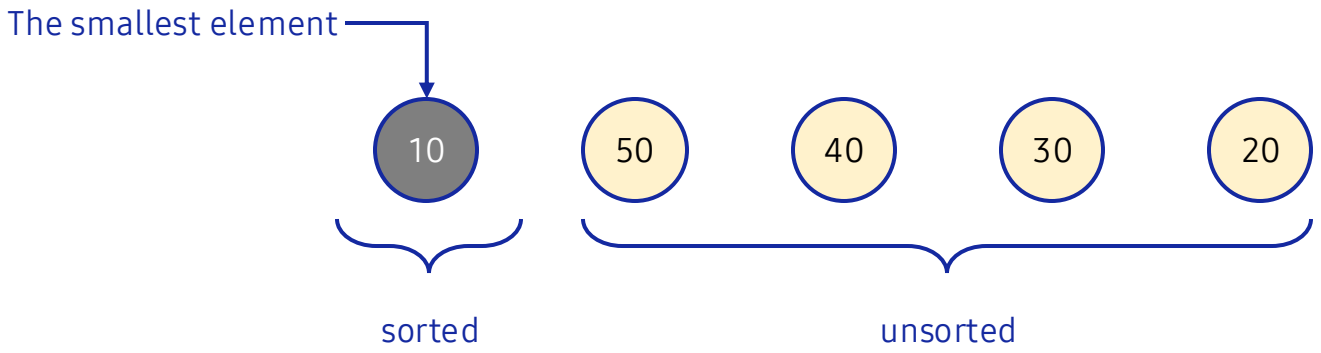
Next, compare the first element 10 with the fifth element 20. Since 10 is less than 20, they do not change their positions.



2. Selection Sort

2.2. The process of selection sort

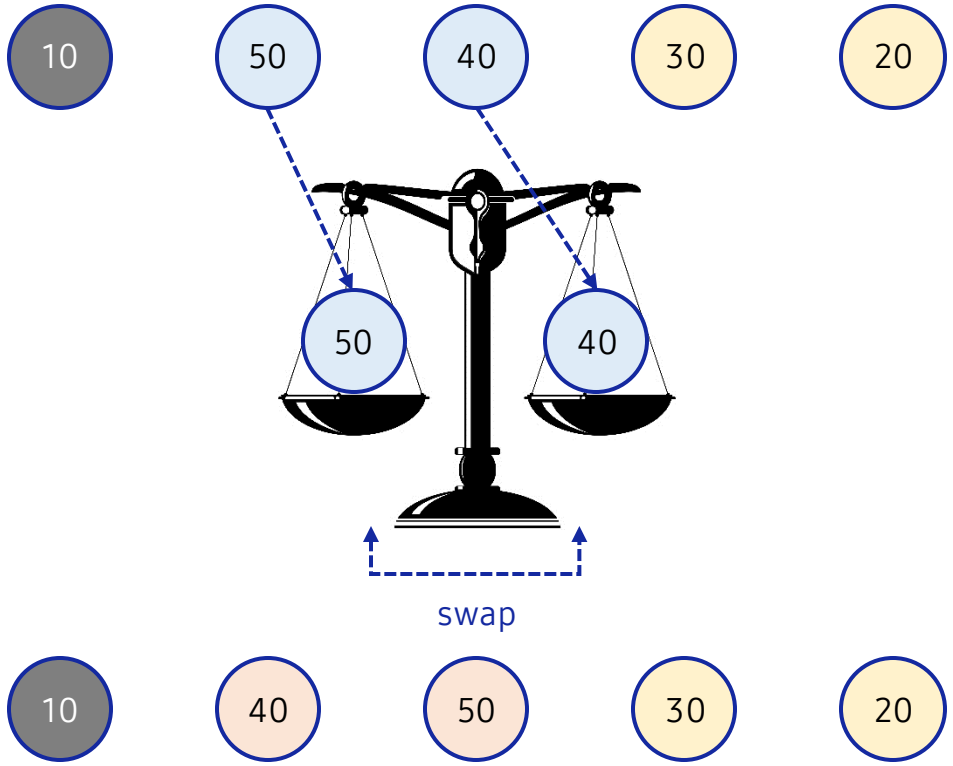
Note that in selection sort, the smallest element 10 is placed at the beginning of the list at the end of the first pass. Since the first element is a sorted list, it can be excluded from selection sorting of the next pass.



2. Selection Sort

2.2. The process of selection sort

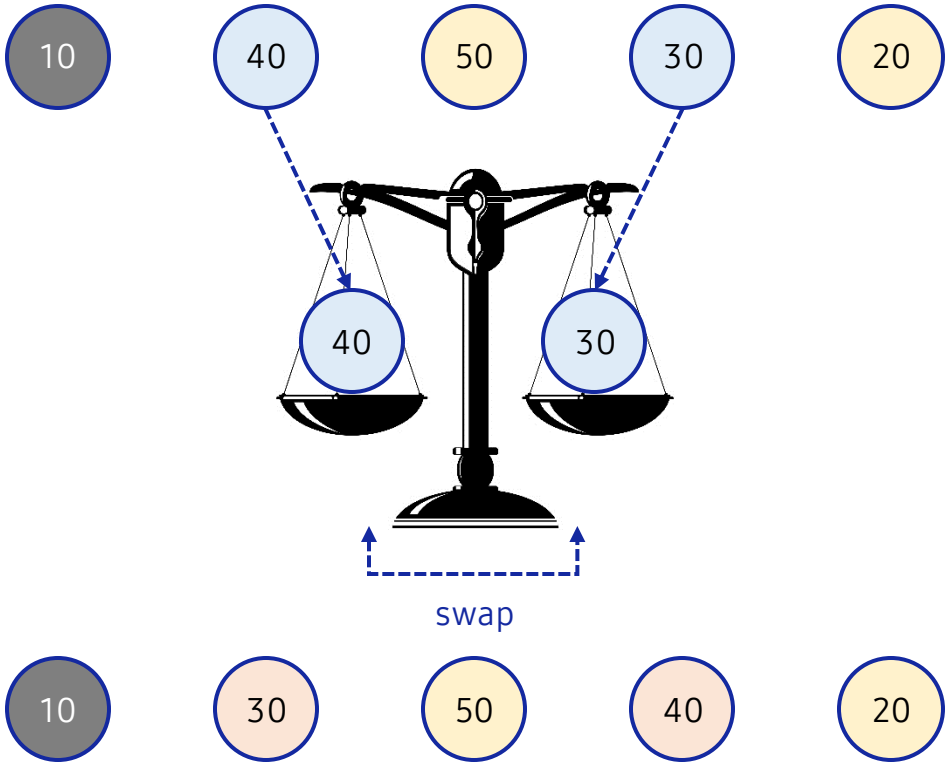
Selection sort can continue on an unsorted list except for the first element.



2. Selection Sort

2.2. The process of selection sort

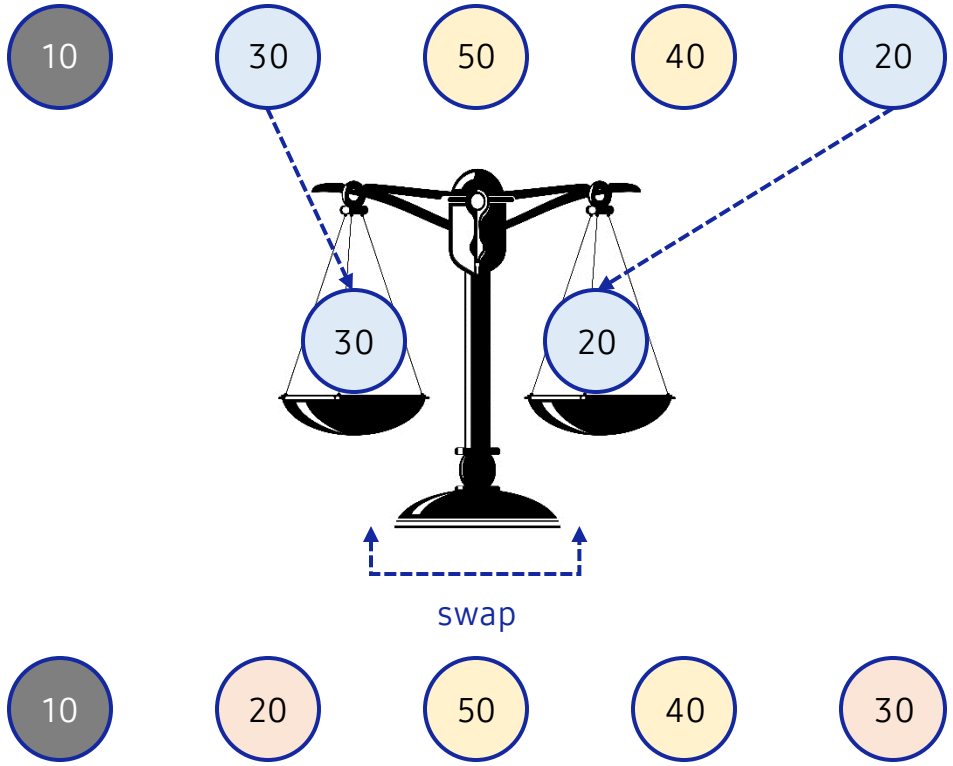
Selection sort can continue on an unsorted list except for the first element.



2. Selection Sort

2.2. The process of selection sort

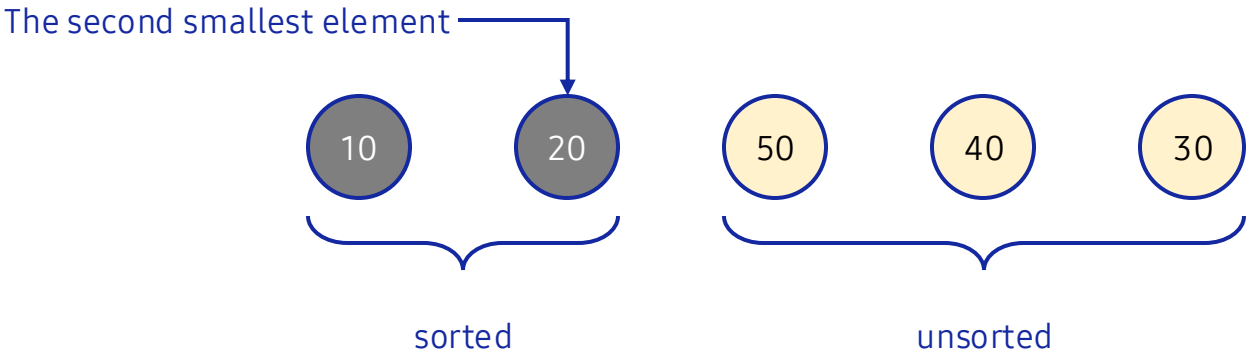
Selection sort can continue on an unsorted list except for the first element.



2. Selection Sort

2.2. The process of selection sort

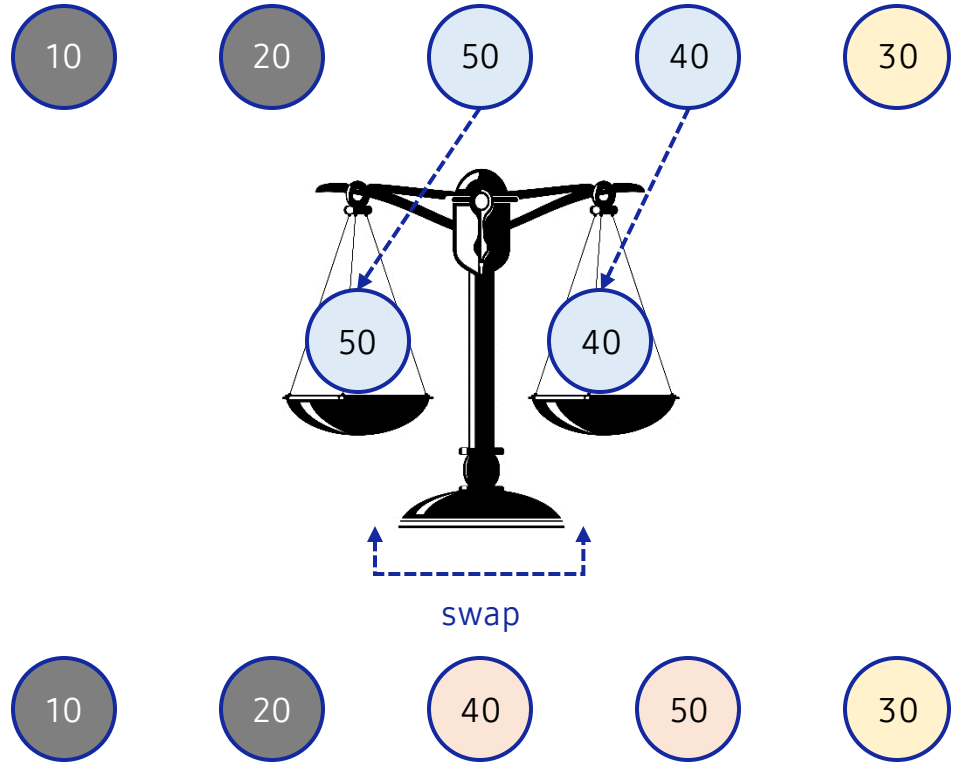
Note that the second pass found the second largest element. Now this second element can also be excluded from sorting.



2. Selection Sort

2.2. The process of selection sort

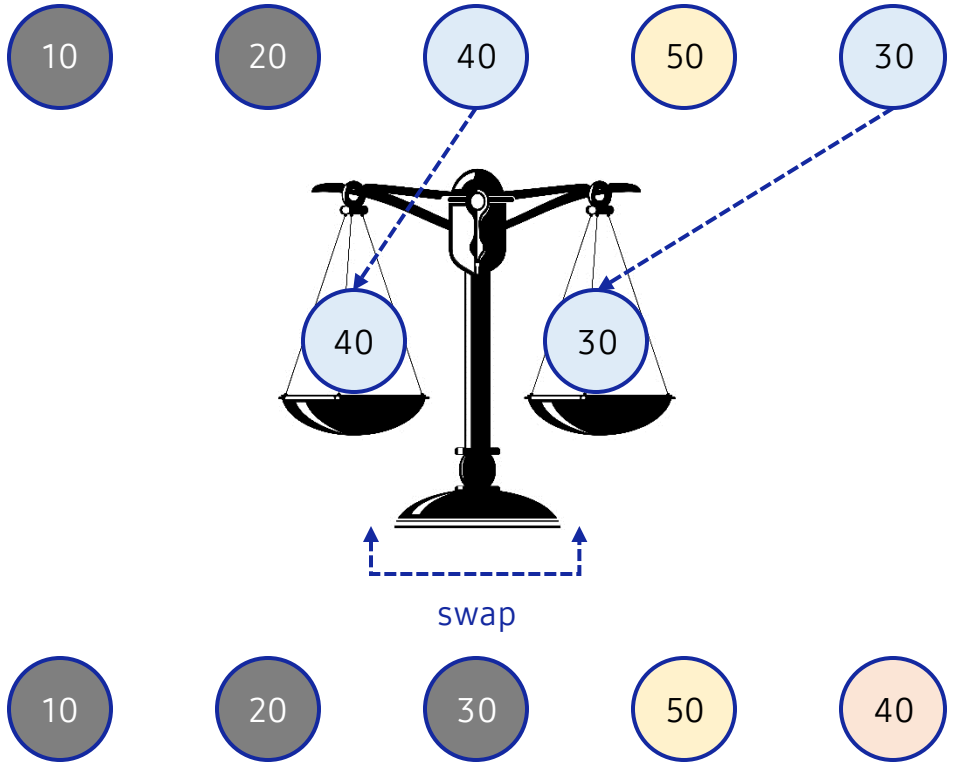
Note that the second pass found the second largest element. Now this second element can also be excluded from sorting.



2. Selection Sort

2.2. The process of selection sort

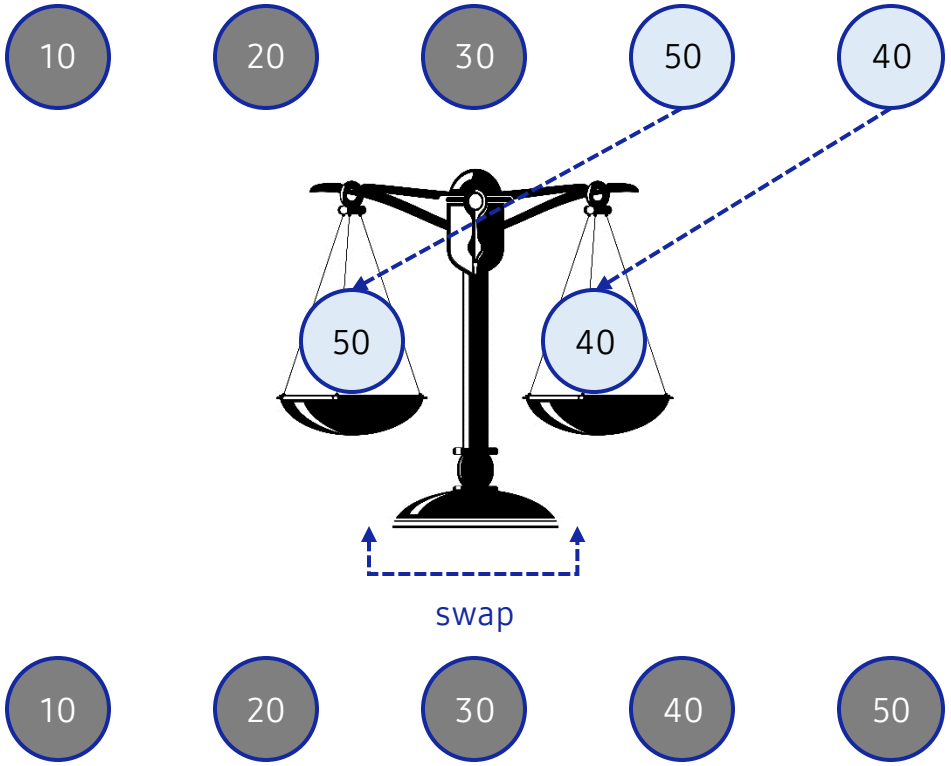
Selection sorting can be continued on an unsorted list except for the sorted list.



2. Selection Sort

2.2. The process of selection sort

Selection sorting can be continued on an unsorted list except for the sorted list.



3. Insertion Sort

3.1. An example of insertion sort

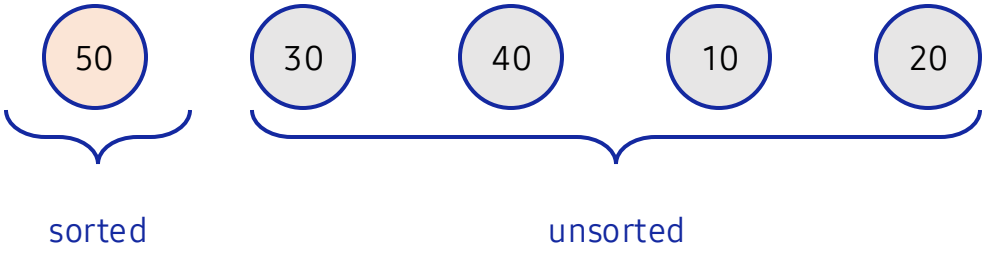
Insertion sort is a sorting algorithm that adds elements from an unsorted list to the sorted list one by one. To do this, it uses a strategy to find the position to add by comparing the element to be added with the rest of the sequentially sorted list.

Ex Suppose we want to sort a list arranged in the order of [50, 30, 40, 10, 20] as follows.

3. Insertion Sort

3.2. The process of insertion sort

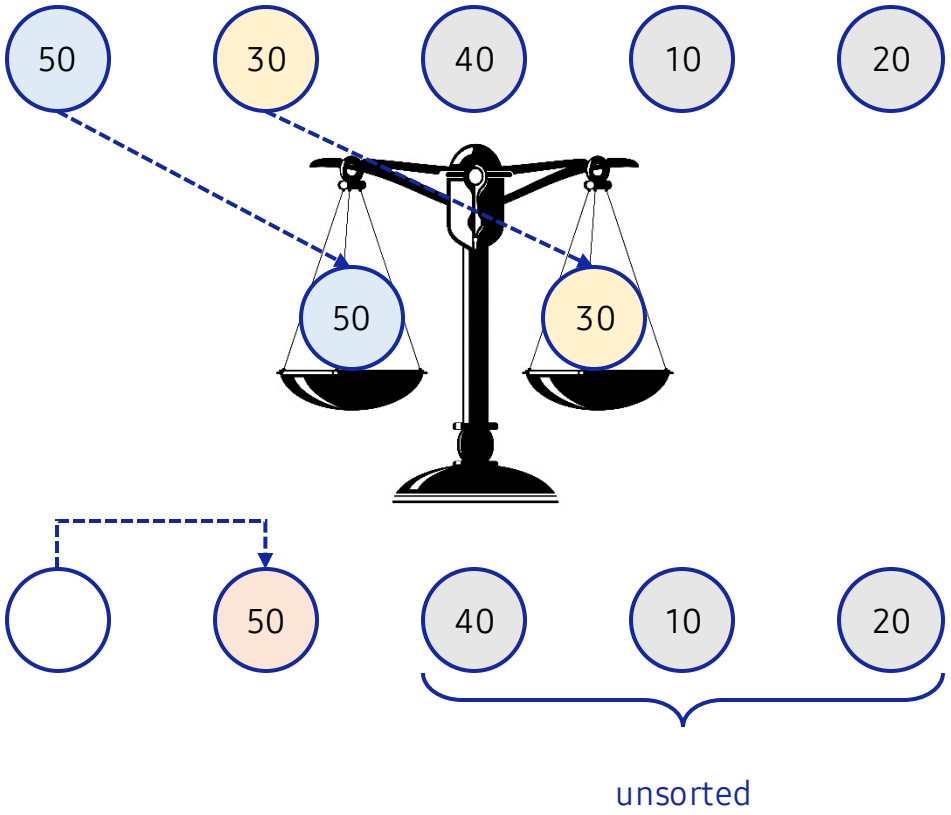
First, the first element 50 can be regarded as a sorted list because there is only one element.



3. Insertion Sort

3.2. The process of insertion sort

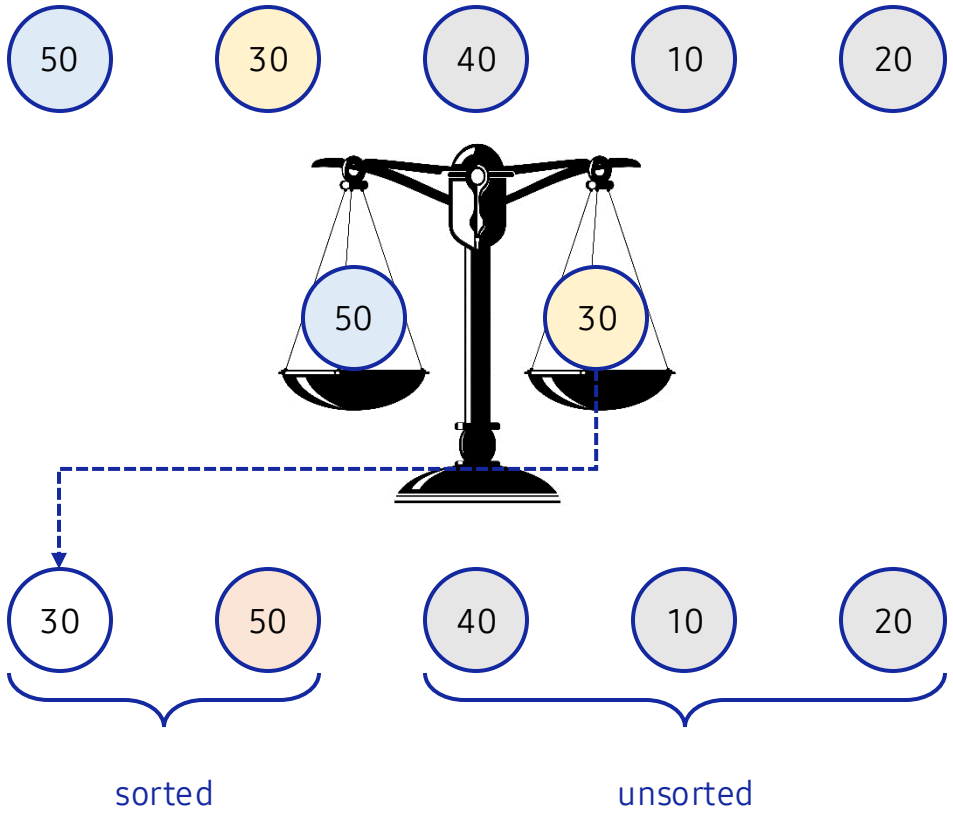
Since the second element 30 is less than 50 in the sorted list, 50 is shifted to the right by one.



3. Insertion Sort

3.2. The process of insertion sort

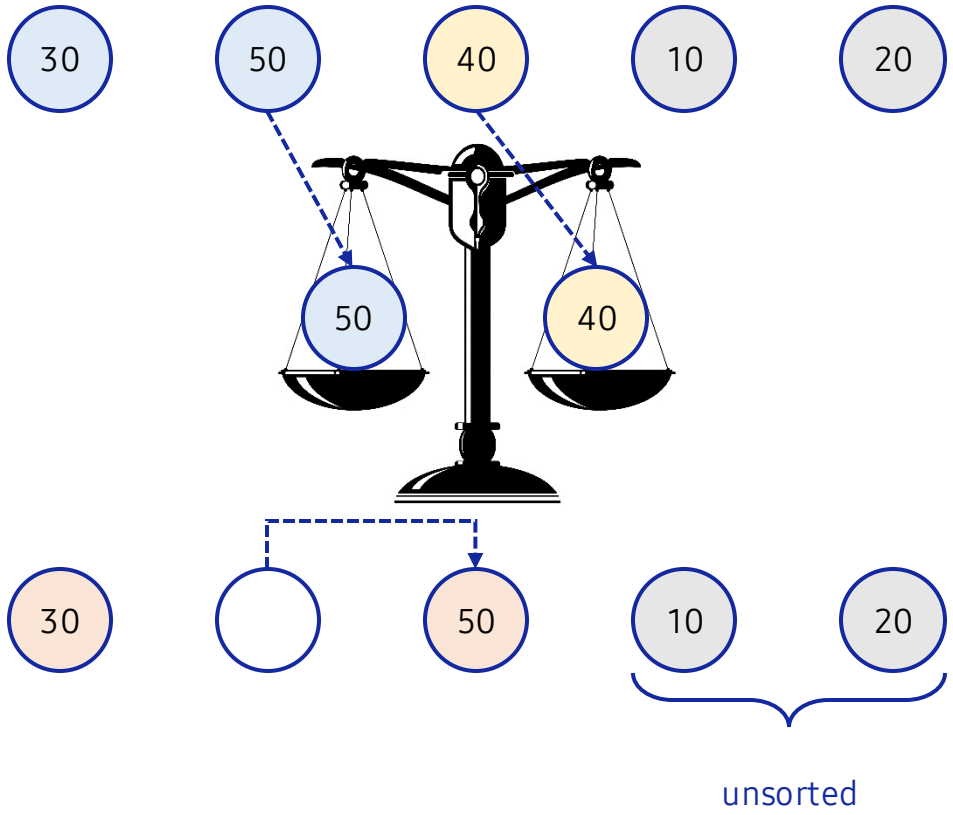
After moving 50, 30 is inserted at that position because there are no more elements to compare.



3. Insertion Sort

3.2. The process of insertion sort

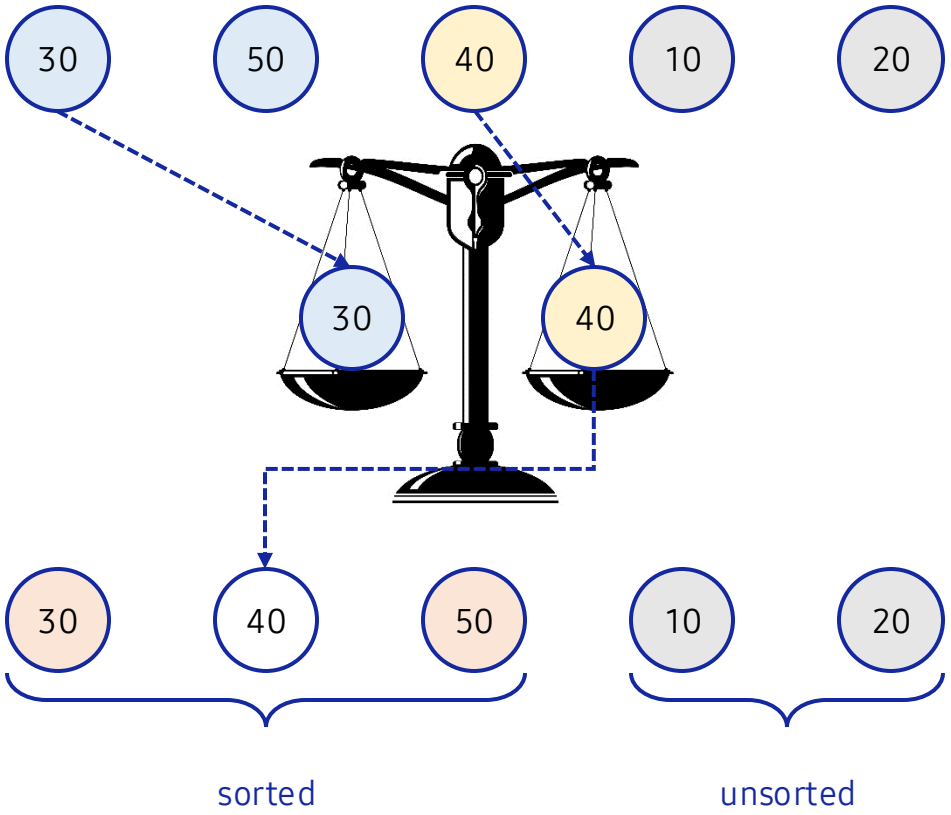
Since the third element 40 is less than 50 in the sorted list, 50 is shifted to the right by one.



3. Insertion Sort

3.2. The process of insertion sort

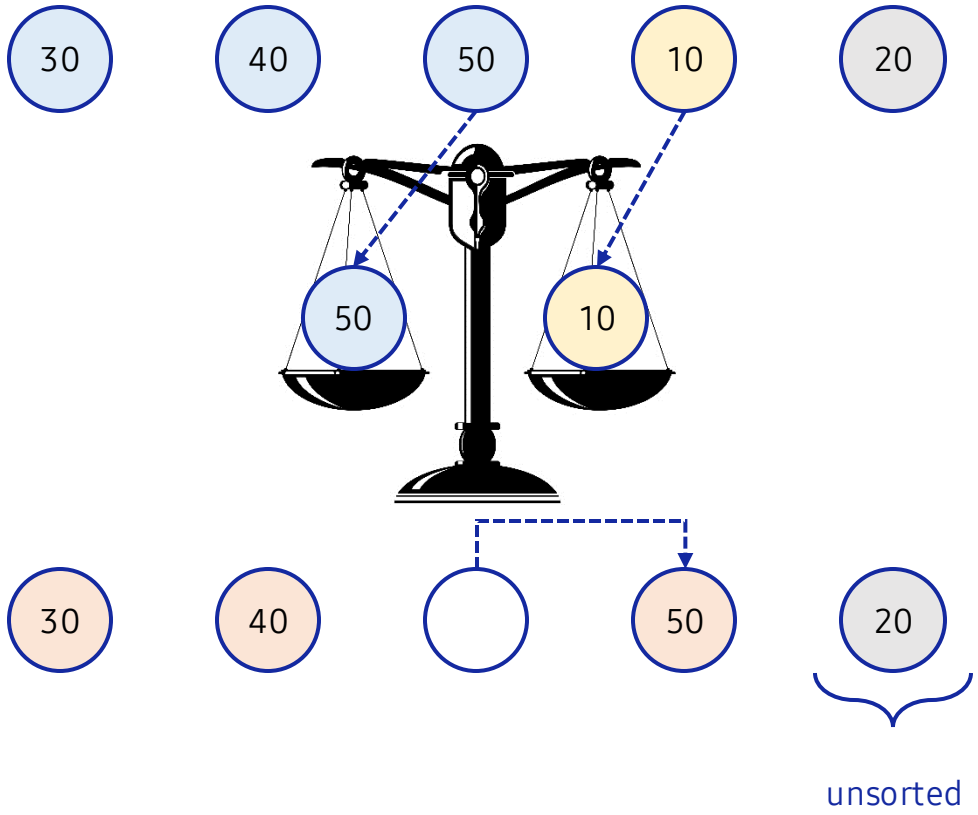
In the sorted list, since the previous element 30 is less than 40, 40 is inserted at the current position.



3. Insertion Sort

3.2. The process of insertion sort

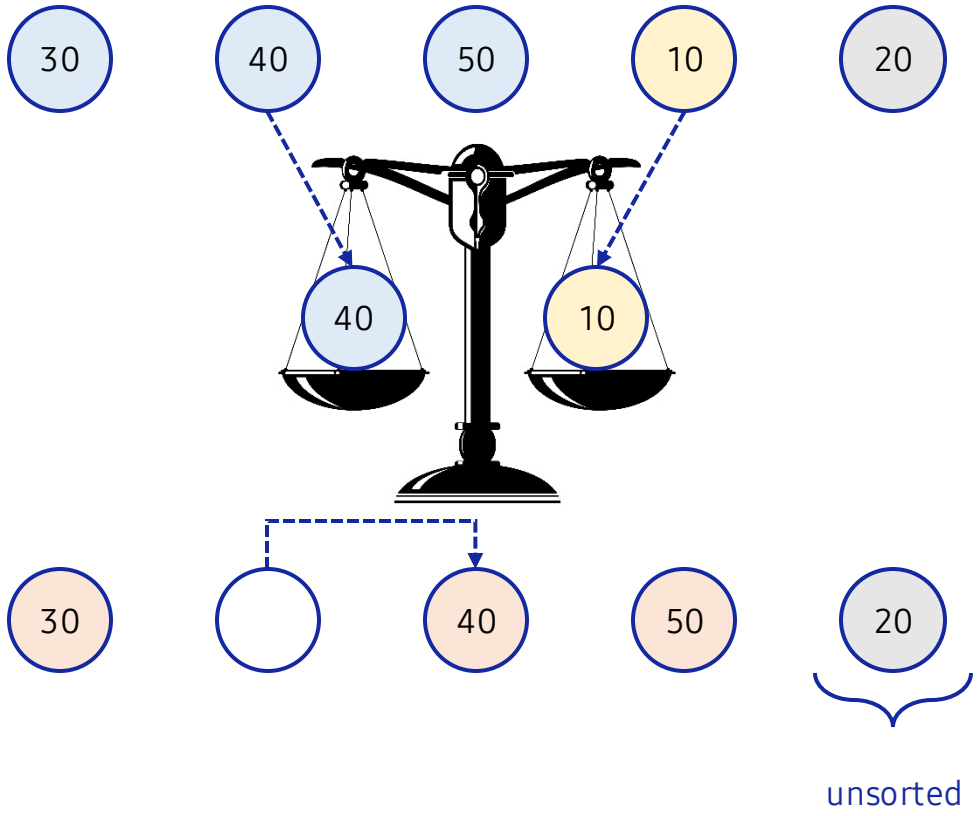
In the sorted list, since the previous element 50 is greater than 10, 50 is shifted to the right by one.



3. Insertion Sort

3.2. The process of insertion sort

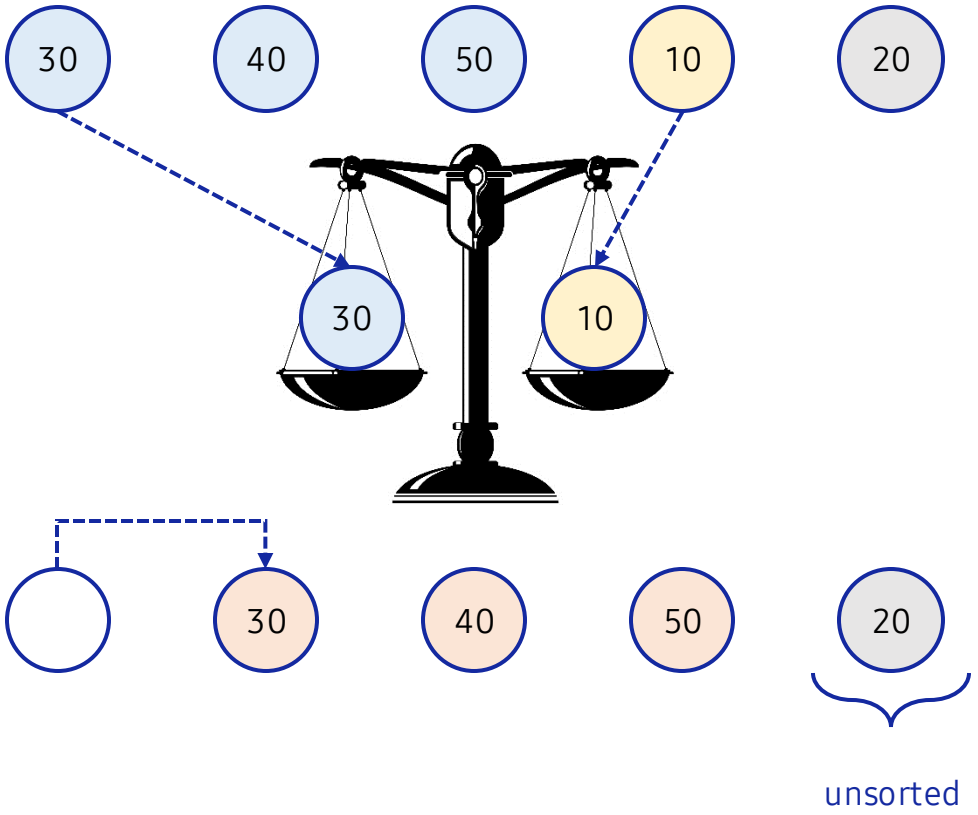
In the sorted list, since the previous element 40 is greater than 10, 40 is shifted to the right by one.



3. Insertion Sort

3.2. The process of insertion sort

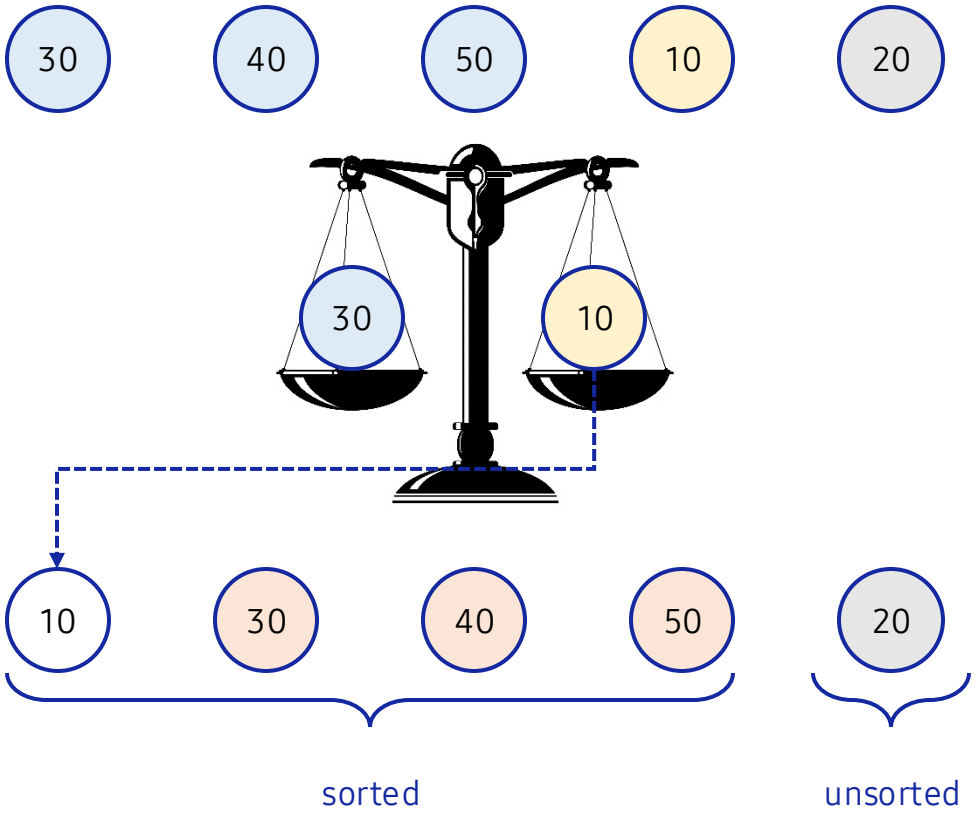
In the sorted list, since the previous element 30 is greater than 10, 30 is shifted to the right by one.



3. Insertion Sort

3.2. The process of insertion sort

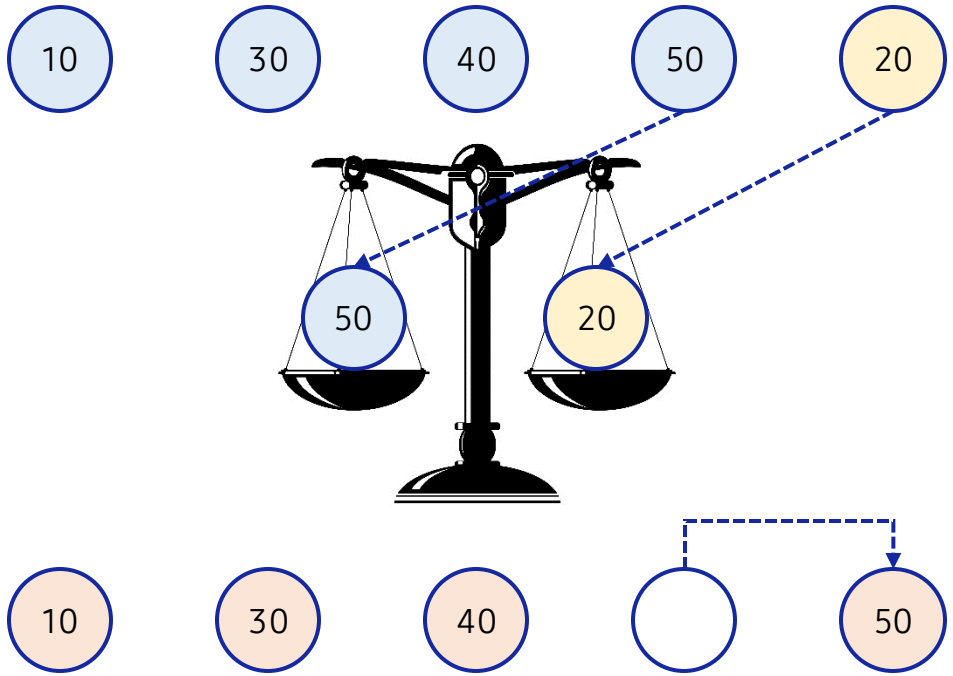
10 is inserted at that position because there are no more elements left to compare in the sorted list.



3. Insertion Sort

3.2. The process of insertion sort

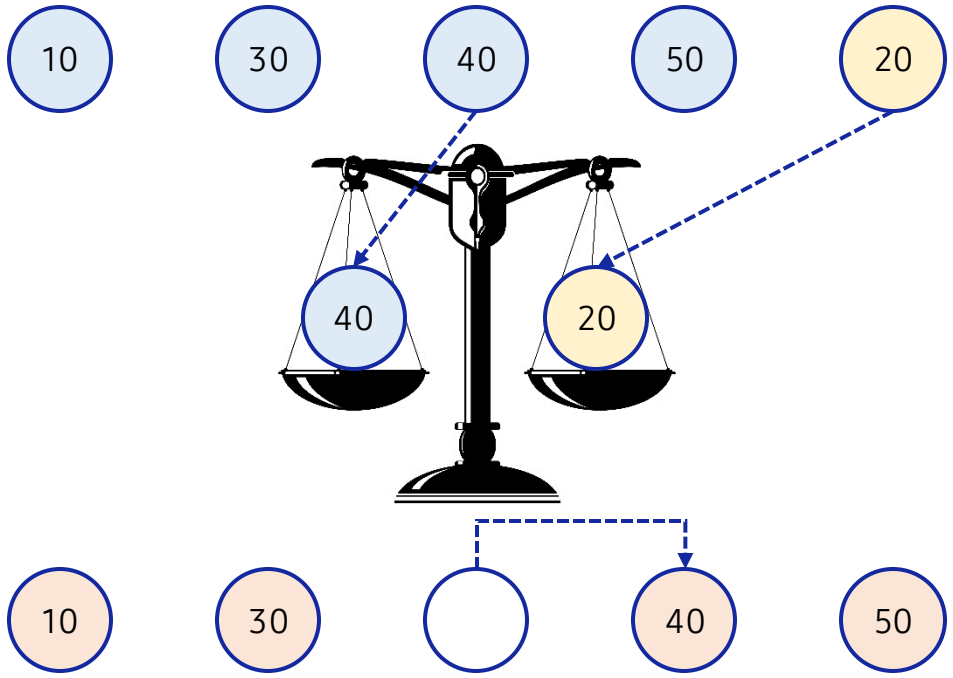
Since the last element 20 is less than 50 in the sorted list, 50 is shifted to the right by one.



3. Insertion Sort

3.2. The process of insertion sort

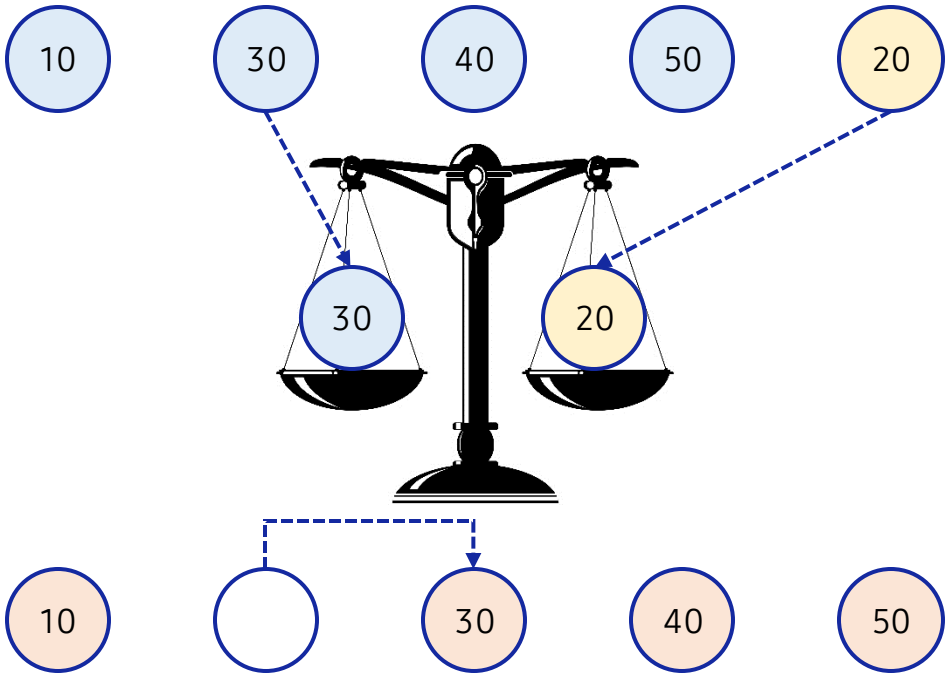
In the sorted list, since the previous element 40 is greater than 20, 40 is shifted to the right by one.



3. Insertion Sort

3.2. The process of insertion sort

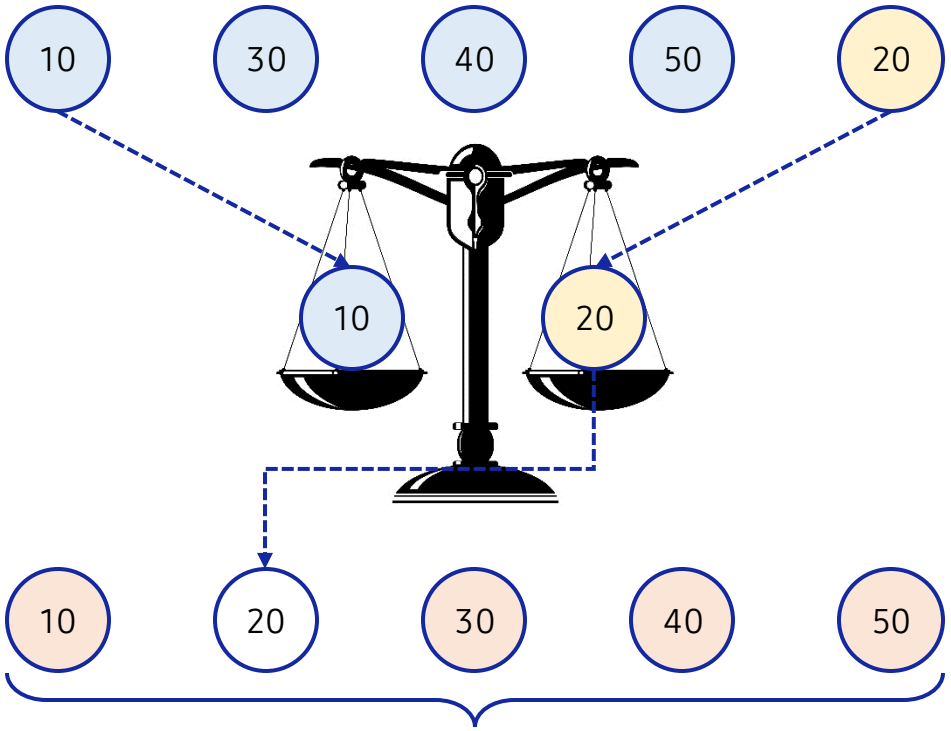
In the sorted list, since the previous element 30 is greater than 20, 30 is shifted to the right by one.



3. Insertion Sort

3.2. The process of insertion sort

In the sorted list, since the previous element 10 is less than 20, 20 is inserted at that position. Note that since we have inserted all the elements, the entire list is a sorted list.



Let's code

1. Swap Operation

1.1. Swap: exchange the values of two elements

The two basic operations in the sorting algorithm are comparison and exchange operations. The exchange operation can be implemented as the following function.

```
1 def swap1(S, x, y):
2     t = S[x]
3     S[x] = S[y]
4     S[y] = t
```

```
5 5 = [10, 20]
6 6rint(S)
7 7wap1(S, 0, 1)
8 8rint(S)
```

```
[10, 20]
[20, 10]
```

Line 1~4

- The swap() function takes a list S and the positions x and y of the list elements as input parameters.
- Store the value of S[x] into a temporary variable t and store the value of S[y] into S[x].
- The value of S[x] stored in the t variable as the temporary storage is stored in S[y].

1. Swap Operation

1.2. Swap in Python

In Python, you can store values in multiple variables at the same time, so you can exchange the values of two variables more simply as follows.

```
1  def swap2(S, x, y):
2      S[x], S[y] = S[y], S[x]
```

```
3  S = [10, 20]
4  print(S)
5  swap2(S, 0, 1)
6  print(S)
```

```
[10, 20]
[20, 10]
```

Line 1~2

- Substitute the values of S[y] and S[x] into S[x] and S[y], respectively.
- At this time, it should be noted that the two values can be exchanged because the value on the right is converted into a tuple and assigned to the value on the left.

2. Bubble Sort

2.1. Implementing bubble sort

Bubble sort is an algorithm that exchanges two adjacent elements by checking them as follows.

```
1  def bubblesort(S):
2      n = len(S)
3      for i in range(n):
4          print(S)
5          for j in range(n - 1):
6              if S[j] > S[j + 1]:
7                  S[j], S[j + 1] = S[j + 1], S[j]
```

Line 3~7

- For every i, the values of adjacent elements are compared for the element j to the right of i.
- If two adjacent elements are out of order, the two values are swapped.

2. Bubble Sort

2.2. Running bubble sort

Bubble sort is an algorithm that exchanges two adjacent elements by checking them as follows.

```
1  def bubblesort(S):
2      n = len(S)
3      for i in range(n):
4          print(S)
5          for j in range(n - 1):
6              if S[j] > S[j + 1]:
7                  S[j], S[j + 1] = S[j + 1], S[j]
```

```
8  S = [50, 30, 40, 10, 20]
9  bubblesort(S)
10 print(S)
```

```
[50, 30, 40, 10, 20]
[30, 40, 10, 20, 50]
[30, 10, 20, 40, 50]
[10, 20, 30, 40, 50]
[10, 20, 30, 40, 50]
[10, 20, 30, 40, 50]
```

Line 4

- To check the operation of bubble sort, the state of S is output every time the i-th loop is executed.

3. Selection Sort

3.1. Implementing selection sort without swap

Selection sort is an algorithm that selects the smallest value from the list and adds it to the sorted list as follows.

```
1  def selectionsort1(S):
2      R = []
3      while len(S) > 0:
4          print(R, S)
5          smallest = S.index(min(S))
6          R.append(S[smallest])
7          S.pop(smallest)
8      return R
```

Line 1~8

- It takes an unsorted list S as an input parameter and returns a sorted list R.
- Find the largest element in S, delete it, add the deleted element to R, and repeat until S becomes an empty list.

3. Selection Sort

3.2. Running selection sort without swap

Check the operation of the selection sort algorithm. They are added to R in order of the smallest element in S.

```
1  def selectionsort1(S):
2      R = []
3      while len(S) > 0:
4          print(R, S)
5          smallest = S.index(min(S))
6          R.append(S[smallest])
7          S.pop(smallest)
8      return R
```

```
9  S = [50, 30, 40, 10, 20]
10 R = selectionsort1(S)
11 print(R)
```

```
[] [50, 30, 40, 10, 20]
[10] [50, 30, 40, 20]
[10, 20] [50, 30, 40]
[10, 20, 30] [50, 40]
[10, 20, 30, 40] [50]
[10, 20, 30, 40, 50]
```

Line 4

- To check the operation of selection sort, the state of S is output whenever the while loop is executed.

3. Selection Sort

3.3. Implementing In-Place selection sort

- In-Place sorting refers to a sorting algorithm that sorts through a swap operation without using additional memory. The selectionsort1() algorithm implemented earlier uses additional memory R, so it is not an in-place sort.
- In order to implement the In-Place sorting algorithm for selection sort, implement it as follows using the swap operation as shown earlier.

```
1 def selectionsort2(S):
2     n = len(S)
3     for i in range(n - 1):
4         print(S)
5         smallest = i
6         for j in range(i + 1, n):
7             if S[j] < S[smallest]:
8                 smallest = j
9         S[i], S[smallest] = S[smallest], S[i]
```

Line 1~9

- In Line 6-8, find the position of the smallest element after the i-th element.
- In Line 9, the i-th element and the element of the smallest position are exchanged.
- In the outside for-loop, i is repeated from the first element of the list to the last element.

3. Selection Sort

3.4. Running In-Place selection sort

Check the operation of the selection sort algorithm. They are sorted in the order of the smallest element without using additional memory.

```
1  def selectionsort2(S):
2      n = len(S)
3      for i in range(n - 1):
4          print(S)
5          smallest = i
6          for j in range(i + 1, n):
7              if S[j] < S[smallest]:
8                  smallest = j
9          S[i], S[smallest] = S[smallest], S[i]
```

```
10 S = [50, 30, 40, 10, 20]
11 selectionsort2(S)
12 print(S)
```

```
[50, 30, 40, 10, 20]
[10, 30, 40, 50, 20]
[10, 20, 40, 50, 30]
[10, 20, 30, 50, 40]
[10, 20, 30, 40, 50]
```

Line 4

- To check the operation of selection sort, the state of S is output every time the i-th loop is executed.

4. Insertion Sort

4.1. Implementing insertion sort with additional memory

Insertion sort is a sorting algorithm that adds a new element to the correct position in the sorted list as follows.

```
1  def insertionsort1(S):
2      n = len(S)
3      R = []
4      while len(S) > 0:
5          print(R, S)
6          x = S.pop(0)
7          j = len(R) - 1
8          while j >=0 and R[j] > x:
9              j -= 1
10         R.insert(j + 1, x)
11     return R
```

Line 1~11

- It takes an unsorted list S as an input parameter and returns a sorted list R.
- We take elements from S one by one, move forward from the last element of R, and add them to the position where the element is to be inserted.

4. Insertion Sort

4.2. Running insertion sort with additional memory

Check the operation of the insertion sort algorithm. Add elements of S to R in sorted order.

```
1  def insertionsort1(S):
2      n = len(S)
3      R = []
4      while len(S) > 0:
5          print(R, S)
6          x = S.pop(0)
7          j = len(R) - 1
8          while j >=0 and R[j] > x:
```

```
9  S = [50, 30, 40, 10, 20]
10 insertionsort1(S)
11 print(R)
```

```
[] [50, 30, 40, 10, 20]
[50] [30, 40, 10, 20]
[30, 50] [40, 10, 20]
[30, 40, 50] [10, 20]
[10, 30, 40, 50] [20]
[10, 20, 30, 40, 50]
```

Line 5

- In order to check the operation of insertion sort, the state of S is output whenever the while loop is executed.

4. Insertion Sort

4.3. Implementing In-Place insertion sort

In order to implement insertion sort as an in-Place sorting algorithm, it can be implemented while moving elements one by one, as shown earlier.

```
1  def insertionsort2(S):
2      n = len(S)
3      for i in range(1, n):
4          print(S)
5          x = S[i]
6          j = i - 1
7          while j >= 0 and S[j] > x:
8              S[j + 1] = S[j]
9              j -= 1
10         S[j + 1] = x
```

Line 1~10

- In Line 5, the i-th element is temporarily stored in x.
- In Line 6-9, decrease j while moving the element until it encounters an element greater than x, and then store x in the corresponding position.
- In the outside for-loop, i is repeated from the first element of the list to the last element.

4. Insertion Sort

4.4. Running In-Place insertion sort

Check the operation of the insertion sort algorithm. It places an element of S in it's right position by insertion.

```
1  def insertionsort2(S):
2      n = len(S)
3      for i in range(1, n):
4          print(S)
5          x = S[i]
6          j = i - 1
7          while j >= 0 and S[j] > x:
8              S[j + 1] = S[j]
```

```
9  S = [50, 30, 40, 10, 20]
10 insertionsort2(S)
11 print(S)
```

```
[50, 30, 40, 10, 20]
[30, 50, 40, 10, 20]
[30, 40, 50, 10, 20]
[10, 30, 40, 50, 20]
[10, 20, 30, 40, 50]
```

Line 4

- In order to check the operation of insertion sort, the state of S is output every time the i-th for loop is executed.

One More Step

- What is the time complexity of the sorting algorithm?
- The bubble sort algorithm executes a double nested for-loop, and the number of comparisons for each pass is N – 1, N – 2, N – 3, ... , 0.
- The number of comparisons in the bubble sort algorithm is T(N) = (N – 1) + (N - 2) + ... + 1 = N(N-1)/2.
- The time complexity of the bubble sort algorithm is $O(N^2)$.
- Since selection sort executes a double for-loop like bubble sort, the time complexity is $O(N^2)$ for the same reason.
- In the case of insertion sort, the number of comparisons differs between the worst case and the best case.
- The best case is an already sorted array. In this case, the time complexity becomes O(N) because the insertion position can be found only by one comparison for each pass.
- The worst case is an array sorted in reverse order. In this case, the time complexity is $O(N^2)$ as in bubble/selection sort, because the insertion position must be compared with all elements of the sorted list for each pass.

Pop quiz

Q1. How many swap operations were executed in the bubble sort process below?

```
1  def bubblesort(S):
2      n = len(S)
3      for i in range(n):
4          print(S)
5          for j in range(n - 1):
6              if S[j] > S[j + 1]:
7                  S[j], S[j + 1] = S[j + 1], S[j]
```

```
1  S = [50, 30, 40, 10, 20]
2  bubblesort(S)
3  print(S)
```

```
[50, 30, 40, 10, 20]
[30, 40, 10, 20, 50]
[30, 10, 20, 40, 50]
[10, 20, 30, 40, 50]
[10, 20, 30, 40, 50]
[10, 20, 30, 40, 50]
```

Q2. How many comparison operations were executed in the insertion sort process below?

```
1  def insertionsort2(S):
2      n = len(S)
3      for i in range(1, n):
4          print(S)
5          x = S[i]
6          j = i - 1
7          while j >= 0 and S[j] > x:
8              S[j + 1] = S[j]
9              j -= 1
10         S[j + 1] = x
```

```
1  S = [50, 30, 40, 10, 20]
2  insertionsort2(S)
3  print(S)
```

```
[50, 30, 40, 10, 20]
[30, 50, 40, 10, 20]
[30, 40, 50, 10, 20]
[10, 30, 40, 50, 20]
[10, 20, 30, 40, 50]
```

Pair programming

Pair Programming Practice

Guideline, mechanisms & contingency plan

Preparing pair programming involves establishing guidelines and mechanisms to help students pair properly and to keep them paired. For example, students should take turns "driving the mouse." Effective preparation requires contingency plans in case one partner is absent or decides not to participate for one reason or another. In these cases, it is important to make it clear that the active student will not be punished because the pairing did not work well.

Pairing similar, not necessarily equal, abilities as partners

Pair programming can be effective when students of similar, though not necessarily equal, abilities are paired as partners. Pairing mismatched students often can lead to unbalanced participation. Teachers must emphasize that pair programming is not a "divide-and-conquer" strategy, but rather a true collaborative effort in every endeavor for the entire project. Teachers should avoid pairing very weak students with very strong students.

Motivate students by offering extra incentives

Offering extra incentives can help motivate students to pair, especially with advanced students. Some teachers have found it helpful to require students to pair for only one or two assignments.

Pair Programming Practice

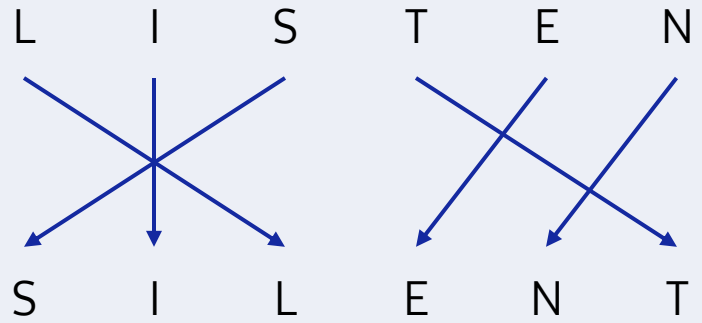
Prevent collaboration cheating

The challenge for the teacher is to find ways to assess individual outcomes, while leveraging the benefits of collaboration. How do you know whether a student learned or cheated? Experts recommend revisiting course design and assessment, as well as explicitly and concretely discussing with the students on behaviors that will be interpreted as cheating. Experts encourage teachers to make assignments meaningful to students and to explain the value of what students will learn by completing them.

Collaborative learning environment

A collaborative learning environment occurs anytime an instructor requires students to work together on learning activities. Collaborative learning environments can involve both formal and informal activities and may or may not include direct assessment. For example, pairs of students work on programming assignments; small groups of students discuss possible answers to a professor's question during lecture; and students work together outside of class to learn new concepts. Collaborative learning is distinct from projects where students "divide and conquer." When students divide the work, each is responsible for only part of the problem solving and there are very limited opportunities for working through problems with others. In collaborative environments, students are engaged in intellectual talk with each other.

Q1. Given two words, write an algorithm that determines whether these two strings are anagrams.
An anagram is a word formed by rearranging the letters of a different word using all the original letters exactly once. (ex. “LISTEN” and “SILENT” are anagrams.)



Q2. Using a sorting algorithm makes it easy to determine if it is an anagram or not.

- Create a function that evaluates anagrams using Python's built-in sorted() function.
- Modify the selection_sort2() function to create a function that determines anagrams.
- Modify the insertion_sort2() function to create a function that determines the anagram.

```
print(is_anagram("listen", "silent"))
print(is_anagram("anagram", "nagaram"))
print(is_anagram("listen", "silence"))
print(is_anagram("anagram", "anagrams"))

```

```
True
True
False
False
```

Unit 28.

Merge Sort

Unit learning objective (1/3)

Learning objectives

- ✓ Understand merge sort algorithm and be able to solve sorting problems using merge sort.
- ✓ Understand and be able to explain the difference between merge sort, bubble sort, selection sort, and insertion sort.
- ✓ Understand and be able to explain the time complexity of merge sort.

Unit learning objective (2/3)

Learning overview

- ✓ Implement merge sort that divides the given unsorted list, sorts each, and merges them back together.
- ✓ Implement a merge function that converts two sorted lists into one sorted list for merge sort.
- ✓ Understand that merge sort algorithm is divide-and-conquer using recursive functions.

Concepts you will need to know from previous units

- ✓ Using comparison operators to compare two numbers.
- ✓ Dividing the given problem and call the recursive function recursively.
- ✓ Applying Big O notation to analyze the time complexity of an algorithm.

Keywords

- Sorting Problem
- Merge Sort
- Merge

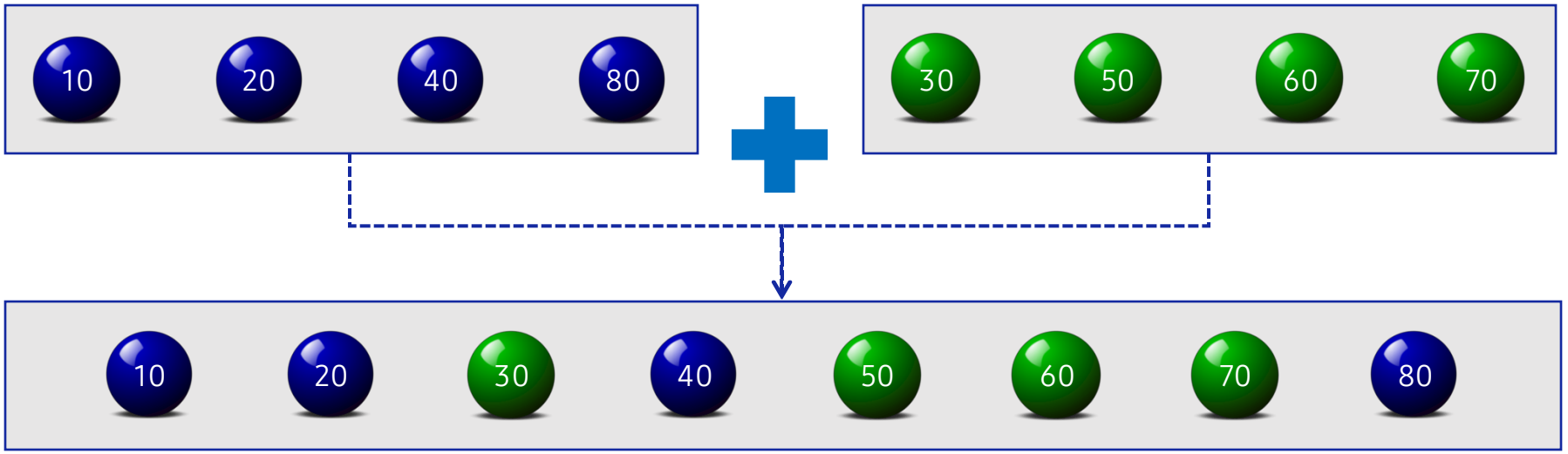
Mission

1. Real world problem

1.1 The problem of merging

In the previous unit, we studied the $O(N^2)$ sorting algorithms. Can we create a more efficient sorting algorithm than these.

Ex Suppose you are given 4 blue and 4 green iron marbles as shown below. These iron marbles are the same shape and size but have different weights. However, if these iron marbles are arranged in ascending order of weight as shown below, how can the whole be sorted?

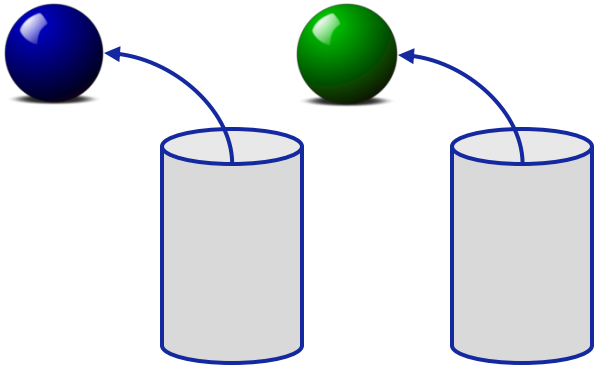


2. Mission

2.1. Merging two sorted list

- Use the balance scale. The balance scale can do the following operations:
  - Take the lightest marble from two barrels of iron marble and compare them.
  - The lighter of the two iron marbles can be placed in a new iron marble barrel.
- Knowing that the two barrels already contain sorted marbles, we can compare the two marbles with the least weight in each of the two barrels and take a strategy to place them in a new barrel.
- With this strategy, how many comparisons would it take to place all the marbles in sorted order by weight?

Sorted



Sorted Sorted

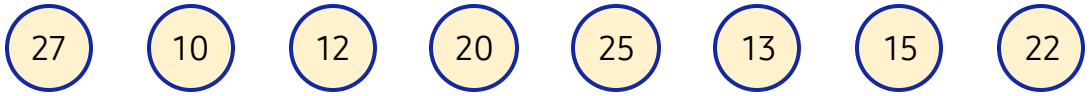
Key concept

1. Merge Sort

1.1. An example of merge sort

Merge sort is a sorting algorithm that divides an unsorted list into two sublists. sorts them, and then merges them into a sorted list.

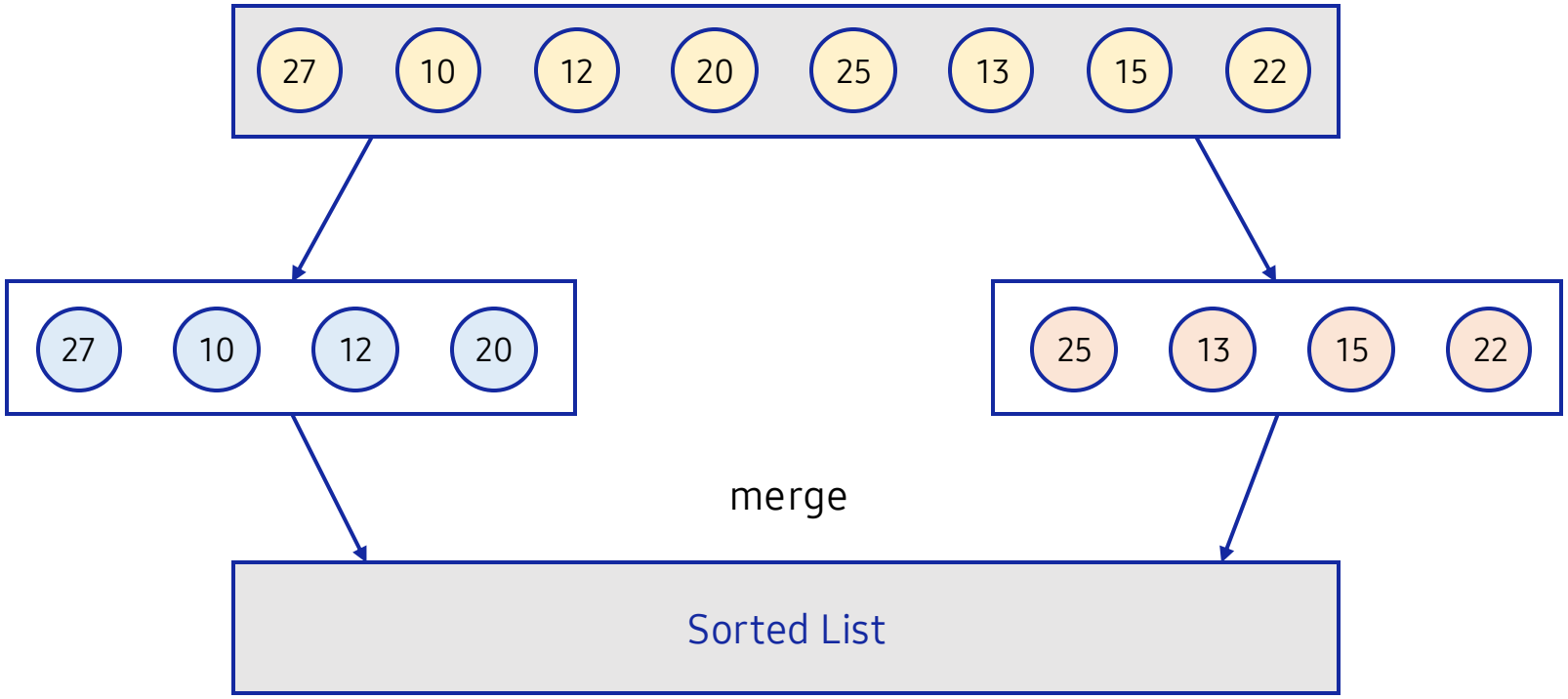
Ex Suppose we want to sort a list arranged in the order of [27, 10, 12, 20, 25, 13, 15, 22] as follows.



1. Merge Sort

1.1. An example of merge sort

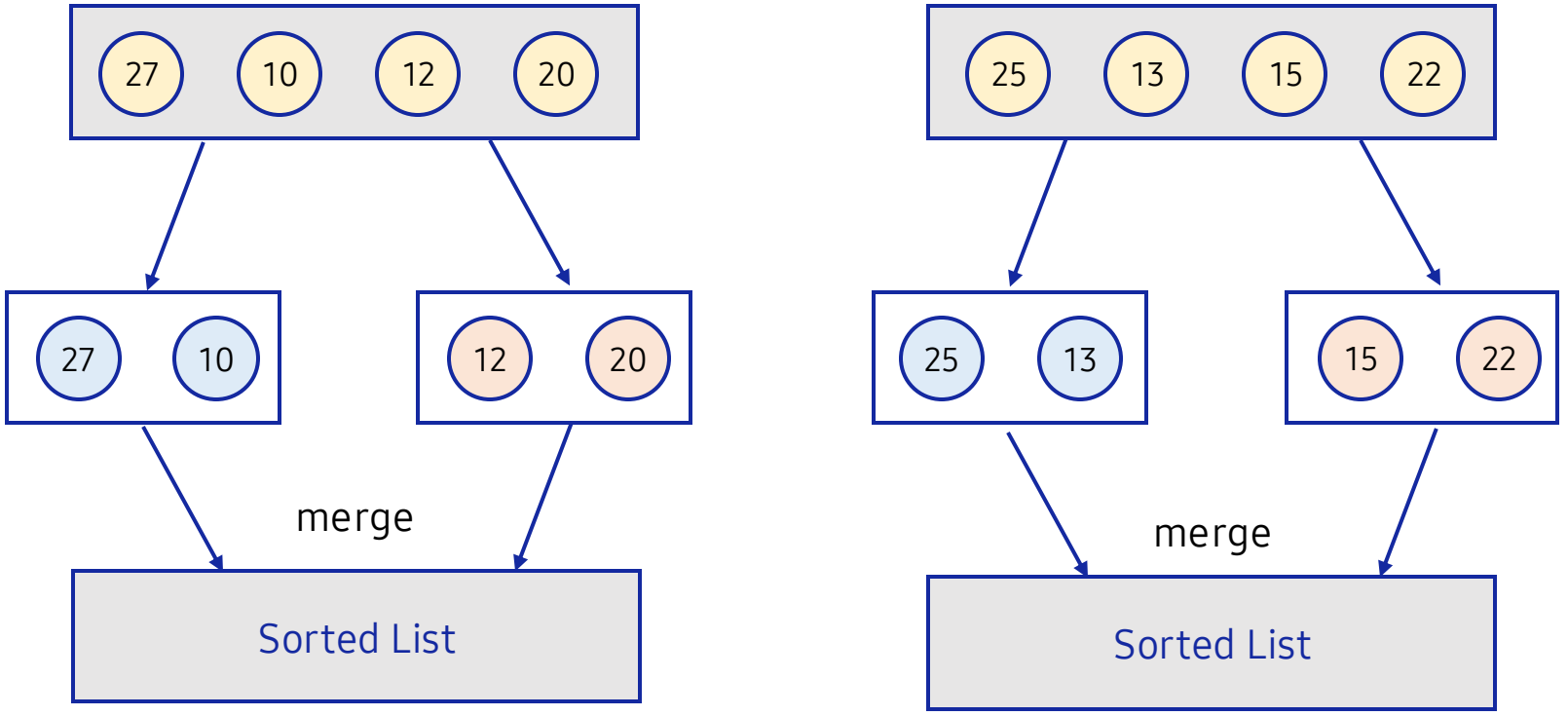
First, divide 8 elements by two to get 4 elements each. If this divides list is in a sorted state, you can merge the two divided lists in a sorted state again. We have to recursively sort the divided list.



1. Merge Sort

1.1. An example of merge sort

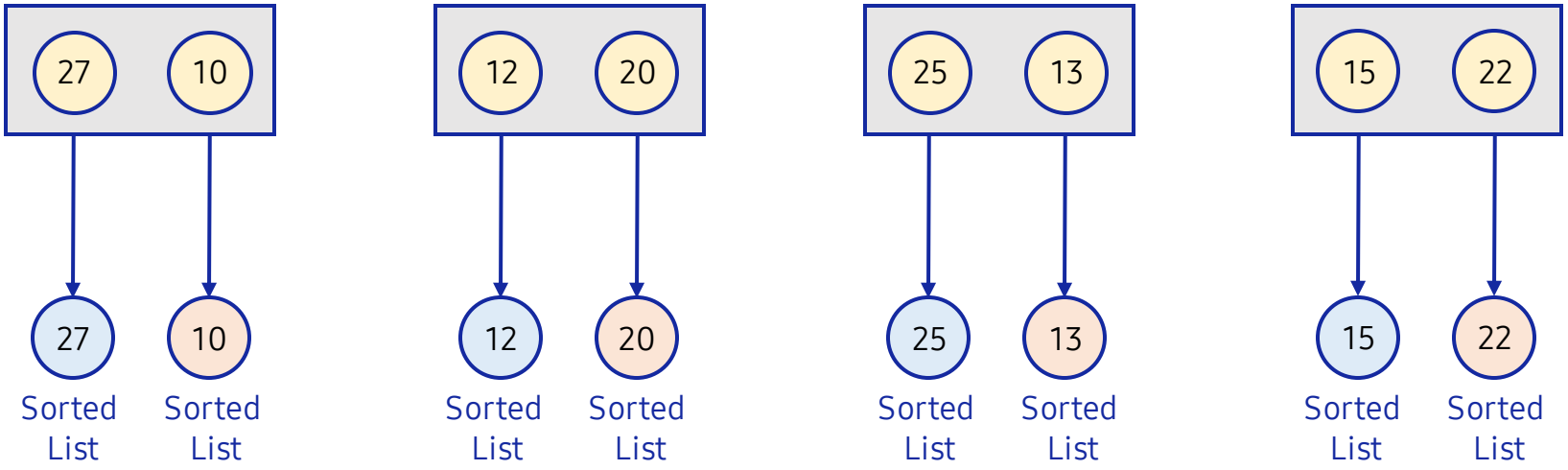
An unsorted list divided into two can be divided into two and merged into a sorted list. Since the divided list is not sorted yet, the recursive call continues.



1. Merge Sort

1.1. An example of merge sort

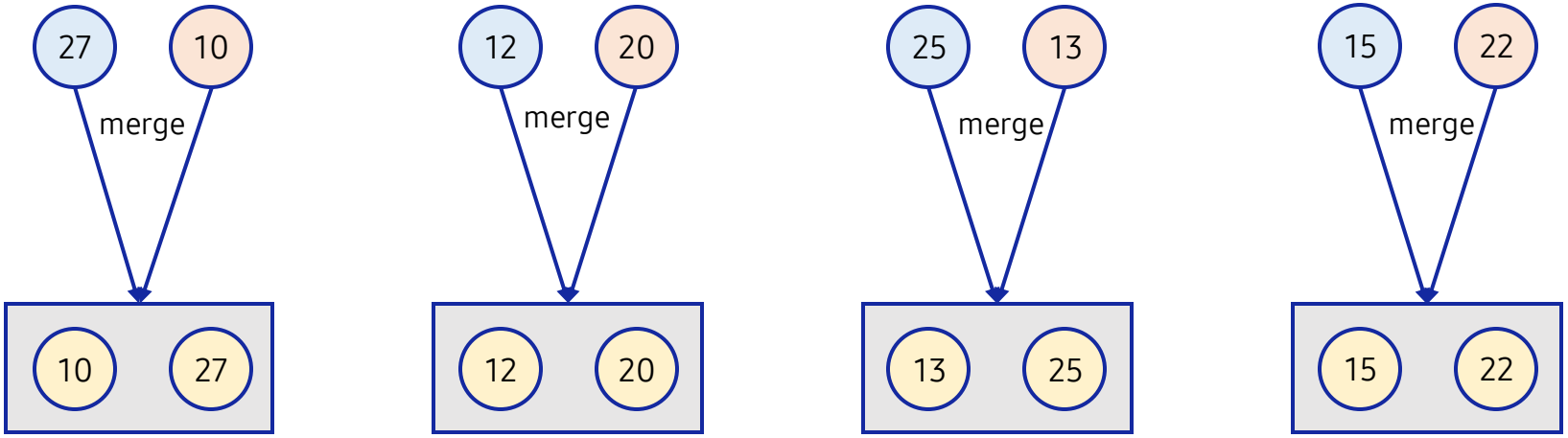
- dividing a two-element list results in a one-element list. In this case, since a list with one element is a sorted list, the lists can be merged again in a sorted state.



1. Merge Sort

1.1. An example of merge sort

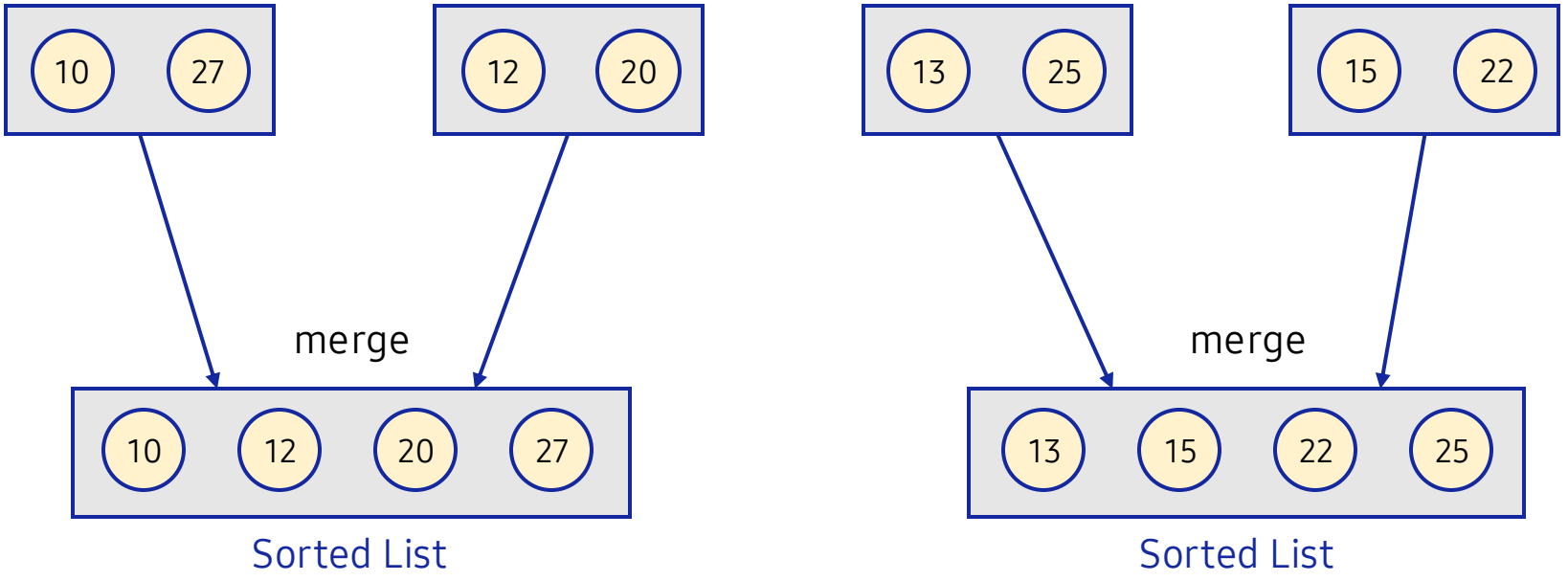
Merging one-element lists creates a two-element ordered list.



1. Merge Sort

1.1. An example of merge sort

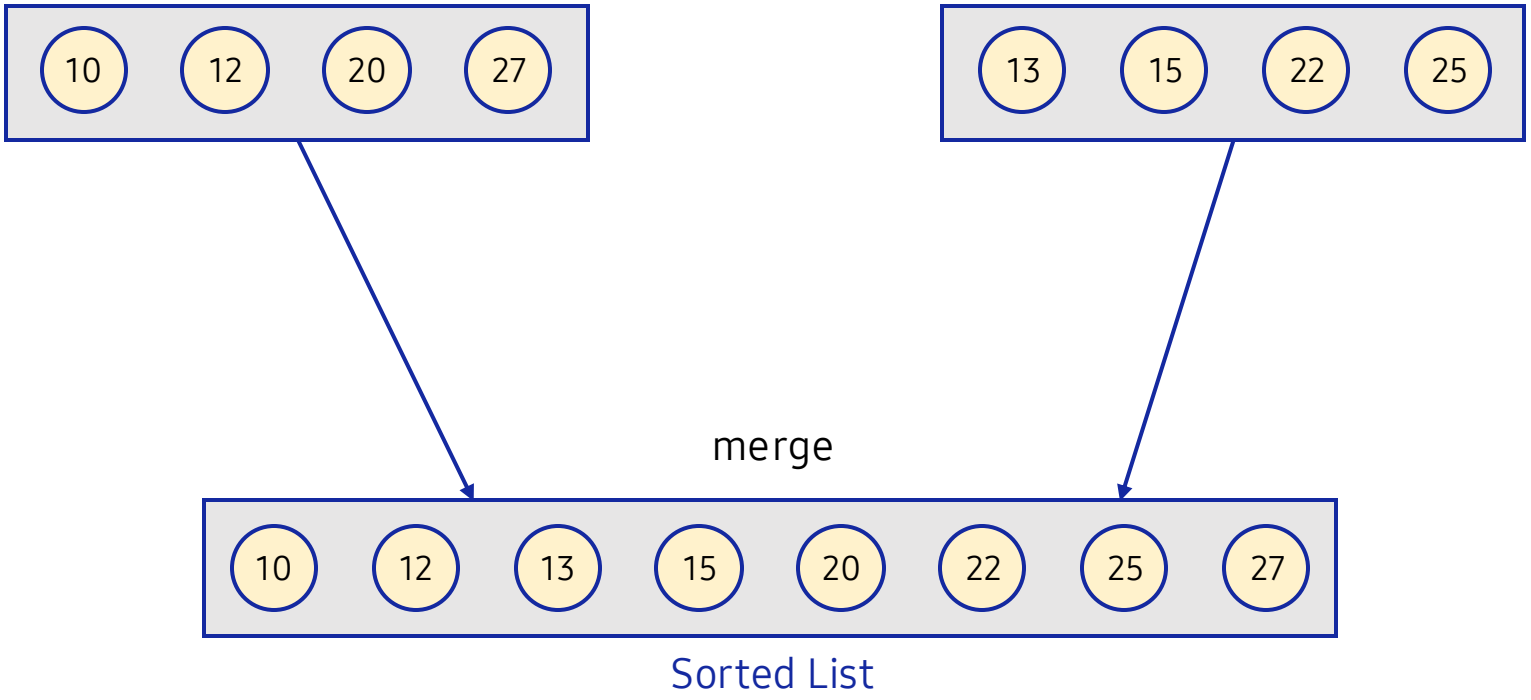
Merging two-element lists creates a four-element sorted list.



1. Merge Sort

1.1. An example of merge sort

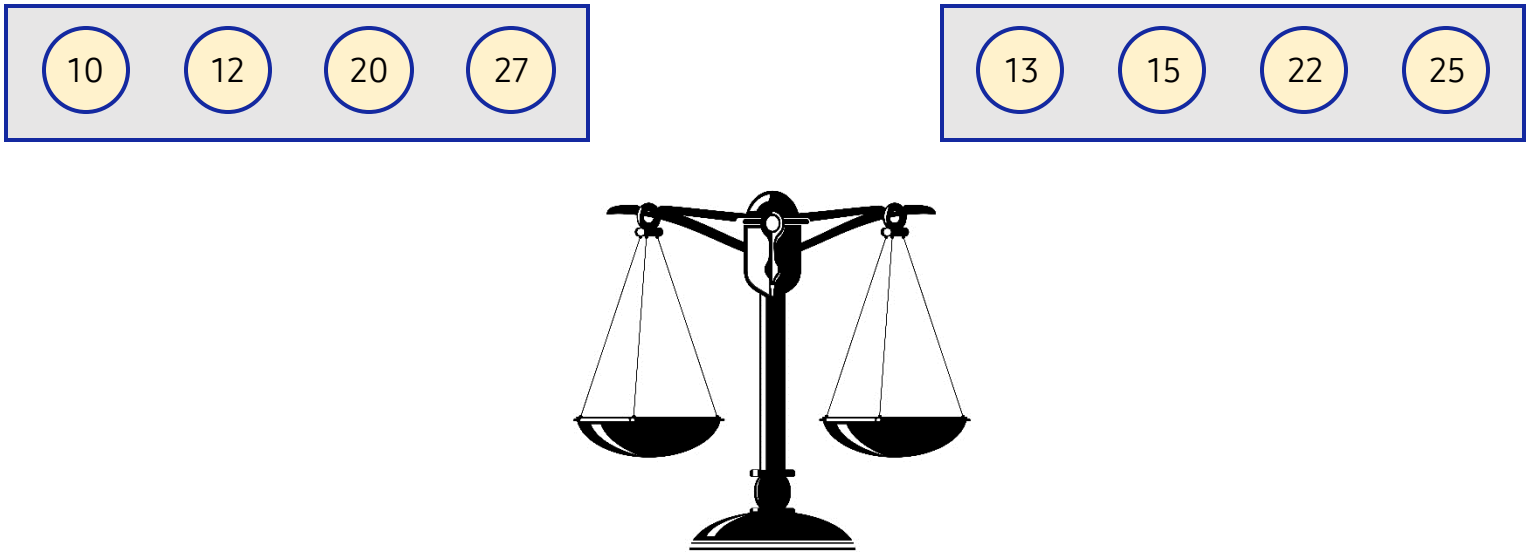
Merging a four-element list creates an eight-element sorted list.



1. Merge Sort

1.2. Merging two sorted lists

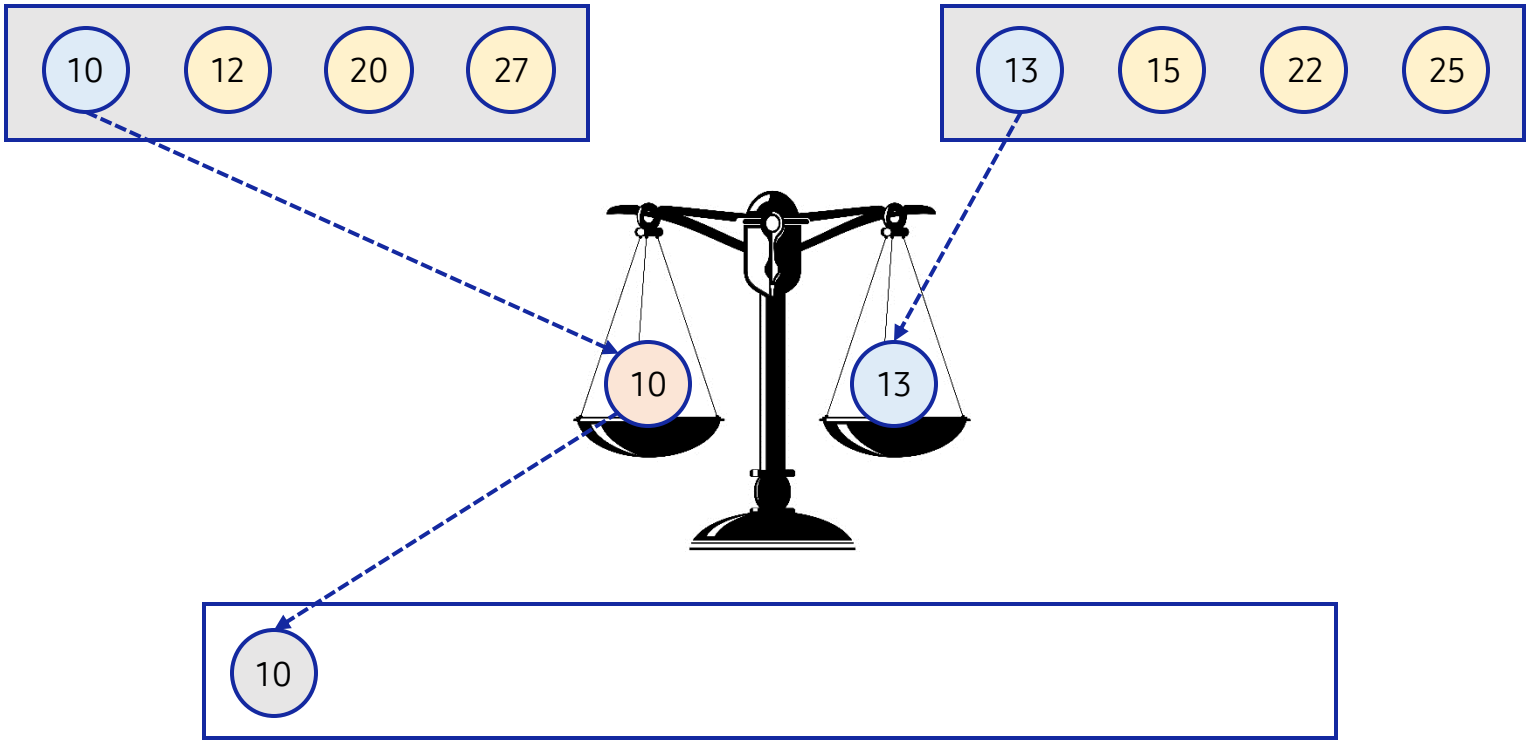
Find a method to merge two sorted lists into one sorted list. Use a balance scale.



1. Merge Sort

1.2. Merging two sorted lists

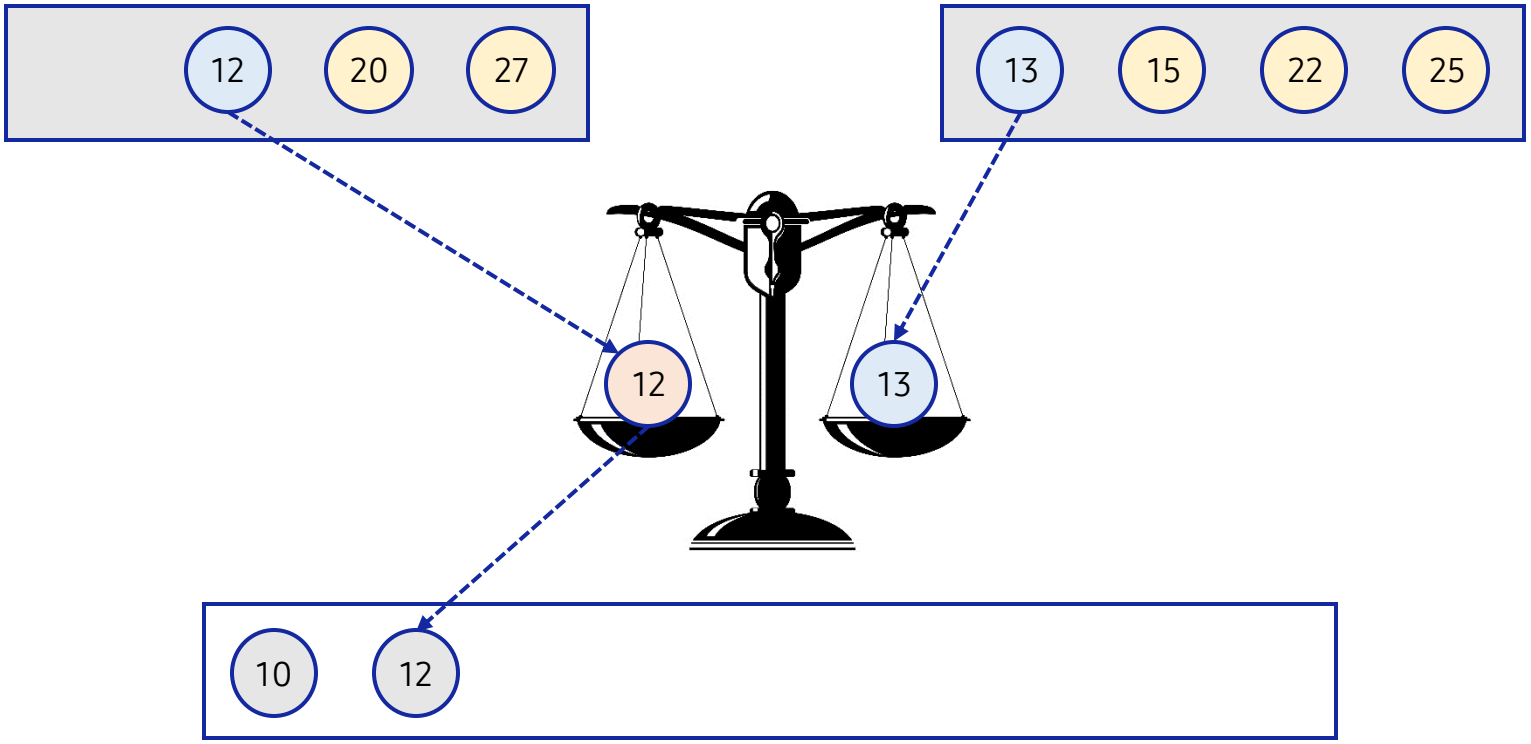
First, take the smallest element of the two sorted lists one by one and compare them. In this case, since 10 is less than 13, 10 is added to the sorted list first.



1. Merge Sort

1.2. Merging two sorted lists

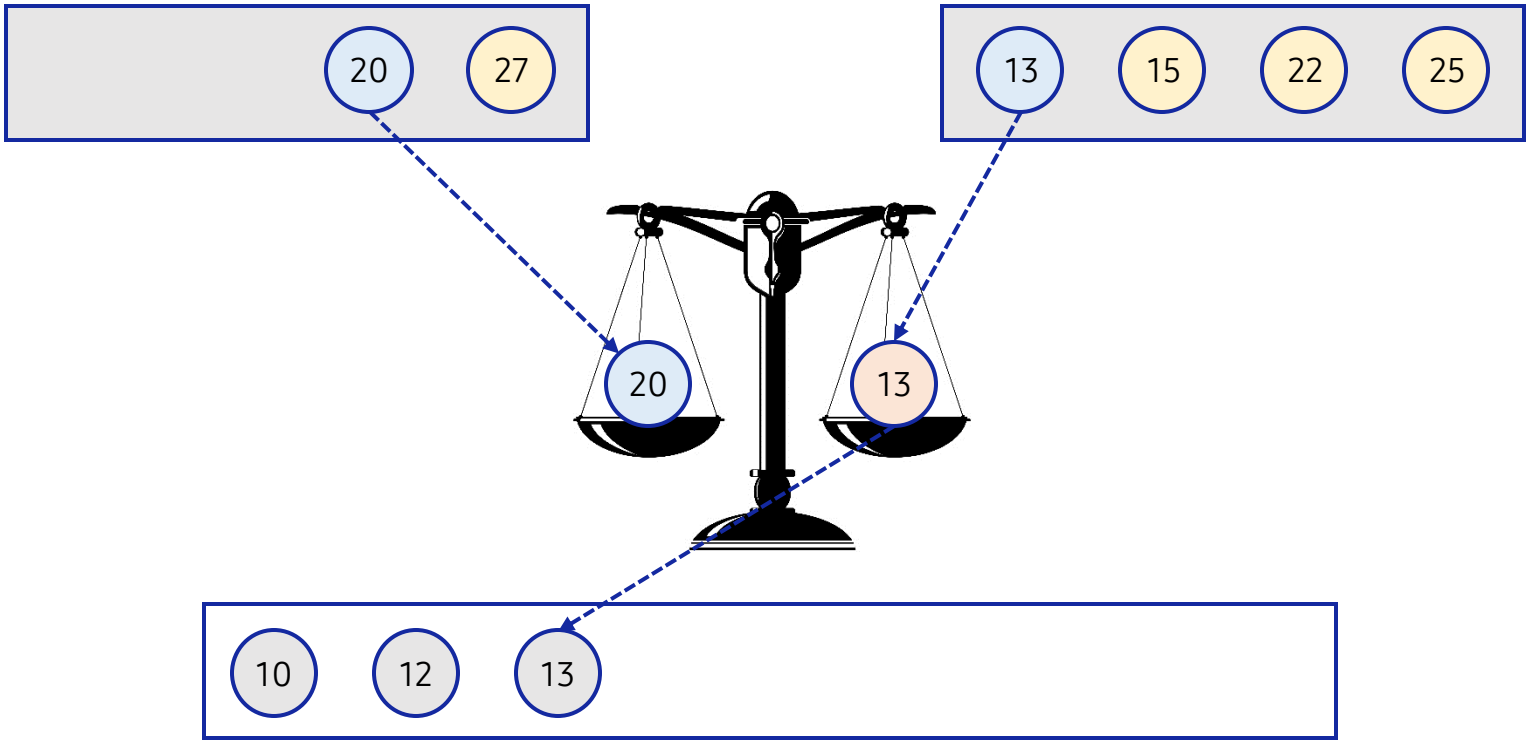
- Remove the element sent to the sorted list and compare the next element. Since 12 is less than 13, 12 is added to the sorted list.



1. Merge Sort

1.2. Merging two sorted lists

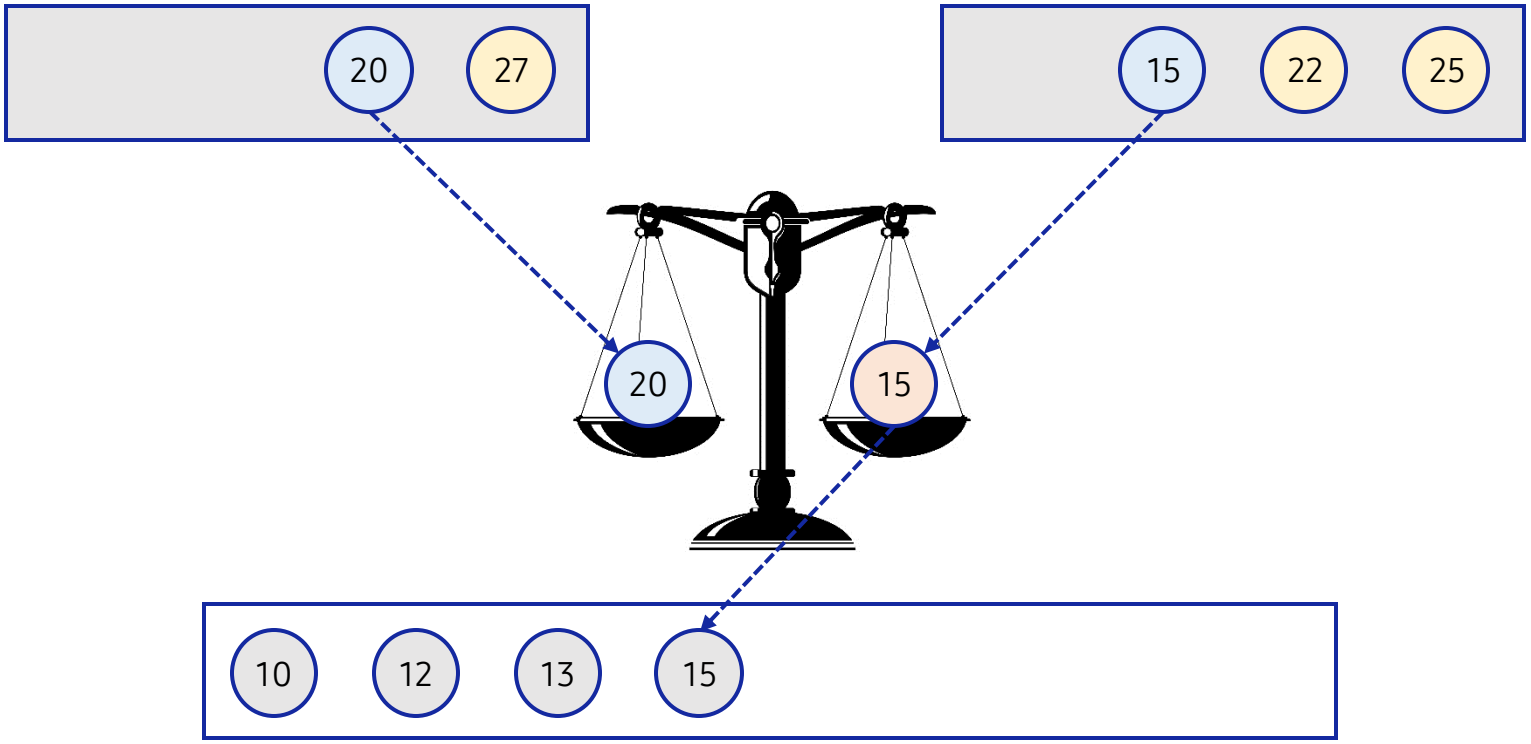
- Remove the element sent to the sorted list and compare the next element. Since 20 is greater than 13, 13 is added to the sorted list.



1. Merge Sort

1.2. Merging two sorted lists

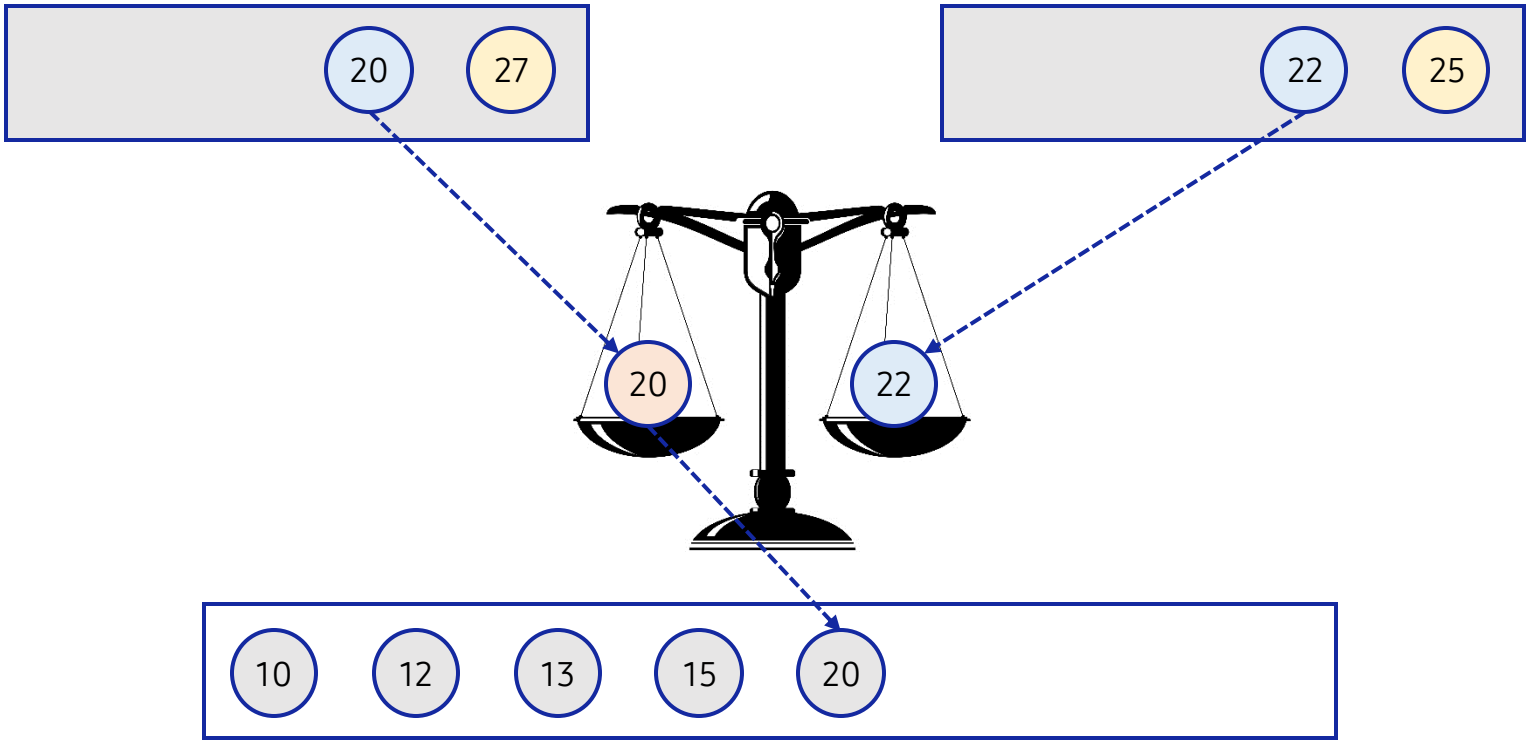
- Remove the element sent to the sorted list and compare the next element. Since 15 is less than 20, 15 is added to the sorted list.



1. Merge Sort

1.2. Merging two sorted lists

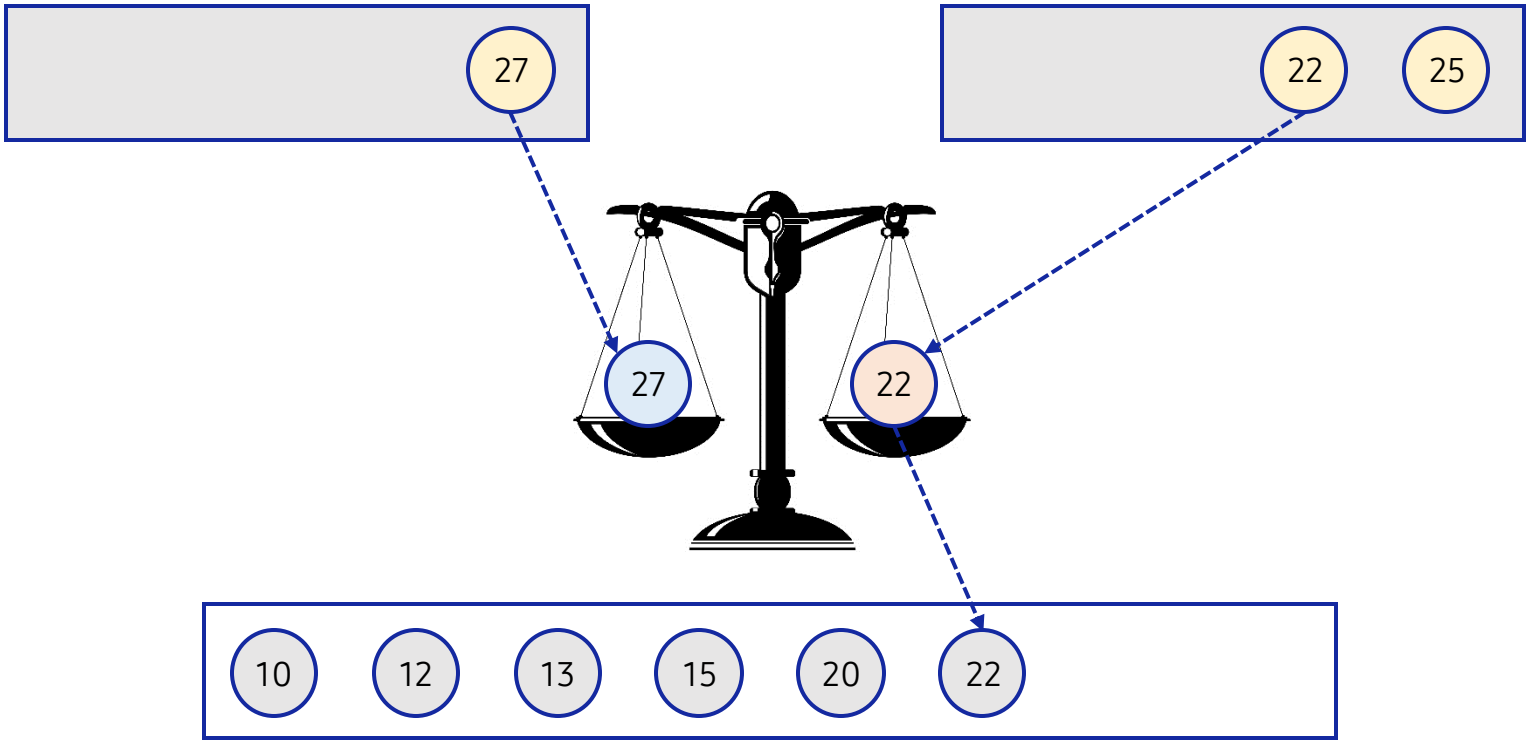
Remove the element sent to the sorted list and compare the next element. Since 20 is less than 22, 20 is added to the sorted list.



1. Merge Sort

1.2. Merging two sorted lists

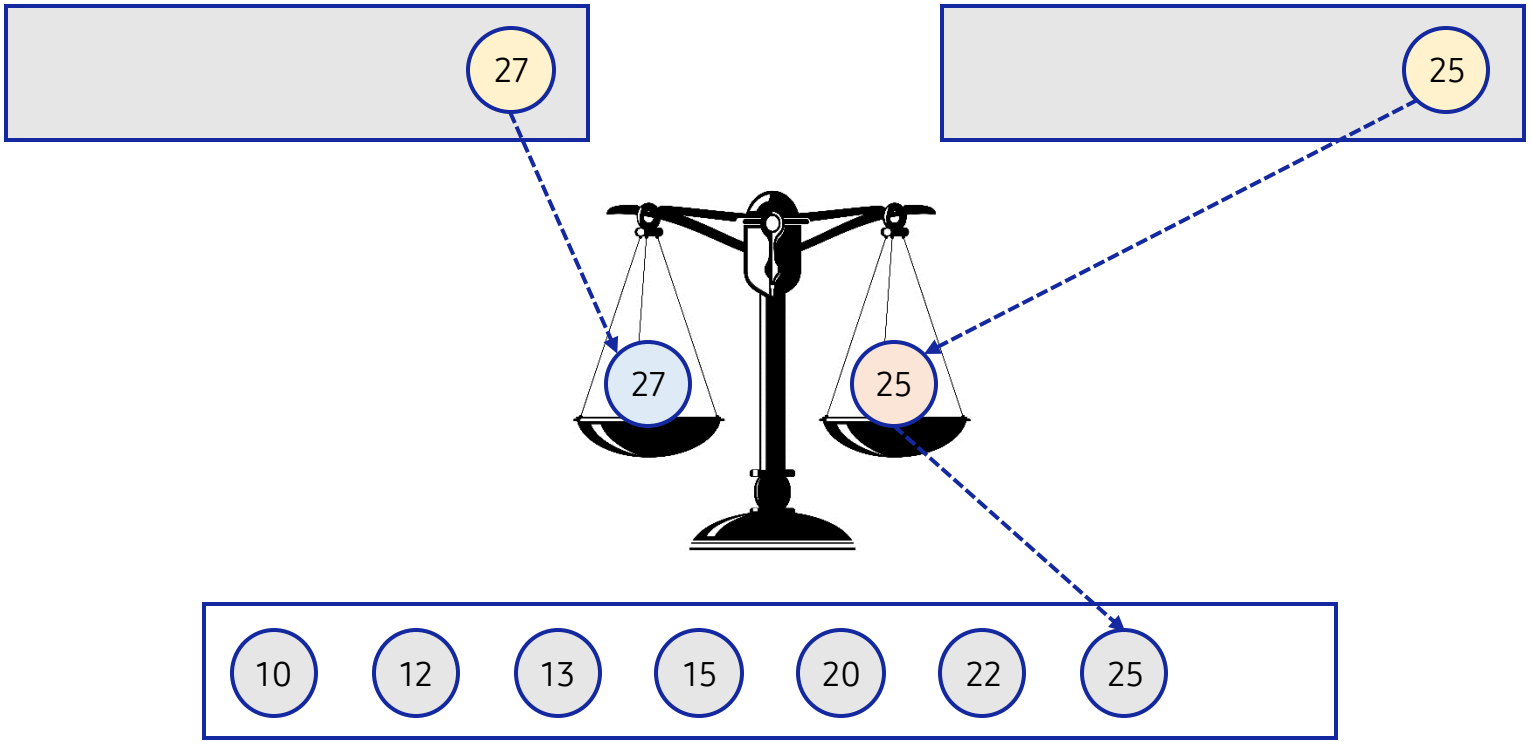
- Remove the element sent to the sorted list and compare the next element. Since 22 is less than 27, 22 is added to the sorted list.



1. Merge Sort

1.2. Merging two sorted lists

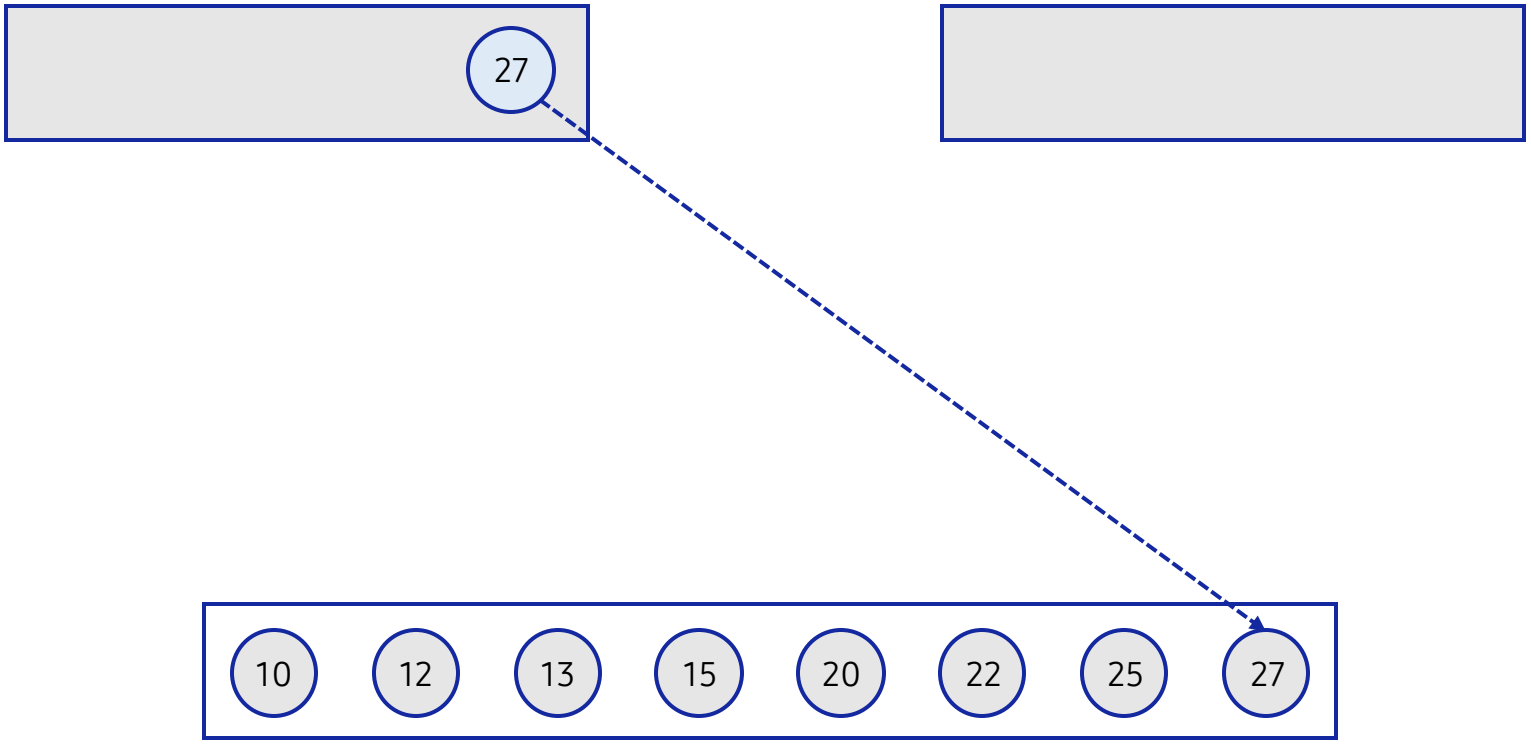
Remove the element sent to the sorted list and compare the next element. Since 22 is less than 27, 22 is added to the sorted list.



1. Merge Sort

1.2. Merging two sorted lists

If one of the two lists is empty, all elements in the other list are added to the sorted list in order.



Let's code

1. Implementing Merge Sort with Additional Memory

1.1. Merge sort by the divide-and-conquer approach

A simple way to implement merge sort is to conquer a given list by dividing it into two lists. This algorithm design method is called divide-and-conquer, and we will study it again in Unit 31.

```
1  def mergesort1(S):
2      n = len(S)
3      if n > 1:
4          print(S)
5          mid = n // 2
6          L, R = S[:mid], S[mid:]
7          mergesort1(L)
8          mergesort1(R)
9          merge1(S, L, R)
```

Line 1~3

- Merge sort conquers a given S by dividing it into two lists if the size of S is greater than 1.
- If the size of the given S is 1 or less, the sorting can be terminated because it has already been sorted.

1. Implementing Merge Sort with Additional Memory

1.1. Merge sort by the divide-and-conquer approach

If the given size of S is greater than 1, it is divided into two lists, sorted into two lists, and merged into one list.

```
1  def mergesort1(S):
2      n = len(S)
3      if n > 1:
4          print(S)
5          mid = n // 2
6          L, R = S[:mid], S[mid:]
7          mergesort1(L)
8          mergesort1(R)
9          merge1(S, L, R)
```

Line 5~8

- To divide a given S into two, after finding the mid value, it is divided into two lists, L and R.
- At this time, it is necessary to note that the list S is created as a separate list for L and R through slicing.
- Separately created L and R sort separately through recursive calls.

1. Implementing Merge Sort with Additional Memory

1.1. Merge sort by the divide-and-conquer approach

If L and R are sorted lists, we can merge these two lists into sorted S.

```
1  def mergesort1(S):
2      n = len(S)
3      if n > 1:
4          print(S)
5          mid = n // 2
6          L, R = S[:mid], S[mid:]
7          mergesort1(L)
8          mergesort1(R)
9          merge1(S, L, R)
```

Line 9

- Merge L and R to create a sorted list S.

1. Implementing Merge Sort with Additional Memory

1.2. Merge two sorted lists into a sorted list

The merge1() function creates a sorted list S by merging L and R, which are sorted lists, respectively.

```
1  def merge1(S, L, R):
2      k = 0
3      while len(L) > 0 and len(R) > 0:
4          if L[0] <= R[0]:
5              S[k] = L.pop(0)
6          else:
7              S[k] = R.pop(0)
8          k += 1
9      while len(L) != 0:
10         S[k] = L.pop(0)
11         k += 1
12     while len(R) != 0:
13         S[k] = R.pop(0)
14         k += 1
```

Line 2~8

- While increasing the index of k, the smaller of the first elements of L and R is stored as the k-th element of S.

1. Implementing Merge Sort with Additional Memory

1.2. Merge two sorted lists into a sorted list

The merge1() function creates a sorted list S by merging L and R, which are sorted lists, respectively.

```
1  def merge1(S, L, R):
2      k = 0
3      while len(L) > 0 and len(R) > 0:
4          if L[0] <= R[0]:
5              S[k] = L.pop(0)
6          else:
7              S[k] = R.pop(0)
8          k += 1
9      while len(L) != 0:
10         S[k] = L.pop(0)
11         k += 1
12     while len(R) != 0:
13         S[k] = R.pop(0)
14         k += 1
```

Line 9~11

- When exiting the while-loop of Line 3-8, one of the two lists L and R has processed all elements.
- If the length of L is non-zero, all remaining elements of L can be added to S.

1. Implementing Merge Sort with Additional Memory

1.2. Merge two sorted lists into a sorted list

The merge1() function creates a sorted list S by merging L and R, which are sorted lists, respectively.

```
1  def merge1(S, L, R):
2      k = 0
3      while len(L) > 0 and len(R) > 0:
4          if L[0] <= R[0]:
5              S[k] = L.pop(0)
6          else:
7              S[k] = R.pop(0)
8          k += 1
9      while len(L) != 0:
10         S[k] = L.pop(0)
11         k += 1
12     while len(R) != 0:
13         S[k] = R.pop(0)
14         k += 1
```

Line 12~14

- If the length of the list R is non-zero, then all L is processed and only R is left unprocessed.
- All the remaining elements of R need to be added to S.

1. Implementing Merge Sort with Additional Memory

1.3. Running merge sort

Once the example used earlier is applied to the mergesort1() function, the output is as follows.

```
1 S = [27, 10, 12, 20, 25, 13, 15, 22]
2 mergesort1(S)
3 print(S)
```

```
[27, 10, 12, 20, 25, 13, 15, 22]
[27, 10, 12, 20]
[27, 10]
[12, 20]
[25, 13, 15, 22]
[25, 13]
[15, 22]
[10, 12, 13, 15, 20, 22, 25, 27]
```

Line 1~3

- The S given by print(S) is printed on the 4th line of the mergesort1() function.
- The S output from Line 3 is the sorted result after the mergesort1() function is executed.

One More Step

- How much memory is additionally used by the mergesort1() function?
- The mergesort1() function creates new lists L and R each time the function is called. If the size of these two lists is added, it becomes the same size as the S originally passed in the call.
- Since the recursive call is called for L and R respectively, the size of the two lists at each recursive call is about half the size of the previous call. Therefore, the size of the additionally used list is as follows.

$$N + \frac{N}{2} + \frac{N}{2^2} + \cdots + \frac{N}{2^k} = 2N$$

  - The mergesort1() function must use an additional memory of size 2N which is twice the size N of a given S.
  - This algorithm can be modified to use only N additional memory.

2. Implementing Merge Sort More Efficiently

2.1. Enhanced merge sort

The mergesort2() sorts S by specifying the ranges of low and high instead of dividing S into L and R.

```
1 def mergesort2(S, low, high):
2     if low < high:
3         print(S)
4         mid = (low + high) // 2
5         mergesort2(S, low, mid)
6         mergesort2(S, mid + 1, high)
7         merge2(S, low, mid, high)
```

Line 1~3

- For a given S, it receives the indices of low and high as input parameters.
- A recursive call is made only if low is less than the high value, otherwise, the sort is terminated.

2. Implementing Merge Sort More Efficiently

2.1. Enhanced merge sort

The mergesort2() sorts S by specifying the ranges of low and high instead of dividing S into L and R.

```
1 def mergesort2(S, low, high):
2     if low < high:
3         print(S)
4         mid = (low + high) // 2
5         mergesort2(S, low, mid)
6         mergesort2(S, mid + 1, high)
7         merge2(S, low, mid, high)
```

Line 4~7

- In order to divide and conquer the elements in the low and high ranges, a recursive call is made based on mid.
- The merge2() function merges two sorted elements into one based on mid.

2. Implementing Merge Sort More Efficiently

2.2. Merging two sorted list for enhanced merge sort

The merge2() function makes elements sorted from left to right into low and high ranges based on mid.

```
1  def merge2(S, low, mid, high):
2      R = []
3      i, j = low, mid + 1
4      while i <= mid and j <= high:
5          if S[i] < S[j]:
6              R.append(S[i]); i += 1
7          else:
8              R.append(S[j]); j += 1
9      if i > mid:
10         for k in range(j, high + 1):
11             R.append(S[j])
12     else:
13         for k in range(i, mid + 1):
14             R.append(S[i])
15     for k in range(len(R)):
16         S[low + k] = R[k]
```

Line 2~8

- Add the sorted elements to a new list called R, starting at i and j at low and mid+1, respectively.
- It should be noted that the elements before mid and after mid + 1 are sorted elements, respectively.

2. Implementing Merge Sort More Efficiently

2.2. Merging two sorted list for enhanced merge sort

The merge2() function makes elements sorted from left to right into low and high ranges based on mid.

```
1  def merge2(S, low, mid, high):
2      R = []
3      i, j = low, mid + 1
4      while i <= mid and j <= high:
5          if S[i] < S[j]:
6              R.append(S[i]); i += 1
7          else:
8              R.append(S[j]); j += 1
9      if i > mid:
10         for k in range(j, high + 1):
11             R.append(S[j])
12     else:
13         for k in range(i, mid + 1):
14             R.append(S[i])
15     for k in range(len(R)):
16         S[low + k] = R[k]
```

Line 9~11

- When exiting the while-loop of Line 3-8, either i will be greater than mid or j will be greater than high, depending on the condition of Line 4.
- If i is greater than mid, the remaining elements must be added to R until the index of j becomes high.

2. Implementing Merge Sort More Efficiently

2.2. Merging two sorted list for enhanced merge sort

The merge2() function makes elements sorted from left to right into low and high ranges based on mid.

```
1  def merge2(S, low, mid, high):
2      R = []
3      i, j = low, mid + 1
4      while i <= mid and j <= high:
5          if S[i] < S[j]:
6              R.append(S[i]); i += 1
7          else:
8              R.append(S[j]); j += 1
9      if i > mid:
10         for k in range(j, high + 1):
11             R.append(S[j])
12     else:
13         for k in range(i, mid + 1):
14             R.append(S[i])
15     for k in range(len(R)):
16         S[low + k] = R[k]
```

Line 12~14

- When exiting the while-loop of Line 3-8, either i will be greater than mid or j will be greater than high, depending on the condition of Line 4.
- If i is not greater than mid, the remaining elements must be added to R until the index of i becomes mid.

2. Implementing Merge Sort More Efficiently

2.2. Merging two sorted list for enhanced merge sort

The merge2() function makes elements sorted from left to right into low and high ranges based on mid.

```
1  def merge2(S, low, mid, high):
2      R = []
3      i, j = low, mid + 1
4      while i <= mid and j <= high:
5          if S[i] < S[j]:
6              R.append(S[i]); i += 1
7          else:
8              R.append(S[j]); j += 1
9      if i > mid:
10         for k in range(j, high + 1):
11             R.append(S[j])
12     else:
13         for k in range(i, mid + 1):
14             R.append(S[i])
15     for k in range(len(R)):
16         S[low + k] = R[k]
```

Line 15~16

- In line 15, a single sorted list from low to high is stored in the R list.
- All elements from low to high of S are copied to the elements of R list.

2. Implementing Merge Sort More Efficiently

2.3. Running enhanced merge sort

Once the example used earlier is applied to the mergesort2() function, the output is as follows.

```
1 S = [27, 10, 12, 20, 25, 13, 15, 22]
2 mergesort2(S, 0, len(S) - 1)
3 print(S)
```

```
[27, 10, 12, 20, 25, 13, 15, 22]
[27, 10, 12, 20, 25, 13, 15, 22]
[27, 10, 12, 20, 25, 13, 15, 22]
[10, 27, 12, 20, 25, 13, 15, 22]
[10, 12, 20, 27, 25, 13, 15, 22]
[10, 12, 20, 27, 25, 13, 15, 22]
[10, 12, 20, 27, 13, 25, 15, 22]
[10, 12, 13, 15, 20, 22, 25, 27]
```

Line 1~3

- The S given by print(S) is printed on the 3rd line of the mergesort2() function.
- The S output from Line 3 is the sorted result after the mergesort2() function is executed.
- It should be noted that the size of S does not change, unlike the output of the mergesort1() function.

Pop quiz

Q1. How many times was the merge2() function executed in the merge sort process below?

```
S = [6, 2, 11, 7, 5, 4, 8, 16, 10, 3]
mergesort2(S, 0, len(S) - 1)
print(S)
```

```
S = [6, 2, 11, 7, 5, 4, 8, 16, 10, 3, 1, 12, 9]
mergesort2(S, 0, len(S) - 1)
print(S)
```

Pair programming

Pair Programming Practice

Guideline, mechanisms & contingency plan

Preparing pair programming involves establishing guidelines and mechanisms to help students pair properly and to keep them paired. For example, students should take turns "driving the mouse." Effective preparation requires contingency plans in case one partner is absent or decides not to participate for one reason or another. In these cases, it is important to make it clear that the active student will not be punished because the pairing did not work well.

Pairing similar, not necessarily equal, abilities as partners

Pair programming can be effective when students of similar, though not necessarily equal, abilities are paired as partners. Pairing mismatched students often can lead to unbalanced participation. Teachers must emphasize that pair programming is not a "divide-and-conquer" strategy, but rather a true collaborative effort in every endeavor for the entire project. Teachers should avoid pairing very weak students with very strong students.

Motivate students by offering extra incentives

Offering extra incentives can help motivate students to pair, especially with advanced students. Some teachers have found it helpful to require students to pair for only one or two assignments.

Pair Programming Practice

Prevent collaboration cheating

The challenge for the teacher is to find ways to assess individual outcomes, while leveraging the benefits of collaboration. How do you know whether a student learned or cheated? Experts recommend revisiting course design and assessment, as well as explicitly and concretely discussing with the students on behaviors that will be interpreted as cheating. Experts encourage teachers to make assignments meaningful to students and to explain the value of what students will learn by completing them.

Collaborative learning environment

A collaborative learning environment occurs anytime an instructor requires students to work together on learning activities. Collaborative learning environments can involve both formal and informal activities and may or may not include direct assessment. For example, pairs of students work on programming assignments; small groups of students discuss possible answers to a professor's question during lecture; and students work together outside of class to learn new concepts. Collaborative learning is distinct from projects where students "divide and conquer." When students divide the work, each is responsible for only part of the problem solving and there are very limited opportunities for working through problems with others. In collaborative environments, students are engaged in intellectual talk with each other.

Q1. Given N sorted lists as input, write a program that merges them into a sorted list.

```
N = int(input("Input the number of list: "))
list_of_nums = []
for i in range(N):
    nums = list(map(int, input("Input a list of numbers: ").split()))
    print(nums)
    list_of_nums.append(nums)
sorted = multiway_merge(list_of_nums)
print("Merged into : ", sorted)
```

```
Input the number of list: 3
Input a list of numbers: 1 5
[1, 5]
Input a list of numbers: 2 6
[2, 6]
Input a list of numbers: 3 4 7
[3, 4, 7]
Merged into :  [1, 2, 3, 4, 5, 6, 7]
```

Unit 29.

Quick Sort

Unit learning objective (1/3)

Learning objectives

- ✓ Understand the quick sort algorithm and be able to solve sorting problems using quick sort.
- ✓ Understand and be able to explain the difference between quick sort and merge sort.
- ✓ Understand and be able to explain the time complexity of quicksort.

Unit learning objective (2/3)

Learning overview

- ✓ Implement a quick sort that divides a given unsorted list based on the pivot and sorts them individually.
- ✓ Implement a partition function that divides an unsorted list based on the pivot for quick sorting.
- ✓ Understand that quick sort is also a divide-and-conquer method using a recursive function like the merge sort algorithm.

Concepts you will need to know from previous units

- ✓ Using comparison operators to compare two numbers.
- ✓ Dividing the given problem and call the recursive function recursively.
- ✓ Applying Big O notation to analyze the time complexity of an algorithm.

Keywords

- Sorting Problem
- Quick Sort
- Pivot and Partitioning

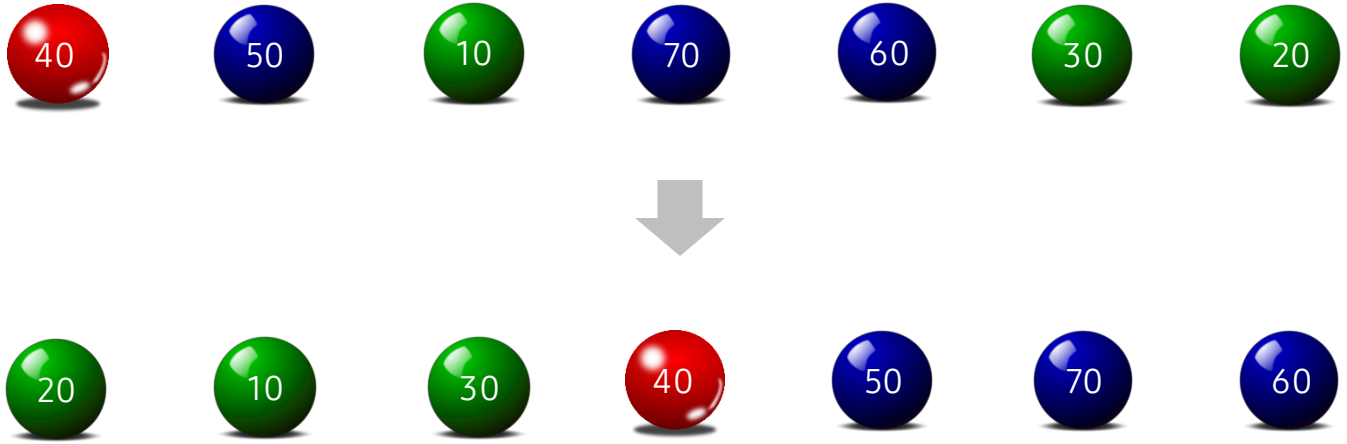
Mission

1. Real world problem

1.1. In-Place sorting

In the previous unit, we studied the merge sort algorithm. However, in order to use merge sort, we had to use separate memory space of the same size as the given list. Can we design an efficient sorting algorithm without extra memory space?

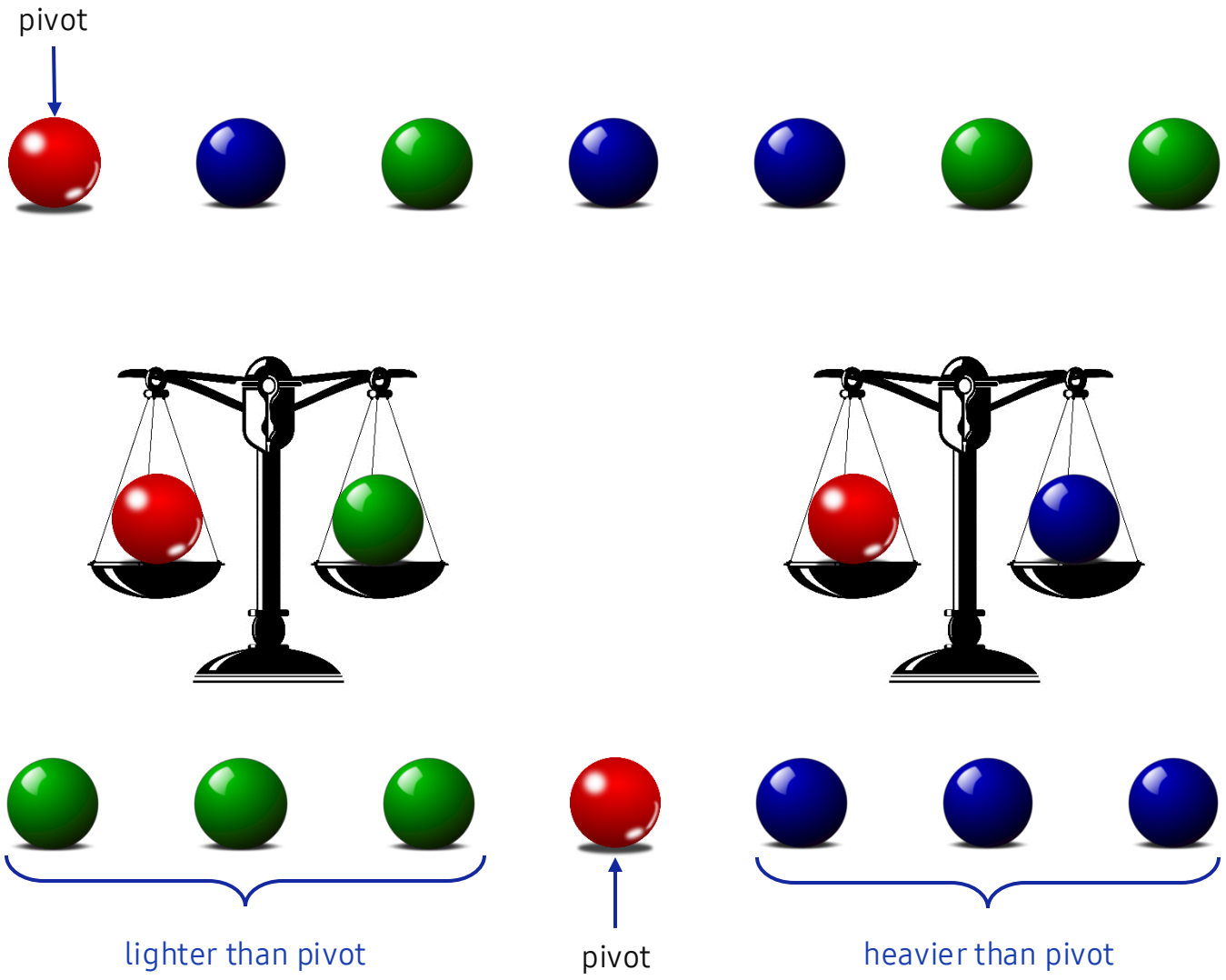
Ex Suppose you are given 7 iron marbles as shown below. These iron marbles are the same shape and size but have different weights. By just swapping these iron marbles, can things lighter than red marbles be sent to the left and things heavier than red marbles to the right?



2. Mission

2.1. Partitioning with pivot

- Use the balance scale. The balance scale can do the following operations:
  - You can compare the weights by taking out two marbles from a barrel of iron marbles.
  - The positions of the two iron marbles can be swapped.
- We have a reference red marble, so we can find a marble lighter and one heavier than the red marble and swap them.
- With this strategy, how many comparisons do all the marbles need to be placed in sorted order by weight?


pivot
lighter than pivot
pivot
heavier than pivot

Key concept

1. Quick Sort

1.1. An example of quick sort

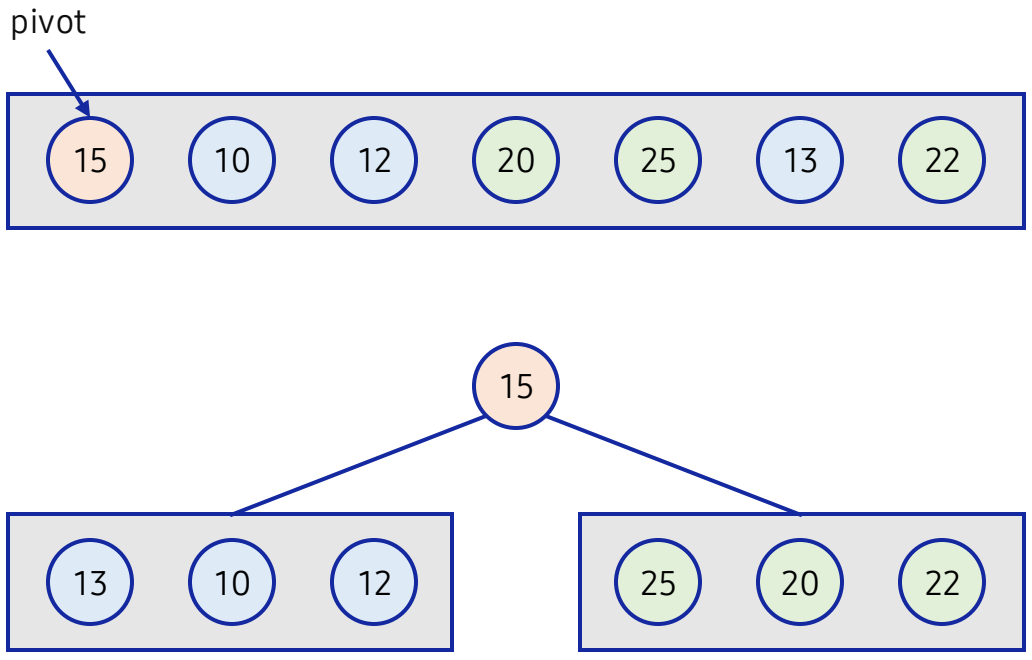
Quick sort is a sorting algorithm that recursively sorts an unsorted list by dividing it into two sublists, according to the value of pivot.

Ex Suppose we want to sort a list arranged in the order [15, 10, 12, 20, 25, 13, 22] as follows.

1. Quick Sort

1.1. An example of quick sort

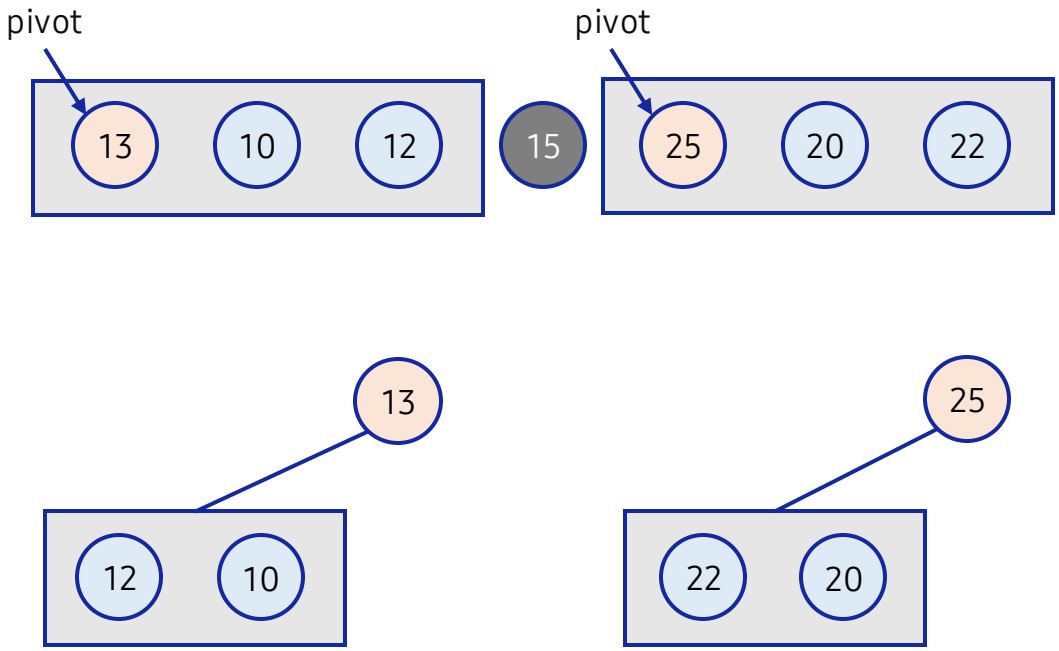
First, the first element 15 is set as the pivot. Elements smaller than the pivot are divided to the left of the pivot, and elements larger than the pivot are divided to the right of the pivot.



1. Quick Sort

1.1. An example of quick sort

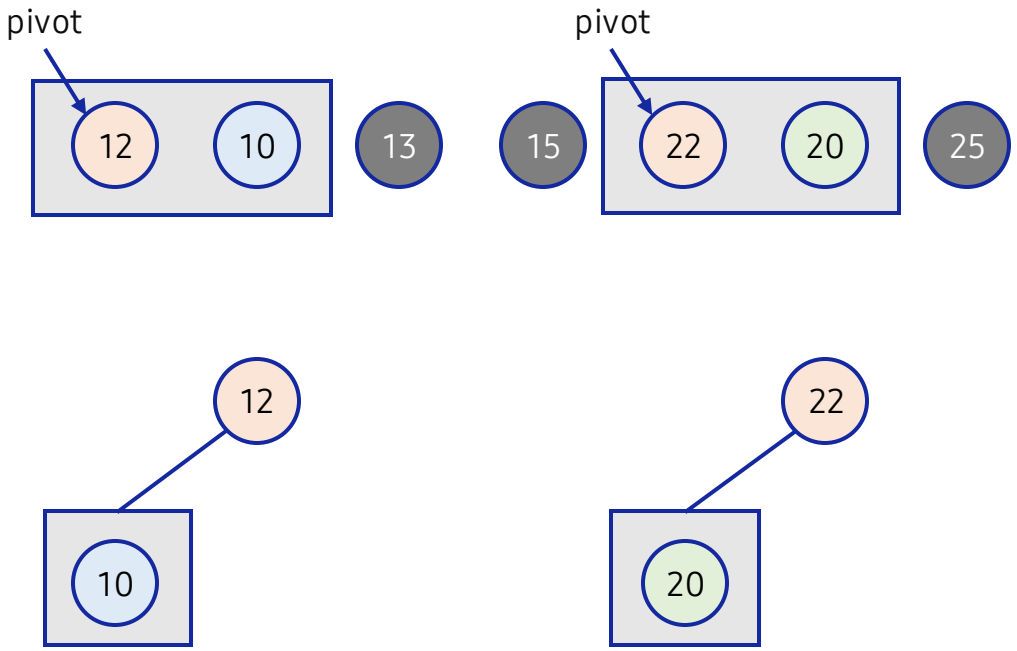
- Divide by pivoting the first element in each divided list.
- Next, divide the sublists with a new pivot in each divided sublist respectively.



1. Quick Sort

1.1. An example of quick sort

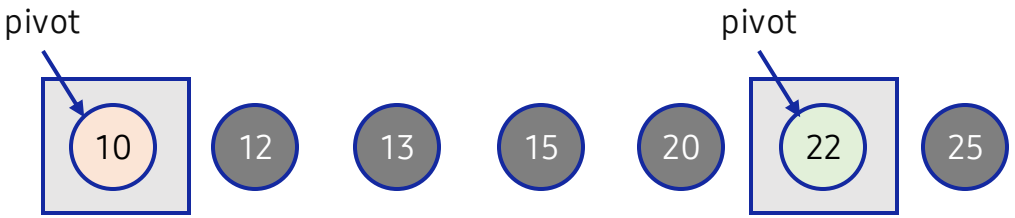
- Divide by pivoting the first element in each divided list.
- Next, divide the sublists with a new pivot in each divided sublist respectively.

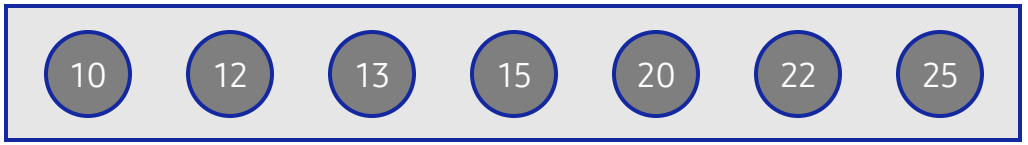


1. Quick Sort

1.1. An example of quick sort

If there is only one element, the recursive call is terminated because it is sorted. At this time, the entire list is sorted.



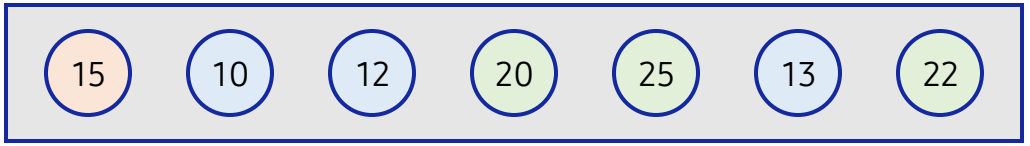


Sorted List

1. Quick Sort

1.2. Partitioning with pivot

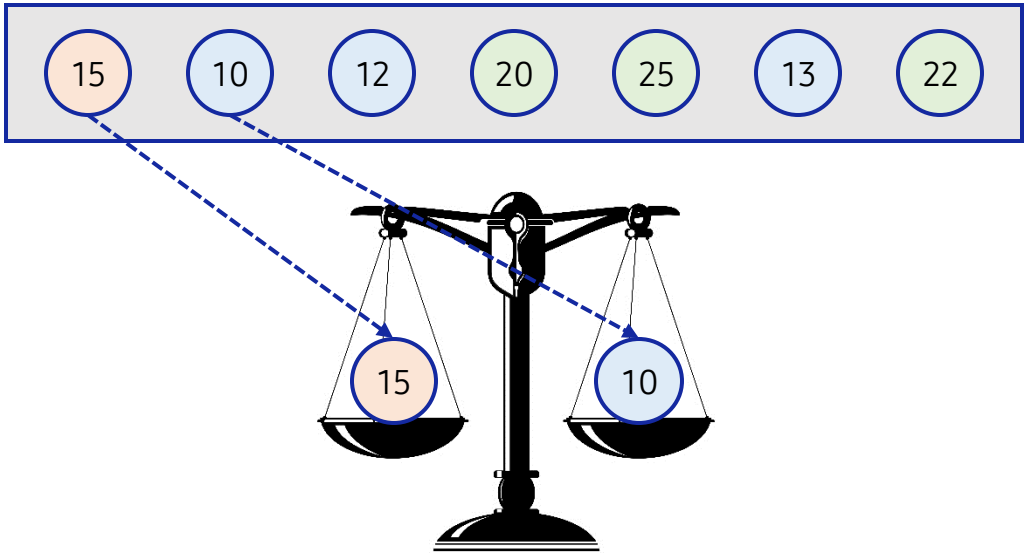
Find a method to divide left and right based on the pivot without using separate storage space. Use the balance scale.



1. Quick Sort

1.2. Partitioning with pivot

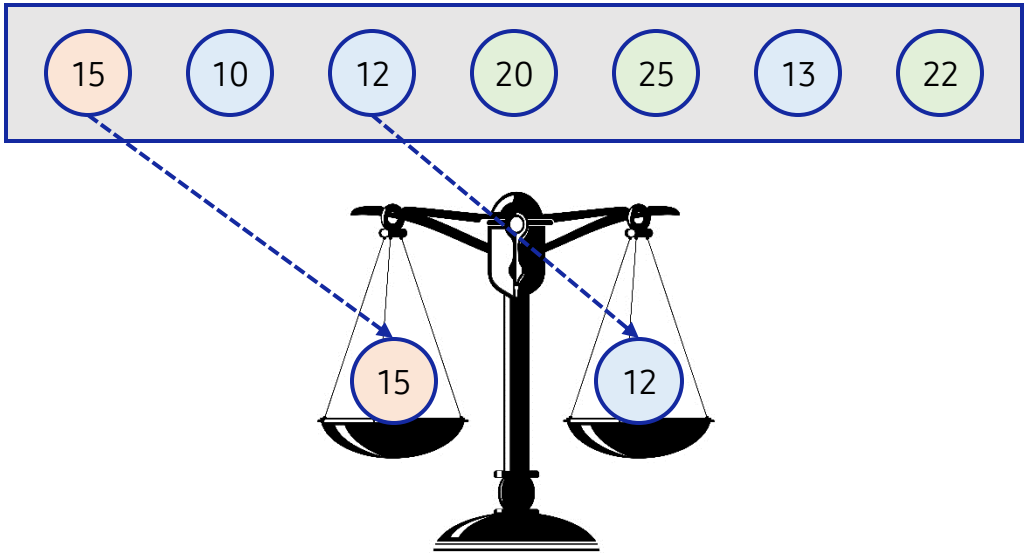
First, find a value greater than the pivot from left to right in the set of elements excluding the pivot. Since 10 is less than 15, it moves to the next element.



1. Quick Sort

1.2. Partitioning with pivot

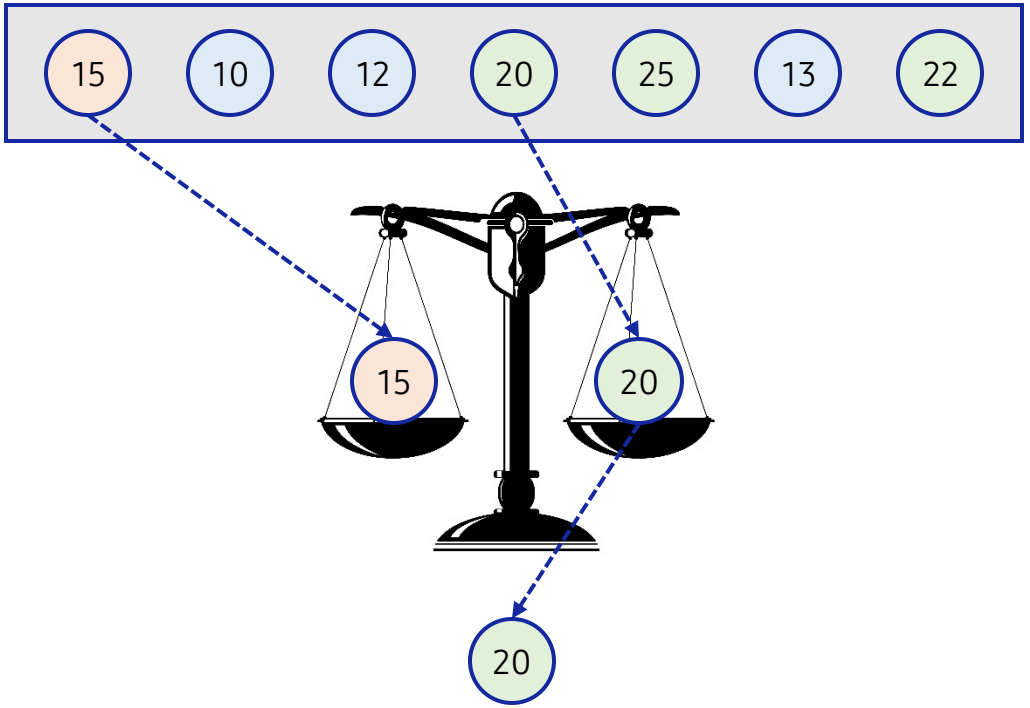
Since 12 is less than 15, it moves to the next element.



1. Quick Sort

1.2. Partitioning with pivot

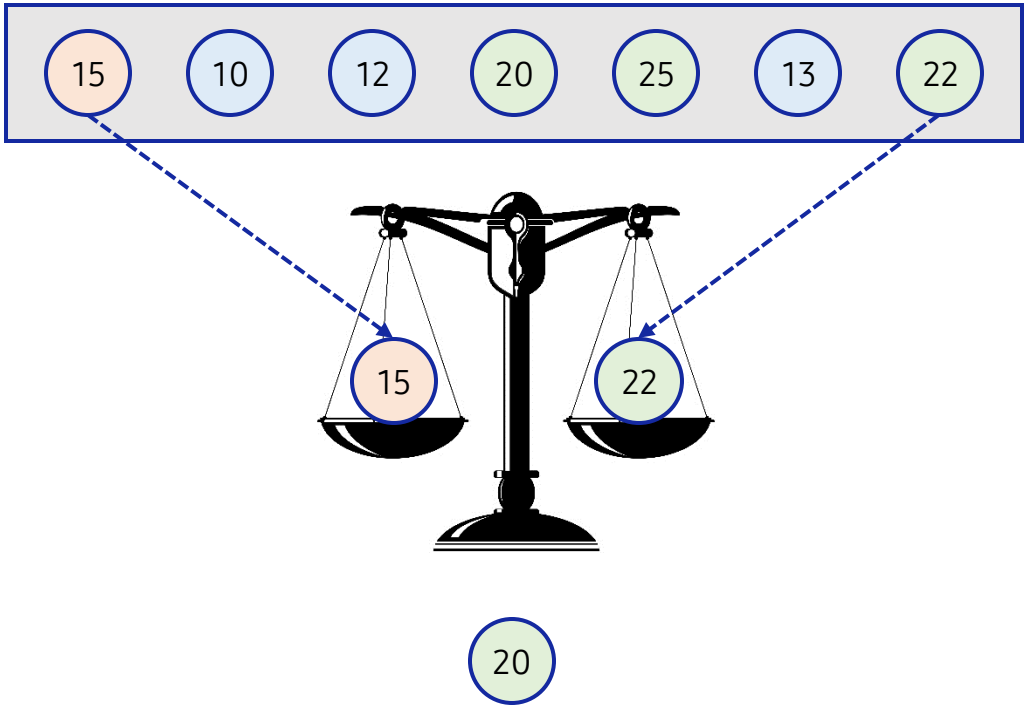
Since 20 is greater than 15, 20 is the first element in the list that is greater than the pivot.



1. Quick Sort

1.2. Partitioning with pivot

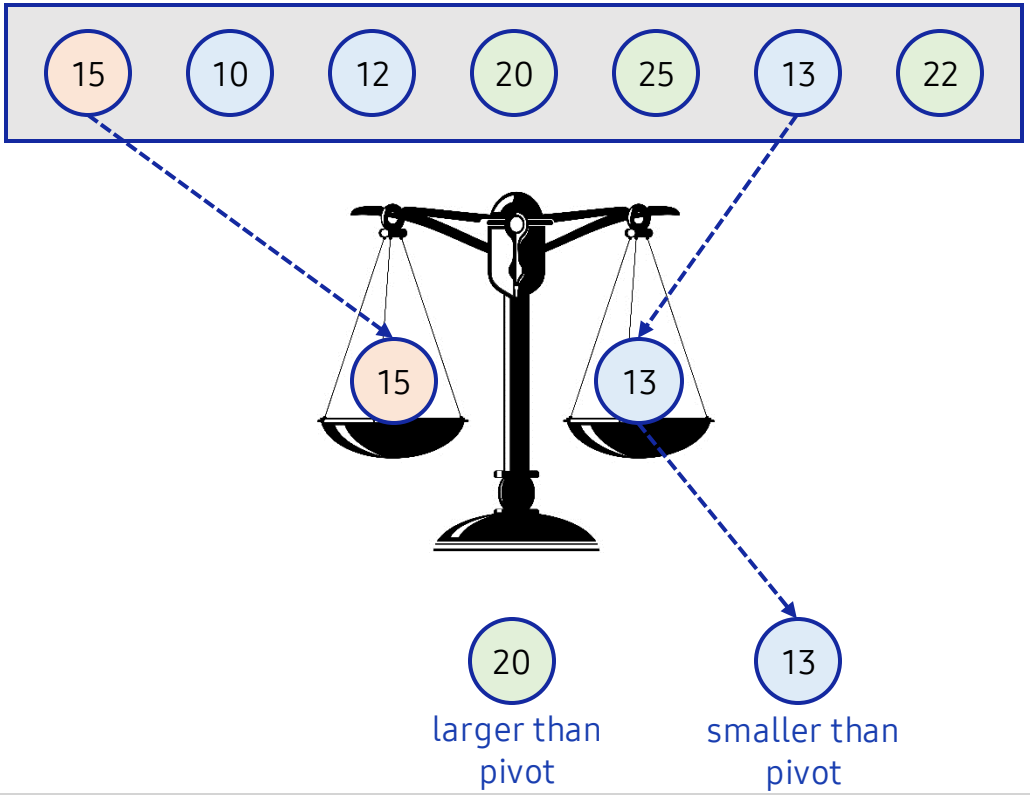
Next, find the first element less than the pivot from right to left. Since 22 is greater than 15, it moves to the next element.



1. Quick Sort

1.2. Partitioning with pivot

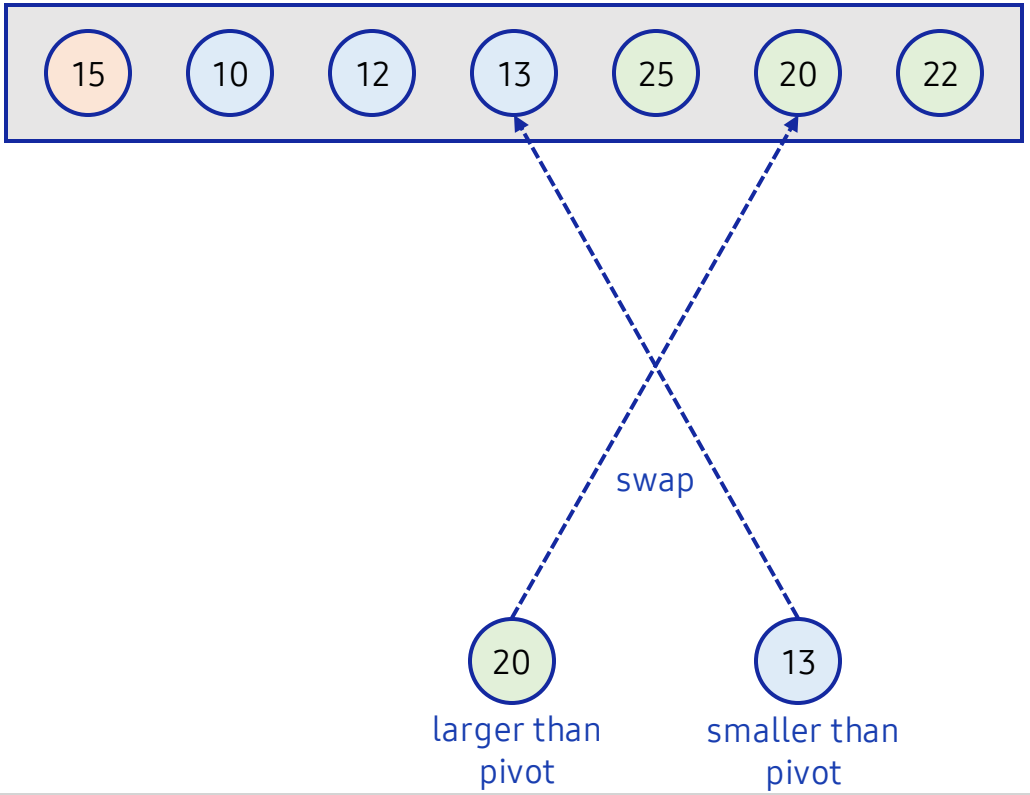
Since 13 is less than 15, 13 is the last element less than 15.



1. Quick Sort

1.2. Partitioning with pivot

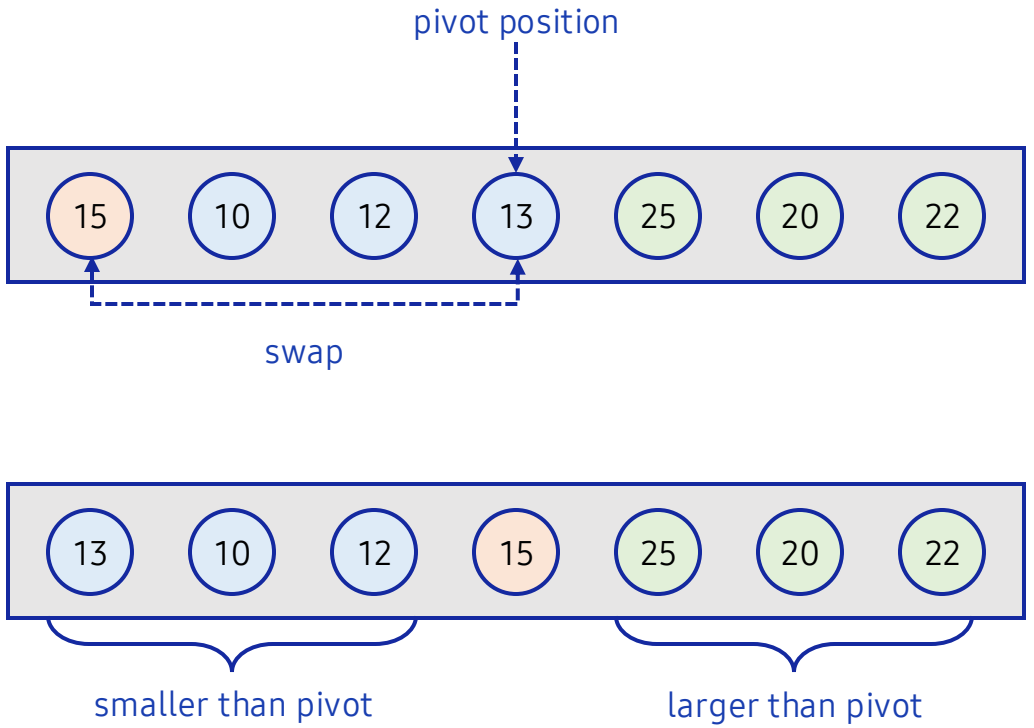
- The first element greater than the pivot and the last element less than the pivot are swapped. By repeating this process, elements smaller than the pivot and elements larger than the pivot are divided into two partitions.



1. Quick Sort

1.2. Partitioning with pivot

In a two-partitioned list, when an element at a pivot position and a pivot item are exchanged with each other, the pivot is placed at the pivot position, elements smaller than the pivot are placed on the left, and elements larger than the pivot are placed on the right.



1. Quick Sort

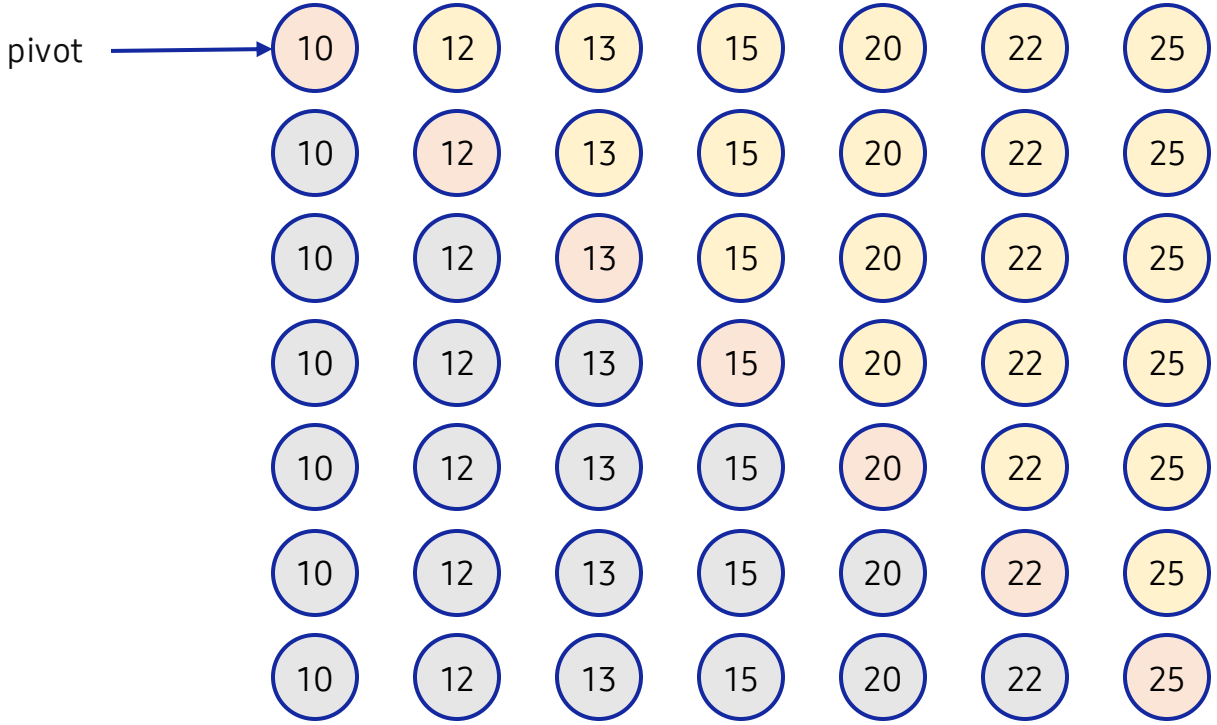
1.3. Choosing a pivot

Previously, we used the strategy of pivoting to select the first item in the list. What would happen if we used this strategy on an already sorted list like this:

1. Quick Sort

1.3. Choosing a pivot

Since each time the first item in the list is the smallest element, the partition() function cannot divide. We have to divide N times as follows.



1. Quick Sort

1.3. Choosing a pivot

What would happen if we set the pivot to an arbitrary element? In the worst case, the sorted order of elements may be selected, but the probability of such a case is very unlikely, so on average, it has an expected value of O(NlogN).

Let's code

1. Implementing Quick Sort with First Pivot

1.1. Quick sort with the divide-and-conquer approach

Quick sort uses a divide-and-conquer strategy like merge sort. The difference from merge sort is that since it divides into two lists based on the pivot, the sizes of the two lists may be different.

```
1  def quicksort1(S, low, high):
2      if low < high:
3          print(S)
4          pivotpoint = partition1(S, low, high)
5          quicksort1(S, low, pivotpoint - 1)
6          quicksort1(S, pivotpoint + 1, high)
```

Line 1~3

- Quick sort sorts the elements from low to high for a given S.
- If low is equal to or less than high, sorting can be terminated because it is in the sorted state.

1. Implementing Quick Sort with First Pivot

1.1. Quick sort with the divide-and-conquer approach

According to the pivot, the partition1() function divides the elements of S into elements smaller than the pivot and elements larger than the pivot. These two divided lists are sorted through a recursive call.

```
1  def quicksort1(S, low, high):
2      if low < high:
3          print(S)
4          pivotpoint = partition1(S, low, high)
5          quicksort1(S, low, pivotpoint - 1)
6          quicksort1(S, pivotpoint + 1, high)
```

Line 4~6

- pivotpoint returns the position of the pivot in the partition1() function.
- Elements smaller than the pivot are placed to the left of the pivotpoint, and elements larger than the pivot are placed to the right of the pivot point.
- The left and right elements are sorted based on the pivotpoint, respectively.

1. Implementing Quick Sort with First Pivot

1.2. Partitioning with the first pivot

The partition1() function divides the elements from low to high in a given S based on the pivot. Assume that the pivot strategy is to select the first element of the given elements.

```
1  def partition1(S, low, high):
2      pivot = S[low]
3      left, right = low + 1, high
4      while left < right:
5          print(S)
6          while left <= right and S[left] <= pivot:
7              left += 1
8          while left <= right and S[right] >= pivot:
9              right -= 1
10         if left < right:
11             S[left], S[right] = S[right], S[left]
12     pivotpoint = right
13     S[low], S[pivotpoint] = S[pivotpoint], S[low]
14     return pivotpoint
```

Line 1~2

- The partition1() function takes S, low, and high as input parameters.
- The first element in the given range S[low] is set as the pivot.

1. Implementing Quick Sort with First Pivot

1.2. Partitioning with the first pivot

The partitioning operation selects a strategy to exchange elements in the range of S by comparing them at the left and right ends, respectively.

```
1  def partition1(S, low, high):
2      pivot = S[low]
3      left, right = low + 1, high
4      while left < right:
5          print(S)
6          while left <= right and S[left] <= pivot:
7              left += 1
8          while left <= right and S[right] >= pivot:
9              right -= 1
10         if left < right:
11             S[left], S[right] = S[right], S[left]
12     pivotpoint = right
13     S[low], S[pivotpoint] = S[pivotpoint], S[low]
14     return pivotpoint
```

Line 3~11

- Among the elements excluding the pivot, left and right are selected as the leftmost and rightmost elements.
- Find the first element from the left that is greater than the pivot, and the first element from the right that is less than the pivot.
- Swap the left and right elements.

1. Implementing Quick Sort with First Pivot

1.2. Partitioning with the first pivot

When the exchange operation is finished, the first element selected by the pivot is moved to the center position.

```
1  def partition1(S, low, high):
2      pivot = S[low]
3      left, right = low + 1, high
4      while left < right:
5          print(S)
6          while left <= right and S[left] <= pivot:
7              left += 1
8          while left <= right and S[right] >= pivot:
9              right -= 1
10         if left < right:
11             S[left], S[right] = S[right], S[left]
12     pivotpoint = right
13     S[low], S[pivotpoint] = S[pivotpoint], S[low]
14     return pivotpoint
```

Line 12-14

- In Line 3-11, when the exchange operation is completed, the element S[pivotpoint] becomes the last element of the elements smaller than the pivot element.
- If the pivot element in the low position and the element in the pivotpoint position are exchanged with each other, the partitioning operation is completed.

1. Implementing Quick Sort with First Pivot

1.3. Running quick sort

When the quick sort algorithm implemented earlier is executed, the quick sort is performed through the following process.

```
1 S = [15, 10, 12, 20, 25, 13, 22]
2 quicksort1(S, 0, len(S) - 1)
3 print(S)
```

```
[15, 10, 12, 20, 25, 13, 22]
[15, 10, 12, 20, 25, 13, 22]
[15, 10, 12, 13, 25, 20, 22]
[13, 10, 12, 15, 25, 20, 22]
[13, 10, 12, 15, 25, 20, 22]
[12, 10, 13, 15, 25, 20, 22]
[10, 12, 13, 15, 25, 20, 22]
[10, 12, 13, 15, 25, 20, 22]
[10, 12, 13, 15, 22, 20, 25]
[10, 12, 13, 15, 20, 22, 25]
```

1. Implementing Quick Sort with First Pivot

1.3. Running quick sort

The worst case of quick sort is when the given S is sorted in descending order. In this case, it can be confirmed that exactly N-1 elements are divided at each recursive call.

```
1 S = [25, 22, 20, 15, 13, 12, 10]
2 quicksort1(S, 0, len(S) - 1)
3 print(S)
```

```
[25, 22, 20, 15, 13, 12, 10]
[25, 22, 20, 15, 13, 12, 10]
[10, 22, 20, 15, 13, 12, 25]
[10, 22, 20, 15, 13, 12, 25]
[10, 22, 20, 15, 13, 12, 25]
[10, 22, 20, 15, 13, 12, 25]
[10, 12, 20, 15, 13, 22, 25]
[10, 12, 20, 15, 13, 22, 25]
[10, 12, 20, 15, 13, 22, 25]
[10, 12, 20, 15, 13, 22, 25]
[10, 12, 13, 15, 20, 22, 25]
[10, 12, 15, 13, 20, 22, 25]
```

2. Implementing Quick Sort with Randomized Pivot

2.1. Quick sort with randomized pivot

The quicksort1() algorithm selects the first element of the list as the pivot. The quicksort2() algorithm selects a random element as a pivot. Pivot selection is implemented in the partition2() function.

```
def quicksort2(S, low, high):
    if low < high:
        pivotpoint = partition2(S, low, high)
        quicksort2(S, low, pivotpoint - 1)
        quicksort2(S, pivotpoint + 1, high)
```

2. Implementing Quick Sort with Randomized Pivot

2.2. Choosing a random pivot

The partition2() function selects a random element from the given element of S as a pivot.

```
1  from random import randint
2
3  def partition2(S, low, high):
4      rand = randint(low, high)
5      S[low], S[rand] = S[rand], S[low]
6      pivot, left, right = S[low], low, high
7      print(S, left, right, "pivot = ", pivot)
8      while left < right:
9          while left < high and S[left] <= pivot:
10             left += 1
11         while right > low  and pivot <= S[right]:
12             right -= 1
13         if left < right:
14             S[left], S[right] = S[right], S[left]
15     S[low], S[right] = S[right], S[low]
16     return right
```

Line 4~6

- A random position within the given low and high range is set as the rand value.
- When S[low] and S[rand] are exchanged, the first element is changed to a random element.

2. Implementing Quick Sort with Randomized Pivot

2.3. Running quick sort with randomized pivot

It should be noted that the quicksort2() algorithm is not the worst case even for input cases sorted in descending order, unlike the quicksort1() algorithm.

```
1 S = [25, 22, 20, 15, 13, 12, 10]
2 quicksort2(S, 0, len(S) - 1)
3 print(S)
```

```
[12, 22, 20, 15, 13, 25, 10] 0 6 pivot =  12
[10, 12, 25, 15, 13, 20, 22] 2 6 pivot =  25
[10, 12, 22, 15, 13, 20, 25] 2 5 pivot =  22
[10, 12, 15, 20, 13, 22, 25] 2 4 pivot =  15
[10, 12, 13, 15, 20, 22, 25]
```

Pop quiz

Q1. Given the list below, write the output after executing the partition1() function.

```
S = [15, 10, 12, 20, 25, 13, 22]
partition1(S, 0, len(S) - 1)
print(S)
```

Pair programming

Pair Programming Practice

Guideline, mechanisms & contingency plan

Preparing pair programming involves establishing guidelines and mechanisms to help students pair properly and to keep them paired. For example, students should take turns "driving the mouse." Effective preparation requires contingency plans in case one partner is absent or decides not to participate for one reason or another. In these cases, it is important to make it clear that the active student will not be punished because the pairing did not work well.

Pairing similar, not necessarily equal, abilities as partners

Pair programming can be effective when students of similar, though not necessarily equal, abilities are paired as partners. Pairing mismatched students often can lead to unbalanced participation. Teachers must emphasize that pair programming is not a "divide-and-conquer" strategy, but rather a true collaborative effort in every endeavor for the entire project. Teachers should avoid pairing very weak students with very strong students.

Motivate students by offering extra incentives

Offering extra incentives can help motivate students to pair, especially with advanced students. Some teachers have found it helpful to require students to pair for only one or two assignments.

Pair Programming Practice

Prevent collaboration cheating

The challenge for the teacher is to find ways to assess individual outcomes, while leveraging the benefits of collaboration. How do you know whether a student learned or cheated? Experts recommend revisiting course design and assessment, as well as explicitly and concretely discussing with the students on behaviors that will be interpreted as cheating. Experts encourage teachers to make assignments meaningful to students and to explain the value of what students will learn by completing them.

Collaborative learning environment

A collaborative learning environment occurs anytime an instructor requires students to work together on learning activities. Collaborative learning environments can involve both formal and informal activities and may or may not include direct assessment. For example, pairs of students work on programming assignments; small groups of students discuss possible answers to a professor's question during lecture; and students work together outside of class to learn new concepts. Collaborative learning is distinct from projects where students "divide and conquer." When students divide the work, each is responsible for only part of the problem solving and there are very limited opportunities for working through problems with others. In collaborative environments, students are engaged in intellectual talk with each other.

Q1. Write an algorithm that finds the Kth largest element, given N unordered elements.
After solving the problem with the above two methods, analyze which algorithm is more efficient.

- You can return the Kth element after using the sort function.
- You can use the partition() function to make a recursive call until the pivot is the Kth element.


End of
Document
Samsung Innovation Campus

SAMSUNG

Together for Tomorrow!

Enabling People

Education for Future Generations