Sort Algorithms

**Malik Saif Islam**

**35748**

Git URL:

https://github.com/studentRiphah/Sorting-Algorithms

* ***Sorting*** is a process that organizes a collection of data into either ascending or descending order.
* An ***internal sort*** requires that the collection of data fit entirely in the computer’s main memory.
* We can use an ***external sort*** when the collection of data cannot fit in the computer’s main memory all at once but must reside in secondary storage such as on a disk.
* We will analyze only internal sorting algorithms.
* Any significant amount of computer output is generally arranged in some sorted order so that it can be interpreted.
* Sorting also has indirect uses. An initial sort of the data can significantly enhance the performance of an algorithm.
* Majority of programming projects use a sort somewhere, and in many cases, the sorting cost determines the running time.
* A comparison-based sorting algorithm makes ordering decisions only on the basis of comparisons.
  + There are many comparison based sorting algorithms, such as:
    - Bubble Sort
    - Selection Sort

Merge Sort

* + - Quick Sort
  + The list is divided into two sublists: sorted and unsorted.
  + Starting from the bottom of the list, the smallest element is bubbled up from the unsorted list and moved to the sorted sublist.
  + After that, the wall moves one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
  + Each time an element moves from the unsorted part to the

sorted part one sort pass is completed.

* + Given a list of n elements, bubble sort requires up to n-1 passes to sort the data.

Original List

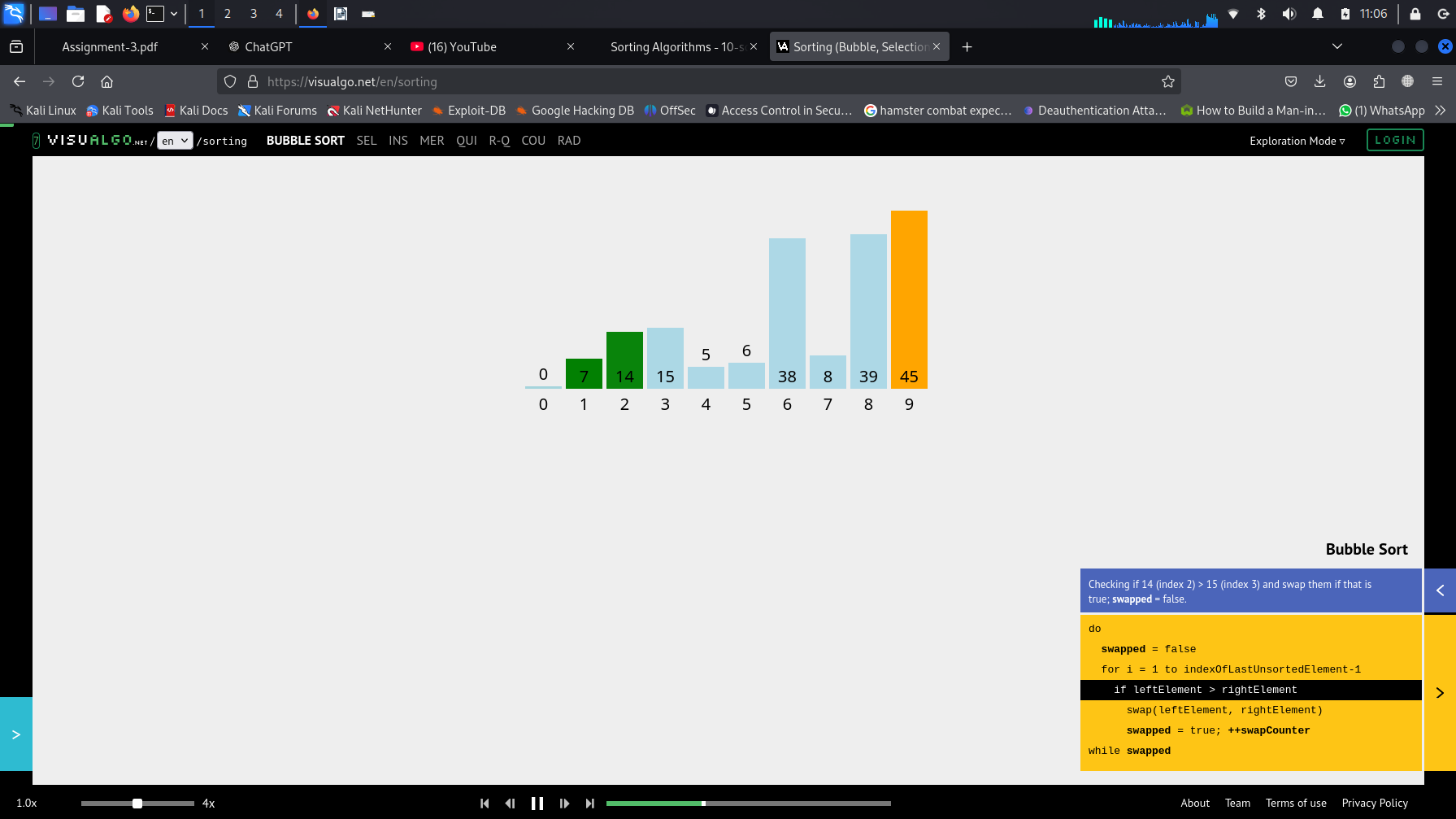
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | | | | |
| 23 | 78 | 45 | 8 | 32 | 56 |
|  | | | | | |
|  | | | | | |
|  |  | | | | |
| 8 | 23 | 78 | 45 | 32 | 56 |
|  |  | | | | |
|  | | | | | |
|  | |  | | | |
| 8 | 23 | 32 | 78 | 45 | 56 |
|  | |  | | | |
|  | | | | | |
|  | | |  | | |
| 8 | 23 | 32 | 45 | 78 | 56 |
|  | | |  | | |
|  | | | | | |
|  | | | |  | |
| 8 | 23 | 32 | 45 | 56 | 78 |
|  | | | |  | |

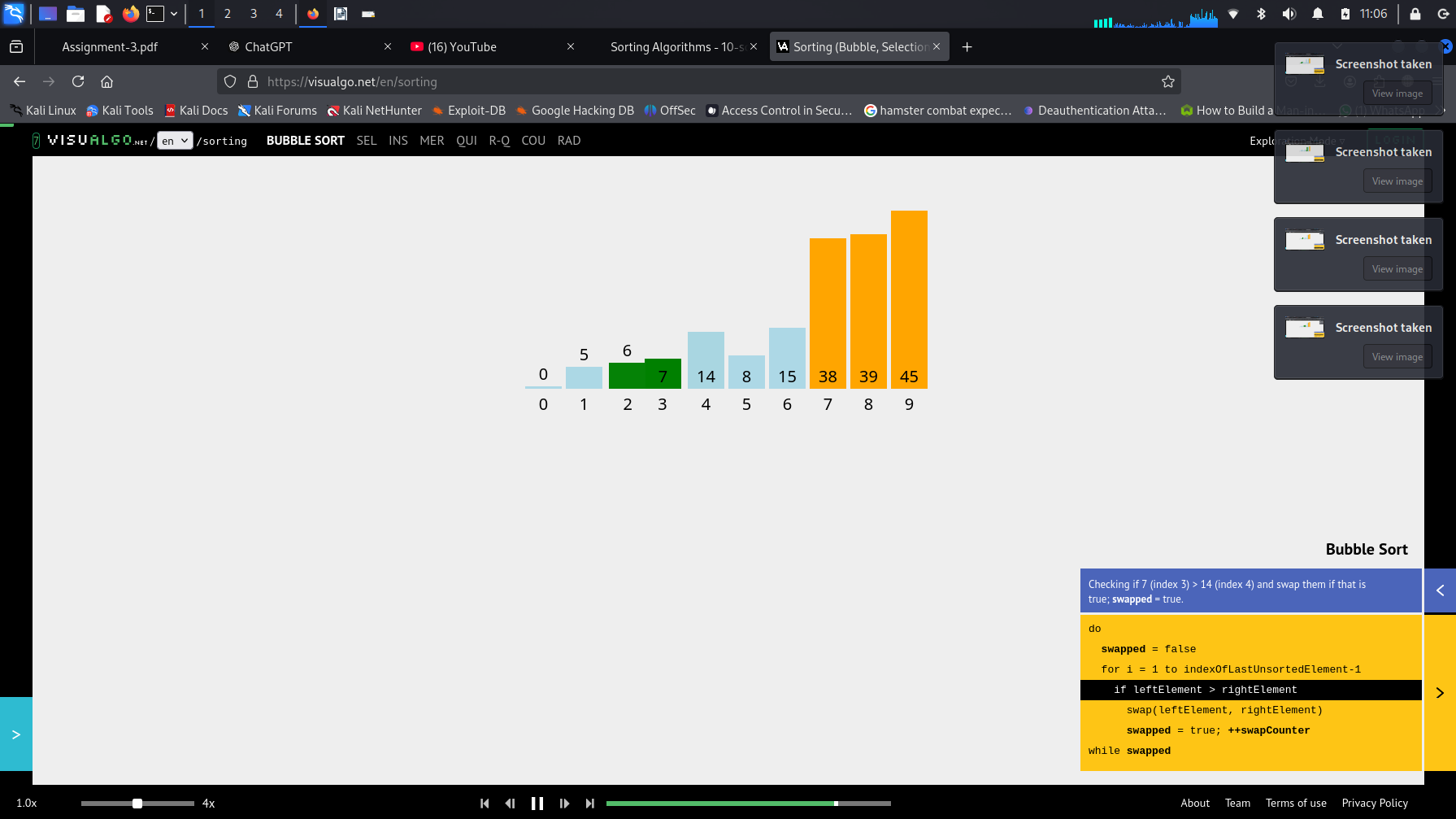
After pass 1

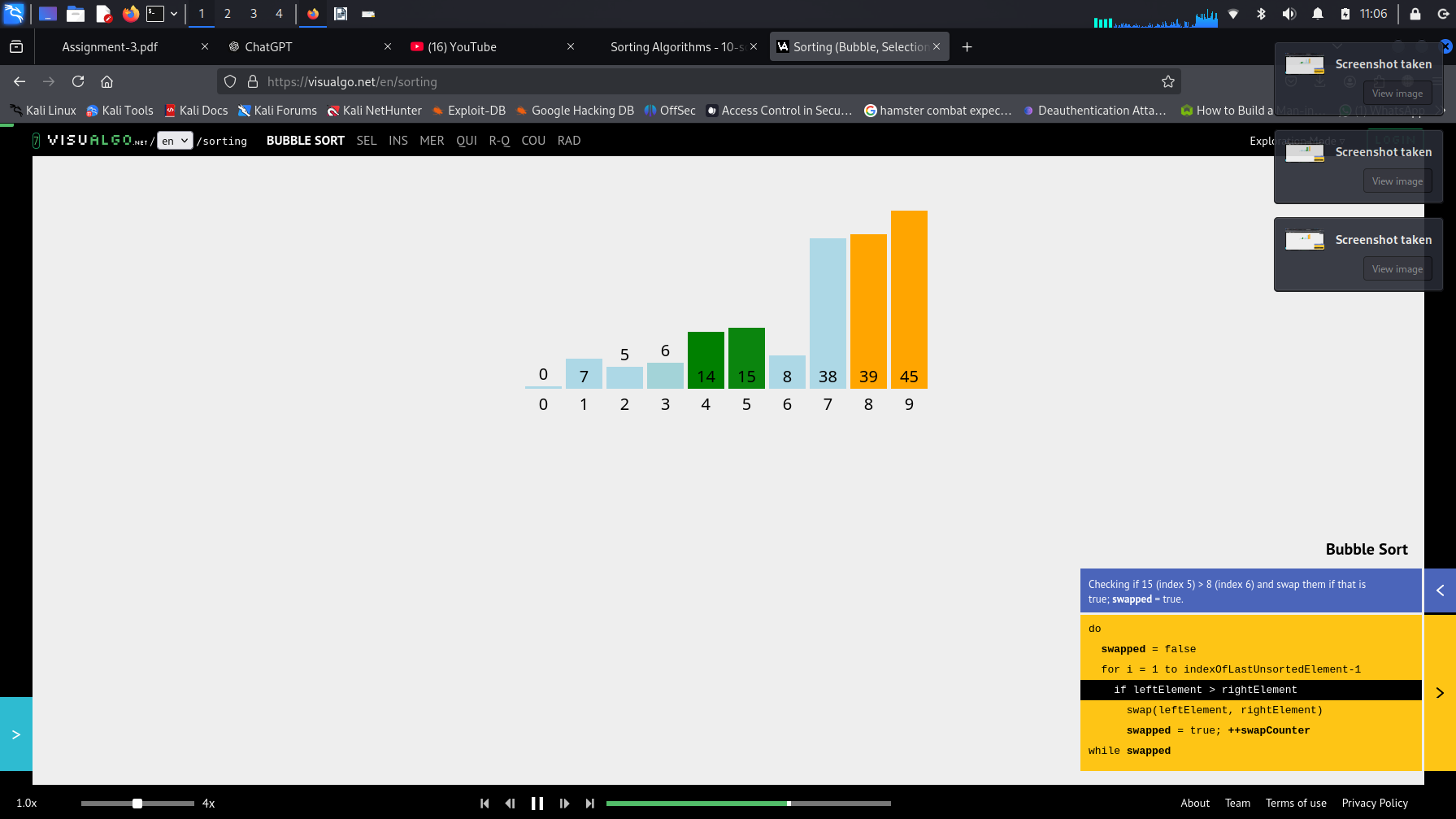
After pass 2

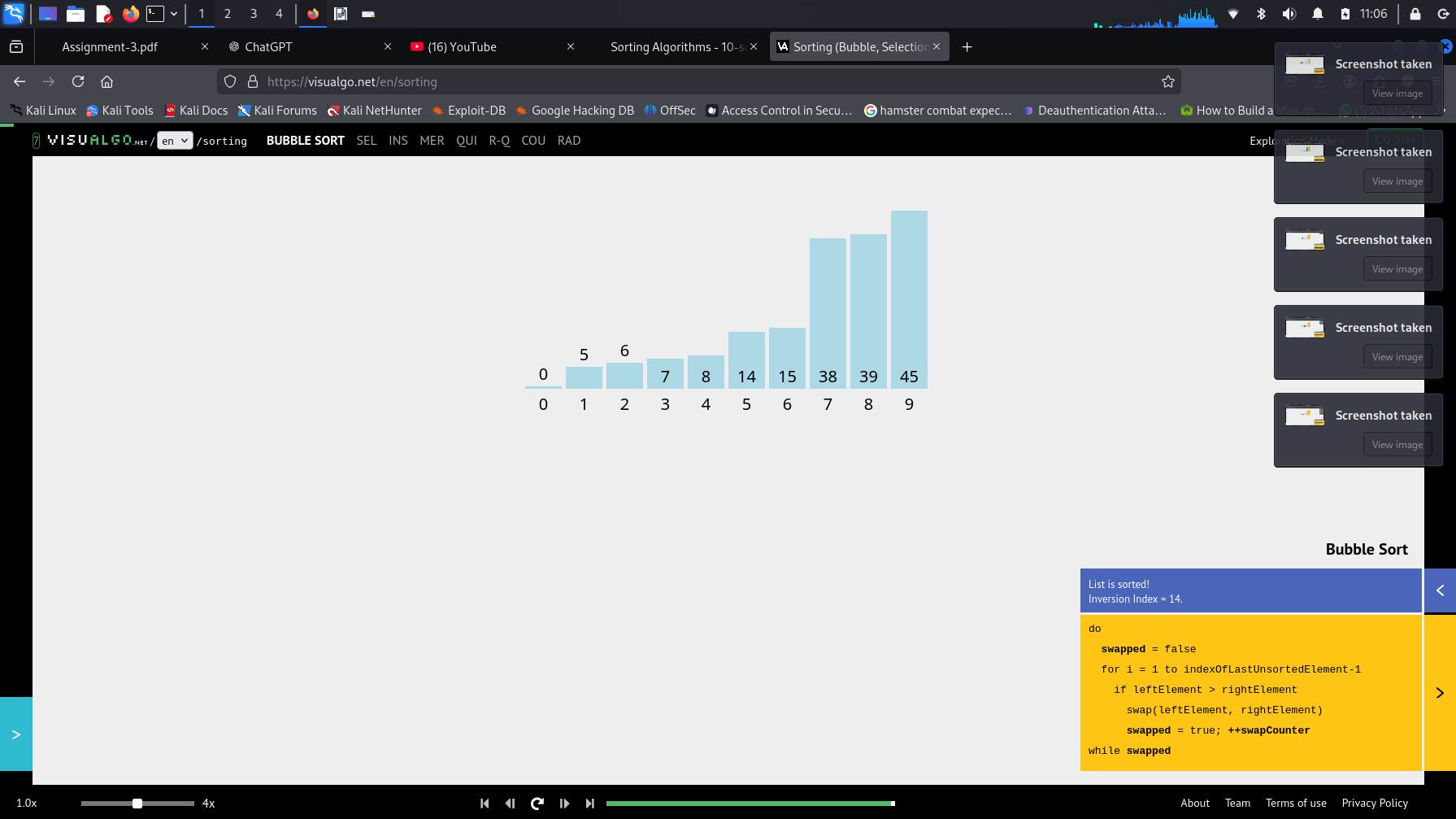
After pass 3

After pass 4









void **swap**( int &lhs, int &rhs );

void **bubleSort**(int a[], int n) { bool sorted = false;

int last = n-1;

for (int i = 0; (i < last) && !sorted; i++){

sorted = true;

for (int j=last; j > i; j--) if (a[j-1] > a[j]{

swap(a[j],a[j-1]);

sorted = false;

}

}

}

void **swap**( int &lhs, int &rhs ){ int tmp = lhs;

lhs = rhs; rhs = tmp;

}

**Tick Function**

int calculateTicks(int arr[], int n)

{ int ticks = 0; for (int i = 0; i < n - 1; i++)

{ bool swapped = false;

for (int j = 0; j < n - i - 1; j++)

{ ticks++;

if (arr[j] > arr[j + 1])

{ swap(arr[j], arr[j + 1]); ticks++;

swapped = true; }

}

if (!swapped) break;

}

return ticks; }

int main()

{ int originalArray[] = {23, 78, 45, 8, 32, 56};

int n = sizeof(originalArray) / sizeof(originalArray[0]);

int arr[n];

**Tick Function**

**// Worst Case:**

copy(begin(originalArray), end(originalArray), arr);

sort(arr, arr + n, greater<int>());

cout << "Ticks (Worst Case): " << calculateTicks(arr, n) << endl;

**// Best Case:**

copy(begin(originalArray), end(originalArray), arr);

sort(arr, arr + n);

/cout << "Ticks (Best Case): " << calculateTicks(arr, n) << endl;

**// Average Case:**

copy(begin(originalArray), end(originalArray), arr);

cout << "Ticks (Average Case): " << calculateTicks(arr, n) << endl;

* In general, we compare keys and move items (or exchange items) in a sorting algorithm (which uses key comparisons).

### So, to analyze a sorting algorithm we should count the number of key comparisons and the number of moves.

* + Ignoring other operations does not affect our final result.

**Time Calculation Function for all cases**

import time

def bubble\_sort(arr):

n = len(arr)

for i in range(n):

swapped = False

for j in range(0, n - i - 1):

if arr[j] > arr[j + 1]:

arr[j], arr[j + 1] = arr[j + 1], arr[j]

swapped = True

if not swapped:

break

def measure\_time(arr):

start\_time = time.time()

bubble\_sort(arr)

return time.time() - start\_time

**Time Calculation Function for all cases**

best\_case = [1, 2, 3, 4, 5]

average\_case = [3, 1, 4, 5, 2]

worst\_case = [5, 4, 3, 2, 1]

print(f"Best-case time: {measure\_time(best\_case[:]):.6f} seconds")

print(f"Average-case time: {measure\_time(average\_case[:]):.6f} seconds")

print(f"Worst-case time: {measure\_time(worst\_case[:]):.6f} seconds")

* ***Best-case:***  **O(n)**
  + Array is already sorted in ascending order.
  + Outer loop executes 1 time and inner loop n-1 times.
  + The number of moves: 0 O(1)
  + The number of key comparisons: (n-1) O(n)
* ***Worst-case:***  **O(n2)**
  + Array is in reverse order:
  + Outer loop is executed n-1 times and inner loop executes (n-1-i) times,

– The number of moves: 3\*((n-1)+(n-2)+...+3+2+1) = 3 \* n\*(n-1)/2 O(n2)

– The number of key comparisons: ((n-1)+(n-2)+...+3+2+1) = n\*(n-1)/2 O(n2)

* **Average-case**: **O(n2)**
  + We have to look at all possible initial data organizations.

### So, Bubble Sort is O(n2)

**Comparison of *N*, *logN* and *N2***

|  |  |  |
| --- | --- | --- |
| **N** | **O(LogN)** | **O(N2)** |
| 16 | 4 | 256 |
| 64 | 6 | 4K |
| 256 | 8 | 64K |
| 1,024 | 10 | 1M |
| 16,384 | 14 | 256M |
| 131,072 | 17 | 16G |
| 262,144 | 18 | 6.87E+10 |
| 524,288 | 19 | 2.74E+11 |
| 1,048,576 | 20 | 1.09E+12 |
| 1,073,741,824 | 30 | 1.15E+18 |

* The list is divided into two sublists, *sorted* and *unsorted*, which are divided by an imaginary wall.
* We find the smallest element from the unsorted sublist and swap it with the element at the beginning of the unsorted data.
* After each selection and swapping, the imaginary wall between the two sublists move one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
* Each time we move one element from the unsorted sublist to the sorted sublist, we say that we have completed a sort pass.
* A list of *n* elements requires *n-1* passes to completely rearrange the data.

#### Sorted Unsorted

Original List

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | |
| 23 | 78 | 45 | 8 | 32 | | 56 |
|  | | | | | | |
|  | | | | | | |
|  |  | | | | | |
| 8 | 78 | 45 | 23 | 32 | | 56 |
|  |  | | | | | |
|  | | | | | | |
|  | |  | | | | |
| 8 | 23 | 45 | 78 | 32 | | 56 |
|  | |  | | | | |
|  | | | | | | |
|  | | |  | | | |
| 8 | 23 | 32 | 78 | 45 | | 56 |
|  | | |  | | | |
|  | | | | | | |
|  | | | | |  | |
| 8 | 23 | 32 | 45 |  | 78 | 56 |
|  | | | | |  | |
|  | | | | | | |
|  | | | | | |  |
| 8 | 23 | 32 | 45 | 56 | | 78 |
|  | | | | | |  |

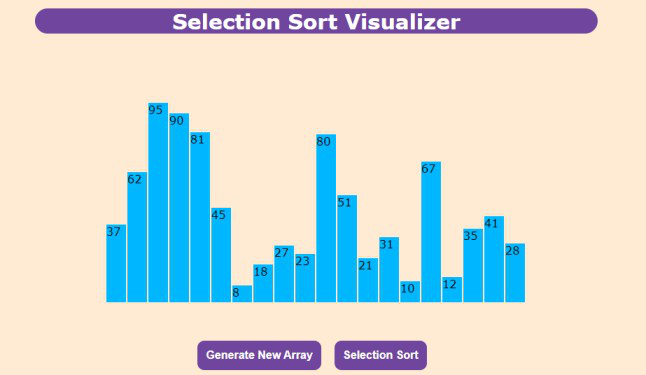
After pass 1

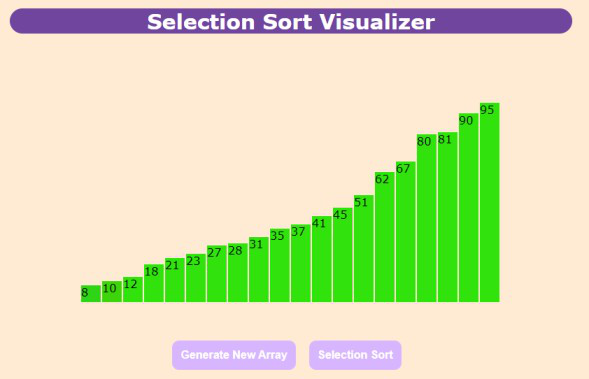
After pass 2

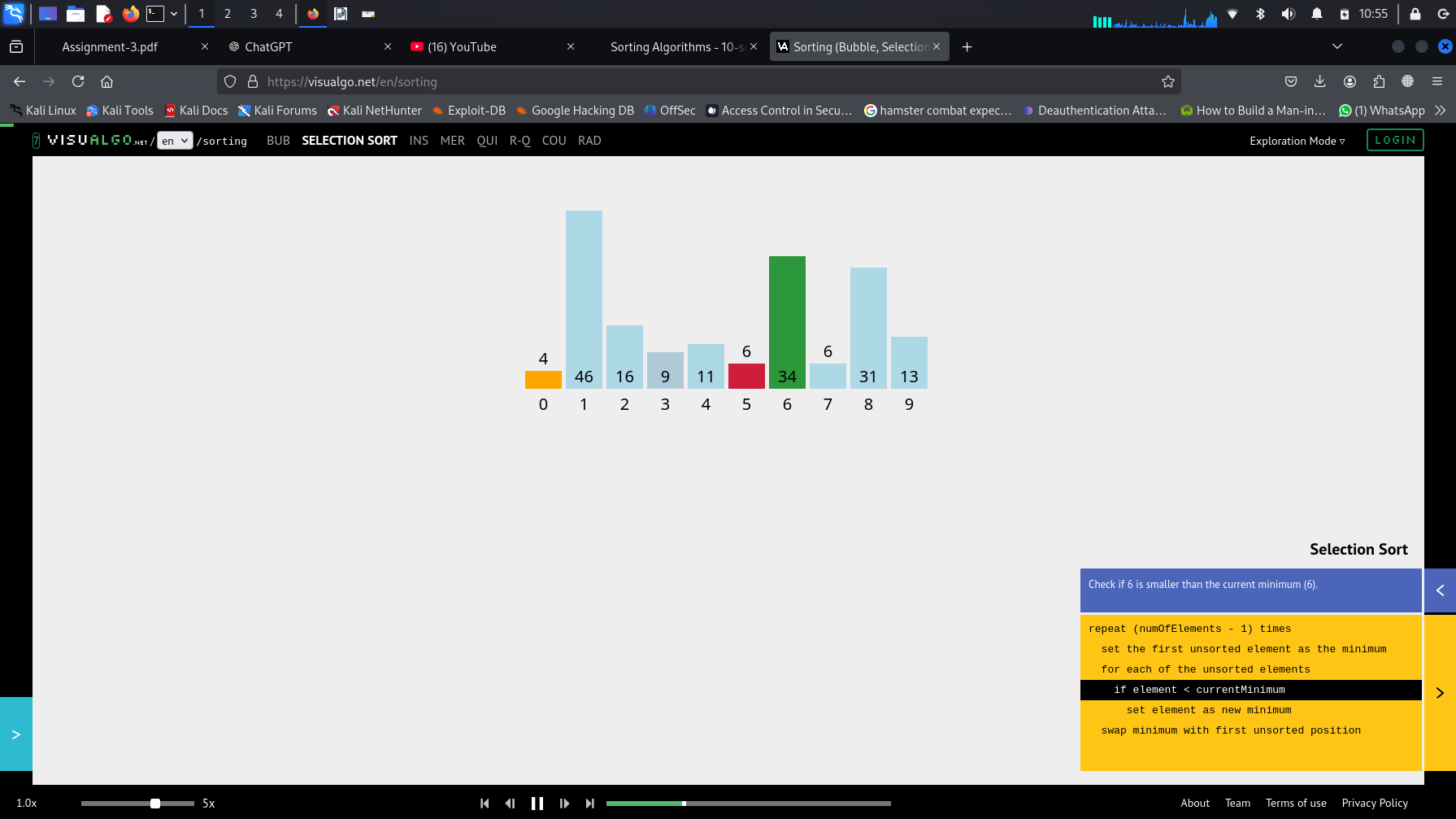
After pass 3

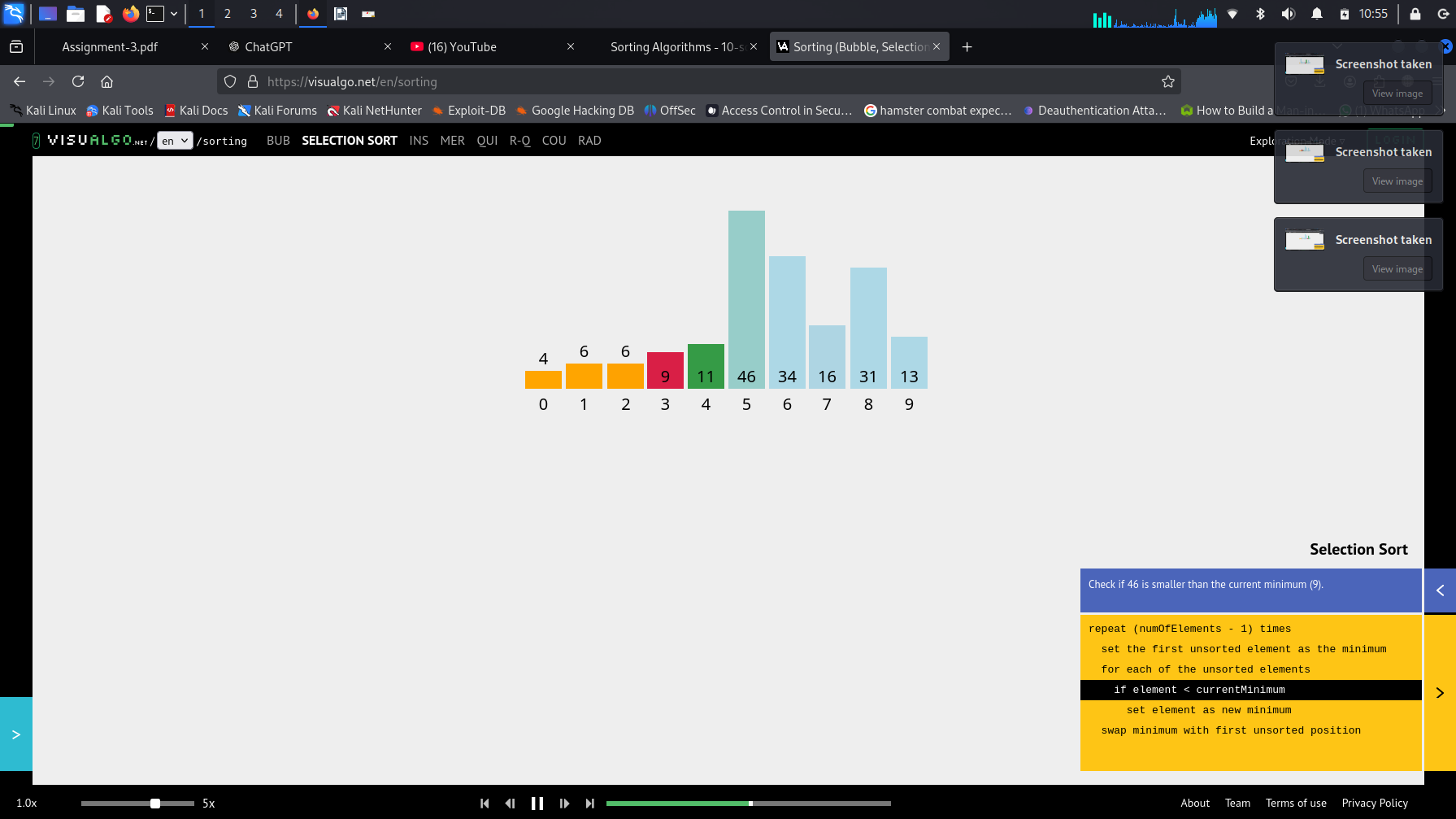
After pass 4

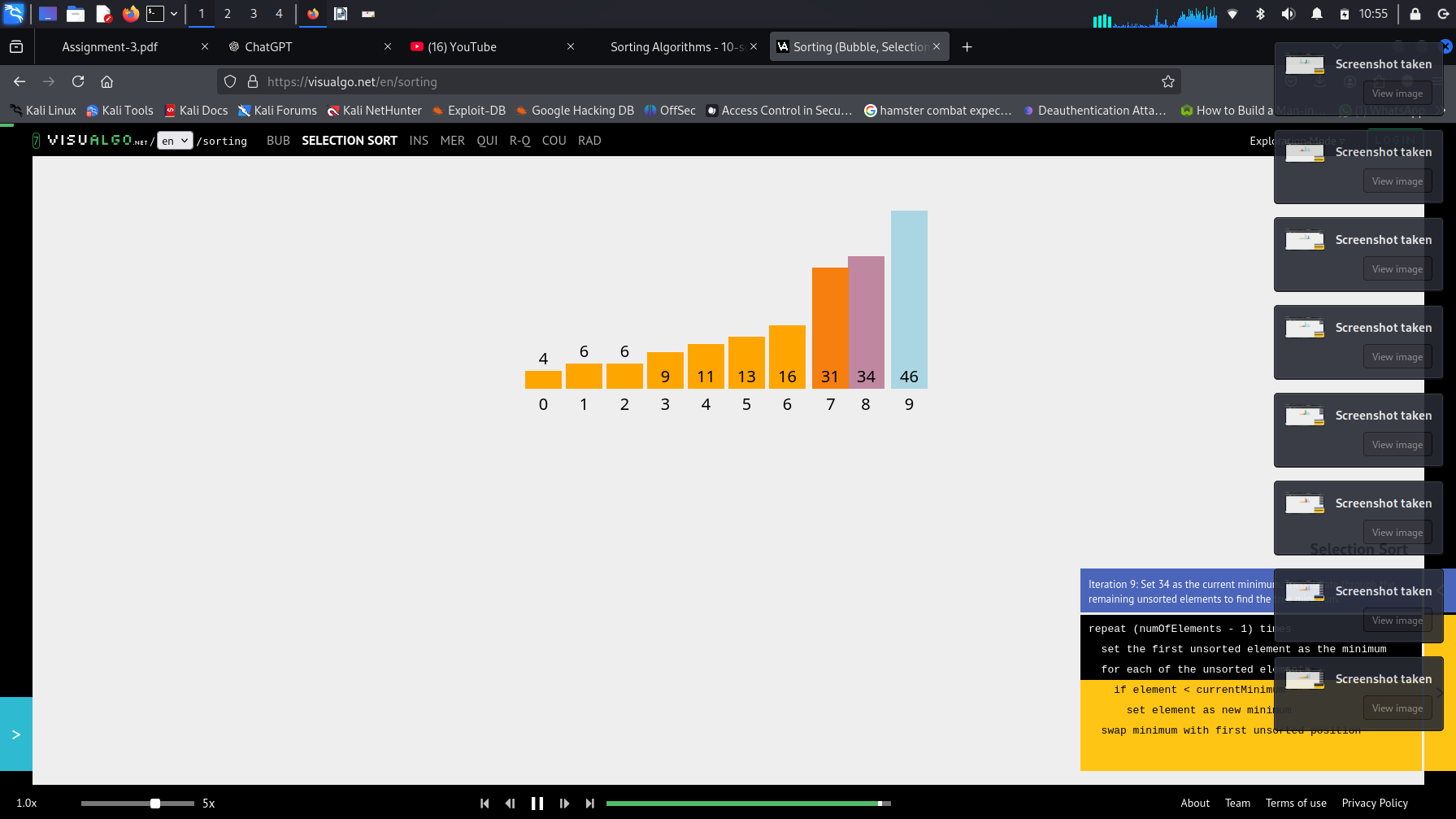
After pass 5











void **swap**( int &lhs, int &rhs );

void **selectionSort**( int a[], int n) { for (int i = 0; i < n-1; i++) {

int min = i;

for (int j = i+1; j < n; j++){ if (a[j] < a[min]) min = j;

}

swap(a[i], a[min]);

}

}

**Tick Function**

def selection\_sort(arr):

ticks = 0

n = len(arr)

for i in range(n):

min\_idx = i

ticks += 1

for j in range(i+1, n):

ticks += 1

if arr[j] < arr[min\_idx]:

min\_idx = j

ticks += 1

arr[i], arr[min\_idx] = arr[min\_idx], arr[i]

ticks += 1

return ticks

arr = [64, 25, 12, 22, 11]

ticks = selection\_sort(arr)

print(arr)

print(ticks)

**Time calculation for all cases(Best, Worst, Average)**

import time

def selection\_sort(arr):

for i in range(len(arr)):

min\_idx = i

for j in range(i+1, len(arr)):

if arr[j] < arr[min\_idx]:

min\_idx = j

arr[i], arr[min\_idx] = arr[min\_idx], arr[i]

def time\_sort(arr):

start = time.time()

selection\_sort(arr[:])

return time.time() - start

arr\_best = [1, 2, 3, 4, 5]

arr\_average = [64, 25, 12, 22, 11]

arr\_worst = [5, 4, 3, 2, 1]

print(f"Best case: {time\_sort(arr\_best):.6f} seconds")

print(f"Average case: {time\_sort(arr\_average):.6f} seconds")

print(f"Worst case: {time\_sort(arr\_worst):.6f} seconds")

* In selectionSort function, the outer for loop executes n-1 times.
* We invoke swap function once at each iteration.

Total Swaps: n-1

Total Moves: 3\*(n-1) (Each swap has three moves)

* The inner for loop executes the size of the unsorted part minus 1 (n-1-i), and in each iteration we make one key comparison.

of key comparisons = ((n-1)+(n-2)+...+3+2+1) = n\*(n-1)/2

#### So, Selection sort is O(n2)

* The best case, the worst case, and the average case of the selection sort algorithm are same. all of them are **O(n2)**
  + This means that the behavior of the selection sort algorithm does not depend on the

initial organization of data.

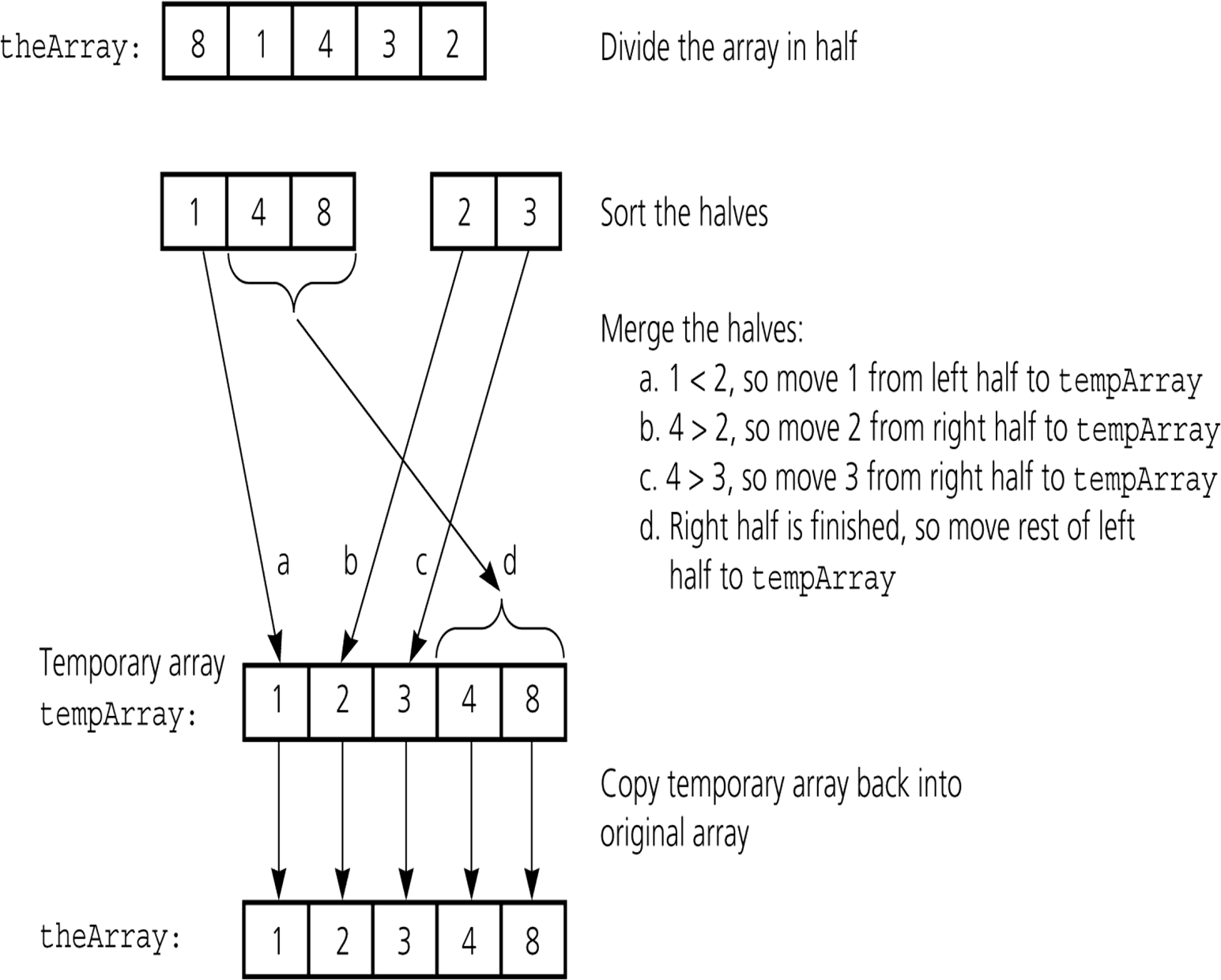
* + Since O(n2) grows so rapidly, the selection sort algorithm is appropriate only for small n.
  + Although the selection sort algorithm requires O(n2) key comparisons, it only requires O(n) moves.
  + A selection sort could be a good choice if data moves are costly but key comparisons are not costly (short keys, long records).

Mergesort algorithm is one of two important divide-and-conquer

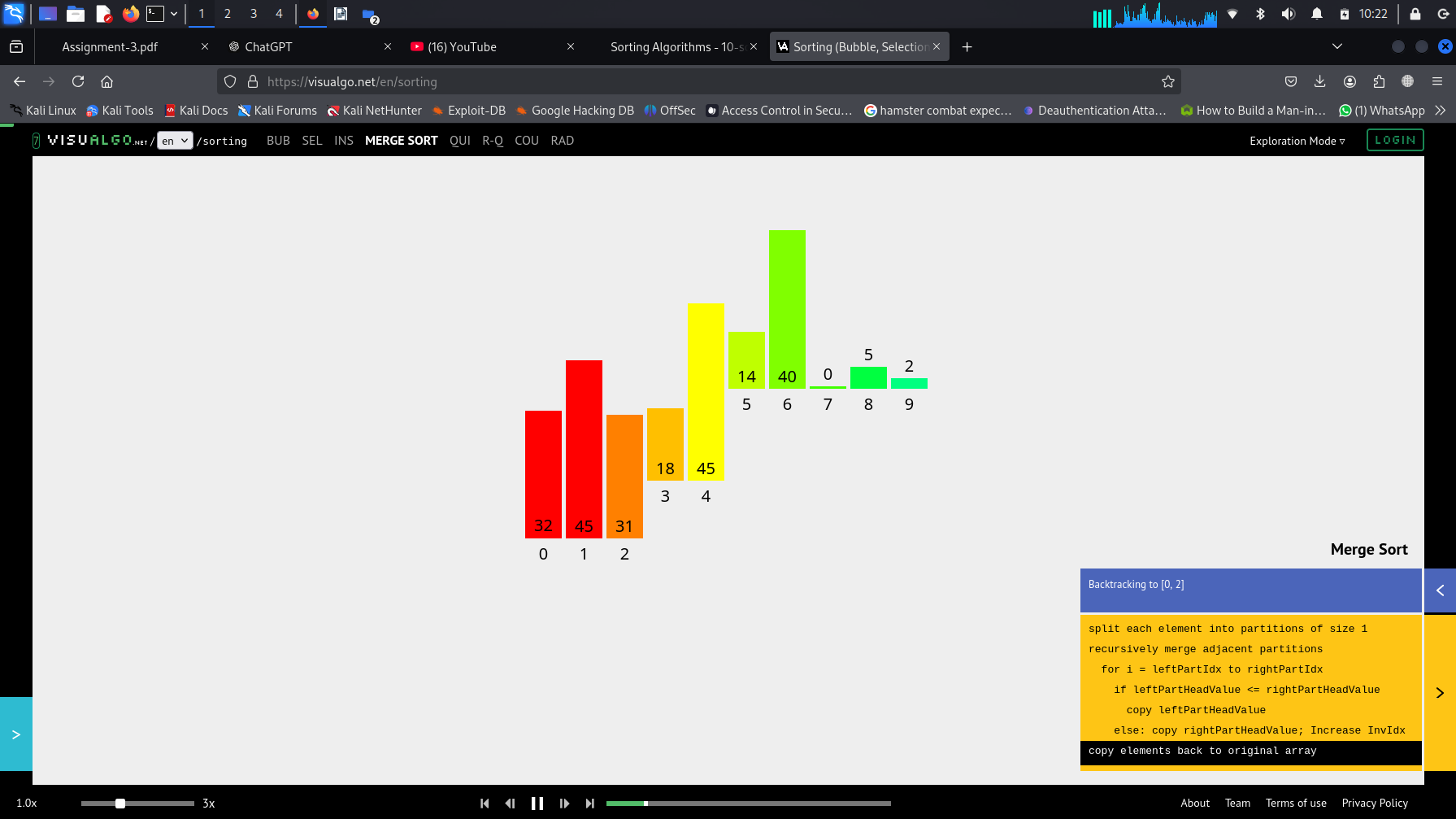
sorting algorithms (the other one is quicksort).

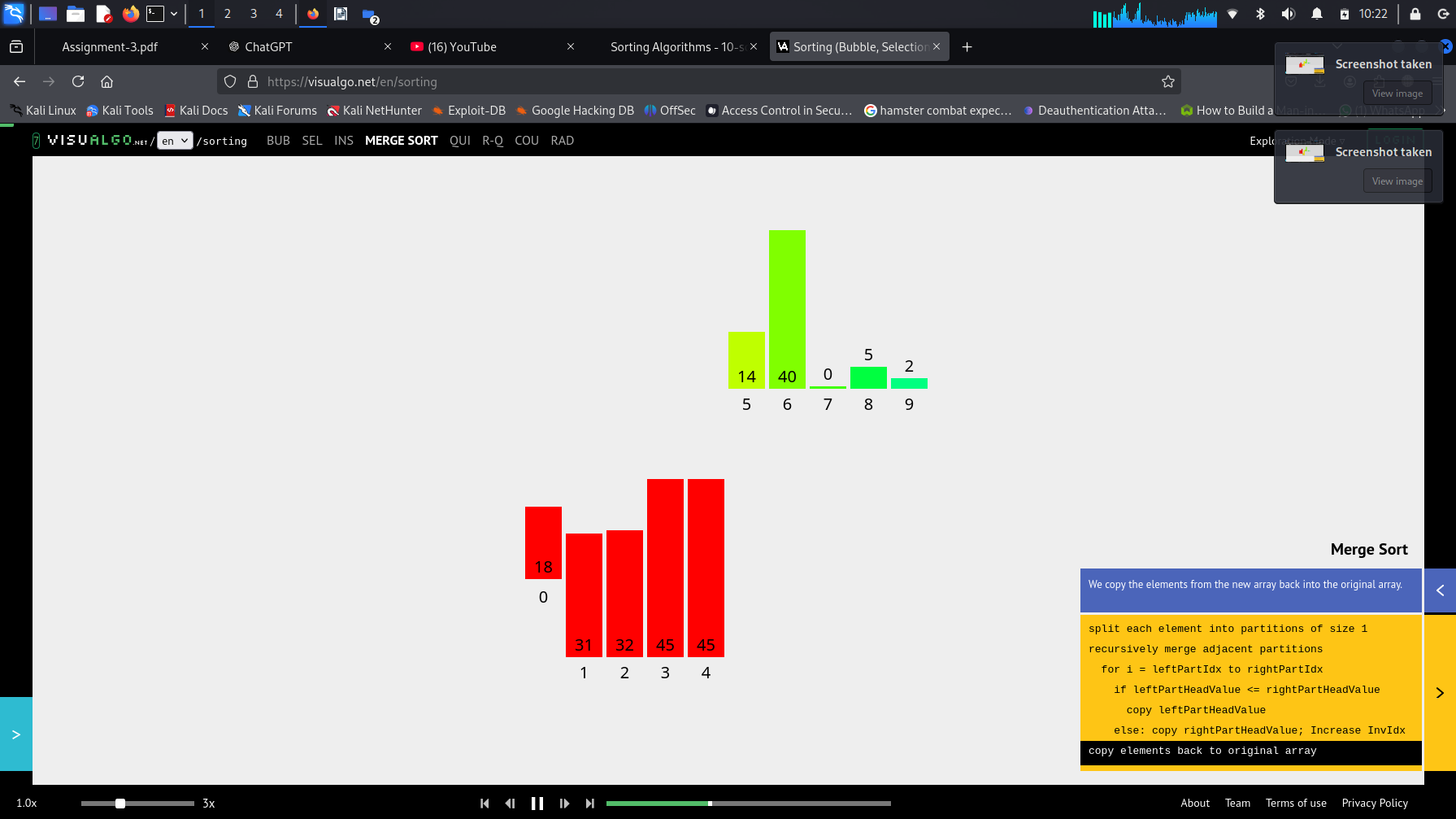
* It is a recursive algorithm.
  + Divides the list into halves,
  + Sort each halve separately, and
  + Then merge the sorted halves into one sorted array.

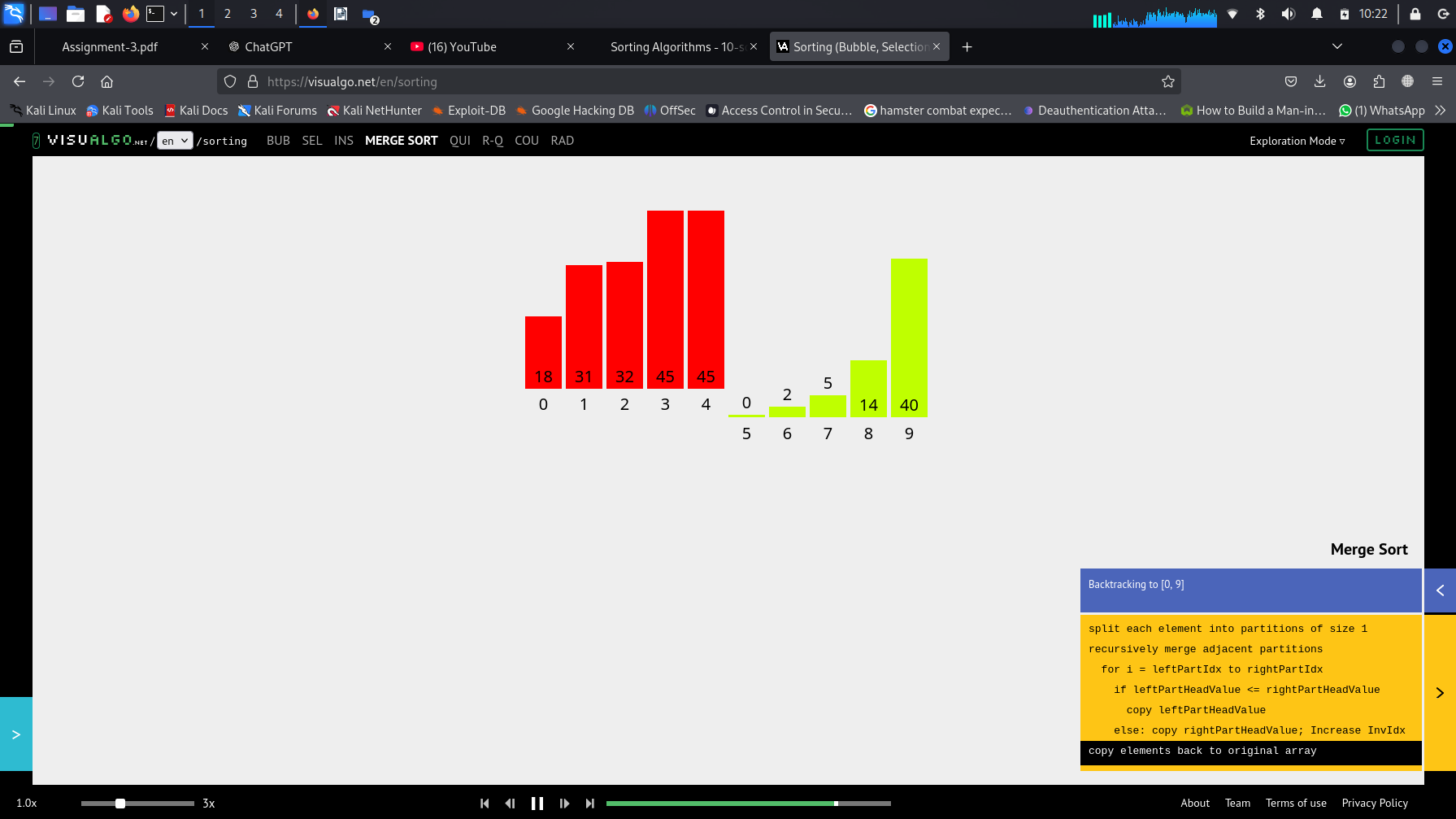
# Mergesort - Example



Merge Sort







void **merge**(int theArray[], int first, int mid, int last) { int tempArray[last+1]; // temporary array

int first1 = first; // beginning of first subarray int last1 = mid; // end of first subarray

int first2 = mid + 1; // beginning of second subarray int last2 = last; // end of second subarray

int index = first1; // next available location in tempArray for ( ; (first1 <= last1) && (first2 <= last2); ++index) {

if (theArray[first1] < theArray[first2]) { tempArray[index] = theArray[first1];

++first1;

}

else {

tempArray[index] = theArray[first2];

++first2;

}

}

# Merge Sort (cont.)

// finish off the first subarray, if necessary

for (; first1 <= last1; ++first1, ++index) tempArray[index] = theArray[first1];

// finish off the second subarray, if necessary for (; first2 <= last2; ++first2, ++index)

tempArray[index] = theArray[first2];

// copy the result back into the original array for (index = first; index <= last; ++index)

theArray[index] = tempArray[index];

}

# Merge Sort

void **mergesort**(int theArray[], int first, int last) {

if (first < last) {

int mid = (first + last)/2; // index of midpoint

// dived into two halves at the middle mergesort(theArray, first, mid); mergesort(theArray, mid+1, last);

// merge the two halves merge(theArray, first, mid, last);

}

}

# **Merge Sort Tick Function**

# def merge\_sort(arr):

# ticks = 0

# if len(arr) <= 1:

# return arr, ticks

# mid = len(arr) // 2

# left\_sorted, left\_ticks = merge\_sort(arr[:mid])

# right\_sorted, right\_ticks = merge\_sort(arr[mid:])

# ticks += left\_ticks + right\_ticks

# sorted\_arr, merge\_ticks = merge(left\_sorted, right\_sorted)

# ticks += merge\_ticks

# return sorted\_arr, ticks

# def merge(left, right):

# result = []

# i = j = ticks = 0

# while i < len(left) and j < len(right):

# ticks += 2

# if left[i] <= right[j]:

# result.append(left[i])

# i += 1

# else:

# result.append(right[j])

# j += 1

# result.extend(left[i:])

# result.extend(right[j:])

# ticks += len(left[i:]) + len(right[j:])

# return result, ticks

# arr = [64, 25, 12, 22, 11]

# sorted\_arr, ticks = merge\_sort(arr)

# print("Sorted array:", sorted\_arr)

# print("Total ticks (operations):", ticks)

# **Merge Sort Time Calculation for (BEST,AVERAGE,WORST)**

# **Cases**

# import time

# import random

# def merge\_sort(arr):

# if len(arr) > 1:

# mid = len(arr) // 2

# L = arr[:mid]

# R = arr[mid:]

# merge\_sort(L)

# merge\_sort(R)

# i = j = k = 0

# while i < len(L) and j < len(R):

# if L[i] < R[j]:

# arr[k] = L[i]

# i += 1

# else:

# arr[k] = R[j]

# j += 1

# k += 1

# while i < len(L):

# arr[k] = L[i]

# i += 1

# k += 1

# while j < len(R):

# arr[k] = R[j]

# j += 1

# k += 1

# def time\_merge\_sort(arr):

# start\_time = time.time()

# merge\_sort(arr)

# return time.time() - start\_time

# def generate\_sorted\_array(size):

# return list(range(size))

# def generate\_reverse\_sorted\_array(size):

# return list(range(size, 0, -1))

# def generate\_random\_array(size):

# return [random.randint(0, 10000) for \_ in range(size)]

# if \_\_name\_\_ == "\_\_main\_\_":

# array\_size = 1000

# best\_case\_time = time\_merge\_sort(generate\_sorted\_array(array\_size))

# print(f"Best case (sorted): {best\_case\_time:.6f} seconds")

# worst\_case\_time = time\_merge\_sort(generate\_reverse\_sorted\_array(array\_size))

# print(f"Worst case (reverse sorted): {worst\_case\_time:.6f} seconds")

# average\_case\_time = time\_merge\_sort(generate\_random\_array(array\_size))

# print(f"Average case (random): {average\_case\_time:.6f} seconds")

# Merge Sort - Example

*divide*

7

1

5 4 7 2

divide

5 4

divide

4

divide

divide

3

divide

merge

6

merge

5

2

9

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 6 | 3 | 9 | 1 | 5 | 4 | 7 | 2 |

|  |  |  |  |
| --- | --- | --- | --- |
| 6 | 3 | 9 | 1 |

|  |  |
| --- | --- |
| 7 | 2 |

|  |  |
| --- | --- |
| 9 | 1 |

|  |  |
| --- | --- |
| 6 | 3 |

*divide*

*merge merge*

|  |  |
| --- | --- |
| 2 | 7 |

|  |  |
| --- | --- |
| 3 | 6 |

|  |  |
| --- | --- |
| 1 | 9 |

|  |  |
| --- | --- |
| 4 | 5 |

*merge merge*

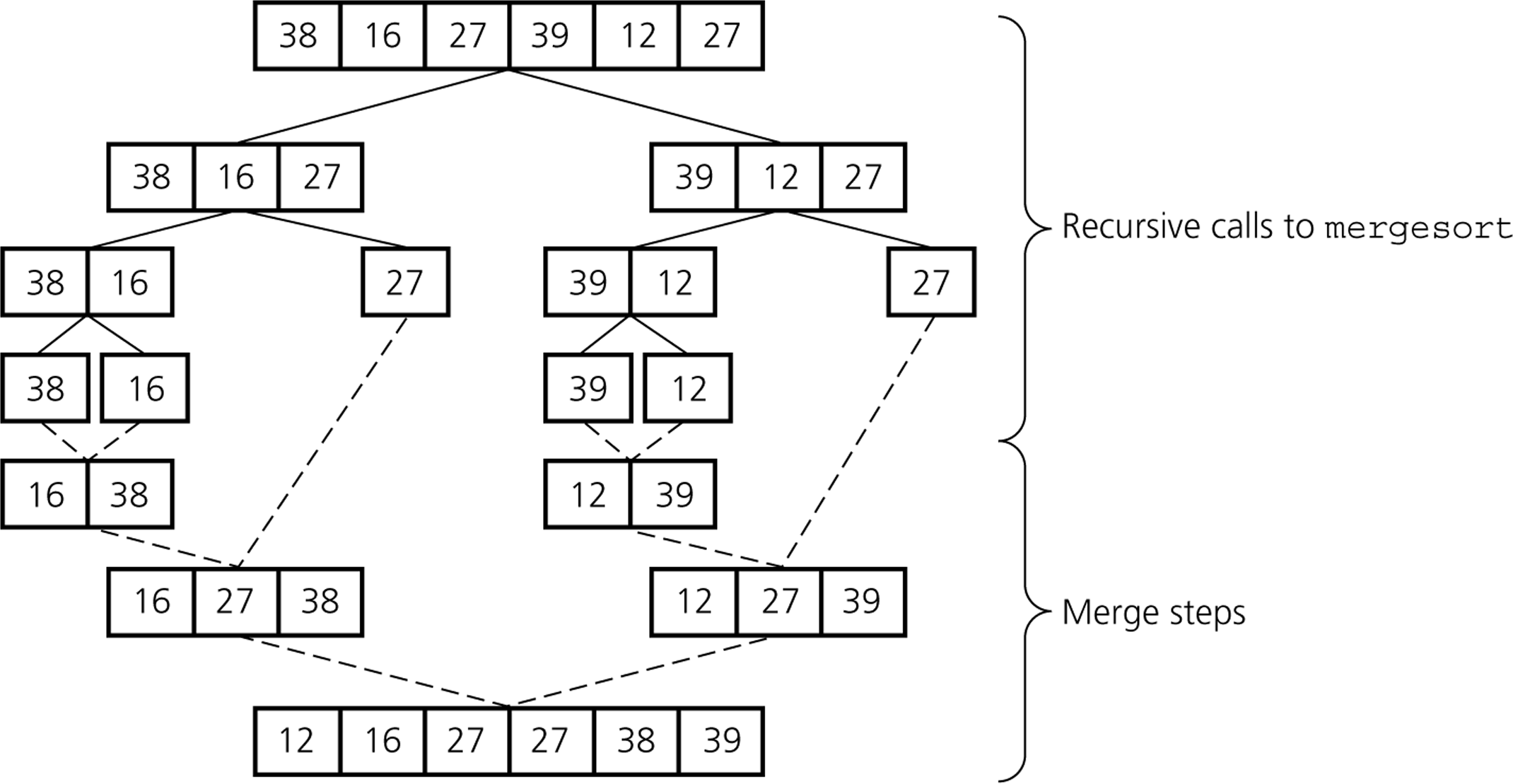
*merge*

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 3 | 6 | 9 |

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 4 | 5 | 7 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 7 | 8 | 9 |

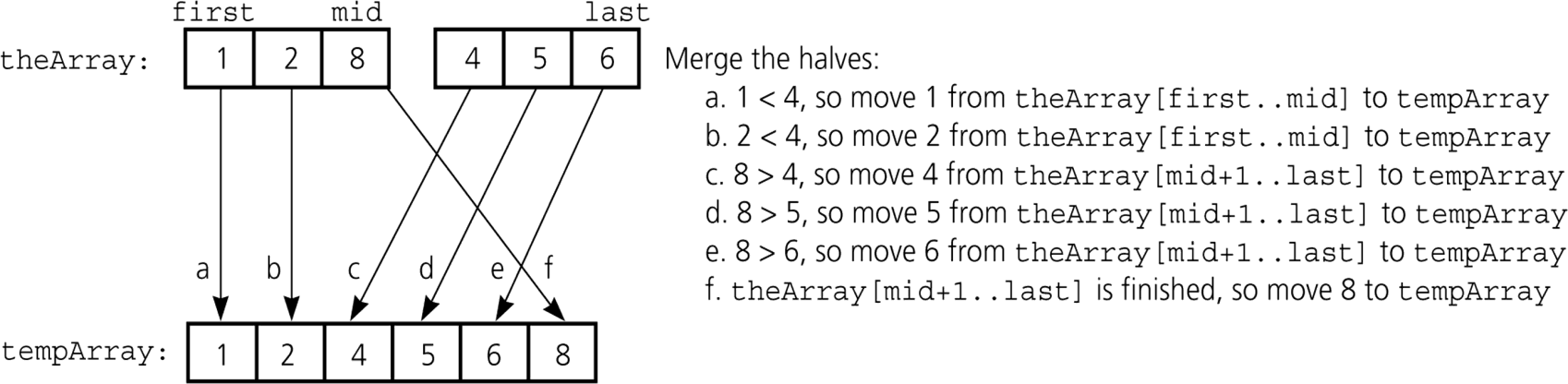
# Mergesort – Example2



Mergesort – Analysis of Merge

**A worst-case instance of the merge step in *mergesort***

Some elements in the first array are smaller and some elements are larger than all the elements in the second array



# Mergesort – Analysis of Merge (cont.)

Merging two sorted arrays of size ***k***

##### Best-case:

1

0 2k-1

0

k-

......

0

k-1

......

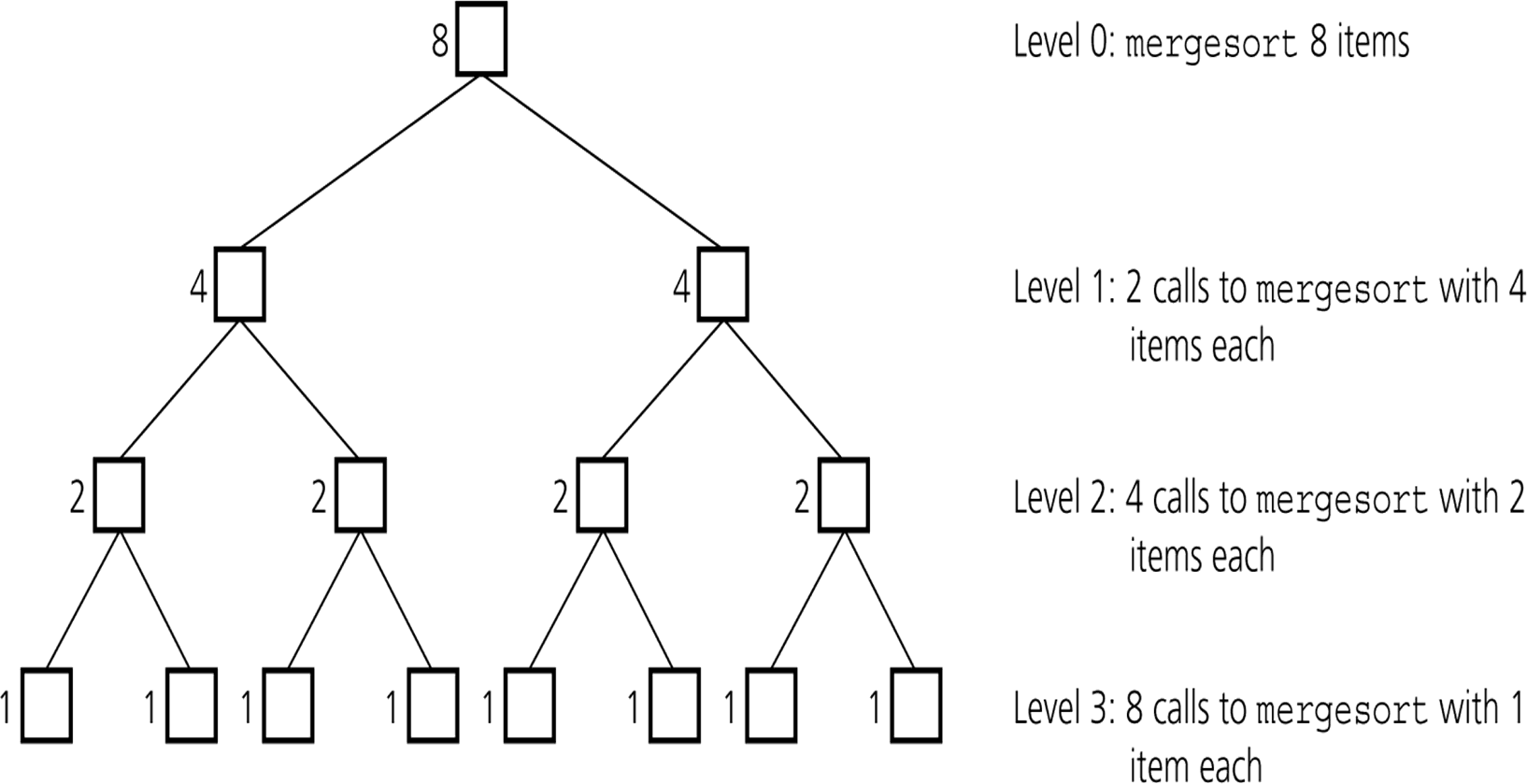
|  |  |  |
| --- | --- | --- |
|  | ...... |  |

* + All the elements in the first array are smaller (or larger) than all the elements in the second array.
  + The number of moves: 2k + 2k
  + The number of key comparisons: k

##### Worst-case:

* + The number of moves: 2k + 2k
  + The number of key comparisons: 2k-1

Levels of recursive calls to *mergesort*, given an array of eight items



##### Worst-case –

The number of key comparisons:

= 20\*(2\*2m-1-1) + 21\*(2\*2m-2-1) + ... + 2m-1\*(2\*20-1)

= (2m - 20) + (2m - 21) + ... + (2m – 2m-1) ( m terms )

= m2m – (20 + 21 + ….. + 2m-1)

= m\*2m –

*m*1

2*i*



*i*0

= m\*2m – 2m – 1

Using m = log n

**= n \* log2n – n – 1**

**O (n \* log2n )**

* Mergesort is extremely efficient algorithm with respect to time.

– Both worst case and average cases are **O (n \* log2n )**

* But, mergesort requires an extra array whose size equals to the size of the original array.

# Quicksort

* Like mergesort, Quicksort is also based on

the *divide-and-conquer* paradigm.

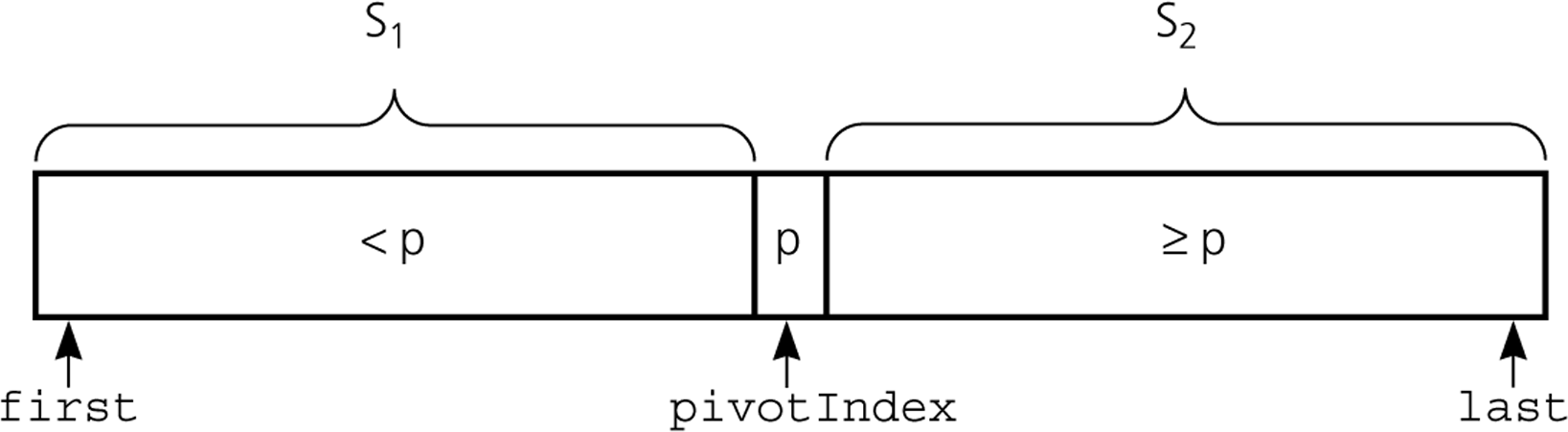
* But it uses this technique in a somewhat opposite manner, as all the hard work is done *before* the recursive calls.
* It works as follows:
  1. First, it partitions an array into two parts with respect to a pivot,
  2. Then, it sorts the parts independently,
  3. Finally, it combines the sorted subsequences by a simple concatenation.

# Quicksort (cont.)

The quick-sort algorithm consists of the following three steps:

1. ***Divide***: Partition the list.
   * To partition the list, we first choose some element from the list for which we hope about half the elements will come before and half after. Call this element the ***pivot***.
   * Then we partition the elements so that all those with values less than the pivot come in one sublist and all those with greater values come in another.
2. ***Recursion***: Recursively sort the sublists separately.
3. ***Conquer***: Put the sorted sublists together.

* Partitioning places the pivot in its correct place position within the array.



Partitions ***theArray[first..last]*** such that:

**S1** = ***theArray[first..pivotIndex-1]*** < ***pivot theArray[pivotIndex]*** == ***pivot***

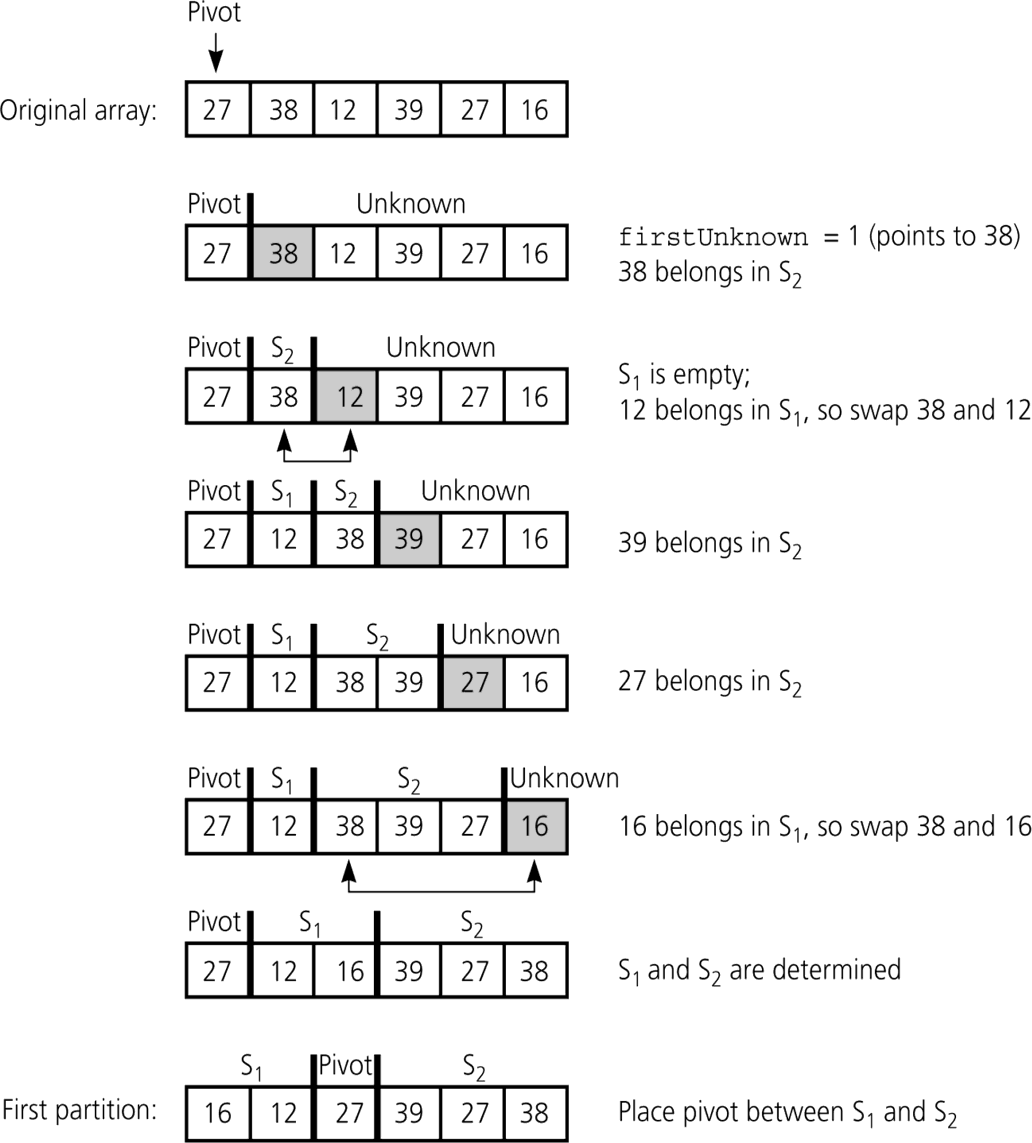
**S2** = ***theArray[pivotIndex+1..last]*** >= ***pivot***

* Generates two smaller sorting problems.
  + Sort the left section of the array
  + Sort the right section of the array
  + Two smaller sorting problems are solved recursively to solve bigger sorting problem.
* Which array item should be selected as pivot?
  + Somehow we have to select a pivot, and we hope that we will get a good partitioning.
  + If the items in the array arranged randomly, we choose a pivot

randomly.

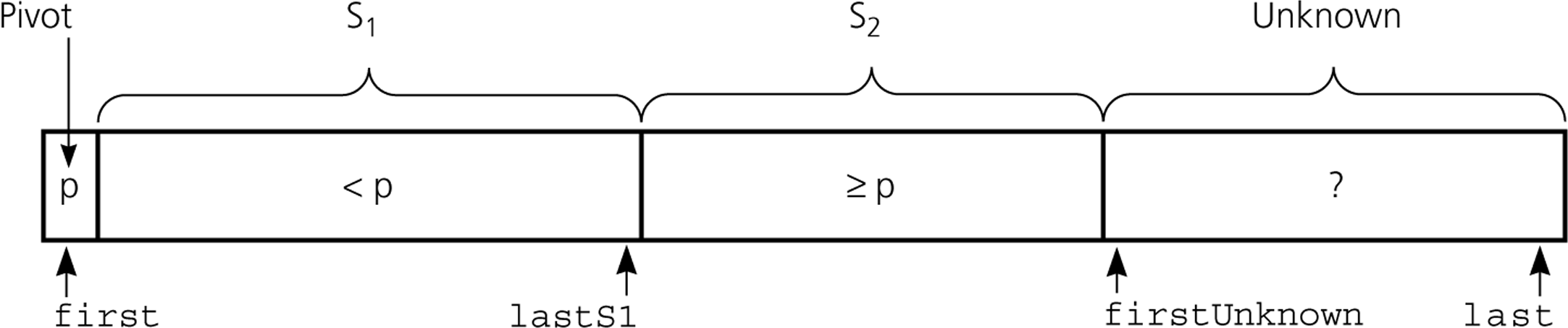
* + We can choose the first or last element as a pivot (it may not give a good partitioning).
  + We can use different techniques to select the pivot.
* Put this pivot into the first location of the array before partitioning

# Partition (cont.)

***Developing the first partition of an array when the pivot is the first item***

## Partition (cont.)

***Invariant for the partition algorithm***

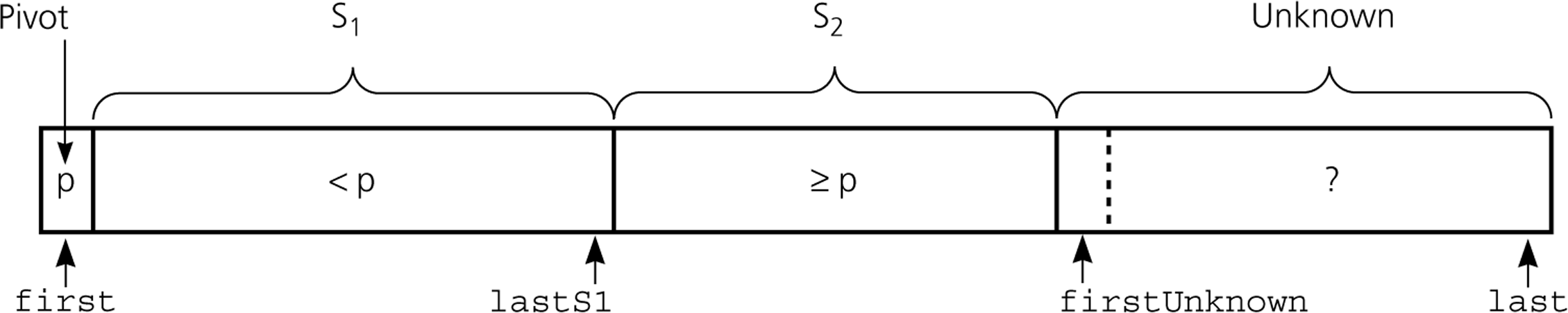
******

S1: theArray[first+1..lastS1] < pivot

S2: theArray[lastS1+1..firstUnknown-1] >= pivot

When ***theArray[firstUnknown]*** >= ***pivot***

Move ***theArray[firstUnknown]*** into ***S2*** by incrementing **firstUnknown**.

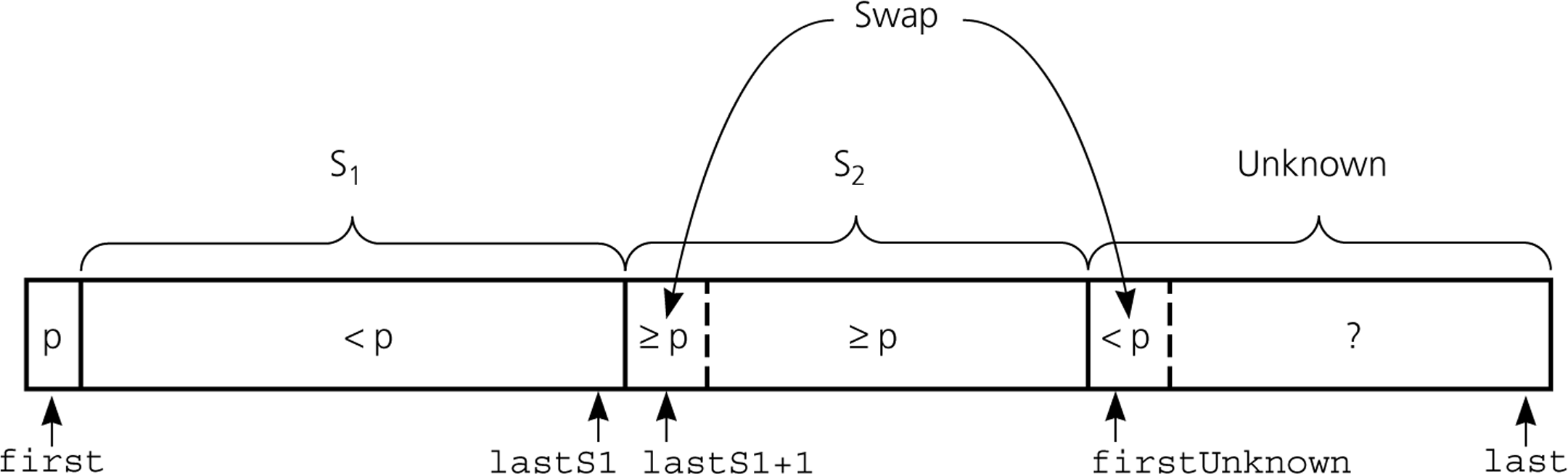


When ***theArray[firstUnknown]*** *<* ***pivot***

Move ***theArray[firstUnknown]*** into **S1** by

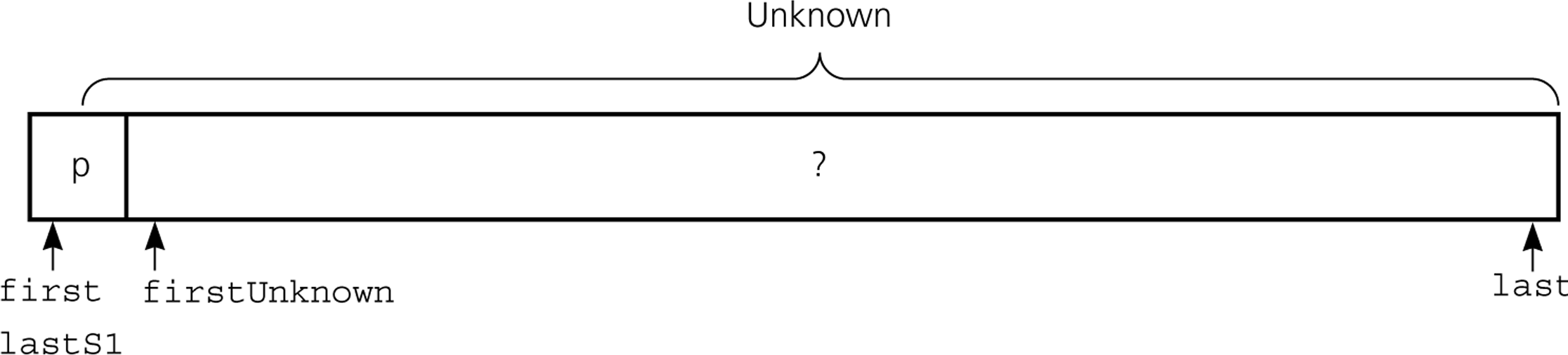
swapping ***theArray[firstUnknown]*** with ***theArray[lastS1+1]*** and

incrementing both ***lastS1*** and ***firstUnknown****.*

**

## Partition (cont.)

***Initial state of the array***

******

**lastS1** = first

**firstUnknown** = first + 1

**S1**: theArray[first+1..lastS1]: **Empty**

**S2**: theArray[lastS1+1..firstUnknown-1]: **Empty**

Quick Sort

# 

# 

# 

# Partition Function

void **swap**( int &lhs, int &rhs );

void **partition**(int theArray[], int first, int last, int &pivotIndex) {

// Choose and place pivot in theArray[first]

choosePivot(theArray, first, last);

// Initialize

int pivot = theArray[first];

int lastS1 = first;

int firstUnknown = first + 1;

# Partition Function (cont.)

// Move one item at a time until unknown region is empty

for (; firstUnknown <= last; ++firstUnknown) {

if (theArray[firstUnknown] < pivot) { // Belongs to S1

++lastS1; // Expands S1 by incrementing lastS1

// Swap firstUnknown with lastS1

swap(theArray[firstUnknown], theArray[lastS1]);

}

// else belongs to S2, ++firstUnknown in the loop

// places it to S2

}

// Place pivot in proper position and mark its location swap(theArray[first], theArray[lastS1]);

pivotIndex = lastS1;

}

void **quicksort**(int theArray[], int first, int last) {

int pivotIndex;

if (first < last) {

// create the partition: S1, pivot, S2 partition(theArray, first, last, pivotIndex);

// sort regions S1 and S2 quicksort(theArray, first, pivotIndex-1); quicksort(theArray, pivotIndex+1, last);

}

}

**Tick Function Quick Sort**

def quick\_sort(arr, low, high):

ticks = 0

if low < high:

pivot\_index, part\_ticks = partition(arr, low, high)

ticks += part\_ticks

ticks += quick\_sort(arr, low, pivot\_index - 1)

ticks += quick\_sort(arr, pivot\_index + 1, high)

return ticks

def partition(arr, low, high):

pivot = arr[high]

i = low - 1

ticks = 0

for j in range(low, high):

ticks += 1

if arr[j] < pivot:

i += 1

arr[i], arr[j] = arr[j], arr[i]

ticks += 1

arr[i + 1], arr[high] = arr[high], arr[i + 1]

ticks += 1

return i + 1, ticks

arr = [64, 25, 12, 22, 11]

ticks = quick\_sort(arr, 0, len(arr) - 1)

print("Sorted array:", arr)

print("Total ticks (operations):", ticks)

**Quick Sort Time Calculation for (BEST,AVERAGE,WORST)**

**Cases**

import time

import random

def quick\_sort(arr):

if len(arr) <= 1:

return arr

pivot = arr[len(arr) // 2]

left = [x for x in arr if x < pivot]

middle = [x for x in arr if x == pivot]

right = [x for x in arr if x > pivot]

return quick\_sort(left) + middle + quick\_sort(right)

def time\_quick\_sort(arr):

start\_time = time.time()

quick\_sort(arr)

return time.time() - start\_time

def generate\_sorted\_array(size):

return list(range(size))

def generate\_reverse\_sorted\_array(size):

return list(range(size, 0, -1))

def generate\_random\_array(size):

return [random.randint(0, 10000) for \_ in range(size)]

if \_\_name\_\_ == "\_\_main\_\_":

array\_size = 1000

best\_case\_time = time\_quick\_sort(generate\_sorted\_array(array\_size))

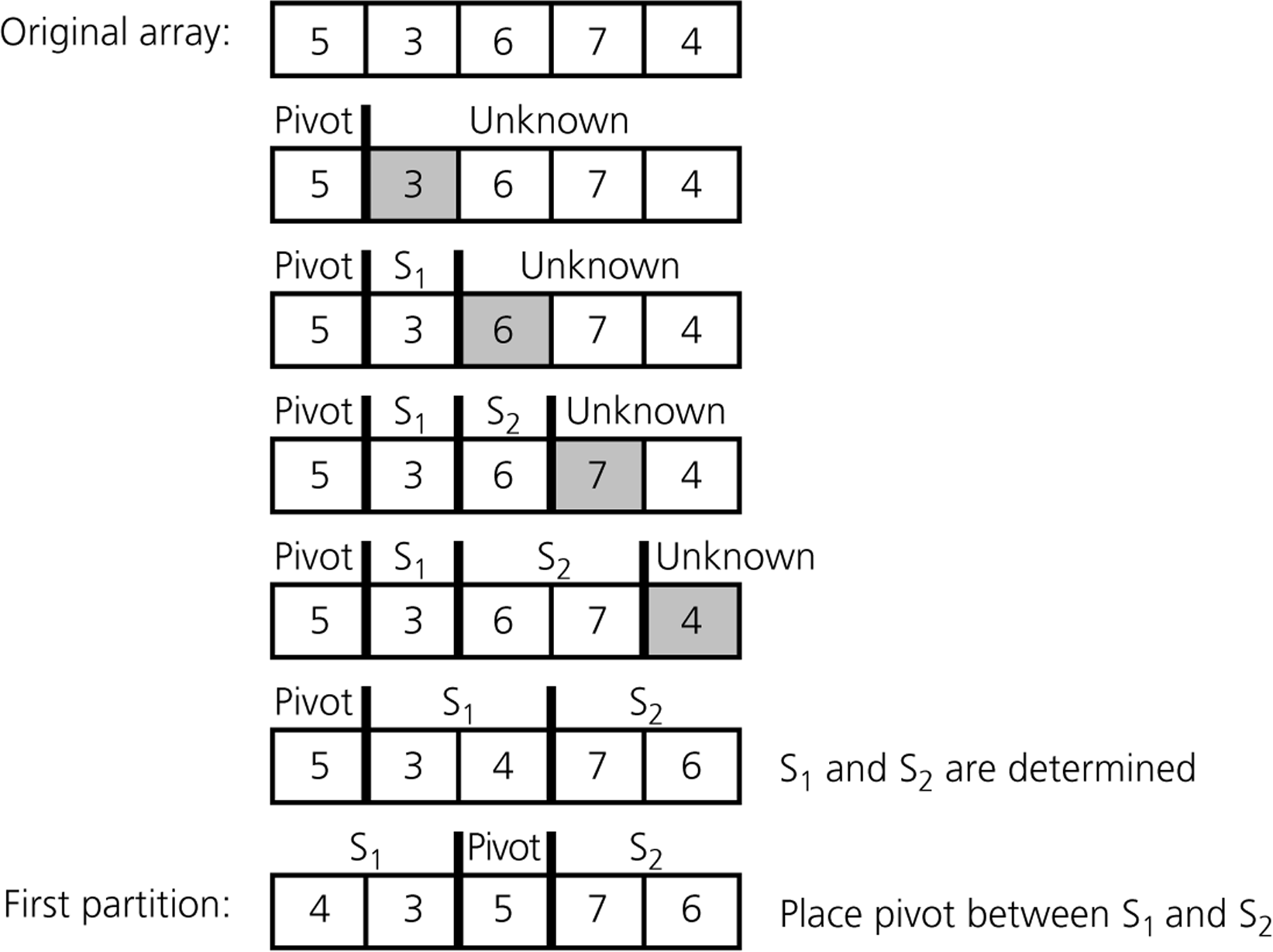
print(f"Best case (sorted): {best\_case\_time:.6f} seconds")

worst\_case\_time = time\_quick\_sort(generate\_reverse\_sorted\_array(array\_size))

print(f"Worst case (reverse sorted): {worst\_case\_time:.6f} seconds")

average\_case\_time = time\_quick\_sort(generate\_random\_array(array\_size))

print(f"Average case (random): {average\_case\_time:.6f} seconds")

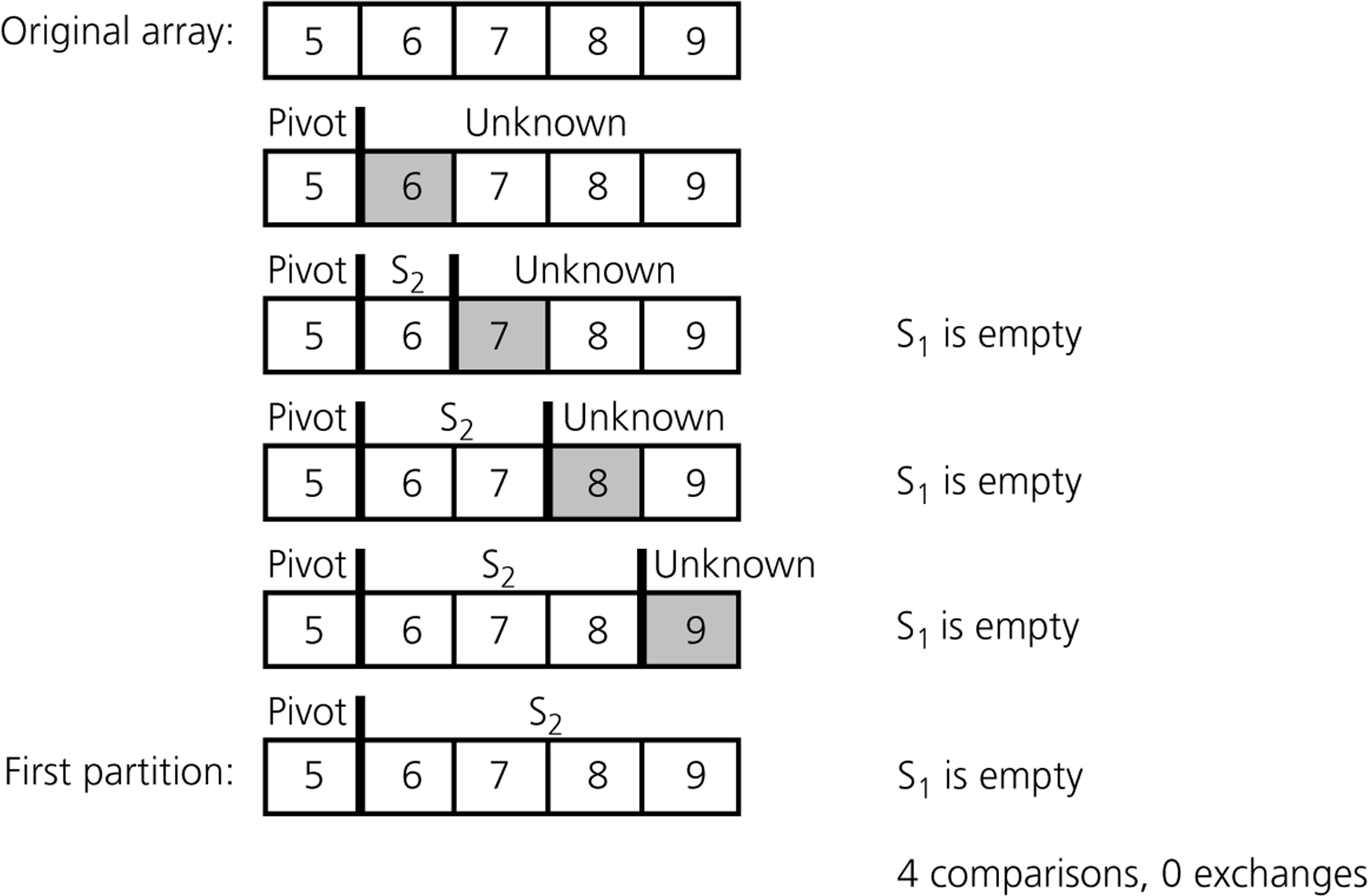
***An average- case partitioning with quicksort***

* Quicksort is **O(n\*log2n)** in the best case and average case.
* Quicksort is slow when the array is sorted and we choose the first element as the pivot.
* Although the worst case behavior is not so good, its average case

behavior is much better than its worst case.

* So, Quicksort is one of best sorting algorithms using key comparisons.

***A worst-case partitioning with quicksort***

******

***Worst Case:*** (assume that we are selecting the first element as pivot)

* The pivot divides the list of size n into two sublists of sizes 0 and n-1.
* The number of key comparisons

= n-1 + n-2 + ... + 1

= n(n-1)/2

= **n2/2 – n/2**  **O(n2)**

* The number of swaps =

= ( n-1 + n-2 + ... + 1) + (n-1)

swaps inside of

the for loop

swaps outside of the for loop

= (n-1) + n(n-1)/2

= **n2/2 + n/2 - 1 O(n2)**

* So, Quicksort is **O(n2)** in worst case

# Comparison of Sorting Algorithms

