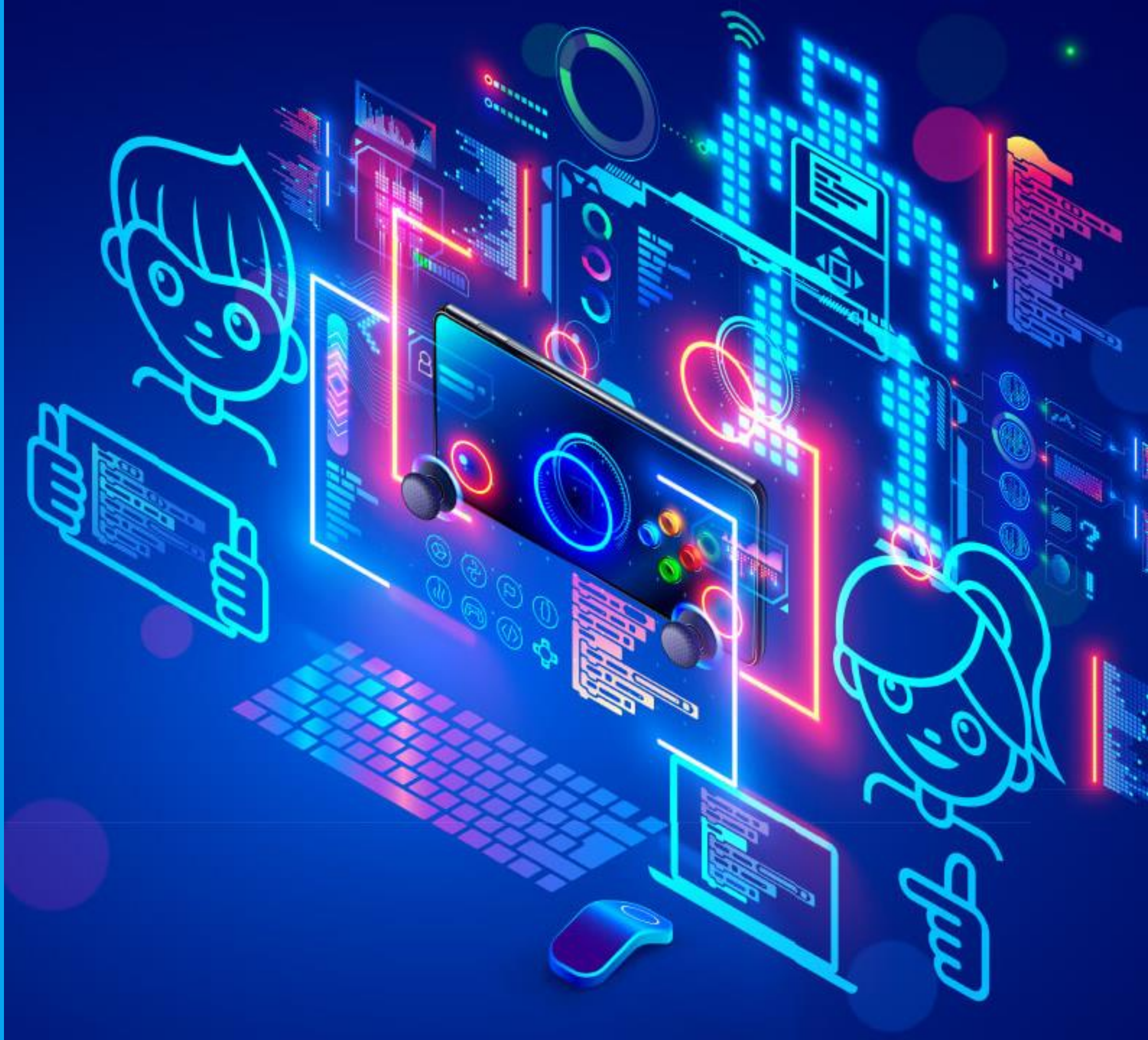
CS 24 AP Computer Science A Review

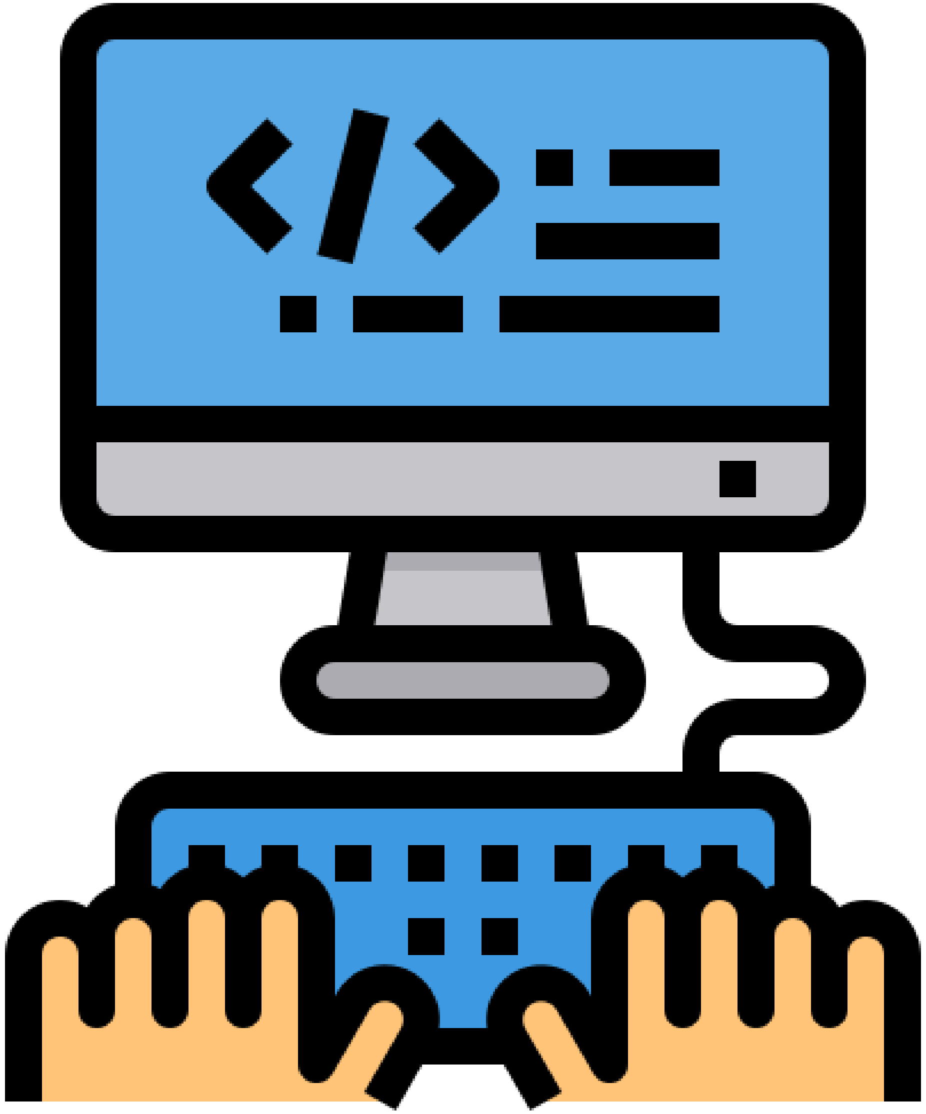
Week 8: Sorting and Searching

DR. ERIC CHOU
IEEE SENIOR MEMBER



Topics

- Sequential Search
- Binary Search
- Selection Sort
- Insertion Sort
- Bubble Sort
- Merge Sort
- Quick Sort



Swap, Shifting and Rotation

Section 1

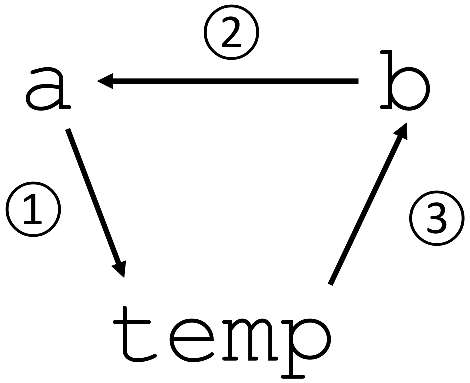
Application for These Algorithms

- Reverse of Array
- Replacement of Object
- Queue, Priority Queue
- Circular Queue
- Permutation of Data
- Bubble Sort
- Selection Sort
- Quick Sort

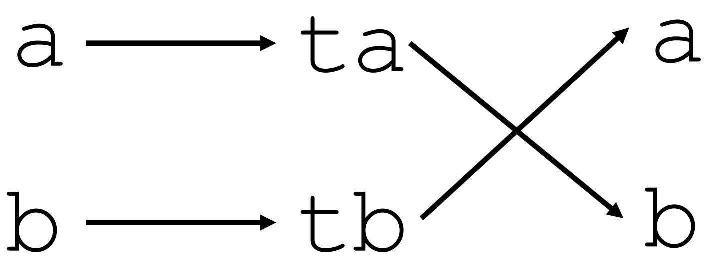
Swap

```
double temp = a;
a = b;
b = temp;
```

```
double ta = a;
double tb = b;
b = ta;
a = tb;
```



Rotational Swap



Swap by Shuffling Network

Swap by Swap Agent

```
class Pack<T>{
    T x;
    T y;
    Pack(T a, T b){
        x = a;
        y = b;
    }

    Pack swap(){
        return new Pack(y, x);
    }
}
```

- Re-usable
- Generic
- Small amount of memory overhead

```java
public class Swap{
    public static void main(String[] args){
        System.out.println("\fBefore Rotational Swap: ");
        int a = 3;
        int b = 4;
        System.out.println("A="+a+" B="+b);
        int temp = a;
        a = b;
        b = temp;
        System.out.println("After Rotational Swap:");
        System.out.println("A="+a+" B="+b);

        System.out.println("\nBefore Shuffle Swap: ");
        a = 3;
        b = 4;
        System.out.println("A="+a+" B="+b);
        int ta = a;
        int tb = b;
        a = tb;
        b = ta;
        System.out.println("After Shuffle Swap:");
        System.out.println("A="+a+" B="+b);

        System.out.println("\nBefore Swap Agent Swap: ");
        a = 3;
        b = 4;
        System.out.println("A="+a+" B="+b);
        Pack<Integer> p = new Pack<Integer>(a, b);
        p=p.swap();
        a = p.x;
        b = p.y;
        System.out.println("After Shuffle Swap:");
        System.out.println("A="+a+" B="+b);
    }
}
```

BlueJ: Terminal Wi...

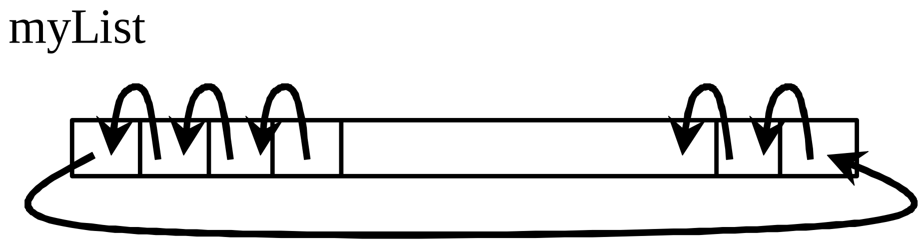
Options

```
Before Rotational Swap:
A=3 B=4
After Rotational Swap:
A=4 B=3

Before Shuffle Swap:
A=3 B=4
After Shuffle Swap:
A=4 B=3

Before Swap Agent Swap:
A=3 B=4
After Shuffle Swap:
A=4 B=3
```

Can only enter input while yo

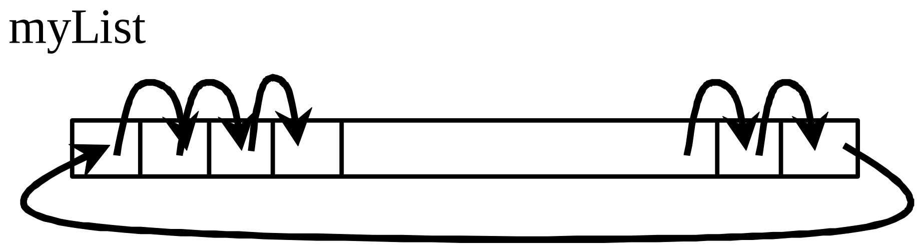
Shifting Elements (Left Shifting)

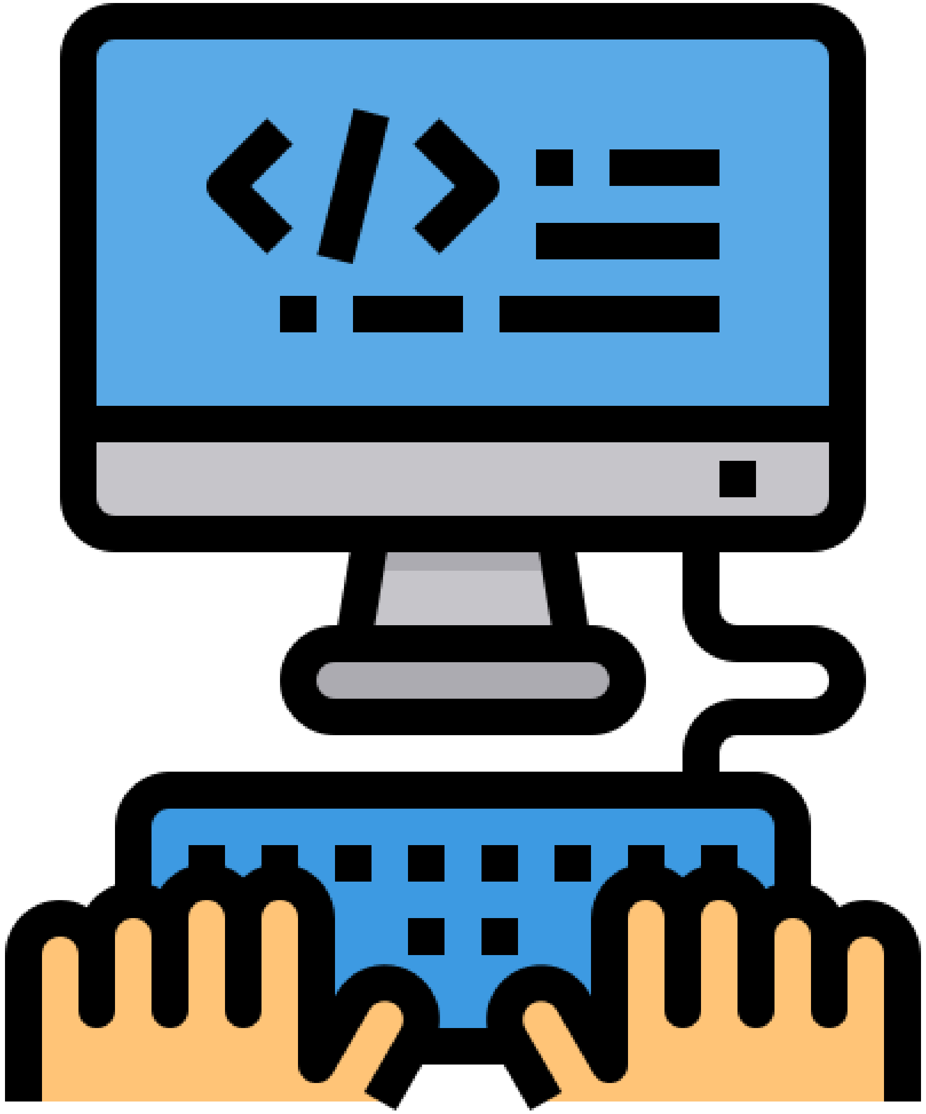
```
double temp = myList[0]; // Retain the first element

// Shift elements left
for (int i = 1; i < myList.length; i++) {
  myList[i - 1] = myList[i];
}

// Move the first element to fill in the last position
myList[myList.length - 1] = temp;
```



Shifting Elements (Right Shifting)

```
double temp = myList[myList.length-1]; // Retain the last element

// Shift elements left
for (int i = myList.length-2; i >=0; i--) {
  myList[i + 1] = myList[i];
}

// Move the last element to fill in the first position
myList[0] = temp;
```





Sequential Search

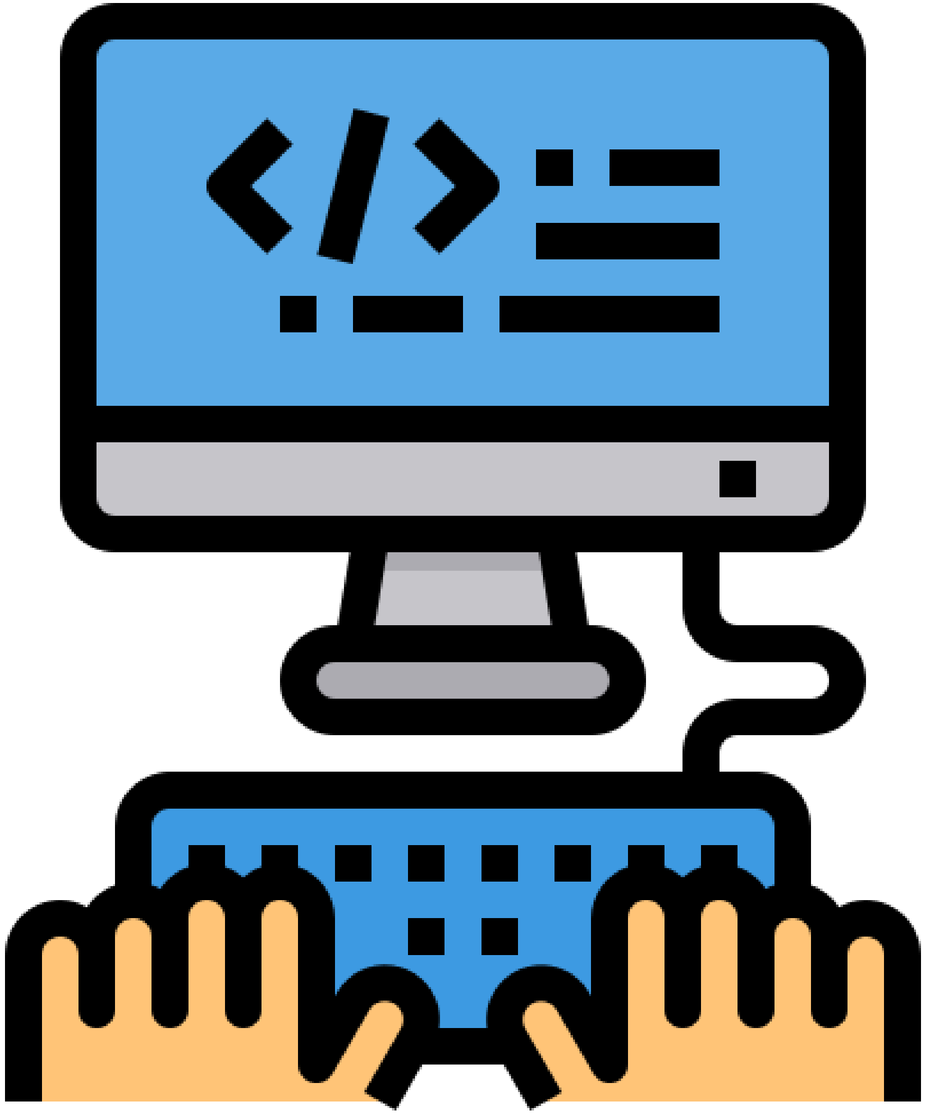
Section 2

Linear Search

- Involves going through each element of a collection until the appropriate value is found.
- This is often implemented using a **simple for loop**, where you begin at index 0 and count up to the last index in the collection.
- This is the simplest search method, and becomes less effective if the collection is very large.

Linear Search (Sequential Search)

```
public static int linearSearch(int[] list, int key) {
   for (int i = 0; i < list.length; i++) {
      if (key == list[i])
         return i;
   }
   return -1;
}
```



Binary Search

Section 3

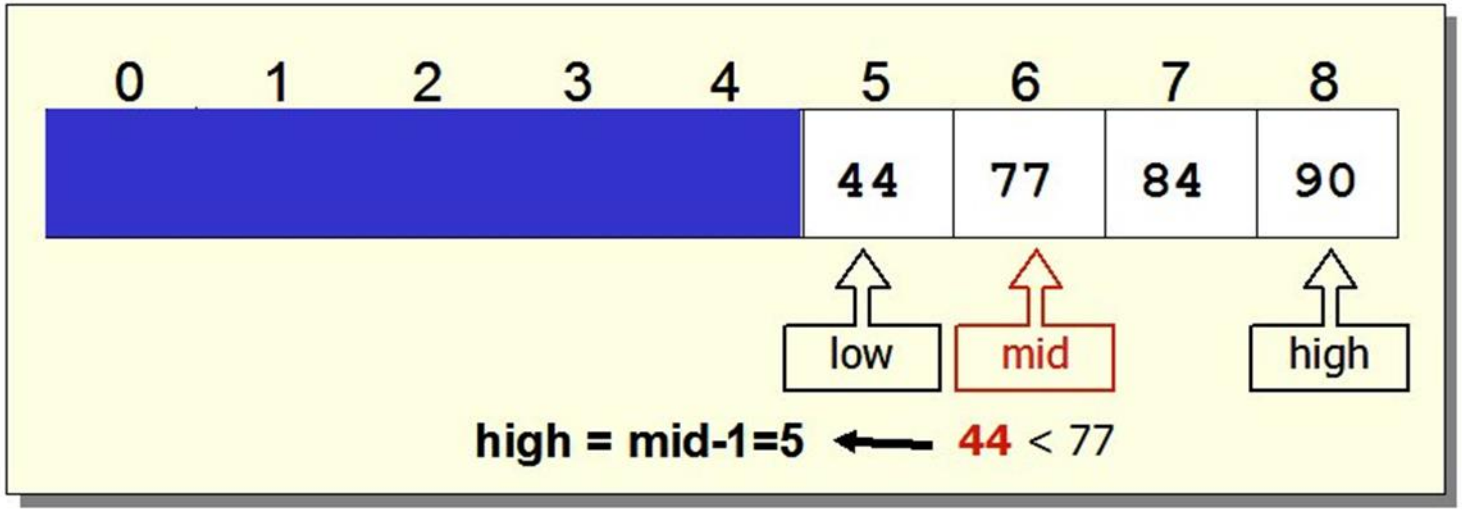
Binary Search

- Involves going through a collection and comparing the middle index against the value. This requires a sorted collection in order to work.
- For Example: You are trying to find the number 12 in an array of 100 integers, with values equal to the index+1 (1,2,3,4,5,..,99,100). Notice that this array is sorted with the smallest values on the left and the largest at the right. You would compare the middle value (50) against the search value (12). Since 50>12, you would exclude all the values 50 to 100 since they are also greater than the search value. You would then check 25 against the search value since that is the new middle between 49 and 1. Once again this value is larger than the search term and so all number 25 to 49 are excluded. This process continues until you finally arrive with 12 as the center value.

	low	high	mid
#1	0	8	4
#2	5	8	6

search(44)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$



Binary Search

```java
public class BinarySearch {
  /** Use binary search to find the key in the list */
  public static int binarySearch(int[] list, int key) {
    int low = 0;
    int high = list.length - 1;

    while (high >= low) {
      int mid = (low + high) / 2;
      if (key < list[mid])
        high = mid - 1;
      else if (key == list[mid])
        return mid;
      else
        low = mid + 1;
    }
    return -low - 1; // Now high < low
  }
}
```

mid = (low+high)/2
= (0+length-1)/2
= (lengh-1)/2

length = 10, mid = 4 (2^{nd} half longer)
length = 9, mid = 4 (balance)

Mid-point belong to lower half.

f(n) function

f(n) function represents the number of elements that can be search by n searching steps.

$f(1) = 1$

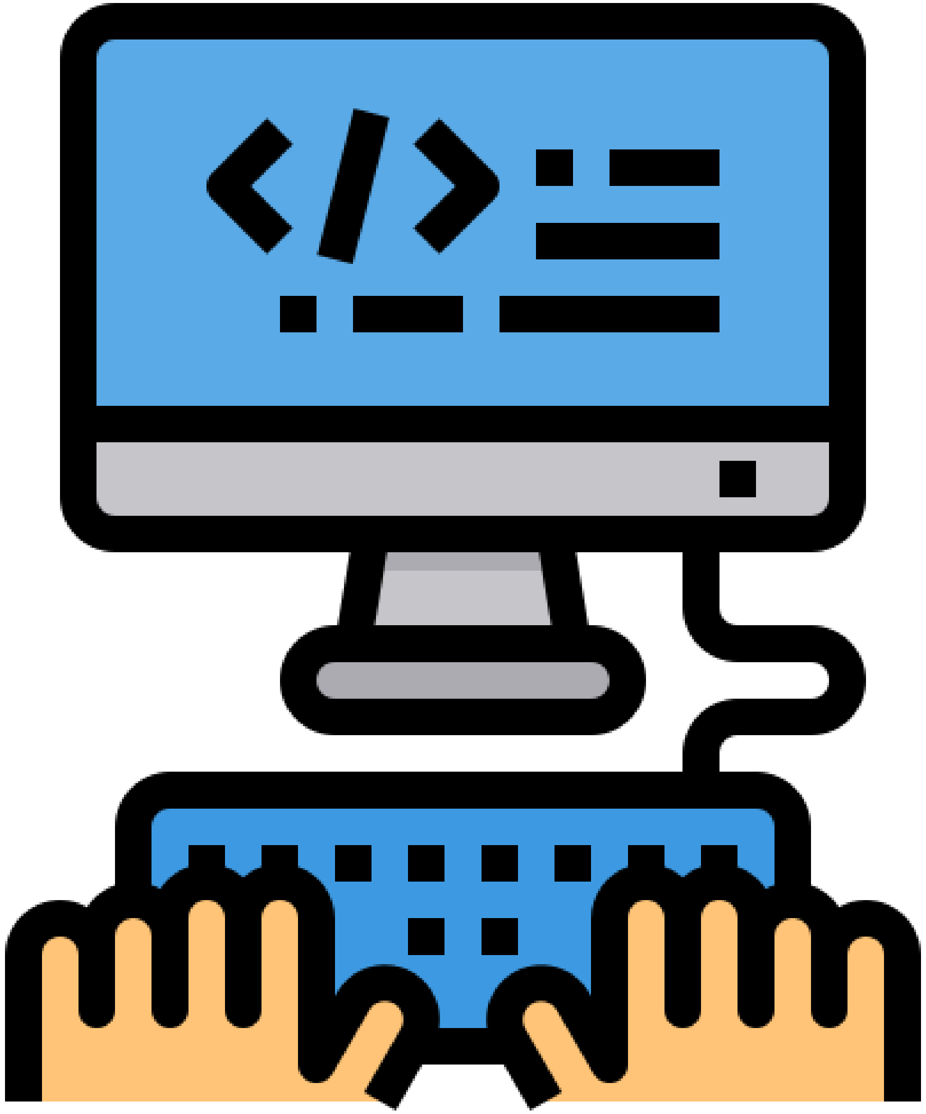
$f(2) = f(1) + 1 + f(1) = 2 * f(1) + 1 = 2 + 1 = 2^2 -1$

$f(3) = 2 * f(2) + 1 = 2 = 2 * (2 + 1) = 4 + 2 + 1 = 2^3 -1$

$f(4) = 2 * f(3) + 1 = 2^4 -1$

…

$f(n) = 2^n -1$

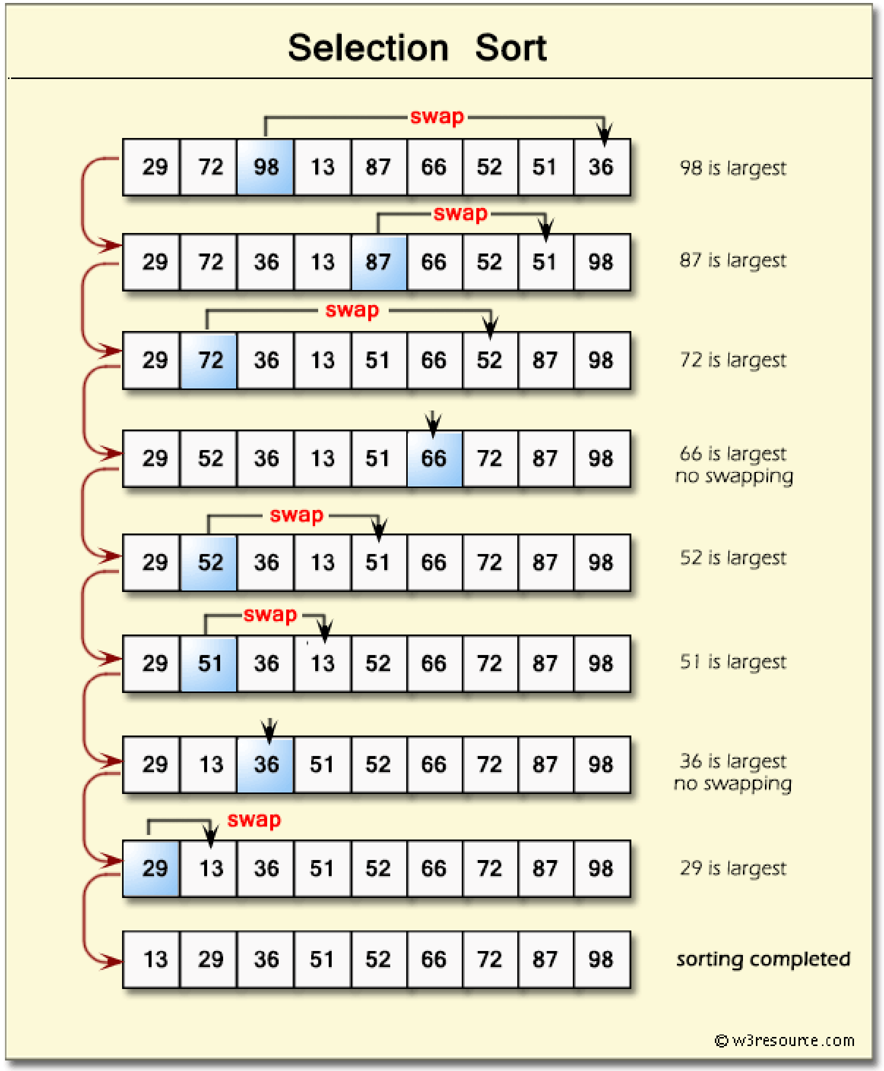


Selection Sort

Section 4

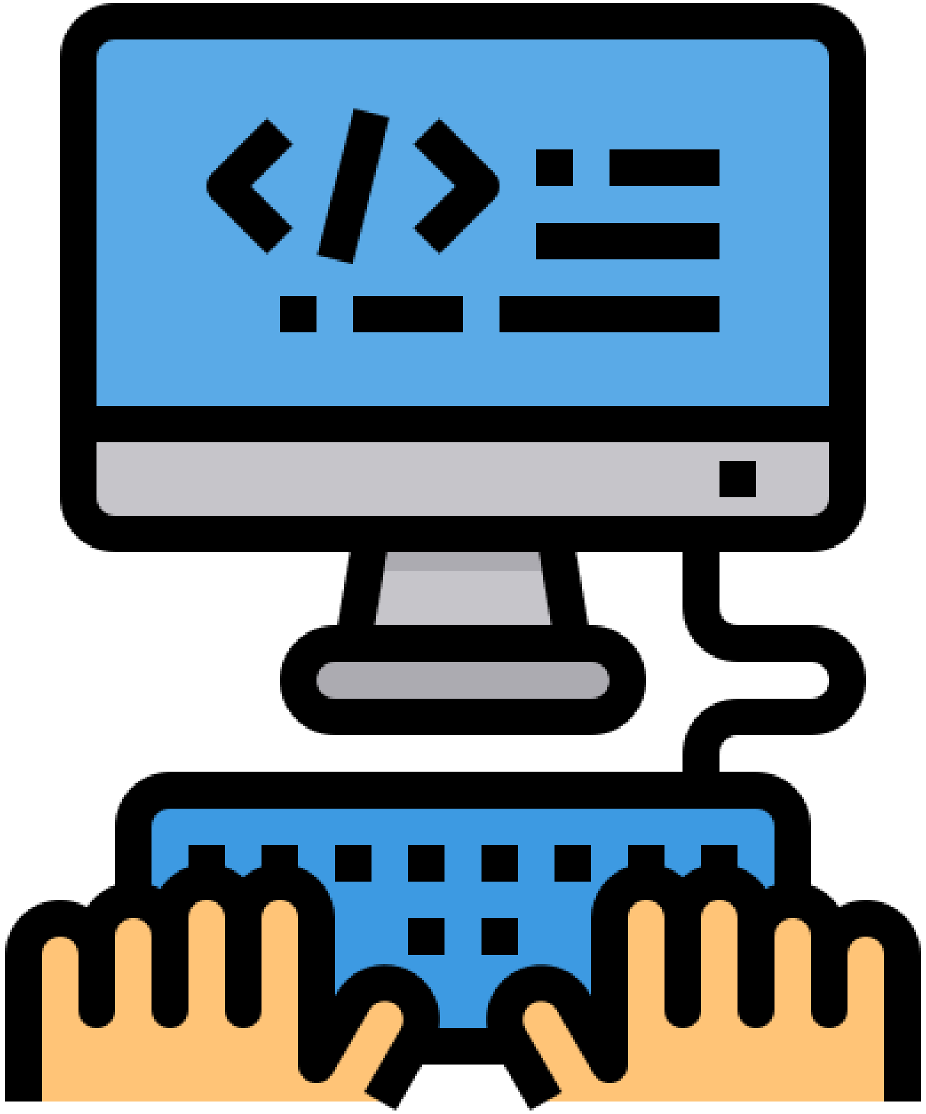
Selection Sort

- Selection sort is an iterative sort algorithm that uses a "search and swap" approach to sort a collection. For each pass through the collection, the algorithm finds the smallest element to be sorted and swaps it with the first unsorted element in the collection.
- The algorithm continues in this manner, finding the next smallest element in the collection and swapping it with the next unsorted element. Finally, when just two unsorted elements remain, they are compared (and if necessary, swapped) and the sort is complete.


Selection Sort
swap
29 72 98 13 87 66 52 51 36
98 is largest
swap
29 72 36 13 87 66 52 51 98
87 is largest
swap
29 72 36 13 51 66 52 87 98
72 is largest
29 52 36 13 51 66 72 87 98
66 is largest
no swapping
swap
29 52 36 13 51 66 72 87 98
52 is largest
swap
29 51 36 13 52 66 72 87 98
51 is largest
29 13 36 51 52 66 72 87 98
36 is largest
no swapping
swap
29 13 36 51 52 66 72 87 98
29 is largest
13 29 36 51 52 66 72 87 98
sorting completed
© w3resource.com

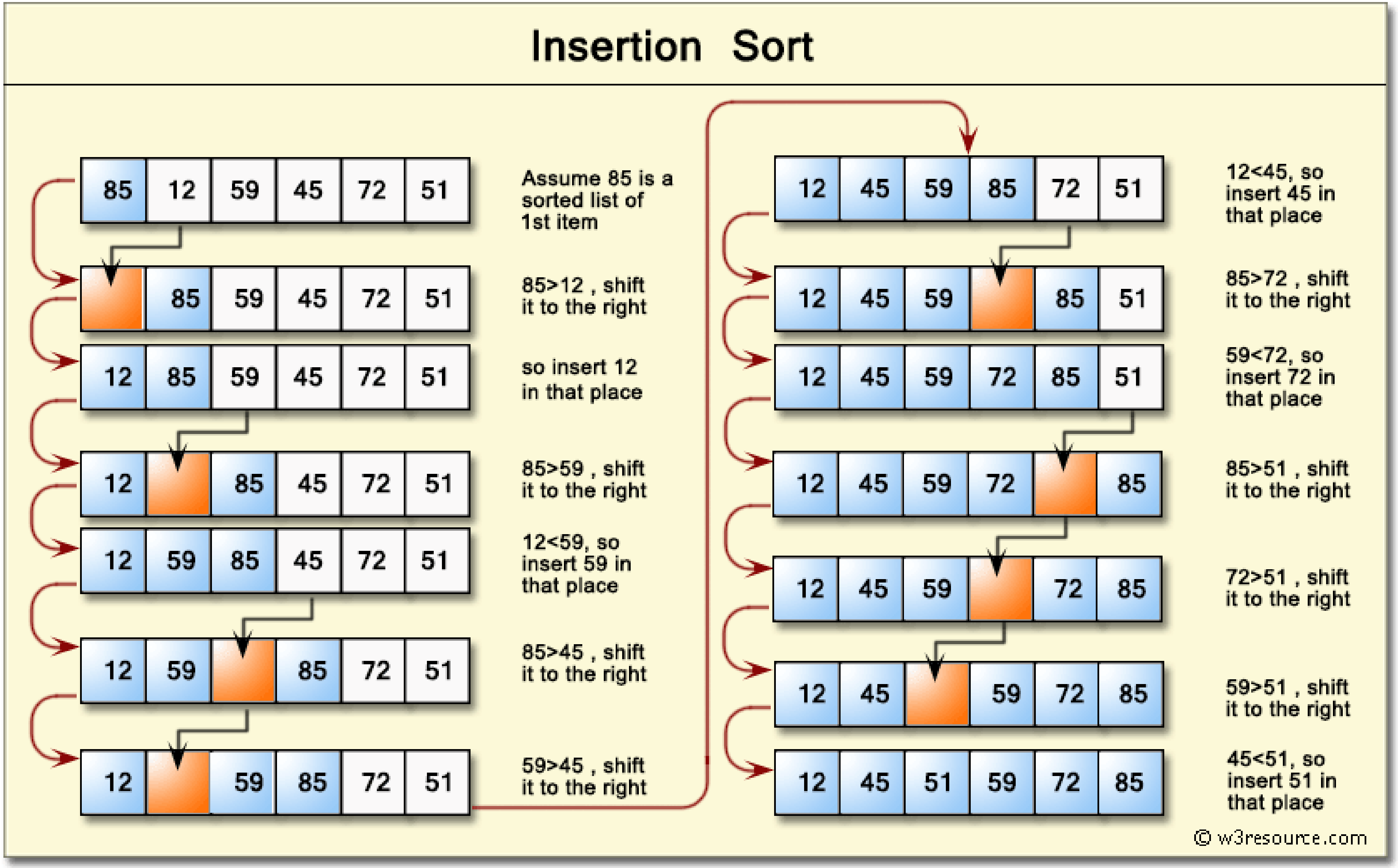
Selection Sort

```java
public class SelectionSort {
  /** The method for sorting the numbers */
  public static void selectionSort(double[] list) {
    for (int i = 0; i < list.length - 1; i++) {
      // Find the minimum in the list[i..list.length-1]
      double currentMin = list[i];
      int currentMinIndex = i;
      for (int j = i + 1; j < list.length; j++) {
        if (currentMin > list[j]) {
          currentMin = list[j];
          currentMinIndex = j;
        }
      }
      // Swap list[i] with list[currentMinIndex] if necessary;
      if (currentMinIndex != i) {
        list[currentMinIndex] = list[i];
        list[i] = currentMin;
      }
    }
  }
```



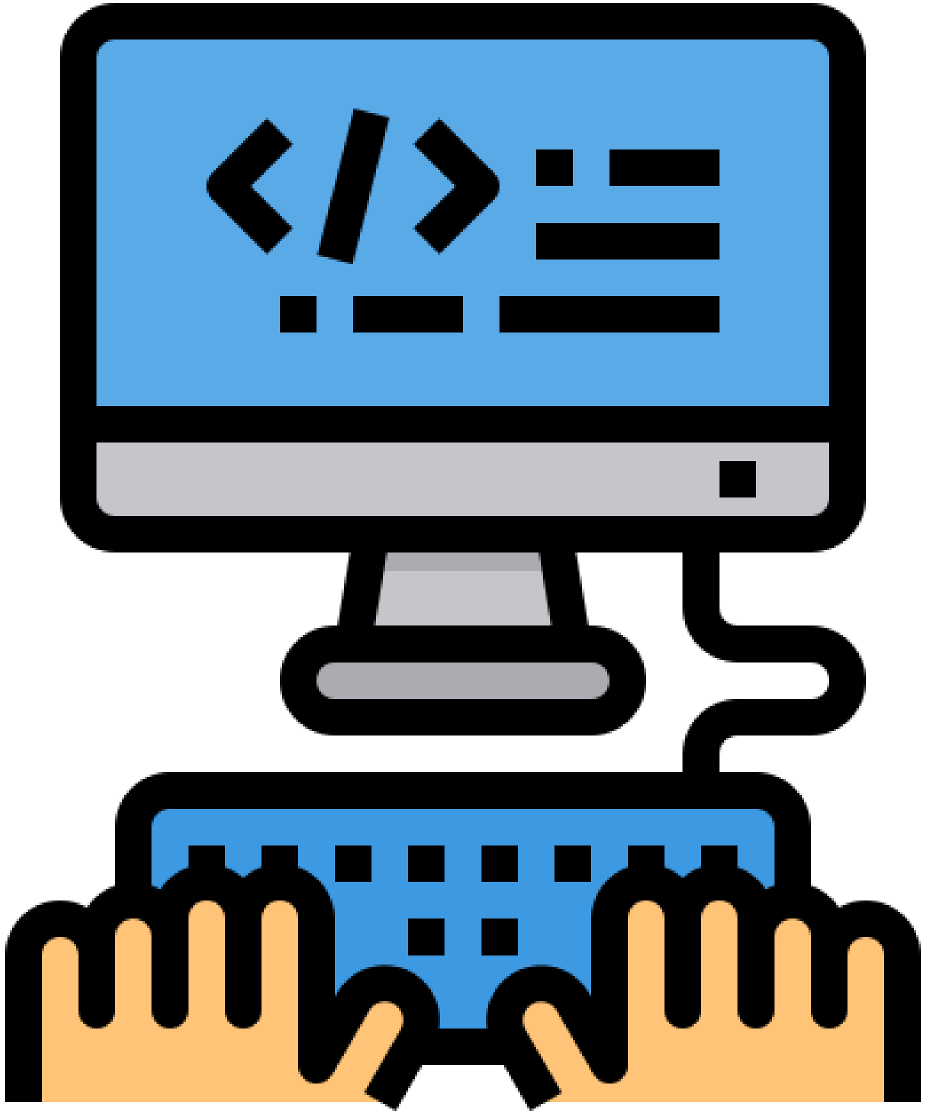
Insertion Sort

Section 5


Insertion Sort
85 12 59 45 72 51
Assume 85 is a sorted list of 1st item
85 59 45 72 51
85>12 , shift it to the right
12 85 59 45 72 51
so insert 12 in that place
12 85 45 72 51
85>59 , shift it to the right
12 59 85 45 72 51
12<59, so insert 59 in that place
12 59 85 72 51
85>45 , shift it to the right
12 59 85 72 51
59>45 , shift it to the right
12 45 59 85 72 51
12<45, so insert 45 in that place
12 45 59 85 51
85>72 , shift it to the right
12 45 59 72 85 51
59<72, so insert 72 in that place
12 45 59 72 85
85>51 , shift it to the right
12 45 59 72 85
72>51 , shift it to the right
12 45 59 72 85
59>51 , shift it to the right
12 45 51 59 72 85
45<51, so insert 51 in that place
© w3resource.com

Insertion Sort

```java
public class InsertionSort {
  /** The method for sorting the numbers */
  public static void insertionSort(int[] list) {
    for (int i = 1; i < list.length; i++) {
      /** insert list[i] into a sorted sublist list[0..i-1] so that
           list[0..i] is sorted. */
      int currentElement = list[i];
      int k;
      for (k = i - 1; k >= 0 && list[k] > currentElement; k--) {
        list[k + 1] = list[k];
      }
      // Insert the current element into list[k+1]
      list[k + 1] = currentElement;
    }
  }

  /** A test method */
  public static void main(String[] args) {
    int[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
    insertionSort(list);
    for (int i = 0; i < list.length; i++)
      System.out.print(list[i] + " ");
  }
}
```

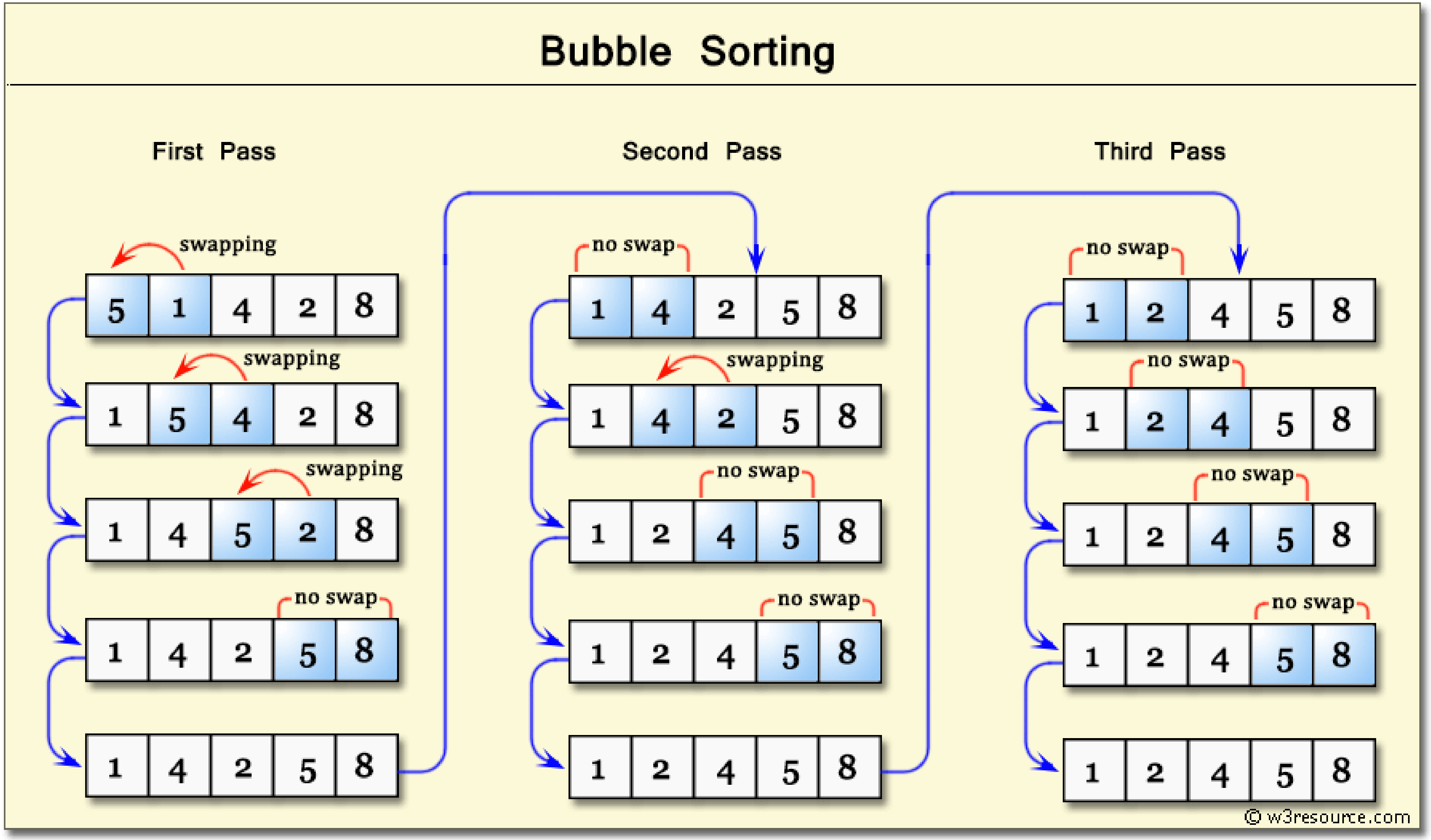


Bubble Sort

Section 6

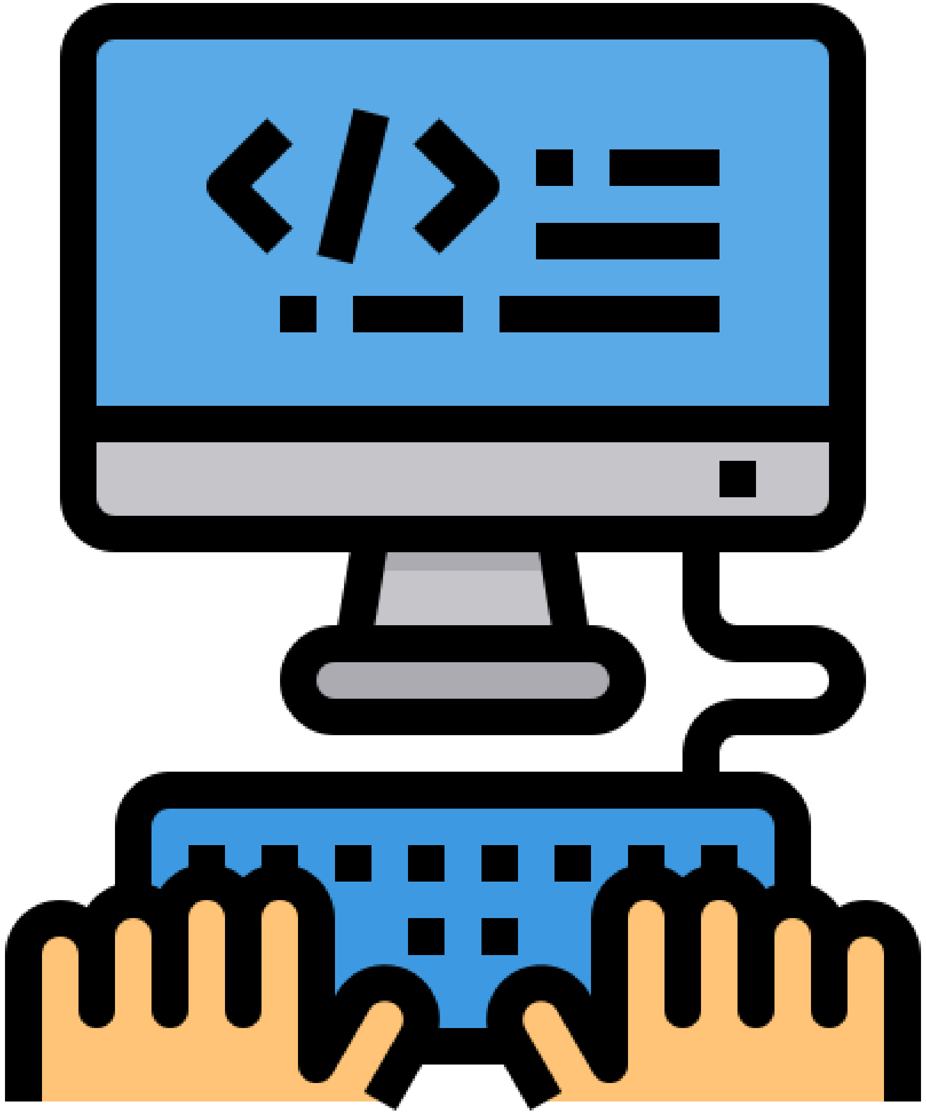
Bubble sort

- Bubble sort, sometimes referred to as **sinking sort**, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order.
- The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list.
- Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort. It can be practical if the input is usually in sorted order but may occasionally have some out-of-order elements nearly in position.


Bubble Sorting
First Pass
Second Pass
Third Pass
swapping
5 1 4 2 8
swapping
1 5 4 2 8
swapping
1 4 5 2 8
no swap
1 4 2 5 8
1 4 2 5 8
no swap
1 4 2 5 8
swapping
1 4 2 5 8
no swap
1 2 4 5 8
no swap
1 2 4 5 8
1 2 4 5 8
no swap
1 2 4 5 8
no swap
1 2 4 5 8
no swap
1 2 4 5 8
no swap
1 2 4 5 8
1 2 4 5 8
© w3resource.com

Bubble Sort

```java
public class BubbleSort {
  /** Bubble sort method */
  public static void bubbleSort(int[] list) {
    boolean needNextPass = true;

    for (int k = 1; k < list.length && needNextPass; k++) {
      // Array may be sorted and next pass not needed
      needNextPass = false;
      for (int i = 0; i < list.length - k; i++) {
        if (list[i] > list[i + 1]) {
          // Swap list[i] with list[i + 1]
          int temp = list[i];
          list[i] = list[i + 1];
          list[i + 1] = temp;

          needNextPass = true; // Next pass still needed
        }
      }
    }
  }
```

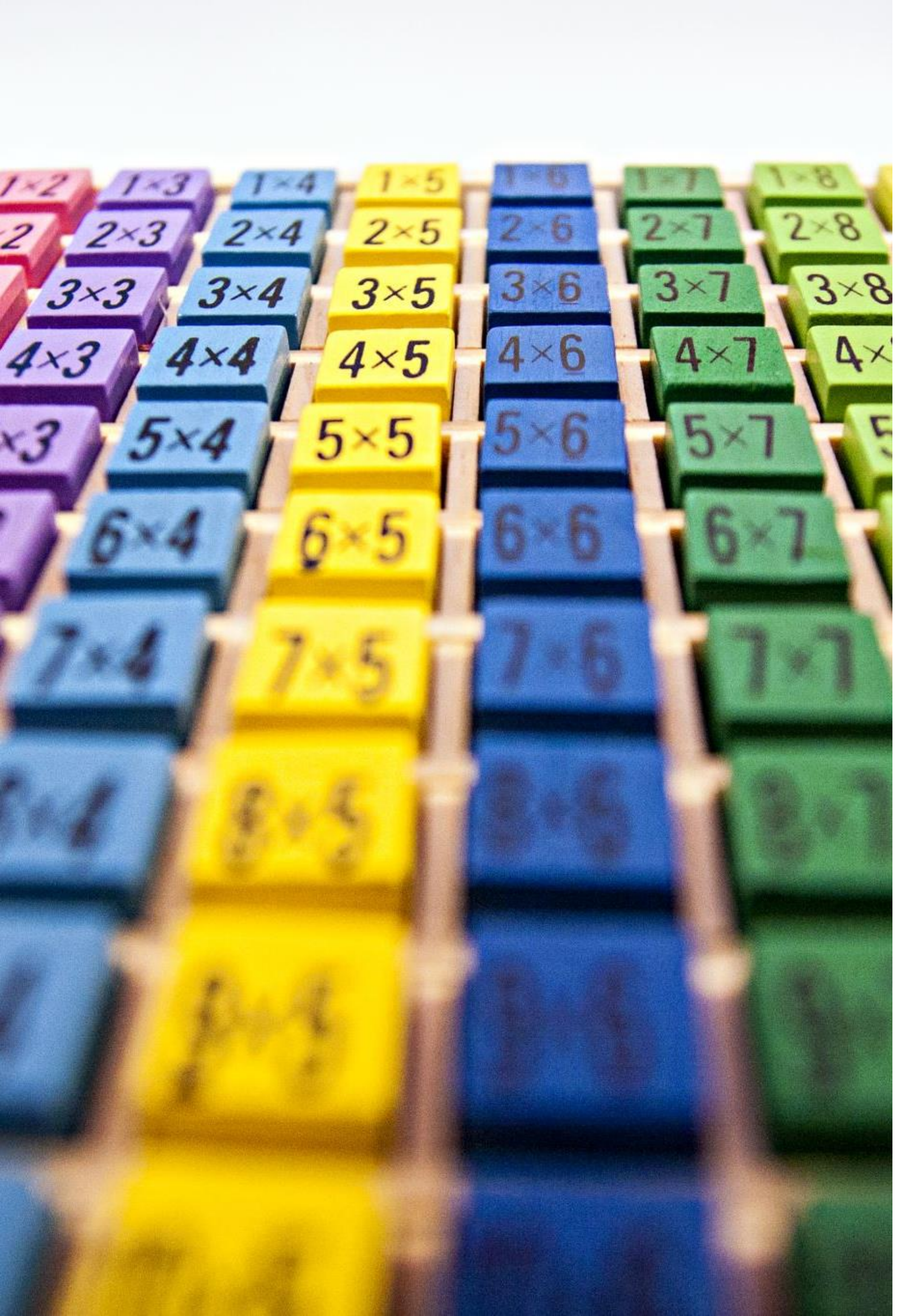


Merge Sort

Section 7

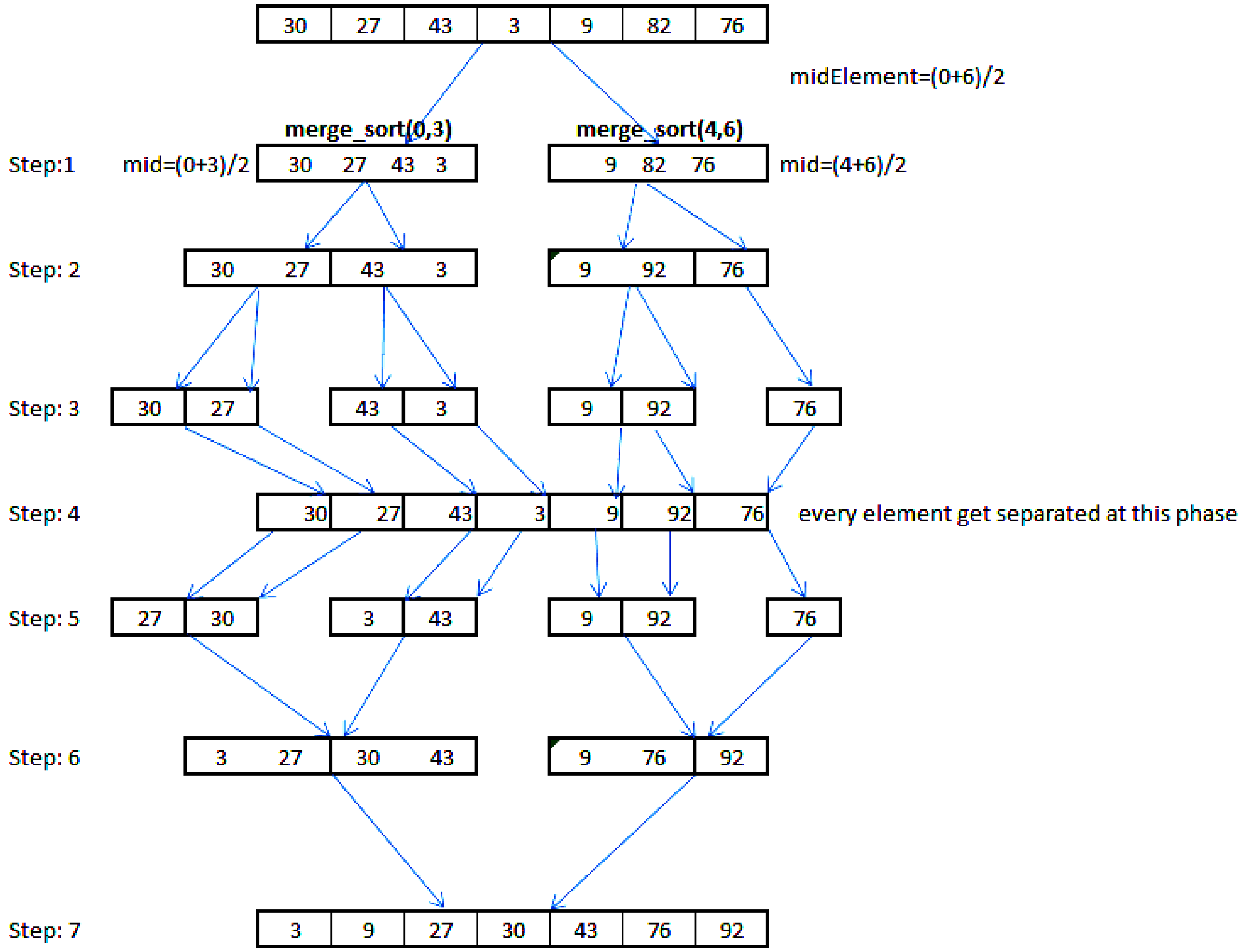
Merge Sort

- Mergesort is a recursive algorithm that uses a "divide and conquer" approach to sorting collections. Each time the algorithm is called, the algorithm checks to see if there is more than one element in the collection, and if there is, the collection is "broken" into two halves.
- If there is even number of elements, the **halves** are equal, but if there is an odd number of elements, the **left half** will contain one more element than the right half.
- At this point, the algorithm uses recursion, calling itself to first mergesort the left half of the collection, then mergesort the right half. When the algorithm calls itself to mergesort one of the halves, it again "breaks" the half into two halves, then calls itself to sort each half. This process is repeated until the entire collection is "broken" into individual elements, or sub-collections of length 1. When the method calls itself to sort one of these, the initial test that sees if there is more than one element in the sub-collection fails, since there is only one element in the sub-collection.



Merge Sort

- This is where the recursion stops, and the algorithm returns to sorting the half by comparing the two adjacent elements, sorting them, and then recombining them into a sorted sub-collection.
- This process continues until all of the individual elements have been sorted and recombined into sorted sub-collections.
- Then the sub-collections themselves are compared, sorted, and recombined.
- This process continues until the sub-collections have been recombined to form two sorted halves. Finally, the left and right halves are compared to each other, sorted, and recombined to create the final, fully sorted collection.



Even number: balanced

Odd number: lower half has one more.

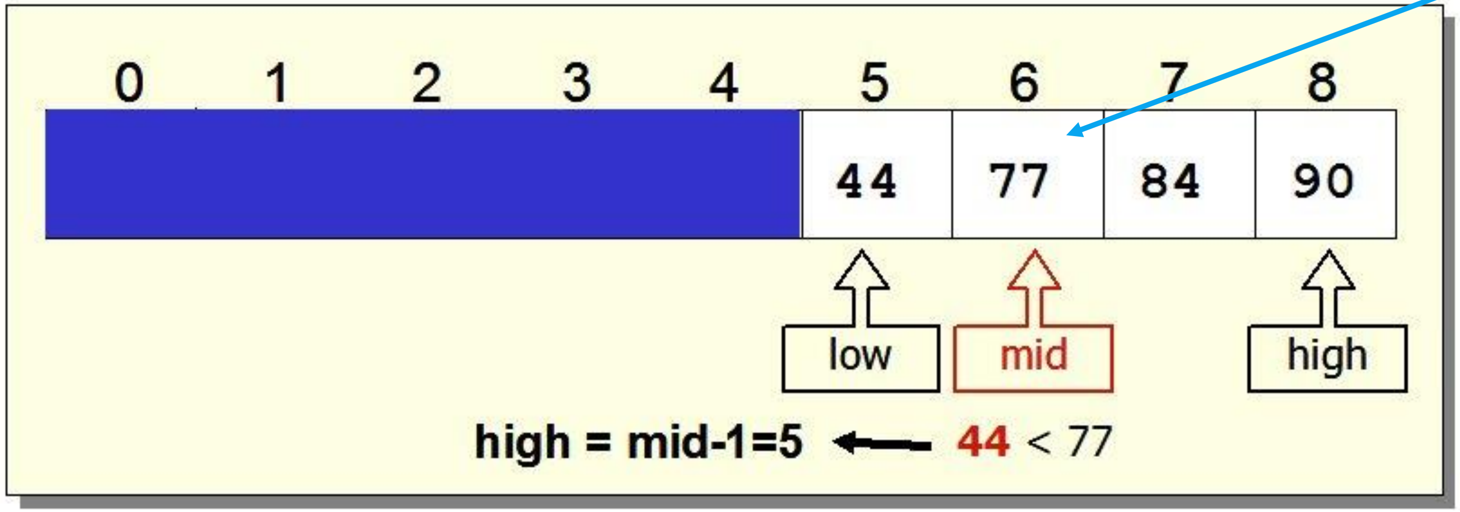
We use the same low, mid, high system as the Binary Search

	low	high	mid
#1	0	8	4
#2	5	8	6

search(44)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$

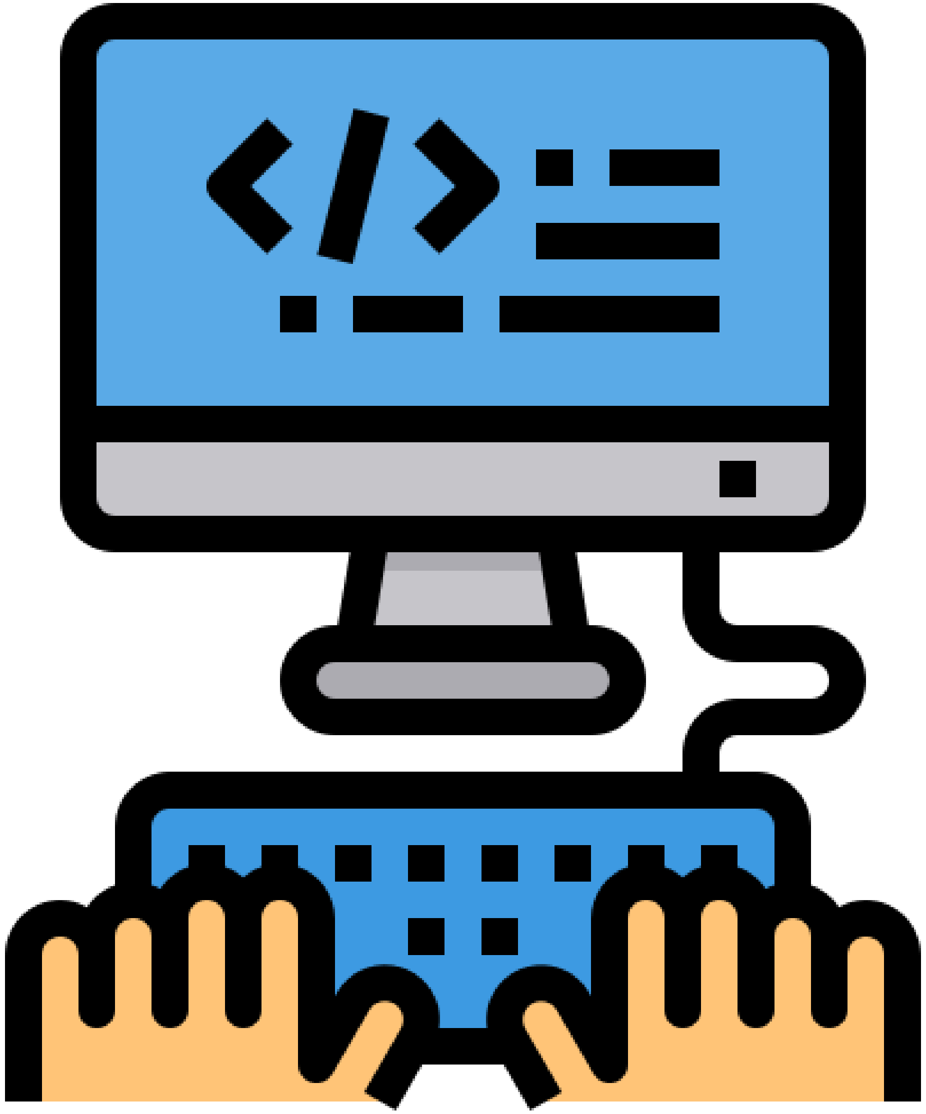
mid belongs to first half



mid belongs to first half

```java
public static void mergeSort(int[] list) {
  if (list.length > 1) {
    // Merge sort the first half
    int low = 0, high = list.length-1;
    int mid = (low+high)/2;
    int[] firstHalf = new int[mid+1];
    System.arraycopy(list, 0, firstHalf, 0, mid+1);
    mergeSort(firstHalf);

    // Merge sort the second half
    int secondHalfLength = list.length - (mid+1);
    int[] secondHalf = new int[secondHalfLength];
    System.arraycopy(list, mid+1,
      secondHalf, 0, secondHalfLength);
    mergeSort(secondHalf);

    // Merge firstHalf with secondHalf into list
    merge(firstHalf, secondHalf, list);
  }
}
```

```java
/** Merge two sorted lists */
public static void merge(int[] list1, int[] list2, int[] temp) {
  int current1 = 0; // Current index in list1
  int current2 = 0; // Current index in list2
  int current3 = 0; // Current index in temp

  while (current1 < list1.length && current2 < list2.length) {
    if (list1[current1] < list2[current2])
      temp[current3++] = list1[current1++];
    else
      temp[current3++] = list2[current2++];
  }

  while (current1 < list1.length)
    temp[current3++] = list1[current1++];

  while (current2 < list2.length)
    temp[current3++] = list2[current2++];
}
```



Quick Sort

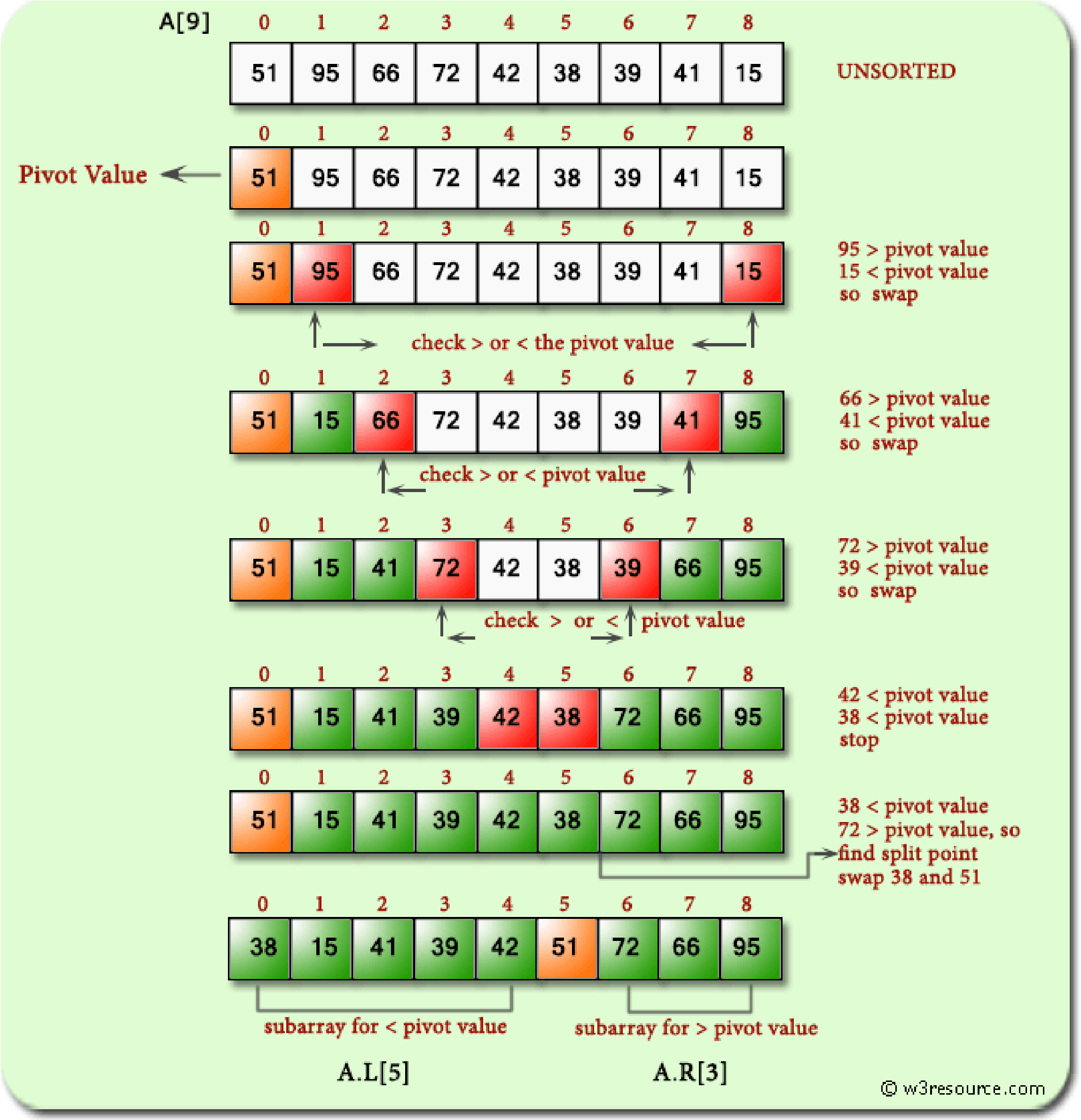
Section 8

Quick Sort

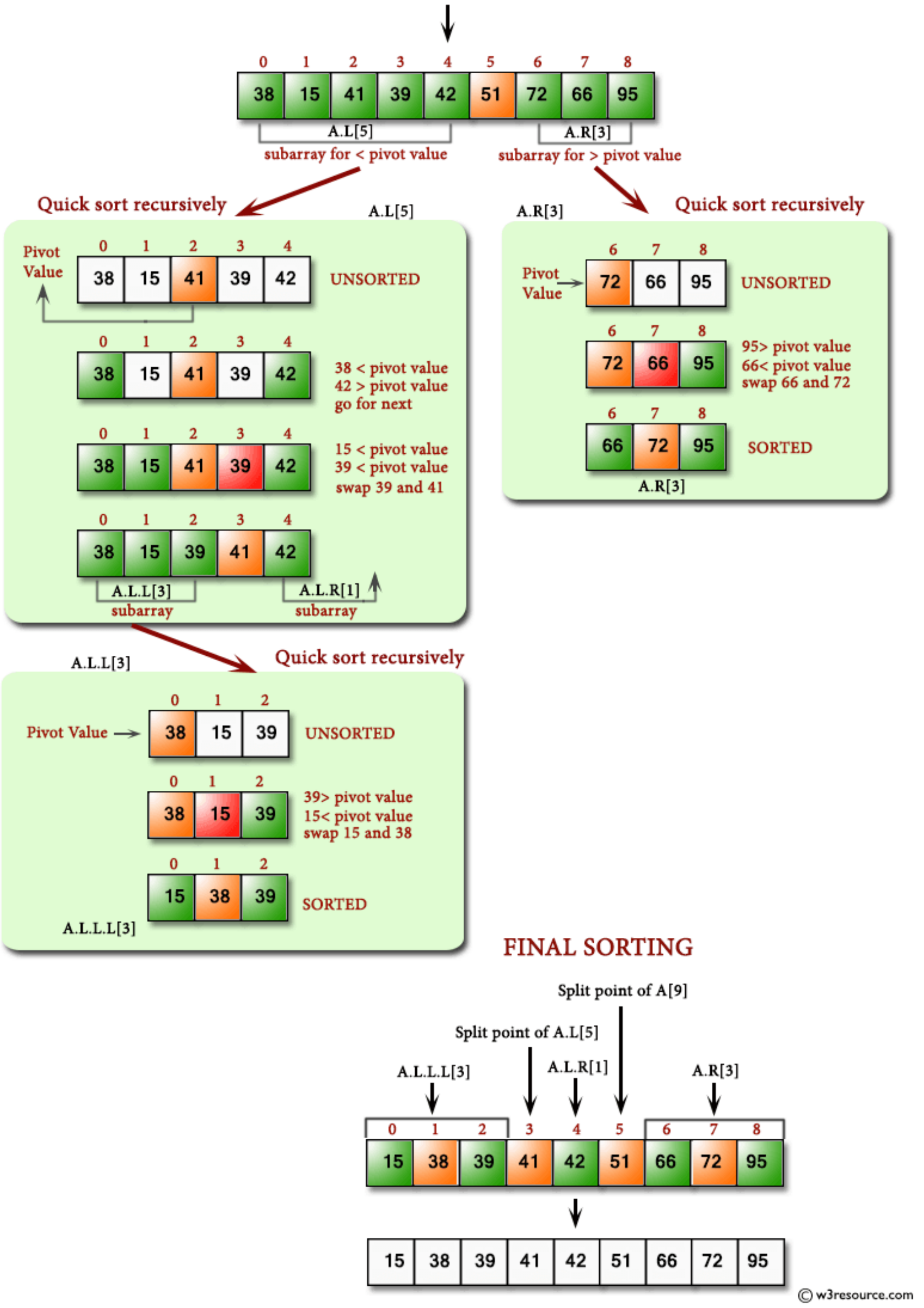
Suitable for ArrayList and Recursive

- Quicksort is, on average, the fastest sorting algorithm for sorting collections with a large number of elements.
- Quicksort is recursive and also uses a "divide and conquer" approach to sorting. This algorithm starts by partitioning the collection, selecting a pivot point, which is usually either the first element in the collection or an element selected at random, then moving all elements less than the pivot point value to the left of the pivot point, and all elements greater than or equal to the pivot point value to the right of the pivot point.
- The algorithm then uses recursion, splitting the array into two halves and calling itself to quicksort each half.
- Finally, the sort is complete when every element has been moved to its correct location.

Quick Sort



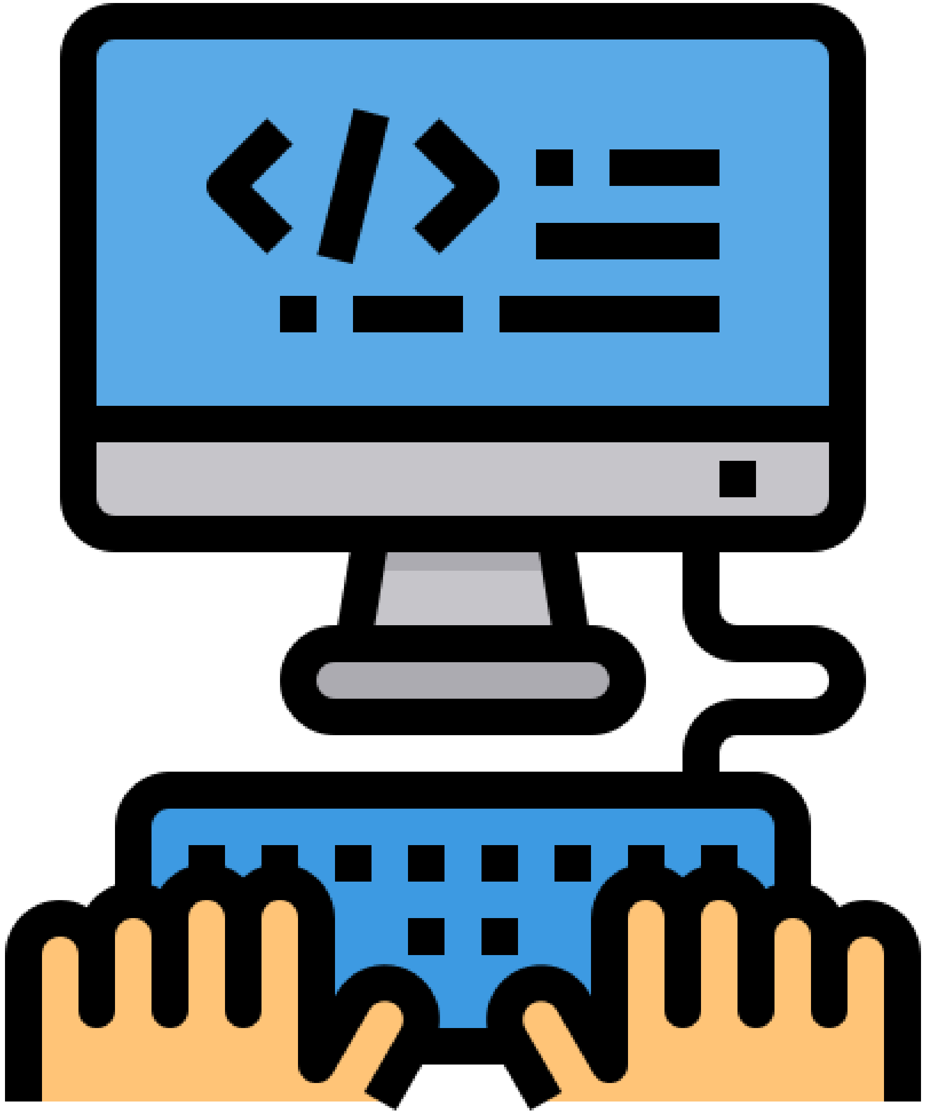
A[9]
51 95 66 72 42 38 39 41 15
UNSORTED
Pivot Value
95 > pivot value
15 < pivot value
so swap
check > or < the pivot value
66 > pivot value
41 < pivot value
so swap
check > or < pivot value
72 > pivot value
39 < pivot value
so swap
check > or < pivot value
42 < pivot value
38 < pivot value
stop
38 < pivot value
72 > pivot value, so
find split point
swap 38 and 51
38 15 41 39 42 51 72 66 95
subarray for < pivot value
subarray for > pivot value
A.L[5]
A.R[3]
© w3resource.com



38 15 41 39 42 51 72 66 95
A.L[5]
A.R[3]
subarray for < pivot value
subarray for > pivot value
Quick sort recursively
A.L[5]
A.R[3]
Quick sort recursively
Pivot Value
38 15 41 39 42
UNSORTED
38 < pivot value
42 > pivot value
go for next
15 < pivot value
39 < pivot value
swap 39 and 41
38 15 39 41 42
A.L.L[3]
subarray
A.L.R[1]
subarray
Pivot Value
72 66 95
UNSORTED
95> pivot value
66< pivot value
swap 66 and 72
66 72 95
SORTED
A.R[3]
A.L.L[3]
Quick sort recursively
Pivot Value
38 15 39
UNSORTED
39> pivot value
15< pivot value
swap 15 and 38
15 38 39
SORTED
A.L.L.L[3]
FINAL SORTING
Split point of A[9]
Split point of A.L[5]
A.L.L.L[3]
A.L.R[1]
A.R[3]
15 38 39 41 42 51 66 72 95
15 38 39 41 42 51 66 72 95
© w3resource.com

```java
public static void quickSort(int[] list) {
  quickSort(list, 0, list.length - 1);
}

private static void quickSort(int[] list, int first, int last) {
  if (last > first) {
    int pivotIndex = partition(list, first, last);
    quickSort(list, first, pivotIndex - 1);
    quickSort(list, pivotIndex + 1, last);
  }
}
```

```java
/** Partition the array list[first..last] */
private static int partition(int[] list, int first, int last) {
  int pivot = list[first]; // Choose the first element as the pivot
  int low = first + 1; // Index for forward search
  int high = last; // Index for backward search

  while (high > low) {
    // Search forward from left
    while (low <= high && list[low] <= pivot) low++;
    // Search backward from right
    while (low <= high && list[high] > pivot) high--;
    // Swap two elements in the list
    if (high > low) {
      int temp = list[high];
      list[high] = list[low];
      list[low] = temp;
    }
  }

  while (high > first && list[high] >= pivot) high--;

  // Swap pivot with list[high]
  if (pivot > list[high]) {
    list[first] = list[high];
    list[high] = pivot;
    return high;
  }
  else { return first;
  }
}
```



Comparison of Sorting Algorithms

Section 9

Time Complexity Comparison

Sorting algorithm	Time complexity			Space complexity	Stability
	Average case	Best case	Worst case		
Bubble sort[2]	$O(N^2)$	$O(N)$	$O(N^2)$	$O(1)$	Yes
Insertion sort[3,4]	$O(N^2)$	$O(N)$	$O(N^2)$	$O(1)$	Yes
Selection sort[5,6]	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(1)$	No
Merge sort[7]	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N)$	Yes
Heap sort[1,5]	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(1)$	No
Quick sort[8,9]	$O(N \log N)$	$O(N \log N)$	$O(N^2)$	$O(N \log N)$	No
Counting sort[10,11]	$O(N)$	$O(N)$	$O(N^2)$	$O(N)$	Yes
Radix sort[3]	$O(N)$	$O(N)$	$O(N)$	$O(N)$	No
Bucket sort[3]	$O(N)$	$O(N)$	$O(N^2)$	$O(N)$	Yes

Benchmarks Comparison