**Day23**

**Sorting in JAVA**

1. **Bubble Sort**

Idea: if arr[i] > arr[i+1] swap them. To place the element in their respective position, we have to do the following operation N-1 times.

Time Complexity: O(N2)

**Code**

import java.util.\*;

class Sorting {

   public static void printArray(int arr[]) {

       for(int i=0; i<arr.length; i++) {

           System.out.print(arr[i]+" ");

       }

       System.out.println();

   }

   public static void main(String args[]) {

       int arr[] = {7, 8, 1, 3, 2};

       //bubble sort

       for(int i=0; i<arr.length-1; i++) {

           for(int j=0; j<arr.length-i-1; j++) {

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

                   //swap

                   int temp = arr[j];

                   arr[j] = arr[j+1];

                   arr[j+1] = temp;

               }

           }

       }

       printArray(arr);

   }

}

1. **Selection Sort**

Idea: The inner loop selects the minimum element in the unsorted array and places the elements in increasing order.

Time complexity: O(N2)

**Code**

**import java.util.\*;**

**class Sorting {**

**public static void printArray(int arr[]) {**

**for(int i=0; i<arr.length; i++) {**

**System.out.print(arr[i]+" ");**

**}**

**System.out.println();**

**}**

**public static void main(String args[]) {**

**int arr[] = {7, 8, 1, 3, 2};**

**//selection sort**

**for(int i=0; i<arr.length-1; i++) {**

**int smallest = i;**

**for(int j=i+1; j<arr.length; j++) {**

**if(arr[j] < arr[smallest]) {**

**smallest = j;**

**}**

**}**

**//swap**

**int temp = arr[smallest];**

**arr[smallest] = arr[i];**

**arr[i] = temp;**

**}**

**printArray(arr);**

**}**

**}**

1. **Insertion Sort**

Idea: Take an element from the unsorted array, place it in its corresponding position in the sorted part, and shift the elements accordingly.

Time Complexity: O(N2)

**Code**

**import java.util.\*;**

**class Sorting {**

**public static void printArray(int arr[]) {**

**for(int i=0; i<arr.length; i++) {**

**System.out.print(arr[i]+" ");**

**}**

**System.out.println();**

**}**

**public static void main(String args[]) {**

**int arr[] = {7, 8, 1, 3, 2};**

**//insertion sort**

**for(int i=1; i<arr.length; i++) {**

**int current = arr[i];**

**int j = i - 1;**

**while(j >= 0 && arr[j] > current) {**

**//Keep swapping**

**arr[j+1] = arr[j];**

**j--;**

**}**

**arr[j+1] = current;**

**}**

**printArray(arr);**

**}**

**}**

**`**

**Questions**

1. What is time complexity of an algorithm?



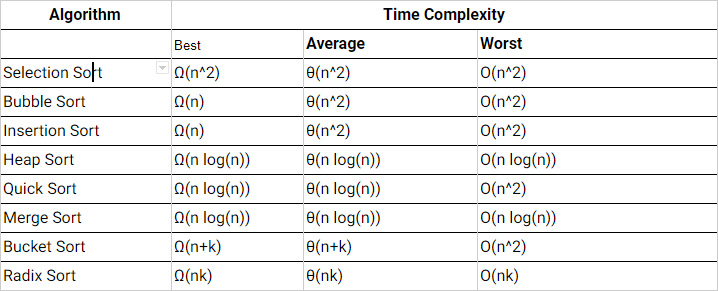
Just how fast can these algorithms go?

Generally speaking, algorithms are designed to solve problems — in our case, problems of sorting elements in a list. To solve this problem, different approaches can be utilized — and the efficiency of each approach is also different. A good scenario to illustrate this point is finding the square of a number: we can either use n\*n or send n into a for loop n times. Naturally, using a method like n\*n is far more efficient because it has better **time complexity**.

The algorithm’s time complexity is a key aspect of its performance. It denotes the total time the algorithm requires to run until it completes. To estimate it, we can count the number of elementary steps that the algorithm performs from start to finish. For n\*n, for instance, the algorithm only performs one step (using the mathematical operator \*), while the for … n loop requires exactly n steps.

Most importantly, the algorithm’s performance is heavily reliant on the input data and its type; therefore, the worst-case time complexity (the maximum time required) is normally used because it’s impossible to predict all variations in the input data.

2. What are the most common types of sorting algorithms? What is their time complexity?



3. What is a data structure?

To organize the data we want to work with, we utilize data structures: this way, we ensure that the data is organized in an efficient manner. These structures come in different shapes and sizes, allowing the developer to work with their application of choice.

**What are linear and non-linear data structures?**

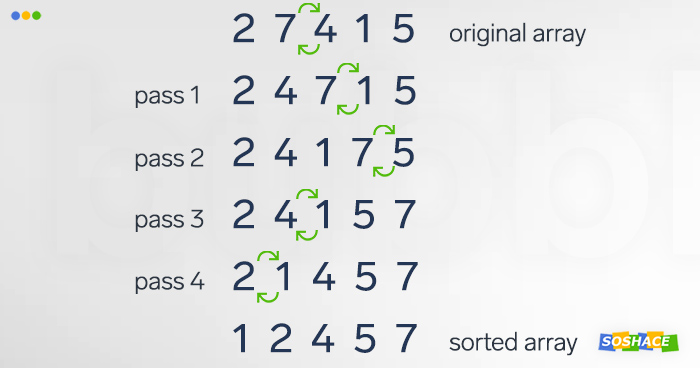
* **Linear structure** organized the elements in a sequence/linear list. Examples of this data structures are: arrays, linked lists, queues, and stacks.
* **Non-linear structure**, as the name suggests, allows the nodes to traverse and intertwine. Examples of this data structures are: graphs and trees.

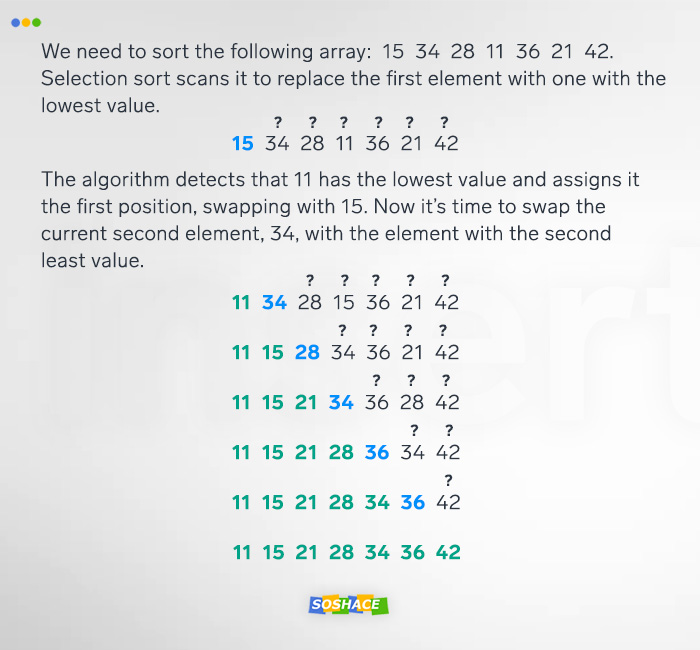
**Which operations can be performed on various data structures?**

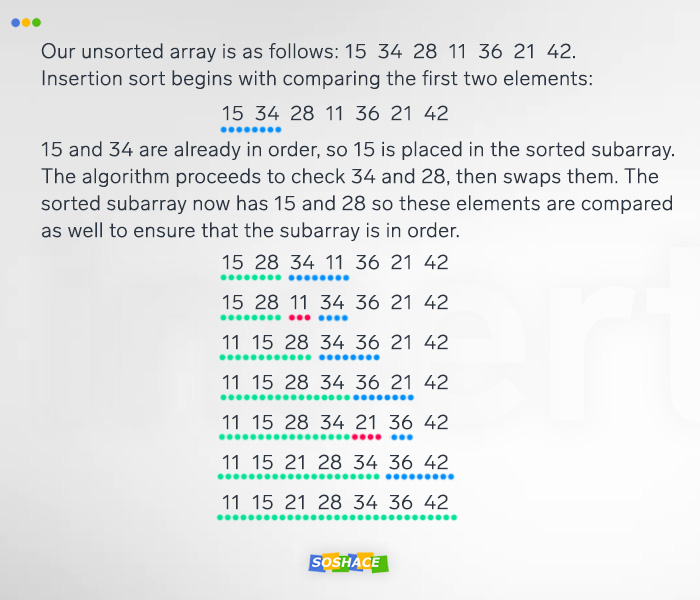
* Insert: Add a new element to the given list.
* Delete: Delete an existing element from the given list.
* Traverse: Access each element once for later processing.
* Search: Locate an element if it exists in the list.
* Sort: Arrange the elements in some order.

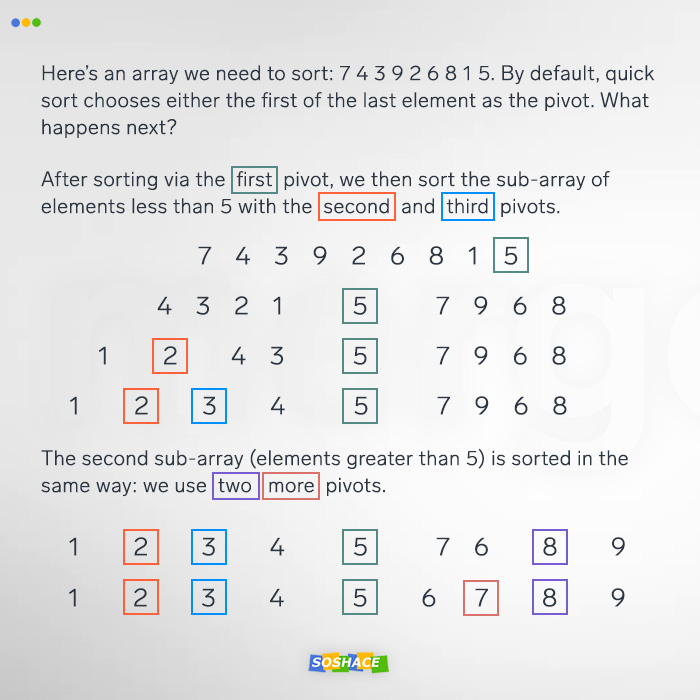
4. How does Algorithm X work?

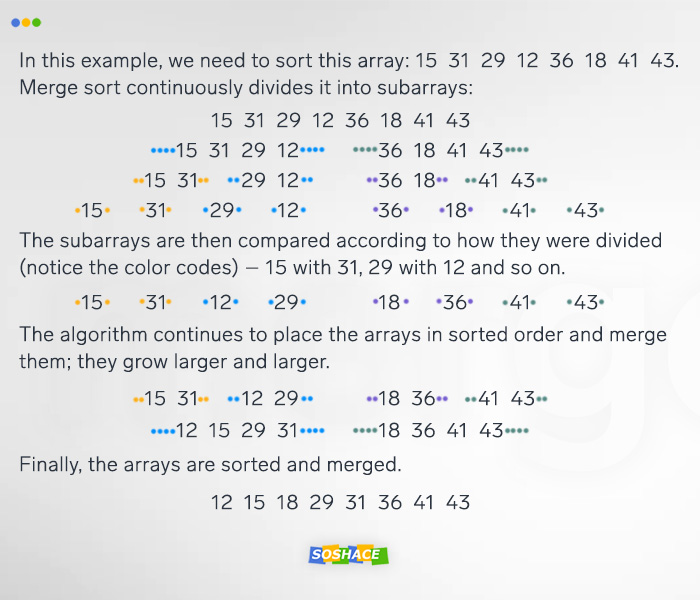
A picture is worth a thousand words — so you’re in luck! In our “Sorting Algorithms Overview” article, we created visualizations of how **bubble sort, selection sort, insertion sort, quick sort,** and **merge sort** work:











5. What is a binary tree?

Binary tree is tree-type data structure. Its nodes only branch out into 2 directions; they are called “left child” and “right child”, while the tree’s main (i.e. topmost) node is called the root.

Practice

Now it’s time to put your knowledge to practice. Let’s get coding!

6. Create a bubble sort algorithm



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public static void sort(int[] input) {  int inputLength = input.length;  int temp;  boolean is\_sorted;  for (int i = 0; i < inputLength; i++) {  is\_sorted = true;  for (int j = 1; j < (inputLength - i); j++) {  if (input[j - 1] > input[j]) {  temp = input[j - 1];  input[j - 1] = input[j];  input[j] = temp;  is\_sorted = false;  }  }  if (is\_sorted) break;  System.out.println("\n");  }  } |

7. Compare time complexity of two algorithms

This function will get an array of integers and return the indices of two numbers, adding them to a specific element.

**Sample 1:**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public int[] twoSum(int[] nums, int target) {      for (int i = 0; i < nums.length; i++) {          for (int j = i + 1; j < nums.length; j++) {              if (nums[j] == target - nums[i]) {                  return new int[] { i, j };              }          }      }      throw new IllegalArgumentException("No two sum solution");  } |

**Sample 2:**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public int[] twoSum(int[] nums, int target) {      Map<Integer, Integer> map = new HashMap<>();      for (int i = 0; i < nums.length; i++) {          int complement = target - nums[i];          if (map.containsKey(complement)) {              return new int[] { map.get(complement), i };          }          map.put(nums[i], i);      }      throw new IllegalArgumentException("No two sum solution");  } |

*Answer for sample 1: Time complexity : O(n^2). For each element, we try to find its complement by looping through the rest of array which takes O(n) time.*

*Answer for sample 2: Time complexity : O(n). We traverse the list containing nn elements only once. Each table access costs only O(1) time.*

8. Write an algorithm to sort the specified array

int[] input = { 4, 1, 2, 7, 10, 1, 2, 4, 4, 7, 1, 2, 1, 10, 1, 2, 4, 1, 2, 7, 10, 1, 2};

**Answer:**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import java.util.Arrays;  public class CountiSorter{    public static void main(String[] args) {      int[] input = { 4, 1, 2, 7, 10, 1, 2, 4, 4, 7, 1, 2, 1, 10, 1, 2, 4, 1, 2, 7, 10, 1, 2};      int k = 10;      // sorting array using Counting Sort Algorithm      countingSort(input, k);      System.out.println(Arrays.toString(input));    }    public static void countingSort(int[] input, int k) {      // create buckets      int counter[] = new int[k + 1];      // fill buckets      for (int i : input) {        counter[i]++;      }      // sort array      int ndx = 0;      for (int i = 0; i < counter.length; i++) {        while (0 < counter[i]) {          input[ndx++] = i;          counter[i]—;        }      }    }  } |

9. Remove duplicates from a sorted list

In a sorted linked list, delete all duplicates.

**Sample:**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class ListNode {      int val;      ListNode next;      ListNode(int x) { val = x; }  }  Input: 1->1->2->3->3->3->4  Output: 1->2->3->4 |

**Answer:**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public ListNode deleteDuplicates(ListNode head) {      ListNode current = head;      while (current != null && current.next != null) {          if (current.next.val == current.val) {              current.next = current.next.next;          } else {              current = current.next;          }      }      return head;  } |

10. Find the element’s first and last positions in a sorted array

In an array of integers sorted in ascending order, find the starting and ending positions of a given element. If the target element cannot be located, return [-1, -1].  
**Sample:**



|  |  |
| --- | --- |
| 1  2  3  4 | Input:  int[] input = {2,3,3,5,5,5,5,6,8,8,8,10,12,14};  int target = 5;  Output: [3, 6] |

**Answer:**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | class Solution {      private int extremeInsertionIndex(int[] nums, int target, boolean left) {          int lo = 0;          int hi = nums.length;          while (lo < hi) {              int mid = (lo + hi) / 2;              if (nums[mid] > target || (left && target == nums[mid])) {                  hi = mid;              }              else {                  lo = mid+1;              }          }          return lo;      }      public int[] searchRange(int[] nums, int target) {          int[] targetRange = {-1, -1};          int leftIdx = extremeInsertionIndex(nums, target, true);          if (leftIdx == nums.length || nums[leftIdx] != target) {              return targetRange;          }          targetRange[0] = leftIdx;          targetRange[1] = extremeInsertionIndex(nums, target, false)-1;          return targetRange;      }  } |