Contents

[1. Array 5](#_Toc34774736)

[1. Find duplicate in the array 5](#_Toc34774737)

[Problem 5](#_Toc34774738)

[Reference 5](#_Toc34774739)

[Approach 5](#_Toc34774740)

[Solution 5](#_Toc34774741)

[Time and space complexity 6](#_Toc34774742)

[2. Find Numbers with Even Number of Digits 6](#_Toc34774743)

[Problem 6](#_Toc34774744)

[Reference 6](#_Toc34774745)

[Approach 6](#_Toc34774746)

[Solution 6](#_Toc34774747)

[Time and space complexity 7](#_Toc34774748)

[3. Largest Number 7](#_Toc34774749)

[Problem 7](#_Toc34774750)

[Reference 7](#_Toc34774751)

[Approach 7](#_Toc34774752)

[Solution 7](#_Toc34774753)

[Time and space complexity 8](#_Toc34774754)

[4. Search Insert Position (Binary Search) 8](#_Toc34774755)

[Problem 8](#_Toc34774756)

[Reference 8](#_Toc34774757)

[Approach 8](#_Toc34774758)

[Solution 9](#_Toc34774759)

[Time and space complexity 9](#_Toc34774760)

[5. Count Negative Numbers in a Sorted Matrix 9](#_Toc34774761)

[Problem 9](#_Toc34774762)

[Reference 9](#_Toc34774763)

[Approach 9](#_Toc34774764)

[Solution 10](#_Toc34774765)

[Time and space complexity 10](#_Toc34774766)

[6. Move Zeroes to end of array 10](#_Toc34774767)

[Problem 10](#_Toc34774768)

[Reference 11](#_Toc34774769)

[Approach 11](#_Toc34774770)

[Solution 11](#_Toc34774771)

[Time and space complexity 11](#_Toc34774772)

[7. Two Sum 11](#_Toc34774773)

[Problem 11](#_Toc34774774)

[Reference 12](#_Toc34774775)

[Approach 12](#_Toc34774776)

[Solution 12](#_Toc34774777)

[Time and space complexity 12](#_Toc34774778)

[8. Merge two sorted array 12](#_Toc34774779)

[Problem 12](#_Toc34774780)

[Reference 13](#_Toc34774781)

[Approach 13](#_Toc34774782)

[Solution 13](#_Toc34774783)

[Time and space complexity 13](#_Toc34774784)

[2. String 13](#_Toc34774785)

[1. Check if a string is substring of source. (Rabin Karp Algorithm) 13](#_Toc34774786)

[Problem 13](#_Toc34774787)

[Reference 14](#_Toc34774788)

[Approach 14](#_Toc34774789)

[Solution 14](#_Toc34774790)

[Time and space complexity 15](#_Toc34774791)

[2. Anagram 15](#_Toc34774792)

[Problem 15](#_Toc34774793)

[Reference 15](#_Toc34774794)

[Approach 15](#_Toc34774795)

[Solution 16](#_Toc34774796)

[Time and space complexity 16](#_Toc34774797)

[3. Math 16](#_Toc34774798)

[1. Palindrome Number 16](#_Toc34774799)

[Problem 16](#_Toc34774800)

[Reference 17](#_Toc34774801)

[Approach 17](#_Toc34774802)

[Solution 17](#_Toc34774803)

[Time and space complexity 17](#_Toc34774804)

[4. Linked List 17](#_Toc34774805)

[1. Reverse single linked list 17](#_Toc34774806)

[Problem 17](#_Toc34774807)

[Reference 17](#_Toc34774808)

[Approach 17](#_Toc34774809)

[Solution 17](#_Toc34774810)

[Time and space complexity 18](#_Toc34774811)

[2. Find middle element of single linked list 18](#_Toc34774812)

[Problem 18](#_Toc34774813)

[Reference 18](#_Toc34774814)

[Approach 18](#_Toc34774815)

[Solution 18](#_Toc34774816)

[Time and space complexity 19](#_Toc34774817)

[3. Detect cycle in linked list 19](#_Toc34774818)

[Problem 19](#_Toc34774819)

[Reference 19](#_Toc34774820)

[Approach 19](#_Toc34774821)

[Solution 19](#_Toc34774822)

[Time and space complexity 20](#_Toc34774823)

[4. Detect cycle in linked list and return starting node of loop 20](#_Toc34774824)

[Problem 20](#_Toc34774825)

[Reference 20](#_Toc34774826)

[Approach 20](#_Toc34774827)

[Solution 20](#_Toc34774828)

[Time and space complexity 21](#_Toc34774829)

[5. Sorting 21](#_Toc34774830)

[1. Insertion Sort in array 21](#_Toc34774831)

[Problem 21](#_Toc34774832)

[Reference 21](#_Toc34774833)

[Approach 22](#_Toc34774834)

[Solution 22](#_Toc34774835)

[Time and space complexity 22](#_Toc34774836)

[2. Insertion Sort in Single Linked list 22](#_Toc34774837)

[Problem 22](#_Toc34774838)

[Reference 22](#_Toc34774839)

[Approach 22](#_Toc34774840)

[Solution 23](#_Toc34774841)

[Time and space complexity 23](#_Toc34774842)

[6. Selection Sort in array 23](#_Toc34774843)

[Problem 23](#_Toc34774844)

[Reference 24](#_Toc34774845)

[Approach 24](#_Toc34774846)

[Solution 24](#_Toc34774847)

[Time and space complexity 24](#_Toc34774848)

[7. Selection Sort in Single linked list 24](#_Toc34774849)

[Problem 24](#_Toc34774850)

[Reference 24](#_Toc34774851)

[Approach 24](#_Toc34774852)

[Solution 24](#_Toc34774853)

[Time and space complexity 25](#_Toc34774854)

[8. Bubble Sort in array 25](#_Toc34774855)

[Problem 25](#_Toc34774856)

[Reference 25](#_Toc34774857)

[Approach 25](#_Toc34774858)

[Solution 25](#_Toc34774859)

[Time and space complexity 26](#_Toc34774860)

[9. Bubble Sort in Single Linked list 26](#_Toc34774861)

[Problem 26](#_Toc34774862)

[Reference 26](#_Toc34774863)

[Approach 26](#_Toc34774864)

[Solution 26](#_Toc34774865)

[Time and space complexity 27](#_Toc34774866)

[10. Merge Sort in array 27](#_Toc34774867)

[Problem 27](#_Toc34774868)

[Reference 27](#_Toc34774869)

[Approach 27](#_Toc34774870)

[Solution 28](#_Toc34774871)

[Time and space complexity 29](#_Toc34774872)

[13. Merge Sort in linked list 29](#_Toc34774873)

[Problem 29](#_Toc34774874)

[Reference 30](#_Toc34774875)

[Approach 30](#_Toc34774876)

[Solution 30](#_Toc34774877)

[Time and space complexity 32](#_Toc34774878)

[6. Dynamic Programming 32](#_Toc34774879)

[1. Maximum sum in Contiguous Sub-Array 32](#_Toc34774880)

[Problem 32](#_Toc34774881)

[Reference 32](#_Toc34774882)

[Approach 32](#_Toc34774883)

[Solution 32](#_Toc34774884)

[Time and space complexity 32](#_Toc34774885)

# 1. Array

## Find duplicate in the array

### Problem

Given an array of integers, 1 ≤ a[i] ≤ *n* (*n* = size of array), some elements appear **twice** and others appear **once**.

Find all the elements that appear **twice** in this array.

Input:

[4,3,2,7,8,2,3,1]

Output:

[2,3]

### Reference

LEETCODE

### Approach

Since content is with-in array index range. We will use itself array as hashTable and when we found item first time we change sign to negative and if second time we get same negative it is duplicate. If data was not in-range we will use hashmap or set.

### Solution

public List<Integer> findDuplicates(int[] nums) {

List<Integer> list = new ArrayList<>();

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

int n=Math.abs(nums[i])-1;

if (nums[n] < 0) {

list.add(n+1);

} else {

nums[n] = -nums[n];

}

}

return list;

}

### Time and space complexity

Time - O(n)

Space – O(1).In space as list needed only for this question

## Find Numbers with Even Number of Digits

### Problem

Given an array nums of integers, return how many of them contain an even number of digits.

Example 1:

Input: nums = [12,345,2,6,7896]

Output: 2

Explanation:

12 contains 2 digits (even number of digits).

345 contains 3 digits (odd number of digits).

2 contains 1 digit (odd number of digits).

6 contains 1 digit (odd number of digits).

7896 contains 4 digits (even number of digits).

Therefore only 12 and 7896 contain an even number of digits.

### Reference

LEETCODE, MATH

### Approach

Approach 1 - can be to iterate over loop and convert each number to String and then check length is even or odd.

Approach 2 - can be to iterate over loop and use Math.log10 method and then check result%2==0. if it is true it is ODD else EVEN.

### Solution

public int findNumbers(int[] nums) {

int c=0;

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

int result = (int)Math.log10(nums[i]);

if(result %2!=0){

c++;

}

}

return c;

}

### Time and space complexity

Time - O(n)

Space – O(1)

## Largest Number

### Problem

Given a list of non-negative integers, arrange them such that they form the largest number.

Example 1:

Input: [10,2]

Output: "210"

Example 2:

Input: [3,30,34,5,9]

Output: "9534330"

### Reference

Leetcode, sort, array

### Approach

We need to sort data smartly i.e. write comparator smartly. So, for two string like 3 and 34 to check which one should come first just contact both combo like – 334 and 343. Now we know number larger can be made if 34 comes first and 3 after that.

So, we use above logic and sort the array.

### Solution

public String largestNumber(int[] nums) {

String[] str = new String[nums.length];

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

str[i]=String.valueOf(nums[i]);

}

Arrays.sort(str,(o1,o2)->{

String s1=o1+o2;

String s2=o2+o1;

return s2.compareTo(s1);

});

if("0".equals(str[0])){

return str[0];

}

StringBuilder sb=new StringBuilder(nums.length);

for(String s:str){

sb.append(s);

}

return sb.toString();

}

### Time and space complexity

Time - O(nlogn)

## Search Insert Position (Binary Search)

### Problem

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You may assume no duplicates in the array.

Example 1:

Input: [1,3,5,6], 5

Output: 2

Example 2:

Input: [1,3,5,6], 2

Output: 1

### Reference

LEETCODE, BINARY-SEARCH, ARRAY

### Approach

Binary search is best algorithm to search in a sorted array. It takes o(logn) time.

1. set start=0 and end=length-1

2. Iterate till start<=end

3. get mid of (start+end)/2 and check if target is in left or right or in the middle.

4. If target<arr[mid] it means target is present in left. So update end=mid-1.

5. So, by this approach we are dividing the items to be searched to half every time.

In this particular problem if element does not exist. In such case start will tell the position of element where it should supposed to be. In classic binary search if item does not found we return -1

### Solution

public int searchInsert(int[] nums, int target) {

int start = 0;

int end = nums.length - 1;

int mid = 0;

while (start <= end) {

mid = (start + end) / 2;

if (nums[mid] == target) {

return mid;

} else if (nums[mid] < target) {

start = mid + 1;

} else {

end = mid - 1;

}

}

return start;

}

### Time and space complexity

O(logn)

O(1)

## Count Negative Numbers in a Sorted Matrix

### Problem

Given m \* n matrix grid which is sorted in non-increasing order both row-wise and column-wise.

Return the number of negative numbers in grid.

Example 1:

Input: grid = [[4,3,2,-1],[3,2,1,-1],[1,1,-1,-2],[-1,-1,-2,-3]]

Output: 8

Explanation: There are 8 negatives number in the matrix.

### Reference

LEETCODE, ARRAY-2D, BINARY-SEARCH

### Approach

We use binary search algorithm row by row and find center if it’s negative update end=center-1 else start=center+1

When loop terminates start will be the index of first negative. So, total negative in that row is row.length – start.

Since we also know that column is also decreasing so, for second row we update end to start-1.so, that we will apply binary search to only 0 to last positive number in previous row.

And we keep it doing till last row.

### Solution

public int countNegatives(int[][] grid)

{

int c = 0;

for (int i = 0, end = grid[i].length - 1; i < grid.length; i++) {

int start = 0;

while (start <= end) {

int mid = (start + end) / 2;

if (grid[i][mid] < 0) {

end = mid - 1;

} else {

start = mid + 1;

}

}

c = c + grid[i].length - start;

end = start - 1;

}

return c;

}

### Time and space complexity

Time - O(n+m)

Space - O(1)

## Move Zeroes to end of array

### Problem

Given an array nums, write a function to move all 0's to the end of it while maintaining the relative order of the non-zero elements.

Example:

Input: [0,1,0,3,12]

Output: [1,3,12,0,0]

### Reference

LEETCODE, ARRAY

### Approach

Here to make code generic we move val to the end of array.

We will keep count of val in c. and if c> 0 means we have at least one val. We move current element to i-c location. And update arr[i] to val.

It works because we make sure we are shifting non zero element to next available index on left.whic will be i-c.

If we does not have any zero we will not shift.

e.g. – 010004 -> in this case 1 will be shift to zero index .

i.e. 100004. Now c=1 and i=1. So, c keep on incrementing to 4. For i=5,arr[5-4]=arr[5] .

so,op will be 140000.

### Solution

public void searchAndShift(int[] arr, int val) {

int c = 0;

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

if (arr[i] == val) {

c++;

} else if (c > 0) {

arr[i - c] = arr[i];

arr[i] = val;

}

}

}

### Time and space complexity

Time - O(n)

Space – O(1)

## Two Sum

### Problem

Given an array of integers, return indices of the two numbers such that they add up to a specific target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

Example:

Given nums = [2, 7, 11, 15], target = 9,

Because nums[0] + nums[1] = 2 + 7 = 9,

return [0, 1].

### Reference

LEETCODE, ARRAY, HASHMAP

### Approach

Take hashmap and check if current item is in map if yes return else put (target-current item) in a loop.

### Solution

public int[] twoSum(int[] numbers, int target) {

Map<Integer, Integer> map = new HashMap<>();

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

if(map.containsKey(numbers[i])) {

return new int[] {map.get(numbers[i]), i};

}

map.put(target - numbers[i], i);

}

return null;

}

### Time and space complexity

Time - O(n)

Space – O(n) (hashmap)

## Merge two sorted array

### Problem

Given two sorted integer arrays nums1 and nums2, merge nums2 into nums1 as one sorted array.

Note:

The number of elements initialized in nums1 and nums2 are m and n respectively.

You may assume that nums1 has enough space (size that is greater or equal to m + n) to hold additional elements from nums2.

Example:

Input:

nums1 = [1,2,3,0,0,0], m = 3

nums2 = [2,5,6], n = 3

Output: [1,2,2,3,5,6]

### Reference

LEETCODE, ARRAY

### Approach

Since we have empty places at end of nums1. We will start comparing from end i.e. from m-1 and n-1 and start filling larger element among two to the end of nums1. E.g. for 6 and 3 -> Output would for nums1=[1,2,3,0,0,6] and we decrement n only and we keep on doing it till the length of nums1 (not m).

### Solution

public void merge(int[] nums1, int m, int[] nums2, int n) {

int i = nums1.length - 1;

n--;

m--;

while (m >= 0 && n >= 0) {

if (nums1[m] > nums2[n]) {

nums1[i--] = nums1[m--];

} else {

nums1[i--] = nums2[n--];

}

}

while (n >= 0) {

nums1[i--] = nums2[n--];

}

}

### Time and space complexity

Time - O(nums1.length)

Space – O(1)

# 2. String

## Check if a string is substring of source. (Rabin Karp Algorithm)

### Problem

Check whether a given string is substring of source.

Example 1-

Input –

helloji,loj

Output –

True

### Reference

STRING, RABIN-KARP, ABDUL BARI

### Approach

Naive approach is to check character by character starting from i=0 and if not matched go back and now check for i=1.

Better approach -

It uses hashcode of a string and instead of matching character one by one.

\* We just match hashcode and once hashcode matched we check the content.

\* If not matched we subtract hashcode of first character and add hashcode of new character

\* It saves time of un-necessary comparison all the time.

\* But in worst case it might be possible that we might get hashcode same on every check.

\* To calculate hashcode again we just subtract hashcode of first character and add hashcode of next character in previous value.

\* For better performance make hash code function better to avoid un-necessary collision.

### Solution

int hSource = 0;

int hStr = 0;

//calculate hashcode of both source and string for first comparison

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

hSource = hSource + hashFunction(source.charAt(i));

hStr = hStr + hashFunction(str.charAt(i));

}

// we compare hash first and if matched return true.

// calculate hash again except for last value of i as we are generating hash in

// advanced.

for (int i = 0; i <= source.length() - str.length(); i++) {

if (hStr == hSource) {

int j = 0;

for (j = 0; j < str.length(); j++) {

if (source.charAt(j + i) != str.charAt(j)) {

break;

}

}

if (j == str.length()) {

return true;

}

}

//to avoid calculation after last index

if (i < source.length() - str.length()) {

hSource = hSource - hashFunction(source.charAt(i)) + hashFunction(source.charAt(i + str.length()));

}

}

return false;

### Time and space complexity

\* worst case - o(n\*m)

\* Best case - o(n+m)

## Anagram

### Problem

An anagram is a word formed by rearranging the letters of a different word. typically using all the original letters exactly once.

Given two strings s and t, write a function to determine if t is an anagram of s.

You may assume the string contains only lowercase alphabets.

Example 1:

Input: s = "anagram", t = "nagaram"

Output: true

Example 2:

Input: s = "rat", t = "car"

Output: false

### Reference

LEETCODE, ARRAY

### Approach

Take array with 26 size and from first string increment counter and for second decrement counter.

After loop finished iterate over table array and check if any non-zero value exists it’s not anagram.

### Solution

public boolean isAnagram(String s, String t) {

int[] table=new int[26];

if(s.length()!=t.length()){

return false;

}

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

table[s.charAt(i)-'a']+=1;

table[t.charAt(i)-'a']-=1;

}

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

if(table[i]!=0){

return false;

}

}

return true;

}

### Time and space complexity

Time - O(n+26)

Space – O(26) means constant

# 3. Math

## 1. Palindrome Number

### Problem

Determine whether an integer is a palindrome. An integer is a palindrome when it reads the same backward as forward.

Example 1:

Input: 121

Output: true

Example 2:

Input: -121

Output: false

Explanation: From left to right, it reads -121. From right to left, it becomes 121-

### Reference

LEETCODE, MATH, MOD

### Approach

Reverse the original number by adding remainder to the original number – res\*10+(num%10);

### Solution

public boolean isPalindrome(int num) {

int res = 0;

int num1 = num;

while (num > 0) {

res = res \* 10 + (num % 10);

num = num / 10;

}

return num1 == res;

}

### Time and space complexity

Time - o(n)

Space – o(1)

# 4. Linked List

## 1. Reverse single linked list

### Problem

Reverse single linked list.

### Reference

SINGLE LINKED LIST, LEETCODE

### Approach

1. It can be done using three pointers. One to hold the current node,one with prev and one next node
2. Start by setting prev=null and current=head.
3. Iterate till current is not null and for every iteration set next = current.next and now set current.next=prev
4. And after that update prev to current and current to next.
5. After loop terminates current is the head of revsered list

### Solution

public void reverse() {

Node<T> prev = null;

while (head != null) {

Node<T> next = head.getNext();

head.setNext(prev);

prev = head;

head = next;

}

head = prev;

}

### Time and space complexity

Time - o(n)

Space – o(1)

## 2. Find middle element of single linked list

### Problem

Middle element of single linked list.

Input: [1,2,3,4,5,6]

Output: 3

### Reference

SINGLE LINKED LIST, LEETCODE

### Approach

1. Take two pointers one run at double speed and other single.
2. Once fast pointer reaches null. At that point slow will be in the middle.
3. If we want to return 4 in above code use below code –

Node<Integer> fast = head;

Node<Integer> slow = head;

while (fast!= null && fast.getNext()!= null) {

slow = slow.getNext();

fast = fast.getNext().getNext();

}

return slow;

### Solution

public static Node<Integer> findMiddleOfLinkedList(Node<Integer> head) {

if (head == null) {

return head;

}

Node<Integer> fast = head;

Node<Integer> slow = head;

while (fast.getNext() != null && fast.getNext().getNext() != null) {

slow = slow.getNext();

fast = fast.getNext().getNext();

}

return slow;

}

### Time and space complexity

Time - o(n)

Space – o(1)

## 3. Detect cycle in linked list

### Problem

Check if cycle exists in single linked list.

Input:



Output:

true

### Reference

SINGLE LINKED LIST, LEETCODE

### Approach

1. Take two pointers one run at double speed and other single.
2. If at any point they meet-> there is a cycle. And if node reaches null value. No cycle.

### Solution

public boolean hasCycle(Node<Integer> head) {

Node<Integer> slow = head;

Node<Integer> fast = head;

while (fast != null && fast.getNext() != null) {

slow = slow.getNext();

fast = fast.getNext().getNext();

if (slow == fast) {

return true;

}

}

return false;

}

### Time and space complexity

Time - o(n)

Space – o(1)

## 4. Detect cycle in linked list and return starting node of loop

### Problem

Check if cycle exists in single linked list. If loop is present then it returns point to first node of loop. Else it returns NULL.



### Reference

SINGLE LINKED LIST, LEETCODE, GEEKSFORGEEKS

### Approach

1. Take two pointers one run at double speed and other single.
2. If at any point they meet-> there is a cycle. And if node reaches null value. No cycle.
3. If cycle exists set any of the two pointer to head and now loop till both of them meets again. That meeting point is cycle starting point.

### Solution

public static Node<Integer> detectCycle(Node<Integer> head) {

Node<Integer> slow = head;

Node<Integer> fast = head.getNext();

while (fast != null && fast.getNext() != null) {

slow = slow.getNext();

fast = fast.getNext().getNext();

if (slow == fast) {

slow=head;

while(slow!=fast) {

slow=slow.getNext();

fast=fast.getNext();

}

return fast;

}

}

return null;

}

### Time and space complexity

Time - o(n)

Space – o(1)

# 5. Sorting

**Stable algorithm** - Stable sorting algorithms maintain the relative order of records with equal keys (i.e. values). That is, a sorting algorithm is stable if whenever there are two records R and S with the same key and with R appearing before S in the original list, R will appear before S in the sorted list.

**In-place algorithm** is an algorithm which transforms input using no auxiliary data structure. However a small amount of extra storage space is allowed for auxiliary variables.

An **adaptive algorithm** is an algorithm that changes its behavior at the time it is run, based on information available and on a priori defined reward mechanism. E.g. shell sort. It takes into consideration some part of data already sorted. and hence work faster on such cases where such type of chunk exists.

**HYBRID ALGORITHM** - it is combination of two or more sorting algorithm to take advantage of both Algorithm.

E.g. 1. *TIMSORT* - *insertion sort + merge sort.*

\* As Insertion sort is faster than both merge and quick sort if elements are small e.g. <10.

\* For large data set it uses merge sort. Collections.sort uses it as in case of linked list no extra memeory needed in merge operation.

E.g. 2. *INTROSORT* - *Quicksort + Heapsort.*

\* As bad pivot selection can lead to O(n^2) in quick sort worst case. it uses hybrid of both 1. Insertion sort in array

## 1. Insertion Sort in array

### Problem

Sort the given 1 dimension array using insertion sort.

### Reference

ARRAY, GEEKSFORGEEKS

### Approach

1. On small data set insertion sort, selection sort works better than merge or quick sort.
2. It is just like arranging cards.
3. Insertion sort requires more swap as compared to selection sort and hence is not preferred where write operation is costly. On other hand insertion sort is more efficient than bubble or selection as it speeds up when array is already partially sorted.
4. Insert ith element in between 0 to i-1 position where it needed to be present. In general Select an item and on the left of that item keep sorted data and on right keep on considering and place it in correct place on left.
5. In-place, Iterative, adaptive, stable.

### Solution

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

for (int j = 0; j < i; j++) {

if (array[i]<array[j]) {

swap(array, i, j);

}

}

}

### Time and space complexity

Time - o(n^2)

Space – o(1)

## 2. Insertion Sort in Single Linked list

### Problem

Sort the given Single linked list using insertion sort.

### Reference

SINGLE LINKED LIST, LEETCODE, GEEKSFORGEEKS

### Approach

Here we will create a dummy list and start adding element to in it in sorted order.

1. curr node will point to current iteration element.

2. prev will point to previous node after which element will be needed to insert

3. dummy - use to hold the result modified sorted list.

Steps -

1. Start curr from head and set dummy as least integer and prev also.

2. If item to be inserted is greater then prev node. we need to start searching from start i.e. from dummy. so, set prev=dummy

3. Now loop till we find position of new item. i.e. where it is less then prev.next or prev.next is null.

In this way we now know after prev we can add that item as after that other elements are greater.

4. Now just add element between prev and prev.next

5. Update curr node to next element of the loop.

### Solution

public static Node<Integer> insertionSort(Node<Integer> head) {

Node<Integer> dummy = new Node<Integer>(Integer.MIN\_VALUE);

Node<Integer> curr = head;

Node<Integer> prev = dummy;

while (curr != null) {

// Store nextNode for next iteration

Node<Integer> nextNode = curr.getNext();

// to save checking from start- below condition is used

if (prev.getData() > curr.getData()) {

prev = dummy;

}

// go to a point where we need to insert new item starting from prev.

while (prev.getNext() != null && prev.getNext().getData() < curr.getData()) {

prev = prev.getNext();

}

// insert current node between prev and prev.next

curr.setNext(prev.getNext());

prev.setNext(curr);

curr = nextNode;

}

return dummy.getNext();

}

### Time and space complexity

Time - o(n^2)

Space – o(1)

## Selection Sort in array

### Problem

On a given array apply selection sort to sort the data.

### Reference

ARRAY, LEETCODE, GEEKSFORGEEKS

### Approach

1. In it we find min in array and swap it with 0 index and then start searching min again from 1 to n and now swap min with 1 and so on.

2. It is an in-place algorithm

### Solution

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

int minIndex = i;

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

if (array[j].compareTo(array[minIndex]) < 0) {

minIndex = j;

}

}

swap(array, i, minIndex);

}

### Time and space complexity

Time - O(n^2)

Space – O(1)

## Selection Sort in Single linked list

### Problem

On a given single linked list apply selection sort to sort the data.

### Reference

SINGLE LINKED LIST, GEEKSFORGEEKS

### Approach

1. In it we find min in Linked list and swap it with 0 index and then start searching min again from 1 to n and now swap min with 1 and so on.
2. For swapping we will swap content of the data not the node itself.

3. It is an in-place algorithm

### Solution

public static Node<Integer> selectionSort(Node<Integer> head) {

Node<Integer> temp = head;

while (temp != null) {

Node<Integer> minNode = temp;

Node<Integer> dummy = temp.getNext();

while (dummy != null) {

if (dummy.getData() < minNode.getData()) {

minNode=dummy;

}

dummy=dummy.getNext();

}

int data = minNode.getData();

minNode.setData(temp.getData());

temp.setData(data);

temp = temp.getNext();

}

return head;

}

### Time and space complexity

Time - O(n^2)

Space – O(1)

## Bubble Sort in array

### Problem

Apply bubble sort algorithm to sort the array.

### Reference

ARRAY

### Approach

1. Compare 0 element with 1 and arrange them. Then it take 1 element with 2 and arrange.

So after 1st iteration largest element moved to the end of array.

From next iteration onwards we will start from 0 and ignore last element as it is already in the correct position.

2. Stable, In-place

### Solution

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

// if after entire below loop no swap happen then it means array is already sorted

boolean swap=false;

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

if (array[j].compareTo(array[j+1]) > 0) {

swap(array, j, j+1);

swap=true;

}

}

//break as array is now sorted.

if(!swap) {

break;

}

}

### Time and space complexity

Time - O(n^2)

Space – O(1)

## Bubble Sort in Single Linked list

### Problem

Apply bubble sort algorithm to sort the linked list.

### Reference

SINGLE LINKED LIST

### Approach

1. Compare 0 element with 1 and arrange them. Then it take 1 element with 2 and arrange.

So after 1st iteration largest element moved to the end of array.

From next iteration onwards we will start from 0 and ignore last element as it is already in the correct position.

2. Stable, In-place

3. We will swap data instead of swapping nodes for simpler solution

### Solution

public static Node<Integer> bubbleSort(Node<Integer> head) {

Node<Integer> end = null;

while (end != head) {

Node<Integer> next = head;

while (next.getNext() != null && next.getNext() != end) {

if (next.getData() > next.getNext().getData()) {

int t = next.getData();

next.setData(next.getNext().getData());

next.getNext().setData(t);

}

next = next.getNext();

}

end = next;

}

return head;

}

### Time and space complexity

Time - O(n^2)

Space – O(1)

## Merge Sort in array

Merge sort has best and worst case both as nlogn. But it takes extra memory. On other hand quick sort has avg case nlogn but worst case n^2

### Problem

Apply merge sort algorithm to sort the array.

### Reference

ARRAY, GEEKSFORGEEKS

### Approach

1. divide and conquer strategy
2. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves.



2. Stable, take extra memory

### Solution

public static void mergeSort(int[] arr) {

mergeSort(arr, 0, arr.length - 1);

}

private static void mergeSort(int[] arr, int i, int j) {

if (i < j) {

int mid = (i + j) / 2;

mergeSort(arr, i, mid); // divide left sub array

mergeSort(arr, mid + 1, j); // divide right sub array

merge(arr, i, j, mid); // merge the two sorted array.

}

}

private static void merge(int[] arr, int l, int r, int mid) {

int[] temp = new int[r - l + 1];

int i = l;

int j = mid + 1;

int count = 0;

while (i <= mid && j <= r) {

if (arr[i] <= arr[j]) {

temp[count++] = arr[i++];

} else {

temp[count++] = arr[j++];

}

}

while (i <= mid) {

temp[count++] = arr[i++];

}

while (j <= r) {

temp[count++] = arr[j++];

}

for (int p = l; p <= r; p++) {

arr[p] = temp[p - l];

}

}

### Time and space complexity

Time - O(nlogn)

Space – O(n)

## 13. Merge Sort in linked list

Merge sort has best and worst case both as nlogn. it takes constant space O(1) in linked list.

### Problem

Apply merge sort algorithm to sort the single linked list.

### Reference

ARRAY, LEETCODE

### Approach

1. divide and conquer strategy
2. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves.
3. So, basically we will find middle element of the list and detach middle.next

So, first half head will have data till middle and middleNext element will be the starting point of second list.

And then we merge them together (for merging we can use both iterative solution as well as recursive solution)

### Solution

public static Node<Integer> mergeSort(Node<Integer> head) {

if (head == null || head.getNext() == null) {

return head;

}

//find middle element

Node<Integer> middle = findMiddleOfLinkedList(head);

//point middleNext to the start of second half of list

Node<Integer> middleNext = middle.getNext();

//set end of first half to null. (for clear separation of two list)

middle.setNext(null);

//now call merge sort for first half

Node<Integer> left = mergeSort(head);

//call merge sort for second half

Node<Integer> right = mergeSort(middleNext);

// merge sorted list

return mergeSortedLists(left, right);

}

private static Node<Integer> mergeSortedLists(Node<Integer> first, Node<Integer> second) {

Node<Integer> dummy = new Node<>(Integer.MAX\_VALUE);

Node<Integer> head=dummy;

while (first != null && second != null) {

if (first.getData() <= second.getData()) {

dummy.setNext(first);

first = first.getNext();

} else {

dummy.setNext(second);

second = second.getNext();

}

dummy = dummy.getNext();

}

while (first != null) {

dummy.setNext(first);

first = first.getNext();

dummy = dummy.getNext();

}

while (second != null) {

dummy.setNext(second);

second = second.getNext();

dummy = dummy.getNext();

}

return head.getNext();

}

public static Node<Integer> findMiddleOfLinkedList(Node<Integer> head) {

if (head == null) {

return head;

}

Node<Integer> fast = head;

Node<Integer> slow = head;

while (fast.getNext() != null && fast.getNext().getNext() != null) {

slow = slow.getNext();

fast = fast.getNext().getNext();

}

return slow;

}

### Time and space complexity

Time - O(nlogn)

Space – O(1)

# 6. Dynamic Programming

## 1. Maximum sum in Contiguous Sub-Array

### Problem

Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

Example:

Input: [-2,1,-3,4,-1,2,1,-5,4],

Output: 6

Explanation: [4,-1,2,1] has the largest sum = 6.

### Reference

LEETCODE, UDEMY, DP

### Approach

\*Take global\_max which hold the max overall

\* And curr\_max will hold the max till curr iteration.

\* We will update curr\_max by this - store max of (current element, curr\_max+current element)

\* By this we make sure that either current is taken or previous one is included in contiguous space.

### Solution

public int maxSubArray(int[] nums) {

int curr\_max = nums[0];

int global\_max = nums[0];

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

curr\_max = MathUtil.max(nums[i], nums[i] + curr\_max);

if (curr\_max > global\_max) {

global\_max = curr\_max;

}

}

return curr\_max;

}

### Time and space complexity

O(n)

O(n)