## Answer of 1

Solution of 1(A):

Algorithm: mergeSortPlus(S)

Input: sequence S with n integers

Output: sequence S sorted

return mergeSortPlus(S,0, S.size() - 1);

Algorithm: mergeSortPlus(S, lower, upper)

Input: sequence S with n integers with upper and lower bounds

Output: sequence S sorted

Size ← upper – lower + 1

Mid ← (upper + lower) / 2

If Size <= 20 then

return insertionSort(S, lower, upper)

else if Size() > 1 then

(S1 , S2 ) ← partition(S, lower, mid, upper)

mergeSortPlus (S1, lower, mid )

mergeSortPlus (S2 , mid + 1, upper)

S ← merge(S1 , S2 )

return S

Algorithm: InsertionSort(arr, lower, upper)

Input: arr is an array of integers with lower and upper bounds to sort

Output: a sorted array

n ← arr.length

temp ← 0

j ← 0

for i ← lower + 1 to upper do

temp ← arr[i]

j ← i

while j > 0 && temp < arr[j - 1]

arr[j] ← arr[j - 1]

j ← j – 1

arr[j] ← temp

return arr

Solution of 1 (B):

public class MergeSortPlus extends Sorter {  
  
 //public sorter  
 public int[] sort(int[] input){  
  
 mergeSortPlus(input, 0, input.length - 1);  
  
 return input;  
 }  
  
 void insertionSort(int[] arr, int lower, int upper) {  
 int temp = 0;  
 int j = 0;  
  
 for (int i = lower + 1; i <= upper; i++) {  
 temp = arr[i];  
 j = i;  
  
 while (j > lower && temp < arr[j - 1]) {

arr[j] = arr[j - 1];  
 j--;  
 }  
  
 arr[j] = temp;  
 }  
 }  
  
 void mergeSortPlus(int[] arr, int lower, int upper) {  
 if (upper - lower + 1 <= 20) {  
 insertionSort(arr, lower, upper);  
 } else {  
 int mid = lower + (upper - lower) / 2;  
  
 mergeSortPlus(arr, lower, mid);  
 mergeSortPlus(arr, mid + 1, upper);  
  
 merge(arr, lower, mid, upper);  
 }  
 }  
  
 void merge(int[] arr, int low, int mid, int high) {  
 int n1 = mid - low + 1;  
 int n2 = high - mid;  
  
 int[] arr1 = new int[n1];  
 int[] arr2 = new int[n2];  
  
 System.*arraycopy*(arr, low, arr1, 0, n1);  
 System.*arraycopy*(arr, mid + 1, arr2, 0, n2);  
  
 int left = 0, right = 0;  
 int index = low;  
 while (left < n1 && right < n2) {  
 if (arr1[left] < arr2[right]) {  
 arr[index++] = arr1[left++];  
 } else {  
 arr[index++] = arr2[right++];  
 }  
 }  
  
 while (left < n1) {  
 arr[index++] = arr1[left++];  
 }  
  
 while (right < n2) {  
 arr[index++] = arr2[right++];  
 }  
 }  
  
 }

Solution of 1 (C):

The MergeSortPlus seems to run faster than the MergeSort due to the enhanced algorithm benefit of using InsertionSort. I compared it on large datasets of 100,000, 200,000, 300,000 and 500,000 elements comparing the time it took for each algorithm to finish. Each time, MergeSortPlus was faster.

The results are conclusive and in favor of MergeSortPlus being the faster algorithm.

## Answer of 2

Solution of 2 (A):

A diagram of a graph

Description automatically generated

A diagram of a tree

Description automatically generated

A diagram of a tree

Description automatically generated with medium confidence

## 

Solution of 2 (B):

Statement: Every binary tree of height 3 has at most 23=8 leaves.

Verdict: It Is true, for a binary tree each node can have 0, 1 or 2 nodes.

Hence, at height 0, we can have at most one element, the root node.

At Height 1, we can only have at most 21 = 2 leaf nodes.

At Height 2, we can only have at most 22 = 4 leaf nodes.

We can conclude, at height 3 we will have at most 23 = 8 leaf nodes.

Solution of 2 (C):

Going on with this pattern, at height n, we will have at most 2n leaf nodes.

## Answer of 3

List<Set<Integer>> powerSet(List<Integer> X){  
 List<Set<Integer>> P = new ArrayList<>();  
 Set<Integer> S = new HashSet<>();  
  
 P.add(S);  
  
 while (!X.isEmpty()){  
 Integer f = X.remove(0);  
  
 List<Set<Integer>> newSubsets = new ArrayList<>();  
  
 for (var x : P) {  
 Set<Integer> newSubset = new HashSet<>(x);  
 newSubset.add(f);  
 newSubsets.add(newSubset);  
 }  
  
 P.addAll(newSubsets);  
 }  
  
 return P;  
}

## Answer of 4

import java.util.ArrayList;  
import java.util.HashSet;  
import java.util.List;  
import java.util.Set;  
  
public class Problem04Solution {  
 public static void main(String[] args) {  
 Problem04Solution solution = new Problem04Solution();  
  
 List<List<Integer>> inputs = new ArrayList<>(List.*of*(  
 new ArrayList<>(List.*of*(1, 3, 9, 4, 8, 5)),  
 new ArrayList<>(List.*of*(1, 3, 9)),  
 new ArrayList<>(List.*of*(1, 3, 9, 4, 8, 5))  
 ));  
  
 int[] ks = new int[]{2, 5, 0};  
  
 for (int i = 0; i < inputs.size(); i++){  
 System.*out*.printf("Array: %s%n", inputs.get(i));  
 System.*out*.printf("Sum of %d exists in subset: %s %n%n",  
 ks[i],  
 solution.subsetSumExists(inputs.get(i), ks[i]));  
 }  
  
  
 }  
  
 Set<Integer> subsetSumExists(List<Integer> X, int k){  
 List<Set<Integer>> P = new ArrayList<>();  
 Set<Integer> S = new HashSet<>();  
  
 if (k == 0)  
 return S;  
  
 P.add(S);  
  
 Integer sum = 0;  
  
 while (!X.isEmpty()){  
 Integer f = X.remove(0);  
  
 List<Set<Integer>> newSubsets = new ArrayList<>();  
  
 sum = 0;  
  
 for (var x : P) {  
 Set<Integer> newSubset = new HashSet<>(x);  
 newSubset.add(f);  
  
 sum = newSubset.stream().mapToInt(Integer::*valueOf*).sum();  
  
 if (sum == k){  
 return newSubset;  
 }  
  
 newSubsets.add(newSubset);  
 }  
  
 P.addAll(newSubsets);  
 }  
  
 return null;  
 }  
}