Question 1

(a) (i)

With the increase of the problem size n, the growth rate of the algorithm execution time is the same as the growth rate of f(n), which is called the asymptotic time complexity of the algorithm, and is referred to as the time complexity. Where f(n) is a function of the problem size n. Time complexity does not include the low order terms and the first term coefficients of this function (ii)

7n2+9n+5

Order：7n^2>9n>5

does not include the low order terms :7n^2

the first term coefficients of this function：n^2

result：O(n^2)

(b)

In the initial state, the array to be sorted is divided into two parts that are sorted and unordered, and then traversal is performed, and the elements in the unsorted sequence are sequentially inserted into the sorted sequence.

the complexity:

The average complexity: the first step traverses the entire array n, then for each number, insert into the previous sorted array, you need to compare the sorted array n, that is, the complexity is n^2

The best complexity: When the element is an array, it only needs to traverse once, comparing only one number at a time. The complexity is n=n\*1.

(c)

non-computable : For a problem, no exact solution can be given.

For example: If there is a set of functions B (that is, this set B is a set of functions f), and B is not an empty set, nor is it all functions f, then whether or not the random program P belongs to B is uncalculable.

Question 2

(a)

public int getNumOfNonLetters(BinaryTree<String> bt) {  
 int num = 0;   
 if (bt.isEmpty()) {   
 return num;  
 } else {   
 num = getNumOfNonLetters(bt.leftChild())+1;  
 num = num+getNumOfNonLetters(bt.rightChild());  
 }  
 return num;  
}

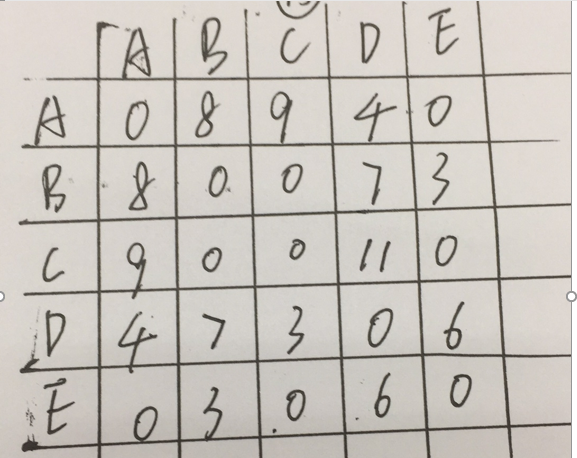
(b)

public class CharStackImpl implements CharStack {  
 private int size;   
 private Node head;   
  
 public CharStackImpl() {   
 }  
  
 @Override  
 public boolean isEmpty() {  
 return size == 0;  
 }  
  
 @Override  
 public void push(char c) {  
 Node current = head;  
 head = new Node();  
 head.value = c;  
 head.next = current;  
 size++;  
 }  
  
 @Override  
 public char top() {   
 if (head == null) {  
 throw new StackException();  
 }  
 return head.value;  
 }  
  
 @Override  
 public void pop() {   
 if (head == null) {  
 throw new StackException();  
 }  
 head = head.next;  
 size--;  
 }  
  
 class Node {

private char value;  
 private Node next;  
 }  
  
 class StackException extends RuntimeException {  
 public StackException() {  
 super("Stack is null");  
 }  
 }  
}

Question 3

(a)



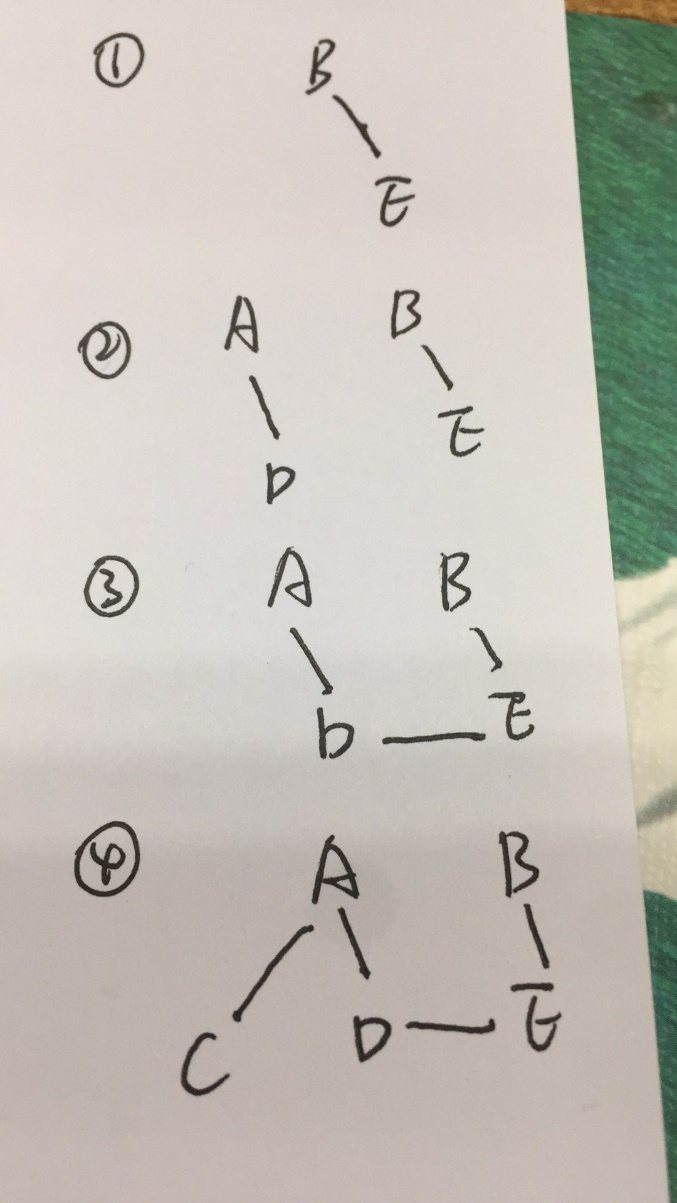
(b)

1. Sort the edges of the graph by their weights

2. The graph is traversed once to find the edge with the smallest weight (condition: the edge found this time cannot form a ring with the edge that has been added to the minimum spanning tree set). If the condition is met, the minimum spanning tree is added. If you do not meet the conditions, continue to traverse the graph to find the next smallest weight edge.

3. Step 1 is repeated recursively until n-1 edges are found (if there are n nodes in the graph, the number of edges in the minimum spanning tree should be n-1), and the algorithm ends. What we get is the minimum spanning tree of this figure.

(c)

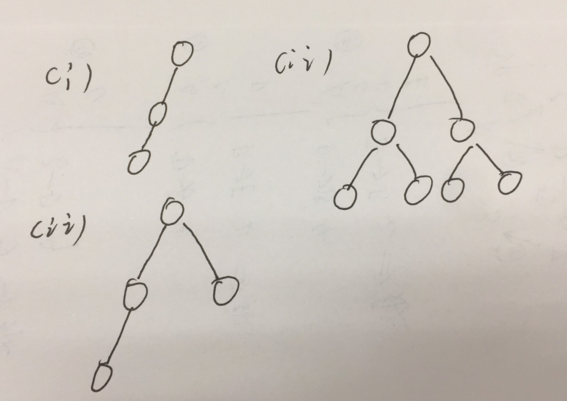


Question 4

(a)

Balanced Binary Tree Requirements For each node, the difference between its left and right subtrees cannot exceed 1. If you insert or delete a node so that the difference in height is greater than 1, rotate the nodes to maintain the binary tree A balanced state.

(b)



(c)

