**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)**

5n^2+11n+9：

Order:5n^2>11n>9

does not include the low order terms :5n^2

the first term coefficients of this function：n^2

result：O(n^2)

**(b)**

Select sort: select a minimum (or maximum) number from the array to be sorted, and place it in front of the array

Time complexity is:n^2：Every time traversing from an array, find the smallest number, n; n times need to be traversed

**(c)**

public class ParseTree {

String element;

ParseTree left;

ParseTree right;

public ParseTree(String element) {

this.element = element;

}

public ParseTree(String element, ParseTree left, ParseTree right) {

this.element = element;

this.left = left;

this.right = right;

}

}

**Question 2**

public class StringQueueImpl implements StringQueue {  
 private QueueCell head;   
 private int size = 0;   
 @Override  
 public boolean isEmpty() {  
 return size == 0;  
 }  
  
 @Override  
 public void add(String c) {  
 if (head == null) {   
 head = new QueueCell(c);  
 } else {   
 QueueCell current = head;   
 while (current != null) {  
 if (current.next == null) {  
 current.next = new QueueCell(c);  
 break;  
 } else {  
 current = head.next;  
 }  
 }  
 }  
 size++;  
 }  
  
 @Override  
 public String front() {   
 if (size == 0 || head == null) {  
 throw new QueueException();  
 }  
 return head.value;  
 }  
  
 @Override  
 public void removeFront() {   
 if (size == 0 || head == null) {  
 throw new QueueException();  
 }   
 if (head.next != null) {  
 head = head.next;  
 } else {  
 head = null;  
 }  
 size--;  
 }  
  
 class QueueCell {  
 private QueueCell next;  
 private String value;  
  
 QueueCell(String str) {  
 value = str;  
 }  
 }  
  
 class QueueException extends RuntimeException {  
 public QueueException() {  
 super("Queue is null");  
 }  
 }  
}

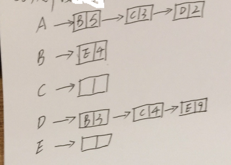
**Question 3**

**(a)**

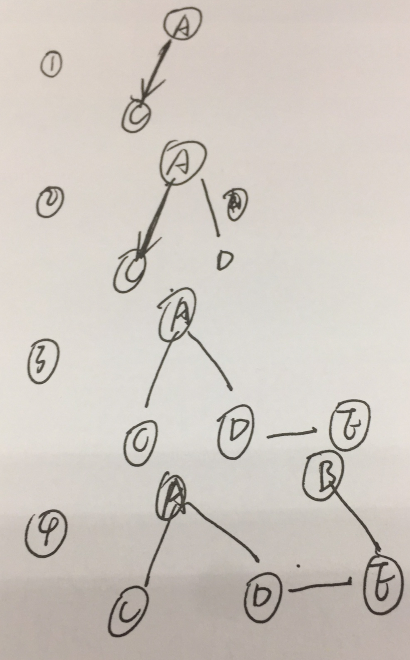
Description: Determine if a program can end or die in a limited amount of time

Prove: If there is a program H (H input required is a program) to determine the shutdown problem, we construct a new program K. This program calls H but the output of H is just the opposite: if the input of K is judged by H In case of stop, K will not stop; if the input of K is judged as not to stop by K, then K will stop. Now a contradiction arises: If we enter K into K (that is, use H to judge K for the program and give the input as K), then what about K? If you follow the logic, the answer should be: If K is not stopped, K will stop; if K is stopped, K will not stop. Contradictions have emerged. The only explanation for the contradiction is that there is no such universal H.

**(b)**



**(c)：**



**Question 4**

**(a)**

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

**(b)**

