1)A programação modular é um exemplo de paradigma orientado a objetos pois lida com classes e objetos.Certo ou Errado.Justifique.

R:Correto.Embora a programação modular não utilize nomes como classe, objeto e métodos, podemos fazer uma analogia entre os elementos dela com os da programação orientada a objeto. Imagine a seguinte situação: Temos o módulo peça que possui uma estrutura com o mesmo nome, a qual contém cor e id. Este também possui diversas funções de acesso. Agora, imagine que declaramos 6 ponteiros para peça em um módulo distinto da mesma aplicação. Podemos considerar esses seis ponteiros como sendo instâncias da classe peça(objetos), os campos cor e id como sendo os atributos da classe e as funções de acesso como sendo os métodos da classe. Portanto, sim, de uma certa perspectiva a programação modular pode ser considerada uma programação orientada a objetos

2)Apresente o esquema de algoritmo do quickSort apontando ao menos um hotspot preenchido com parâmetros do tipo ponteiro para função.
R: Esquema de algoritmo:

```
//Recebe um vetor (vet) a ser ordenado
//Recebe o inteiro esq
//Recebe o inteiro dir
//Recebe a função compara como parametro que retorna 0 se os dois parametros recebidos por ela forem iguais, 1 se o segundo for maior que o primeiro e -1 se o primeiro for maior que o segundo.

pivo <-- esq
i <-- esq + 1
Enquanto i<= dir
 j <-- i
resCompara <- compara(vet[j],vet[pivo]);
se resCompara == 1 então
ch <-- vet[j]
Enquanto j > pivo
vet[j] = vet[j - 1]
j <-- j - 1
Fim Enquanto
vet[j] <-- ch
pivo <-- pivo + 1
Fim Se
i <-- i + 1
Fim Enquanto
Se pivo - 1 >= esq
quick(vet, esq, pivo - 1)
Fim se
Se pivo + 1 <= dir
quick(vet,pivo + 1, dir)
Fim se
Fim
```

3)Faça a argumentação de corretude completa do algoritmo abaixo:

```
INICIO
Ind <— 1
ENQUANTO IND <= LL   FAÇA
INICIO
ATUAL <— ELE[IND]
AUX <— IND - 1
ENQUANTO AUX >= 1   E     ELE[AUX]  >  ATUAL
INICIO
ELE[AUX+1] <— ELE[AUX]
AUX <— AUX + 1
FIM
ELE[AUX + 1] <— ATUAL
IND <— IND + 1
FIM
FIM
```

R:
Argumentação de Corretude

```
=>AE
INICIO
Ind <— 1
=>AI1
ENQUANTO IND <= LL   FAÇA
INICIO
ATUAL <— ELE[IND]
AUX <— IND - 1
ENQUANTO AUX >= 1   E     ELE[AUX]  >  ATUAL
INICIO
ELE[AUX+1] <— ELE[AUX]
AUX <— AUX + 1
FIM
ELE[AUX + 1] <— ATUAL
IND <— IND + 1
FIM
FIM
=>AS
```

Argumentação de Sequencia:

-AE:
-o vetor ELE pode estar vazio, ou com um ou mais elementos.

-AS:
O vetor ELE está ordenado em ordem crescente se não for vazio. Caso ele seja vazio não faz nada

-AI1:
IND vale 1, ou seja, aponta para o primeiro elemento do vetor caso ele não seja vazio

BLOCO B1:

INICIO

FIM

Argumentação de repetição:
AE: AI1
AS: AI2
AINV:
a)O vetor ELE está dividido em dois grupos, o já pesquisado e o a pesquisar.
b)Ind aponta para um elemento do vetor ELE.

-AE => AINV
Pela AE, IND aponta para o primeiro elemento do vetor, portanto o conjunto dos já pesquisados é vazio e o dos a pesquisar corresponde ao vetor inteiro.

-AE && (C == F) => AS
Pela AE o vetor é vazio, logo LL = 0 e ind = 1, portando a repetição não é executada. Vale AS.

-AE && (C == T) + B1 => AINV
Pela AE o vetor não é vazio e ind é incrementado em uma unidade, o que faz com que ele aponte para o segundo elemento do vetor. O conjunto dos elementos já pesquisados possui agora o primeiro elemento do vetor e o de a pesquisar corresponde ao resto do vetor, portanto vale AINV.

-AINV && (C == T) + B1 => AINV
Para que AINV continue valendo, B1 deve garantir que um elemento passe do conjunto a pesquisar para o já pesquisado e IND aponte para o próximo elemento de a pesquisar

-AINV && (C == F) => AS
Nesse caso IND vale LL + 1, logo o conjunto a pesquisar corresponde ao vetor inteiro e o já pesquisado é vazio e o vetor está ordenado em ordem crescente, logo AS é válida.

-Término:
Como a cada ciclo o elemento apontado por IND vai do conjunto a pesquisar para o já pesquisado e a quantidade de elementos deste conjunto é finita, a repetição termina após um número finito de passos.

```
ATUAL <— ELE[IND]
=>AI2
AUX <— IND - 1
=>AI3
ENQUANTO AUX >= 1   E     ELE[AUX]  >  ATUAL
INICIO
ELE[AUX+1] <— ELE[AUX]
AUX <— AUX + 1
FIM
=>AI4
ELE[AUX + 1] <— ATUAL
=>AI5
IND <— IND + 1
=>AI6
FIM
```

Argumentação de Sequencia:
AI2: Atual recebe o valor do elemento apontado por IND

AI3: AUX aponta para o o elemento anterior a AUX, ou vale 0, se IND estiver apontando para o primeiro elemento do vetor
AI4: AUX aponta para um elemento no vetor, anterior a IND, o qual é menor ou igual ao elemento apontado por IND(ATUAL), ou aux = 0. A porção do vetor entre o segundo elemento do vetor e o apontado por IND está ordenada em ordem crescente
AI5: O elemento posterior ao apontado por AUX possui o mesmo valor que o elemento apontado por IND(ATUAL). A porção do vetor entre o primeiro elemento do vetor e o apontado por IND está ordenado em ordem crescente.
AI6: IND aponta para o próximo elemento a ser pesquisado no vetor.

B2:
ENQUANTO AUX >= 1 E ELE[AUX] > ATUAL
INICIO
ELE[AUX+1] <— ELE[AUX]
AUX <— AUX + 1
FIM

Argumentação de Repetição:

AE: AI3
AS: AI4

AINV:
a)O subvetor de ELE compreendido entre o primeiro elemento do vetor e o elemento apontado por AUX está dividido em dois grupos, o já pesquisado e o a pesquisar.
b)AUX aponta para um elemento do subvetor.

AE => AINV
-Pela AE, AUX está apontando para o ultimo elemento do subvetor e, portanto, o conjunto já pesquisado do subvetor é vazio e o a pesquisar contém o subvetor inteiro, portando, vale AINV

AE && (C == F) => AS
- Pela AE, caso a condição seja falsa ou aux = 0, ou aux>0 e o elemento apontado por AUX é menor ou igual ao valor de ATUAL. Em Ambos os casos vale AS.

AE && (C == T) + B2 => AINV
-Pela AE, IND terá que apontar para um elemento no vetor posterior ao primeiro e o elemento apontado por AUX é maior do que o valor de ATUAL. Assim o bloco B2 será executado, o que fará com que o conjunto dos já visitados contenha somente o último elemento do subvetor e o de a visitar contenha o resto do subvetor.

AINV && (C == T) + B2 => AINV
-Para AINV ser válida, B2 deve garantir que um elemento do subvetor seja transferido do conjunto dos a pesquisar para os já pesquisados e AUX aponte para o próximo elemento do subvetor.

AINV && (C == F) => AS
-No último ciclo, teremos três opções ou AUX é igual a zero, ou elemento apontado por AUX é menor ou igual ao valor de ATUAL, ou ambas as condições são verdadeiros. Em qualquer um dos casos a porção do subvetor entre o segundo elemento e o último está ordenado em ordem crescente. Em todos os casos vale AS.

Término: Como a cada ciclo um elemento do conjunto a pesquisar passa pro conjunto de já pesquisados e quantidade de elementos deste conjunto é finita, a repetição termina em um número finito de passos.

B3:

INICIO
ELE[AUX+1] <— ELE[AUX]
=>AI7
AUX <— AUX - 1=>AI8

Argumentação de sequencia:
-AI7
Os elemento posterior ao apontado por AUX possui o mesmo valor do elemento apontado por AUX.
-A18
Aux aponta para o elemento anterior ao apontado anteriormente por AUX.

4)Transforme uma estrutura grafo em auto-verificável
R:
Na imagem separada.

5)Gere quatro deturpações possíveis para a estrutura da questão 4.
R:Deturpações:
a)Trocar o tipo do elemento corrente do grafo.
b)Atribuir NULL ao ponteiro para a cabeça do grafo no vértice corrente.
c)Atribuir lixo ao ponteiro para o elemento corrente.
d) Atribuir lixo ao ponteiro para a cabeça do vértice corrente.

6)Apresente o código do verificador para a estrutura da questão 4.
R:

```
/*imagine que o tipo apontado pelo vértice é char que será representado como o caracter 'c' */
GRA_tpCondRet GRA_Verifica(GRA_tppGrafo graph) {
	if(graph->pVerticeCorrente == LIXO) {
		return GRA_CondRetLixoVerticeCorrente;
	}
	if(graph->pVerticeCorrente->idTipo != 'C') {
		return GRA_CondRetErroTipoVerticeErrado
	}
	if(graph->pVerticeCorrente->pCabeca == NULL) {
		return GRA_CondRetErroPtCabecaNulo;
	}
	if(graph->pVerticeCorrente->pCabeca == LIXO) {
		return GRA_CondRetErroptCabecaLixo;
	}


}
```

7)Distribua controladores de cobertura no verificador e informe o relatório final de contagem executando apenas as deturpações da questão 5.OBS:Deve ser inserido o menor número de controladores necessários para testar completamente o verificador pelo critério caixa aberta.

R:

```
GRA_tpCondRet GRA_Verifica(GRA_tppGrafo graph) {
	if(graph->pVerticeCorrente == LIXO) {
		CNT_Contar("erro-lixoVerticeCorrente");
		return GRA_CondRetVericeCorrenteLixo;
	}  else {
		CNT_Contar("ok-naoHaLixoNoVerticeCorrente");
	}
	if(graph->pVerticeCorrente->idTipo != 'C') {
		CNT_Contar("erro-tipoDoVerticeEstaErrado");
	} else {
		CNT_Contar("ok-tipoDoVerticeEstaCerto");
	}
	if(graph->pVerticeCorrente->pCabeca == NULL) {
		CNT_Contar("erro-ponteiroParaCabecaNoVerticeNulo");
		return GRA_CondRetCabecaNoVerticeNula;
```

```
        } else {
                CNT_Contar("ok-ponteiroParaCabecaNoVerticeNaoNulo");
        }
        if(graph->pVerticeCorrente->pCabeca == LIXO) {
                CNT_Contar("erro-ponteiroParaCabecaNoVerticeLixo");
                return GRA_CondRetCabecaNoVerticeLixo;
        } else {
                CNT_Contar("ok-ponteiroParaCabecaNoVerticeNaoLixo");
        }


}
```

Valor dos contadores: Teríamos que testar algumas separadamente, visto que o programa voaria se testássemos todas juntas. Se executacemos somente a primeira deturpação:

erro-lixoVerticeCorrente /= 0
ok-naoHaLixoNoVerticeCorrente /= 1
erro-tipoDoVerticeEstaErrado /= 1
ok-tipoDoVerticeEstaCerto /= 0
erro-ponteiroParaCabecaNoVerticeNulo /= 0
ok-ponteiroParaCabecaNoVerticeNaoNulo /= 1
erro-ponteiroParaCabecaNoVerticeLixo /= 0
ok-ponteiroParaCabecaNoVerticeNaoLixo /= 1

Se executacemos somente a segunda:
erro-lixoVerticeCorrente /= 0
ok-naoHaLixoNoVerticeCorrente /= 1
erro-tipoDoVerticeEstaErrado /= 0
ok-tipoDoVerticeEstaCerto /= 1
erro-ponteiroParaCabecaNoVerticeNulo /= 1
ok-ponteiroParaCabecaNoVerticeNaoNulo /= 0
erro-ponteiroParaCabecaNoVerticeLixo /= 0
ok-ponteiroParaCabecaNoVerticeNaoLixo /= 1

Se executacemos somento a terceira:
erro-lixoVerticeCorrente /= 1
ok-naoHaLixoNoVerticeCorrente /= 0
erro-tipoDoVerticeEstaErrado /= 0
ok-tipoDoVerticeEstaCerto /= 0
erro-ponteiroParaCabecaNoVerticeNulo /= 0
ok-ponteiroParaCabecaNoVerticeNaoNulo /= 0
erro-ponteiroParaCabecaNoVerticeLixo /= 0
ok-ponteiroParaCabecaNoVerticeNaoLixo /= 0

Se executacemos somente a quarta:
erro-lixoVerticeCorrente /= 0
ok-naoHaLixoNoVerticeCorrente /= 1
erro-tipoDoVerticeEstaErrado /= 0
ok-tipoDoVerticeEstaCerto /= 1
erro-ponteiroParaCabecaNoVerticeNulo /= 0
ok-ponteiroParaCabecaNoVerticeNaoNulo /= 1
erro-ponteiroParaCabecaNoVerticeLixo /= 1
ok-ponteiroParaCabecaNoVerticeNaoLixo /= 0

8)Apresente um exemplo de código em C utilizando trace de evolução.

R: int i = 0,j=9;
if(j + i <10) {
    i = 3;
    j = i*2 + 9;
}
printf(“%d\n”,j);
while(i<9) {
    j++;
    i++;
}
printf(“%d\n”,j);

9)Explique a vantage da instrumentação em relação aos testes convencionais no que diz respeito ao esforço de diagnose.
R: Utilizando a instrumentação, vários erros que são extremamente difíceis de identificar e que estão fortemente relacionados com violações de assertivas estruturais, tornam-se bem mais fáceis de serem percebidos, pois com a instrumentação sempre estamos verificando se a estrutura instrumentada está de acordo com suas assertivas estruturais.

10)Qual é o objetivo de se definir a assertiva invariante de uma repetição?Apresente o código de uma assertiva invariante executável.
R: A assertiva invariante tem como objetivo validar o estado(descritos de estados) da repetição após um número qualquer de iterações. Assim, sabemos que a iteração está progredindo.

Assertiva invariante executável para busca em uma lista duplamente encadeada, que possui a quantidade de elementos da lista em sua cabeça:

```
LIS_PesquisarNaLista(Lis_tppLista pLista) {

#ifdef _DEBUG
int i;
LIS_tppLista pListaJaVistos;
LIS_tppLista pListaAVer;
LIS_tpCondRet retLis;
/*caso a lista não seja vazia, os elementos a ver correspondem ao vetor todo e os já vistos formar um conjunta vazio(no caso uma lista vazia)*/
if(pLista->pElemCorr != NULL) {
   CriaListaVazia(&pListaJaVistos);
   CriaCopiaLista(pLista, &pListaAVer);
   if(pLista->numElem != pListaAVer->numElem) {
         return LIS_CondRetAssertivaInvarianteViolada;
   }
}
if(pLista->pElemCorr != pLista->pOrigem) {
    return LIS_CondRetAssertivaInvarianteViolada;
}
retLis = LIS_ChecaListaVazia(pListaJaVistos);
if(retLis != LIS_CondRetListaEhVazia) {
    return LIS_CondRetAssertivaInvarianteViolada;
}

#endif
```

```
while(pLista->pElemCorr != NULL) {

    retLis = LIS_ComparaValor(pLista->pElemCorr->pValor,buscado);
     if(retLis == LIS_CondRetIguais) {
          return LIS_CondRetAchou;
     }

     pLista->pElemCorr = pLista->pElemCorr->pProx;
     #ifdef _DEBUG
     pListaAVer->pElemCorr = pListaAVer->pElemCorr->pProx;
     LIS_EliminaCorrente(pListaAver);
     LIS_InserirNo(pListaJaVisto,pLista->pElemCorr->pValor);
     #endif
}

#ifdef _DEBUG

retLis = LIS_ChecaListaVazia(pListaAVer);
if(retLis != LIS_CondRetListaEhVazia) {
    return LIS_CondRetAssertivaInvarianteViolada;
}
IrInicio(pLista);
while(pLista->pElemCorr != NULL) {
      retLis= LIS_ComparaValor(pLista->pElemCorr->pValor, pListaJaVistos->pElemCorr->pValor);
      if(retLis != LIS+CondRetIguais) {
           return LIS_CondRetAssertivaInvarianteViolada;
      }
      pListaJaVistos->pElemCorr = pLista->pElemCorr->pProx;
      pLista->pElemCorr = pLista->pElemCorr->pProx;
}


#endif

return LIS_CondRetNaoAchou;

}
```

11)Faça a argumentação de corretude de um esquema de algoritmo de pesquisa em uma árvore binária genérica instanciado para uma árvore de estruturas Nome/Telefone em que o critério de pesquisa é Nome.
R:

```
//Procurar um determinado nome dentro de uma arvore binária chamada de arvo
//arvo contem nome/telefone em seu no
//aux = recebe um ponteiro com o ponteiro da raiz
AE
{
          // Vai para o primeiro elemento
          AI1
          ARV_Ir_Raiz(arvo);
          // aux aponta para o ponteiro
          AI2
          ARV_ObtemNoCorr(arvo, aux)
          A_De_Repetição
          enquanto(aux != NULL)
          {
```

```
                A_De_Seleção
                se(aux->nome == nome)
                        A_de_Sequencia
                        A14
                        retorno ARV_tp_CondRet_Achou
                A_De_Seleção
                se(aux->nome > nome)
                        //Vai para o no da esquerda
                        A_de_Sequencia
                        AI5
                        ARV_Ir_Esq(aux)
                senao
                        //Vai para o no da direita
                        A_de_Sequencia
                        AI6
                        ARV_Ir_Dir(aux)
        }
        AI3
        retorna ARV_tp_ContRet_Nao_Achou
}
AS
```

Argumentação de sequencia
AE: Recebe uma árvore e um nome
AS: aux aponta para o elemento achado ou aux não aponta pra ninguém
AI1: Árvore é inicializada na raiz
AI2: aux aponta para o elemento corrente

Argumentação de repetição
AE: Recebe um no corrente
AS: para a repetição
AE -> AINV
-Pela AE corrente então não é nulo
AE && (C == F) -> AS
-Pela AE corrente é nulo
AE && (C == V) + B2 -> AINV
-Pela AE corrente não é nulo e irá executar o bloco B2, primeira repetição está ok
AINV && (C == V) + B2 -> AINV
-A AINV é valida e B2 executa o garantindo uma próxima repetição
AINV && (C == F) -> AS
-A AINV neste caso saiu correto depois das n iterações
Termino da repetição

Argumentação de seleção
AE && (C == TRUE) + B4 -> AS
-Vale a AE e executará o B4, isso vai resultar na AS
AE && (C == FALSE) -> AS
-Vale a AE e esse teste vai gerar a AS
Argumentação de sequencia
AE:Recebe o nome corrente e o nome procurado
AS: Sai do teste
AI4: Retorna que achou o nome procurado
Argumentação de seleção
AE && (C == TRUE) + B5 -> AS
-Vale a AE e executará o B5, isso vai resultar na AS
AE && (C == FALSE) + B6 -> AS

-Vale a AE e executará o B6, isso vai resultar na AS
Argumentação de sequencia
AE: Recebe o nome corrente e o nome procurado
AS: Sai do teste
AI5: Vai para o filho a esquerda
AI6: Vai para o filho a direita

AI3: retorna que não achou o elemento