Questão 1 (a)

```
//método insere nó no ínico da lista passada por parâmetro
void insereNoInicio(ListaLigada L, No novo){
    novo.prox = L.cabeca.prox;
     L.cabeca = novo;
}

//método cria nova lista que é a inversa da lista passada por parâmetro
ListaLigada inverte(ListaLigada L){
   No itr = L.cabeca.prox;
   No aux;
   ListaLigada nova = new ListaLigada();
   while(itr != null){ //enquanto não chegar no fim da lista
        aux = new No();
        aux.valor = itr.valor;
        insereNoInicio(nova, aux);
        itr = itr.prox;
   }
   return nova;
}
```

Questão 1 (b)

```
//método copia os primeiros i nós da lista ligada passada por parâmetro
ListaLigada copia(ListaLigada L, int i){
   ListaLigada nova = new ListaLigada();
   No ultimo = nova.cabeca; //ultimo sempre aponta para último nó da nova lista
   No aux, itr;
   itr = L.cabeca.prox;
   while(itr != null && i>0){
         aux = new No();
         aux.valor = itr.valor;  //cria novo nó com mesmo valor da lista passada por parâmetro
         ultimo.prox = aux; //insere no fim da nova lista
         ultimo = ultimo.prox;
         itr = itr.prox;
         i--;
   }
   return nova;
}
```

Questâo 2

Definição Recursiva. Seja no um nó da ListaLigada. Se nó é igual a null não imprima nada.
Se nó não é null, imprima primeiro a sublista começando no nó seguinte e depois imprima este nó.

Você pode fazer a chamada de uma lista L da seguinte forma: imprime(L.cabeca.prox)

```
void imprime(No no){
  if(no != null){
     imprime(no.prox); //imprime primeiro a sub-lista começando no próximo nó
     System.out.println(no.valor);
  }
}
```

Questão 3

```
//A parte utilizada do vetor vai de 0 até n. Temos 0<= i <=n, ou seja, i é uma
//posição válida.
void remove(int[] v, int i, int n){
    while(i<n){
       v[i] = v[i+1];
        i++;
    }
    n--; //indica a nova parte util do vetor. n deveria ser da classe
}
```

No pior caso i=0 e o laço é executadao O(n) vezes sendo esta a complexidade do pior caso.
No melhor caso temos i=n e portanto a complexidade é O(1) pois não há execução do laço.

Questão 4

```
boolean repete(int[] v){
    int i,j;
1   for(i=0;i<v.length;i++)
 2     for(j=0; j<v.length; j++)
          if(i!=j && v[i] == v[j])
              return true; //há elemento repetido
    return false;
}
```

No pior caso não há elemento repetido. O laço da linha 2 é executado n vezes para cada vez que o laço da linha 1 é executado. Como o laço da linha 1 é executado n vezes, temos custo total O(n^2) (n ao quadrado).

Um algoritmo melhor seria este abaixo mas com complexidade ainda O(n^2):

```
boolean repete(int[] v){
  int i,j;
  for(i=0;i<v.length;i++)
    for(j=i+1; j<v.length; j++)
        if(v[i] == v[j])
            return true; //há elemento repetido
  return false;
}
```

Um algoritmo ainda melhor ordenaria o vetor v (ou uma cópia deste, caso v não possa ser modificado) usando por exemplo o mergeSort cuja complexidade é O(n log n). Depois o algoritmo checa se existe duas posições seguidas com mesmo valor em O(n). A complexidade do algoritmo seria então O(n log n).

```
boolean repete(int[] v){
  mergeSort(v); //ordena v
  for(int i=0; i< v.length -1; i++)
     if(v[i] == v[i+1] )
        return true;

  return false;
}
```