Questão 1: Na questão 1 é pedido que seja implementado uma função insertionSort mas o que essa função tem que fazer? Essa função ela ordena os elementos na hora da inserção, ou seja, a função escolhe um valor onde ela isso é a chave, desse modo ele vai verificando se essa chave é menor do que o que está atrás dela ou se ela já está no local que é para ficar, assim a função vai verificar todos o valores e reorganizar os necessários até que o vetor esteja organizado. No meu código vou ter dois for aninhados, desse modo o primeiro for ele vai funcionar para disponibilizar a posição dita como a chave, já o segundo for ele vai percorrer o vetor e verificar se aquele valor dado como chave se ele está na sua posição devida, ou seja, desse modo ele vai conseguir reorganizar todo o vetor, porém irá ter a pior complexidade possível que vai ser a $O(n^2)$ pois tem dois for aninhados percorrendo o vetor n vezes, esse é o motivo de ter a pior complexidade possível.

```
37  void insertionSort(vetor *v){
38
39      produto * chave = (produto*) malloc(sizeof(produto));
40
41      int i,j;
42
43      for(i = 1; i < v->n; i++){
44          chave = v->vet[i];
45          for(j = i-1; j >= 0 && chave->codigo < v->vet[j]->codigo; j--){
46              v->vet[j+1] = v->vet[j];
47          }
48          v->vet[j+1] = chave;
49      }
50
51  }
```

Questão 2: Na questão 2 é pedido que seja implementado uma função selectionSort mas o que essa função tem que fazer? bem a função selectionSort ele percorre todo o vetor procurando o espaço do vetor onde se encontra o menor valor e seleciona ele, desse modo quando o menor valor for encontrado ele irá colocar ele o mais a esquerda possível, verificando se na esquerda num a um menor do que ele. Desse modo, o selectionSort reorganiza todo o vetor selecionando os valores e os colocando já no local certo. A minha forma de implementar esse código foi com dois for aninhados, assim o primeiro for irá servir para garantir o vetor seja percorrido e que o menor valor do vetor seja encontrado para assim poder haver a troca de lugares entre os vetores, já o segundo for ele irá percorrer o vetor a partir de onde o primeiro vetor está parado até onde tem valores no vetor, dessa forma vemos que ele sempre irá pegar o vetor menor a partir daquele ponto pois, não tem como pegar um vetor que já foi reorganizado pois o segundo for onde tem a função de pegar o menor valor sempre irá depender do primeiro for então ele sempre irá reorganizar o vetor jogando o menor valor sempre o mais a esquerda possível. Sendo assim a função que fica dentro do segundo for ela tem a funcionalidade de receber as posições dos for aninhados e compara-las para vê se aquele é o menor valor naquele ponto do vetor, assim fazendo

a troca dos valores do vetor e assim reorganizando todo o vetor. Dessa forma essa função tem uma das piores complexidades que um sistema pode ter, que é $O(n^2)$.

```
37  void selectionSort(vetor *v){
38
39      int salvar = 0,aux;
40
41      for(int i=0; i<v->n;i++){
42              salvar = i;
43          for(int j=i+1;j<v->n;j++){
44              if(v->vet[j]->codigo < v->vet[salvar]->codigo){
45                  salvar = j;
46              }
47              aux = v->vet[salvar];
48              v->vet[salvar] = v->vet[i];
49              v->vet[i] = aux;
50          }
51      }
52
53  }
```

Questão 3: Na questão 4 é pedido que seja implementado uma função dubbleSort mas o que essa função tem que fazer? Essa função ela tem que pegar o maior valor e ir fazendo a mudança de lugar dele no vetor até chegar na sua posição reorganizando o vetor em ordem crescente. Desse modo eu implementei um código com dois for aninhados de modo que o primeiro for serve apenas para garantir que o segundo for irá executar todas as vezes necessárias para eu todo o seu vetor seja reorganizado, já o segundo for ele irá percorrer do 0 até a quantidade de vetores -1 pois ele não poderá ir até o final pois a função de comparação compara a posição do segundo for com a posição posterior para saber se tem um valor maior que o outro em posição errada, assim a função que está dentro do segundo for ela irá verificar qual é o maior e ir alternando ele para sua posição adequada, ou seja, dessa forma ele reorganiza esse vetor espaço por espaço um por um. Dessa forma essa função tem uma das piores complexidades que um sistema pode ter, que é $O(n^2)$.

```
37  void dubbleSort(vetor *v){
38
39      int aux;
40
41      for(int i=0; i<v->n;i++){
42          for(int j=0;j<v->n-1;j++){
43              if(v->vet[j]->codigo > v->vet[j+1]->codigo && v->vet[j+1]->codigo != NULL){
44                  aux = v->vet[j+1];
45                  v->vet[j+1] = v->vet[j];
46                  v->vet[j] = aux;
47              }
48          }
49      }
50
51  }
```

Questão 4: Na questão 4 é pedido que seja implementado uma função mergeSort mas o que essa função tem que fazer? Essa função usa o conceito de dividir para conquistar, ou seja, ela divide todo o vetor e quando vai juntar de novo a função vai juntando os valores do vetor ordenadamente, ou seja, essa função irá usar recursão para que ela consiga dividir o vetor, temos que no contigo implementado tem duas funções para merge, a mergeSort e a merge, A função mergeSort divide o vetor e passa para a função merge o vetor separado e nessa função ela vai juntar os valores ordenando-os, ou seja, a função merge já vai deixar o vetor completamente ordenado. Essa função terá complexidade igual a $O(n \log_2 n)$ essa complexidade é devido a forma como ele reorganiza, dividindo para conquistar.

```c
void mergeSort(vetor *v,int esquerda,int direita){

    if(esquerda >= direita){
        return;
    }

    int  meio= esquerda+(direita-esquerda)/2;
    mergeSort(v, esquerda, meio);
    mergeSort(v, meio+1, direita);
    merge(v, esquerda, meio, direita);

}
```

```c
void merge(vetor* v, int esquerda, int meio, int direita){

    int num1 = meio-esquerda+1;
    int num2 = direita-meio;

    int i, j;

    vetor* S= create(num1);
    vetor* T= create(num2);

    for(i = 0; i < num1; i++){
        S->vet[i] = v->vet[esquerda + i];
    }
    for(j = 0; j < num2; j++){
        T->vet[j] = v->vet[meio + 1 + j];
    }

    i = 0;
    j = 0;
    int k = esquerda;

    while(i < num1 && j <num2){
        if(S->vet[i]->codigo <= T->vet[j]->codigo){
            v->vet[k] = S->vet[i];
            i++;
        }
        else{
            v->vet[k] = T->vet[j];
            j++;
        }
        k++;
    }

    while(i < num1){
        v->vet[k] = S->vet[i];
        i++;
        k++;
    }

    while(j < num2){
        v->vet[k] = T->vet[j];
        j++;
        k++;
    }
}
```

Questão 5: Na questão 5 é pedido que seja implementado uma função quickSort mas o que essa função tem que fazer? Essa função ela é bem parecida com a mergeSort, ou seja, ela também usa o conceito de dividir para conquistar, porém ela divide mais a

parte dividida já vai ser organizada, desse modo ele separa o vetor em pedaços e organiza esses pedaços, a função implementada usa recursão para conseguir dividir o vetor em setores e desse modo organizar esse setores com dois whiles, que irão verificar se aquele valor do vetor deveria está onde está, se aquele setor já estiver ordenado como deveria a função passa para o próximo pedaço do vetor até conseguir ordenar o vetor completamente e resultar em um vetor ordenado como era desejado, já a complexidade dessa função será de $O(n^2)$ pois ela no pior caso vai ter que passar por todos os vetores duas vezes para reorganizar os valores.

```
37  void quickSort(vetor *v,int n,int k){
38
39      int i, j;
40
41          produto *x = (produto *)malloc(sizeof(produto));
42          produto *y = (produto *)malloc(sizeof(produto));
43
44          i = n;
45          j = k;
46          x = v->vet[(n + k) / 2];
47
48          while(i <= j){
49              while(v->vet[i]->codigo < x->codigo && i < k){
50                  i++;
51              }
52              while(v->vet[j]->codigo > x->codigo && j > n){
53                  j--;
54              }
55              if(i <= j){
56                  y = v->vet[i];
57                  v->vet[i] = v->vet[j];
58                  v->vet[j] = y;
59                  i++;
60                  j--;
61              }
62          }
63
64          if(j > n){
65              quickSort(v, n, j);
66          }
67          if(i < k){
68              quickSort(v, i, k);
69          }
70
71  }
```

Questão 6: Com base no que foi visto no trabalho, eu considero a função mergeSort a mais eficiente pois sua complexidade é $O(n \log_2 n)$ pois ela está dividindo o vetor para conseguir ordena-lo de forma correta, já as outras complexidades no pior caso ficará $O(n^2)$, ou seja, essa é a pior complexidade existente, pois ela percorre o vetor com no

mínimo duas repetições n vezes, desse modo quando a função merge é executada ela sempre terá a complexidade de $O(n \log_2 n)$ pois sempre ela vai usar o conceito dividir para conquistar, ou seja, quando é dividido na hora de juntar de novo ele já organiza e se já tiver organizado ela só junta o vetor de novo, deixando assim essa função mais eficiente do que os outras.