Q1

```
void insertion(Item v[], int n){
    int j;                                  O(1)
    for(int i = 2; i <= n; i++){            O(n)
        Item x = v[i];                      O(1)*n
        j = i – 1;                          O(1)*n
        v[0] = x;                           O(1)*n
        while(x.compara[v[j] < 0){          O(n)*n  = O(n^2)
            v[j+1] = v[j];                  O(1)*n^2
            j--;                            O(1)*n^2
        }
        v[j+1] = x;                         O(1)*n
    }
}
```

Pior caso: $O(n^2)$

O vetor está em ordenado de forma inversa. Todos os elementos serão deslocados $n – 1$ vezes.

Médio caso: $O(n^2)$

O vetor está ordenado pela metade. Os elementos não ordenados se deslocarão, no máximo $(n – 1) / 2$ vezes.

Melhor caso: $O(n)$

O vetor está completamente ordenado. Nenhum elemento é deslocado.

Q2

Exiba os passos da execução do método de ordenação HeapSort para o seguinte arranjo de dados: {2, 5, 16, 10, 23, 39, 18}. Qual a complexidade do HeapSort para o pior caso?

1º passo: Construir um heap.

```
           2
       /       \
    5            16
   / \          /  \
10     23     39     18
```

2º passo: Converter para max heap. As modificações no heap se refletem no vetor.

```
           2                          2                          39
       /       \                  /       \                   /       \
    5            39           23            39            23             2
   / \          /  \         /  \          /  \          /  \          /  \
10     23     16     18    10     5      16     18     10     5      16     18
```

{2, 5, 39, 10, 23, 16, 18}   {2, 23, 39, 10, 5, 16, 18}   {39, 23, 2, 10, 5, 16, 18}

```
        39
      /    \
   23        18
  / \       / \
10   5    16   2
```
{39, 23, 18, 10, 5, 16, 2}

3º passo: Após construir o max heap, deve-se trocar o primeiro e o último nó, apagar o último nó e construir um novo max heap. Esse passo deve ser repetido até sobrar 1 nó.

Troca primeiro com último
```
        2
      /    \
   23        18
  / \       / \
10   5    16   39
```
{2, 23, 18, 10, 5, 16, 39}

Apaga o último nó do heap
```
        2
      /    \
   23        18
  / \       /
10   5    16
```
{2, 23, 18, 10, 5, 16, 39}

Max heap
```
        23
      /    \
   2         18
  / \       /
10   5    16
```
{23, 2, 18, 10, 5, 16, 39}

Max heap
```
        23
      /    \
   10        18
  / \       /
2    5    16
```
{23, 10, 18, 2, 5, 16, 39}

Troca primeiro com último
```
        16
      /    \
   10        18
  / \       /
 2   5    23
```
{16, 10, 18, 2, 5, 23, 39}

Apaga o último nó do heap
```
        16
      /    \
   10        18
  / \
 2   5
```
{16, 10, 18, 2, 5, 23, 39}

Max heap
```
        18
      /    \
   10        16
  / \
2    5
```
{18, 10, 16, 2, 5, 23, 39}

Troca primeiro com último
```
        5
      /    \
   10        16
  / \
 2   18
```
{5, 10, 16, 2, 18, 23, 39}

Apaga o último nó do heap
```
        5
      /    \
   10        16
  /
2
```
{5, 10, 16, 2, 18, 23, 39}

Max heap
```
        16
      /    \
   10        5
  /
2
```
{16, 10, 5, 2, 18, 23, 39}

Troca primeiro com último
```
        2
      /    \
   10        5
  /
16
```
{2, 10, 5, 16, 18, 23, 39}

Apaga o último nó do heap
```
        2
      /    \
   10        5
```
{2, 10, 5, 16, 18, 23, 39}

Max heap
```
        10
      /    \
    2        5
```
{10, 2, 5, 16, 18, 23, 39}

Troca primeiro com último
```
        5
      /    \
   2         10
```
{5, 2, 10, 16, 18, 23, 39}

Apaga o último nó do heap
```
      5
     /
   2
```
{5, 2, 10, 16, 18, 23, 39}

Troca primeiro com último   Apaga o último nó do heap

```
    2            2
   /
  5
```

{2, 5, 10, 16, 18, 23, 39}   {2, 5, 10, 16, 18, 23, 39}

Pior caso do HeapSort é O(n log n).

Q3

QuickSort faz mais comparações em relação a outros métodos de ordenação. Essas comparações a mais, faz com que mais tempo de CPU seja gasto. Em contrapartida, QuickSort utiliza memória apenas para armazenar endereços das chamadas recursivas, fazendo-o consumir pouca memória.

Segue uma imagem do consumo de tempo e espaço de cada algoritmo de ordenação:

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Q4

a) Sim, pois existem subvetores criados a partir do vetor original e eles são resolvidos separadamente por uma função.

b)

```
se p < r                                O(1)
    então q ← ⌊(p + r)/2⌋               O(1)
    algumMetodoSort(A; p; q)            O(q – p)
    algumMetodoSort(A; q+1; r)          O(r - (q + 1))
    intercala(A; p; q; r)               O(r – p)
```

O(r – p), que seria O(n).

<u>Q5</u>

```
void acharMediana(Vetor v[]) {
    int tamanhoVetor = v.length();
    float mediana;

    bucketSort(v);

    if(tamanhoVetor % 2 == 0) { // Com o vetor de tamanho par
        int posicao1, posicao2;

        posicao1 = tamanhoVetor / 2 – 1;
        posicao2 = tamanhoVetor / 2;

        mediana = (v[posicao1] + v[posicao2]) / 2;

    }else { // Com o vetor de tamanho ímpar
        int posicaoVetor;

        posicaoVetor = Math.floor(tamanhoVetor / 2);
        mediana = v[posicaoVetor];
    }
}
```

<u>Q6</u>

Radix Sort. Ele ordena os números por significatividade de bits, começando do bit menos significativo para o mais significativo. Como, somente os 4 bits menos significativos foram alterados, então limitaria o algoritmo para ordenar somente os 4 últimos bits de cada número. Seria a unidade, dezena, centena e milhar de cada número.

<u>Q7</u>

e

<u>Q8</u>

b

<u>Q9</u>

d

<u>Q10</u>

b

<u>Q11</u>

c

<u>Q12</u>

b

<u>Q13</u>

c