

Q1

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

$O(1)$
 $O(n)$
 $O(1)*n$
 $O(1)*n$
 $O(1)*n$
 $O(n)*n = O(n^2)$
 $O(1)*n^2$
 $O(1)*n^2$
 $O(1)*n$

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.

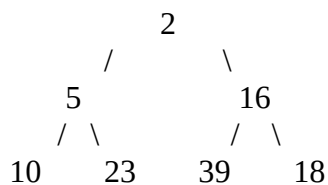
Melhor caso: $O(n)$

O vetor está completamente ordenado. Nenhum elemento é deslocado.

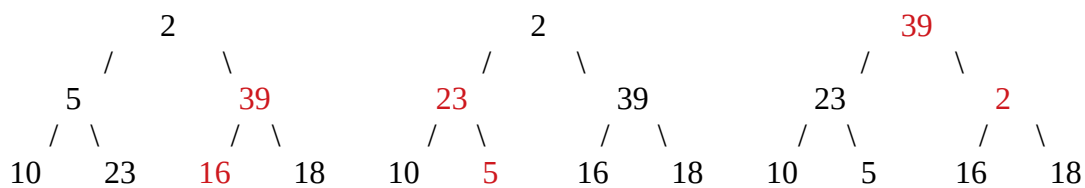
Q2

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?

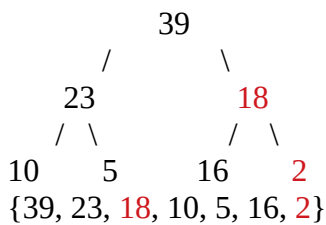
1º passo: Construir um heap.



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

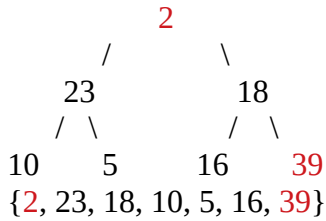


{2, 5, 39, 10, 23, 16, 18} {2, 23, 39, 10, 5, 16, 18} {39, 23, 2, 10, 5, 16, 18}

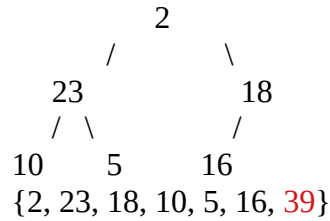


3º passo: Após construir o max heap, deve-se trocar o primeiro e o último nó, apagar o último nó e construir um novo max heap. Esse passo deve ser repetido até sobrar 1 nó.

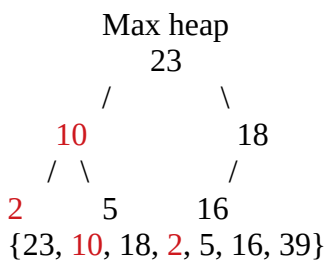
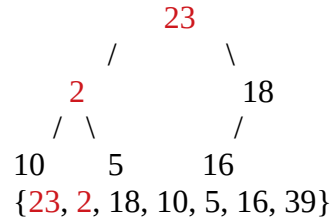
Troca primeiro com último



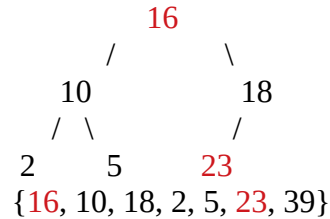
Apaga o último nó do heap



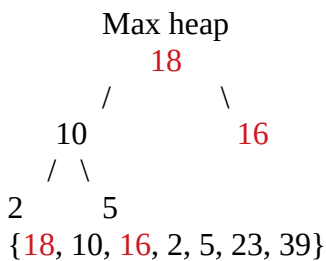
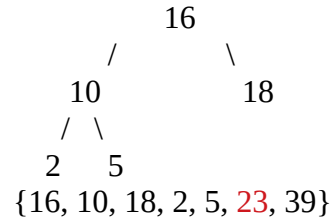
Max heap



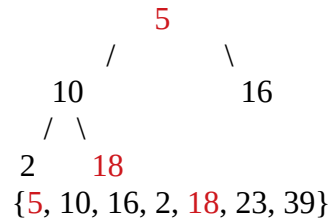
Troca primeiro com último



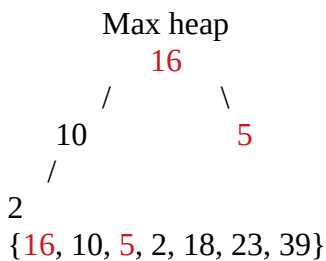
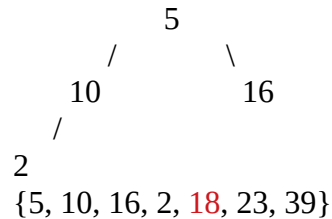
Apaga o último nó do heap



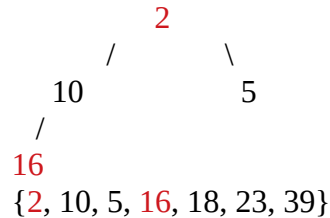
Troca primeiro com último



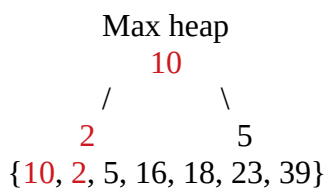
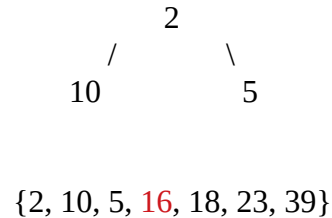
Apaga o último nó do heap



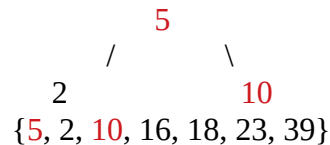
Troca primeiro com último



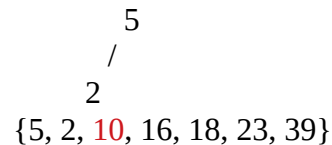
Apaga o último nó do heap



Troca primeiro com último



Apaga o último nó do heap



Troca primeiro com último Apaga o último nó do heap

2
/
5

2

{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	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n^2)$	$O(\log(n))$
Mergesort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Timsort	$\Omega(n)$	$\theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Heapsort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n \log(n))$	$O(1)$
Bubble Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$\Omega(n^2)$	$\theta(n^2)$	$O(n^2)$	$O(1)$
Tree Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n^2)$	$O(n)$
Shell Sort	$\Omega(n \log(n))$	$\theta(n(\log(n))^2)$	$O(n(\log(n))^2)$	$O(1)$
Bucket Sort	$\Omega(n+k)$	$\theta(n+k)$	$O(n^2)$	$O(n)$
Radix Sort	$\Omega(nk)$	$\theta(nk)$	$O(nk)$	$O(n+k)$
Counting Sort	$\Omega(n+k)$	$\theta(n+k)$	$O(n+k)$	$O(k)$
Cubesort	$\Omega(n)$	$\theta(n \log(n))$	$O(n \log(n))$	$O(n)$

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 $q \leftarrow \lfloor (p + r)/2 \rfloor$	$O(1)$
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 $O(n)$.

Q5

```
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];
    }
}
```

Q6

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.

Q7

e

Q8

b

Q9

d

Q10

b

Q11

c

Q12

b

Q13

c