# Programação II

## Selection Sort

A cada passo, procura o menor valor do array e o coloca na primeira posição do array. Descarta-se a primeira posição do array e repete-se o processo para segunda posição. Isso é feito para todas as posições do Array. O método selection sort é ineficiente para grande volume de dados.

**Implementação Selection Sort**

```
void metodoSelecao (int vet[TAM]){
    int menor, aux;
    for (int i = 0; i<TAM; i++){
        menor = i;
        for (int j = i+1; j < TAM; j++){
            if (vet[menor]>vet[j])
            menor = j;
        }
        if (i!=menor){
            aux = vet[i];
            vet[i] = vet[menor];
            vet[menor] =aux;
        }
    }
}
```

**Execução Selection Sort**

Vetor a ordenar: **{29, 26, 20, 15, 16, 24,10}**

Primeiro passo é testar até encontrar o menor número, em seguida o valor da primeira posição é armazenado em uma variável auxiliar e o menor valor será armazenado na primeira posição, após a troca, o valor que antes estava na primeira posição irá para a posição onde o menor valor foi encontrado.

**aux = 29**
**{29, 26, 20, 15, 16, 24, 10}**
**{10, 26, 20, 15, 16, 24, 29}**
**aux = 26**
**{10, 26, 20, 15, 16, 24, 29}**
**{10, 15, 20, 26, 16, 24, 29}**
**aux = 20**
**{10, 15, 20, 26, 16, 24, 29}**
**{10, 15, 16, 26, 20, 24, 29}**
**aux = 26**
**{10, 15, 16, 26, 20, 24, 29}**
**{10, 15, 16, 20, 26, 24, 29}**
**aux = 26**
**{10, 15, 16, 20, 26, 24, 29}**
**{10, 15, 16, 20, 24, 26, 29}**
**aux = 26**
**{10, 15, 16, 20, 24, 26, 29}**
**26<29**

## Bubble Sort

O Bubble Sort é um algoritmo de ordenação simples e popular usado para ordenar uma lista de elementos. Ele funciona comparando elementos adjacentes e trocando de posição aqueles que estão fora de ordem, percorrendo a lista várias vezes até que todos os elementos estejam em ordem crescente ou decrescente. O Bubble Sort é fácil de entender e implementar, mas

sua eficiência é baixa, especialmente em listas grandes, devido ao alto número de comparações e trocas que podem ser necessárias.

**Implementação Bubble Sort**

```
void metodoBolha (int vet[TAM]){
    int aux;
    for (int i = 0; i < TAM; i++){
        for (int j = 0; j < TAM-i-1; j++){
            if (vet[j] > vet [j+1]){
                aux = vet[j];
                vet[j] = vet[j+1];
                vet[j+1] = aux;
            }
        }
    }
```

**Execução Bubble Sort**

Vetor a ordenar: **{29, 26, 20, 15, 16, 24, 10}**

**26 < 29**
**{26, 29, 20, 15, 16, 24, 10}**
**20 < 29**
**{26, 20, 29, 15, 16, 24, 10}**
**15 < 29**
**{26, 20, 15, 29, 16, 24, 10}**
**16 < 29**
**{26, 20, 15, 16, 29, 24, 10}**
**24 < 29**
**{26, 20, 15, 16, 24, 29, 10}**
**10 < 29**
**{26, 20, 15, 16, 24, 10, 29}**

Dessa forma, sabemos que o maior elemento já está posicionado na última posição do vetor, não sendo necessário compará-lo nos próximos loops.

**20 < 26**
**{20, 26, 15, 16, 24, 10, 29}**
**15 < 26**
**{20, 15, 26, 16, 24, 10, 29}**
**16 < 26**
**{20, 15, 16, 26, 24, 10, 29}**
**24 < 26**
**{20, 15, 16, 24, 26, 10, 29}**
**10 < 26**
**{20, 15, 16, 24, 10, 26, 29}**

Nesse ponto o penúltimo elemento já está na sua posição correta, não sendo necessário compará-lo.

**15 < 20**
**{15, 20, 16, 24, 10, 26, 29}**
**16 < 20**
**{15, 16, 20, 24, 10, 26, 29}**
**20 < 24 não troca**
**comparado com próximo 24 < 10**
**{15, 16, 20, 10, 24, 26, 29}**

Nos próximos loops não precisaremos mais comparar as três últimas posições

**20 > 10**
**{15, 16, 10, 20, 24, 26, 29}**

Elemento 20 foi posicionado em sua posição correta

**16 > 10**

# Programação II

**{15, 10, 16, 20, 24, 26, 29}**

Elemento 16 foi posicionado em sua posição correta

**15 > 10**

**{10, 15, 16, 20, 24, 26, 29}**

Menor elemento está posicionado na primeira posição e vetor está ordenado de forma crescente.

## Insertion Sort

O método de inserção ordena o vetor utilizando a lógica de um baralho, onde o jogador vai posicionando a carta que tira da pilha do morto de acordo com as cartas que já possui em mãos, com comparações sucessivas.

### Implementação Insertion Sort

```
void metodoInsercaoDireta (int vet[TAM]){
    int aux, pos;

    for (int i = 1; i < TAM; i++){
        aux = vet[i];
        pos = i;
        while (pos > 0 && vet[pos-1]>aux){
            vet[pos] = vet[pos-1];
            pos--;
        }
        vet[pos] = aux;
    }
}
```

### Execução Insertion Sort

Vetor a ordenar: **{29, 26, 20, 15, 16, 24, 10}**

Inicialmente o laço externo iniciará a partir do segundo elemento, ou seja i = 1

**{29, 26, 20, 15, 16, 24, 10}**

Como o 29 > 26, os elementos serão trocados de posição, com uso de variável auxiliar

**{26, 29, [20] | 15, 16, 24, 10}**

O próximo elemento a ser tirado da pilha para comparação é 20, 20 < 29, então será copiado para posição do 29 e 20 ocupará a posição do 20, em seguida feita nova comparação, 20 < 26, portanto, 26 ocupará a posição do 20 que agora está na posição 1 do vetor

**{20, 26, 29, [15] | 16, 24, 10}**

O mesmo se repetirá com o 15, 29 > 15, portanto 29 assume a posição do 29 e 15 assume posição do 29. Novo teste 26 > 15, portanto a mesma troca ocorre com os valores, 20 > 15, com é a última comparação, o número 15 assume a primeira posição no vetor. O estado do vetor nesse passo é a seguinte:

**{15, 20, 26, 29, [16] | 24, 10}**

Após sucessivas comparações validando que o 16 é menor que os valores, 29, 26 e 20 e realizando as trocas, verificado que 15 < 16

**{15, 16, 20, 26, 29, [24] | 10}**

**{15, 16, 20, 24, 26, 29, [10]}**

**{10, 15, 16, 20, 24, 26, 29}**

# Programação II

Repetindo os mesmos passos elemento por elemento, chegamos ao final do vetor ordenado.

## Shell Sort

O método consiste em dividir o vetor em subvetores com base em um determinado intervalo h, e ordenar cada subvetor com base em um algoritmo de inserção. Em seguida, o tamanho do intervalo é reduzido pela metade e o processo é repetido até que o intervalo seja reduzido a 1, momento em que o vetor estará completamente ordenado.

### Implementação Shell Sort

```
void metodoShellSort (int vet[TAM]){
int intervalo, i, j, aux;
    for (intervalo = TAM/2; intervalo > 0; intervalo /= 2) {
        for (i = intervalo; i < TAM; i++) {
        aux = vet[i];
        for (j = i; j >= intervalo && vet[j-intervalo] > aux;
        j -= intervalo) {
        vet[j] = vet[j-intervalo];
}
        vet[j] = aux;
    }
    }
}
```

### Execução Shell Sort

Vetor a ordenar: **{29, 26, 20, 15, 16, 24, 10}**

Definido intervalo inicial como metade do tamanho do vetor: intervalo = TAM/2 = 7/2 = 3

1° Passo: Gera subvetor a partir do elemento da posição 0 {29, 15, 10}

2° Passo: Troca de posições após sucessivas comparações a fim de ordenar subvetor {15, 29, 10} -> { 15, 10, 29} -> {10, 15, 29}

3° Passo: Gera subvetor a partir do elemento da posição 1 {26, 16}

4° Passo: Troca de posições após sucessivas comparações a fim de ordenar subvetor {16, 26}

5° Passo: Gera subvetor a partir do elemento da posição 1 {20, 24}

6° Passo: Feito comparações e verificado subvetor já ordenado, não sendo necessário efetuar trocas {20, 24}

Definido intervalo: intervalo = 3/2 = 1
A partir desse ponto será aplicado inserção

7° Passo: Verificado que 16>10, não sendo necessário a troca. Gera subvetor a partir do elemento da posição 2 {10, 16, 20, 15, 26, 24, 29}

8° Passo: 20 > 15, troca de posição {10, 16, 15, 20, 26, 24, 29}

9° Passo: 16 > 15 e 15 > 10, troca de posição {10, 15, 16, 20, 26, 24, 29}

# Programação II

10° Passo: 26 > 20, não troca de posição {10, 15, 16, 20, 26, 24, 29}

11° Passo: 26 > 24, troca de posição {10, 15, 16, 20, 24, 26, 29}

12° Passo: 29 > 26, não troca de posição {10, 15, 16, 20, 24, 26, 29}

Ordenação finalizada: {29, 26, 20, 15, 16, 24, 10}

## Heap Sort

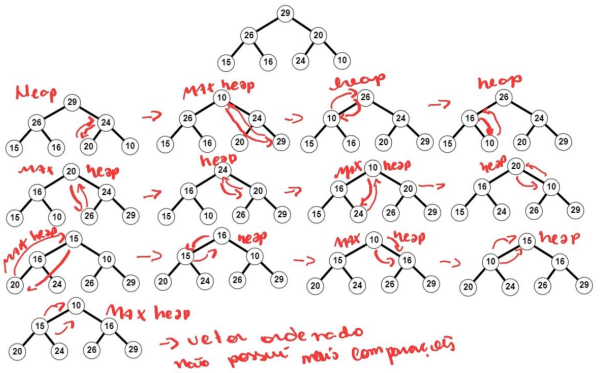
O método HeapSort ordena o vetor utilizando dois passos simples. Fase 1: Construção do Heap
Fase 2: seleção dos elementos na ordem desejada (geralmente de forma crescente).

### Implementação Heap Sort

```c
void metodoHeapSort (int vet[TAM]){
    int tam = TAM;
   int i = tam / 2, pai, filho, t;
    while (true) {
        if (i > 0) {
            i--;
            t = vet[i];
        } else {
            tam--;
            if (tam <= 0) return;
            t = vet[tam];
            vet[tam] = vet[0];
        }
        pai = i;
        filho = i * 2 + 1;
        while (filho < tam) {
            if ((filho + 1 < tam) && (vet[filho + 1] > vet[filho]))
                filho++;
            if (vet[filho] > t) {
                vet[pai] = vet[filho];
                pai = filho;
                filho = pai * 2 + 1;
            } else {
                break;
            }
        }
        vet[pai] = t;
    }
}
```

### Execução Heap Sort

Vetor a ordenar: **{29, 26, 20, 15, 16, 24, 10}**




1° Passo: HEAP - subárvore direita, filho > pai, realizado troca entre 20 e 24.
**{29, 26, 24, 15, 16, 20, 10}**

2° Passo: MAX HEAP - o nó pai é maior que qualquer um dos filhos, portanto é realizada a troca com elemento da última posição. Valor 29 chegou a sua posição correta e não será mais movido.
**{ 10, 26, 24, 15, 16, 20, 29}**

3° Passo: HEAP com maior elemento 26 troca com 10 pois é o maior filho **{26, 10, 24, 15, 16, 20, 29}**

4° Passo: HEAP na subárvore a esquerda, alterado 16 com 10 para que nó pai seja maior que seus filhos. **{26, 16, 24, 15, 10, 20, 29}**

5° Passo:MAX HEAP - Nó pai é maior que qualquer um dos seus filhos, portando é

# Programação II

realizado troca com penúltima posição do vetor. **{20, 16, 24, 15, 10, 26, 29}**

6° Passo: HEAP aplicado, trocado nó pai com filho a direita, 24 > 20 **{24, 16, 20, 15, 10, 26, 29}**

7° Passo: MAX HEAP - Nó pai maior que qualquer um dos seus filhos, realizado troca com elemento da última posição não ordenada. **{10, 16, 20, 15, 24, 26, 29}**

8° Passo: HEAP - Nó pai menor que seus filhos, alterado de posição com maior filho à direita **{20, 16, 10, 15, 24, 26, 29}**

9° Passo: MAX HEAP - Nó pai maior que qualquer um dos seus filhos, realizado troca com elemento da última posição não ordenada. **{15, 16, 10, 20, 24, 26, 29}**

10° Passo: HEAP - Nó pai menor que seus filhos, alterado de posição com maior filho à esquerda **{16, 15, 10, 20, 24, 26, 29}**

11° Passo: MAX HEAP - Nó pai maior que qualquer um dos seus filhos, realizado troca com elemento da última posição não ordenada. **{10, 15, 16, 20, 24, 26, 29}**

12° Passo: HEAP - Nó pai menor que seus filhos, alterado de posição com maior filho à esquerda **{15, 10, 16, 20, 24, 26, 29}**

13° Passo: MAX HEAP - Nó pai maior que qualquer um dos seus filhos, realizado troca com elemento da última posição não ordenada, não tendo mais elementos para comparação, vetor já está ordenado :D
**{10, 15, 16, 20, 24, 26, 29}**

# Merge Sort

O Merge Sort é um algoritmo de ordenação eficiente que segue a abordagem "dividir para conquistar". Ele divide uma lista desordenada em n sublistas, cada uma contendo um elemento, e depois repete o processo até que todas as sublistas tenham apenas um elemento. Então, ele as junta novamente em ordem crescente, comparando os elementos das sublistas. O processo de "mesclagem" é realizado recursivamente, até que a lista inteira esteja ordenada.

**Implementação Heap Sort**

```
void metodoMergeSort(int vet[], int inicio, int fim) {
    if (inicio < fim) {
        int meio = (inicio + fim) / 2;
        metodoMergeSort(vet, inicio, meio);
        metodoMergeSort(vet, meio + 1, fim);
        mesclar(vet, inicio, meio, fim);
    }
}
```

# Programação II

```c
void mesclar(int vet[], int inicio, int meio, int fim) {
    int tam = fim - inicio + 1;
    int *vet_aux = (int *) malloc(tam * sizeof(int));
    int i = inicio, j = meio + 1, k = 0;

    for (k = 0; k < tam; k++) {
        if (i > meio) {
            vet_aux[k] = vet[j++];
        } else if (j > fim) {
            vet_aux[k] = vet[i++];
        } else if (vet[i] < vet[j]) {
            vet_aux[k] = vet[i++];
        } else {
            vet_aux[k] = vet[j++];
        }
    }

    for (k = 0; k < tam; k++) {
        vet[inicio + k] = vet_aux[k];
    }

    free(vet_aux);
}
```

**Execução Merge Sort**

Vetor a ordenar: **{29, 26, 20, 15, 16, 24, 10}**
1° Passo: Dividir o vetor em subvetores menores ao meio.
**{29, 26, 20, 15} {16, 24, 10}**

2° Passo: Dividir novamente o subvetor ao meio:
**{29, 26} {20, 15}**

3° Passo: Dividir novamente os subvetores ao meio
**{29} {26}**

4° Passo: Uma matriz de comprimento 1 não pode ser dividida, realizar mesclagem em ordem
**{26, 29}**

5° Passo: Selecionar subvetor à direita
**{20, 15}**

6° Passo: Dividir novamente os subvetores ao meio
**{20} {15}**

7° Passo: Uma matriz de comprimento 1 não pode ser dividida, realizar mesclagem em ordem
**{15, 20}**

8ª Passo: Comparar e selecionar o menor valor na primeira posição dos dois subvetores, e então inserir subvetor primeiro e logo em seguida o subvetor de maior peso (à esquerda)
**{15, 20, 26, 29}**

9° Passo: Selecione o subvetor a direita
**{16, 24, 10}**

10° Passo: Divida-o. **{16, 24} {10}**

11° Passo: Dividido novamente **{16} {24}**

12° Passo: Como **16 < 24**, os elementos são inseridos de forma **{16, 24}**

13° Passo: Selecionado próximo subvetor {10}, não é necessário mais divisões pois o tamanho já é mínimo

14° Passo: Comparado e selecionado primeiro elemento de cada subvetor, reinserido o menor dentre eles **{10}**

# Programação II

15° Passo: Reinserido subvetor restante **{10, 16, 24}**

16° Passo: Comparado e selecionado primeiro elemento de cada subvetor, reinserido o menor dentre eles **{10}**

17° Passo: Comparado e selecionado primeiro elemento de cada subvetor, reinserido o menor dentre eles {15} na próxima posição disponível **{10, 15}**

18° Passo: Comparado e selecionado primeiro elemento de cada subvetor, reinserido o menor dentre eles **{16}** na próxima posição disponível **{10, 15, 16}**

19° Passo: Comparado e selecionado primeiro elemento de cada subvetor, reinserido o menor dentre eles **{20}** na próxima posição disponível **{10, 15, 16, 20}**

20° Passo: Comparado e selecionado primeiro elemento de cada subvetor, reinserido o menor dentre eles **{24}** na próxima posição disponível **{10, 15, 16, 20, 24}**

21° Passo: Como subvetor à direita ficou vazio, é inserido todos os elementos do vetor restante sem necessário mais comparações pois já está ordenado. **{10, 15, 16, 20, 24, 26}**

22° Passo: Como subvetor à direita ficou vazio, é inserido todos os elementos do vetor restante sem necessário mais comparações pois já está ordenado. **{10, 15, 16, 20, 24, 26, 29}**

23° Passo: Vetor ordenado: **{10, 15, 16, 20, 24, 26, 29}**

# Quick Sort

O Quicksort é um algoritmo de ordenação muito utilizado por sua eficiência, sendo capaz de ordenar grandes quantidades de dados em pouco tempo. Ele utiliza a técnica de dividir para conquistar, que consiste em dividir o problema em subproblemas menores, resolvê-los e combiná-los para obter a solução final.

O algoritmo funciona da seguinte forma: escolhe-se um elemento como pivô, que é usado para particionar o vetor em duas partes, uma com elementos menores que o pivô e outra com elementos maiores. Essa etapa é repetida recursivamente em cada subvetor até que todos estejam ordenados. O pivô pode ser escolhido de diferentes formas, como o primeiro elemento do vetor, o último ou um elemento aleatório.

**Implementação Heap Sort**

# Programação II

```c
void metodoQuickSort (int vet[], int inicio, int fim) {
    int aux, i, j, pivo;
    i = inicio; j = fim;
    pivo = vet[(i + j) / 2];
    do {
        while (vet[i] < pivo) i++;
        while (vet[j] > pivo) j--;
        if (i <= j) {
            aux = vet[i];
            vet[i] = vet[j];
            vet[j] = aux;
            i++;
            j--;
        }
    } while (i <= j);
    if (j > inicio) metodoQuickSort(vet, inicio, j);
    if (i < fim) metodoQuickSort(vet, i, fim);
}

void startaQuick(int vet [], int tam) {
    metodoQuickSort (vet, 0, tam - 1);
}
```

**Execução Quick Sort**

Vetor a ordenar: **{29, 26, 20, 15, 16, 24, 10}**

1° Passo: Selecionado posição central como pivô **{29, 26, 20, [15], 16, 24, 10}**

2° Passo: Pivô trocado de posição com elemento da última posição **{29, 26, 20, 10, 16, 24, 15}**

3° Passo: Particionar subvetor **{*29, 26, 20, 10, 16, *24, 15}**

4° Passo: Movido o limite esquerdo para direito até atingir um valor maior ou igual ao pivô **{*29, 26, 20, 10, 16, *24, 15}**

5° Passo: Movido o limite direito para esquerda até cruzar o limite esquerdo ou encontrar um valor menor que o pivô **{*29, 26, 20, 10, *16, 24, 15}**

6° Passo: Movido a esquerda **{*29, 26, 20, *10, 16, 24, 15}**

7° Passo: Realizado troca de valores **{*10, 26, 20, *29, 16, 24, 15}**

8° Passo: Movido à direita **{10, *26, 20, *29, 16, 24, 15}**

9° Passo: Movido o limite direito para esquerda até cruzar o limite esquerdo ou encontrar um valor menor que o pivô **{10, *26, 20, *29, 16, 24, 15}**

10° Passo: Movido à esquerda **{10, *26, *20, 29, 16, 24, 15}**

11° Passo: Movido à esquerda **{10, *26, 20, 29, 16, 24, 15}**

12° Passo: Movido à esquerda, limites se cruzaram, isso signifca que todos os elementos à esquerda do limite esquerdo são menores que o pivô e todos os elementos à direita são maiores ou iguais ao pivô. **{*10, *26, 20, 29, 16, 24, 15}**

13° Passo: Movido pivô para posição final **{10, 15, 20, 29, 16, 24, 26}**

14° Passo: Chamado novamente quicksort na subvetor à direita **{10, 15, 20, 29, 16, 24, 26}**

# Programação II

15° Passo: Novo pivô definido como 16 **{10, 15, 20, 29, [16], 24, 26}**

16° Passo: Movido pivô para o final **{10, 15, 20, 29, 26, 24, 16}**

17° Passo: Particionar subvetor **{10, 15, 20*, 29, 26, 24*, 16}**

18° Passo: Movido o limite direito para esquerda até cruzar o limite esquerdo ou encontrar um valor menor que o pivô **{10, 15, 20*, 29, 26, 24*, 16}**

19° Passo: Movido à esquerda **{10, 15, 20*, 29, *26, 24, 16}**

20° Passo: Movido à esquerda **{10, 15, 20*, *29, 26, 24, 16}**

21° Passo: Movido à esquerda **{10, 15, 20*, 29, 26, 24, 16}**

22° Passo: Movido à esquerda, limites se cruzaram, isso signifca que todos os elementos à esquerda do limite esquerdo são menores que o pivô e todos os elementos à direita são maiores ou iguais ao pivô.
**{10, 15, 20*, 29, 26, 24, 16}**

23° Passo: Movido pivô para posição final
**{10, 15, 16, 29, 26, 24, 20}**

24° Passo: Chamado subvetor a sublista à direita e selecionado 26 como pivô
**{10, 15, 16, 29, [26], 24, 20}**

25° Passo: Movido pivô para o final
**{10, 15, 16, 29, 20, 24, 26}**

26° Passo: Particionar subvetor
**{10, 15, 16, *29, 20, *24, 26}**

27° Passo: Movido o limite esquerdo para direito até atingir um valor maior ou igual ao pivô **{10, 15, 16, *29, 20, *24, 26}**

28° Passo: Movido o limite direito para esquerda até cruzar o limite esquerdo ou encontrar um valor menor que o pivô **{10, 15, 16, *29, 20, *24, 26}**

29° Passo: Trocado valores **{10, 15, 16, *24, 20, *29, 26}**

30° Passo: Movido o limite esquerdo para direito até atingir um valor maior ou igual ao pivô **{10, 15, 16, *24, 20, *29, 26}**

31° Passo: Movido para direita **{10, 15, 16, 24, *20, *29, 26}**

32° Passo: Movido para direita **{10, 15, 16, 24, 20, *29, 26}**

33° Passo: Movido o limite direito para esquerda até cruzar o limite esquerdo ou

encontrar um valor menor que o pivô **{10, 15, 16, 24, 20, *29, 26}**

34° Passo: Movido à esquerda **{10, 15, 16, 24, *20, *29, 26}**

35° Passo: Movido à esquerda, limites se cruzaram, isso signifca que todos os elementos à esquerda do limite esquerdo são menores que o pivô e todos os elementos à direita são maiores ou iguais ao pivô.
**{10, 15, 16, 24, *20, *29, 26}**

36° Passo: Movido pivô para posição final **{10, 15, 16, 24, 20, 26, 29}**

37° Passo: Chamado novamente quicksort na subvetor à esquerda e selecionado 24 como pivô **{10, 15, 16, [24], 20, 26, 29}**

38° Passo: Movido pivô para final **{10, 15, 16, 20, 24, 26, 29}**

39° Passo: Particionar subvetor **{10, 15, 16, *20, 24, 26, 29}**

40° Passo: Movido à direita **{10, 15, 16, *20, *24, 26, 29}**

41° Passo: Movido o limite direito para esquerda até cruzar o limite esquerdo ou encontrar um valor menor que o pivô **{10, 15, 16, *20, *24, 26, 29}**

42° Passo: Movido à direita, limites se cruzaram, isso signifca que todos os elementos à esquerda do limite esquerdo são menores que o pivô e todos os elementos à direita são maiores ou iguais ao pivô. **{10, 15, 16, *20, *24, 26, 29}**

43° Movido pivô para final (sua posição atual) **{10, 15, 16, 20, 24, 26, 29}**

44° Subvetor à esquerda e sublista à direita ordenadas **{10, 15, 16, 20, 24, 26, 29}**