



Mestrado Profissional em Matemática, Estatística e Computação Aplicado à Indústria
(MECAI)

Trabalho de MAI5001 - Introdução a Ciências da Computação

Análise e Desempenho dos Algoritmos de Ordenação

Prof.Dr.: Adenilso Simões

Aluno: João Carlos Batista
NºUSP: 6792197

USP – São Carlos 05/05/2017

Sumário

LISTA DE FIGURAS.....	III
LISTA DE TABELAS.....	IV
CAPÍTULO 1: BUBBLESORT.....	1
1.1.DESCRICÃO DO ALGORITMO.....	1
1.1.1 CARACTERÍSTICAS.....	1
1.1.2 CODIFICAÇÃO EM C#.....	1
1.1.3 COMPLEXIDADE DE TEMPO E DE ESPAÇO.....	2
1.2 MELHOR CASO.....	2
1.3 PIOR CASO.....	3
CAPÍTULO 2: INSERÇÃO	5
2.1. DESCRICÃO DO ALGORITMO.....	5
2.1.1. CARACTERÍSTICAS.....	5
2.1.2. CODIFICAÇÃO EM C#.....	5
2.1.5 COMPLEXIDADE DE TEMPO E DE ESPAÇO.....	6
2.2. MELHOR CASO.....	6
2.3. PIOR CASO.....	8
CAPÍTULO 3: SELEÇÃO	9
3.1. DESCRICÃO DO ALGORITMO.....	9
3.1.1 CARACTERÍSTICAS.....	9
3.1.2 CODIFICAÇÃO EM C#.....	10
3.1.3 COMPLEXIDADE DE TEMPO E DE ESPAÇO.....	10
3.2. MELHOR CASO.....	10
3.3. PIOR CASO.....	12
CAPÍTULO 4: QUICKSORT	14
4.1. DESCRICÃO DO ALGORITMO.....	14
4.1.1 CARACTERÍSTICAS.....	14
4.1.2 CODIFICAÇÃO EM C#.....	14
4.1.3 COMPLEXIDADE DE TEMPO E DE ESPAÇO.....	15

4.2. MELHOR CASO.....	15
4.3. PIOR CASO.....	16
CAPÍTULO 5: HEAPSORT	21
5.1. DESCRIÇÃO DO ALGORITMO	21
5.1.1 CARACTERÍSTICAS.....	21
5.1.2 CODIFICAÇÃO EM C#	21
5.1.2 COMPLEXIDADE DE TEMPO E DE ESPAÇO.....	22
5.2. MELHOR CASO.....	22
5.3. PIOR CASO.....	25
CAPÍTULO 6: MERGESORT	27
6.1. DESCRIÇÃO DO ALGORITMO.....	27
6.1.1 CARACTERÍSTICAS.....	27
6.1.2 CODIFICAÇÃO EM C#	27
6.1.2 COMPLEXIDADE DE TEMPO E DE ESPAÇO.....	28
6.2. MELHOR CASO.....	28
6.3. PIOR CASO.....	29
CAPÍTULO 7: TESTES E ANALISE DE DESEMPENHO	33
REFERÊNCIAS	36
ANEXO: PROJETO EM C#	36

Lista de Figuras

FIGURA 1: COMPARAÇÃO DE DESEMPENHO DOS ALGORITMOS DE ORDENAÇÃO	34
--	----

Lista de Tabelas

TABELA 1 COMPLEXIDADE DE TEMPO E DE ESPAÇO. ALGORITMO BUBBLESORT.....	2
TABELA 2 COMPLEXIDADE DE TEMPO E DE ESPAÇO. ALGORITMO INSERÇÃO	6
TABELA 3 COMPLEXIDADE DE TEMPO E DE ESPAÇO. ALGORITMO SELEÇÃO	10
TABELA 4 COMPLEXIDADE DE TEMPO E DE ESPAÇO. ALGORITMO QUICKSORT	15
TABELA 5 COMPLEXIDADE DE TEMPO E DE ESPAÇO. ALGORITMO HEAPSORT	22
TABELA 6 COMPLEXIDADE DE TEMPO E DE ESPAÇO. ALGORITMO MERGESORT	28
TABELA 7 TABELA DE TESTE	33

CAPÍTULO 1: BUBBLESORT

1.1. Descrição do algoritmo

O bubblesort é conhecido por fazer ordenação flutuante ou aplica o método da “bolha”. A lógica de ordenação dos números em um vetor procede em percorrer um vetor com n elementos do início até o final várias vezes, sendo assim a cada comparação verifica-se se o elemento anterior é maior ou menor que o elemento seguinte. A cada comparação verifica se os elementos estão fora de ordem, caso esteja ocorre a troca de posições dos elementos, deixando os elementos maiores à direita. Esse processo ocorre até que o vetor esteja totalmente ordenado .

1.1.1 Características

- É fácil de implementar, tem uma alta complexidade e suas comparações ocorre entre posições adjacentes do vetor.
- É um algoritmo estável.
- É recomendado o usar em pequena quantidade de dados.

1.1.2 Codificação em C#

```
static public void bubblesort(int[] vector)
{
    int temp = 0; // complexidade de espaço
    for (int i = 0; i < vector.Length; i++) // n
    {
        for (int percorre_vetor = 0; percorre_vetor < vector.Length - 1; percorre_vetor++) // n-1
        {
            if (vector[percorre_vetor] > vector[percorre_vetor + 1]) //troca
            {
                temp = vector[percorre_vetor + 1];
                vector[percorre_vetor + 1] = vector[percorre_vetor];
                vector[percorre_vetor] = temp;
            }
        }
    }
} // O(n*(n-1)) → n*n
```

1.1.3 Complexidade de tempo e de espaço

Tabela 1 Complexidade de tempo e de espaço. Algoritmo Bubblesort

Melhor caso	$\frac{n*n}{2}$	→	$O(n^2)$
Caso médio	$\frac{5*n*n}{2}$	→	$O(n^2)$
Pior caso	$2 * n * n$	→	$O(n^2)$
Complexidade de espaços		→	$O(1)$

1.2 Melhor Caso

Fase 1

i	0	1	2	3	4	5		não troca	troca
vector[i]	1	7	10	15	20	52	posicao	vector[i] < vector[i+1]	vector[i] > vector[i+1]
vector[i]	1	7	10	15	20	52	0	vector[0] < vector[1]	
vector[i]	1	7	10	15	20	52	1	vector[1] < vector[2]	
vector[i]	1	7	10	15	20	52	2	vector[2] < vector[3]	
vector[i]	1	7	10	15	20	52	3	vector[3] < vector[4]	
vector[i]	1	7	10	15	20	52	4	vector[4] < vector[5]	

Com uma fase e várias iteração já está ordenado, só tivemos comparação e nenhuma troca de elemento de posição

Tivemos 0 trocas e 30 comparações

1.3 Pior Caso

Fase 1

i	0	1	2	3	4	5		não troca	troca
vector[i]	52	20	15	10	7	1	posicao	vector[i] < vector[i+1]	vector[i] > vector[i+1]
vector[i]	52	20	15	10	7	1	0		vector[0] > vector[1]
vector[i]	20	52	15	10	7	1	1		vector[1] > vector[2]
vector[i]	20	15	52	10	7	1	2		vector[2] > vector[3]
vector[i]	20	15	10	52	7	1	3		vector[3] > vector[4]
vector[i]	20	15	10	7	52	1	4		vector[4] > vector[5]
vector[i]	20	15	10	7	1	52			

Observa-se que o elemento 52 da posição 0 foi deslocado até a posição 5. Neste caso tivemos (n-1) comparações e (n-1) trocas de posições.

Fase 2:

i	0	1	2	3	4	5		não troca	troca
vector[i]	20	15	10	7	1	52	posicao	vector[i] < vector[i+1]	vector[i] > vector[i+1]
vector[i]	20	15	10	7	1	52	0		vector[0] > vector[1]
vector[i]	15	20	10	7	1	52	1		vector[1] > vector[2]
vector[i]	15	10	20	7	1	52	2		vector[2] > vector[3]
vector[i]	15	10	7	20	1	52	3		vector[3] > vector[4]
vector[i]	15	10	7	1	20	52	4	vector[4] < vector[5]	
vector[i]	15	10	7	1	20	52			

Fase 3

i	0	1	2	3	4	5		não troca	troca
vector[i]	15	10	7	1	20	52	posicao	vector[i] < vector[i+1]	vector[i] > vector[i+1]
vector[i]	15	10	7	1	20	52	0		vector[0] > vector[1]
vector[i]	10	15	7	1	20	52	1		vector[1] > vector[2]
vector[i]	10	7	15	1	20	52	2		vector[2] > vector[3]
vector[i]	10	7	1	15	20	52	3	vector[3] < vector[4]	
vector[i]	10	7	1	15	20	52	4	vector[4] < vector[5]	
vector[i]	10	7	1	15	20	52			

Fase 4

i	0	1	2	3	4	5		não troca	troca
vector[i]	10	7	1	15	20	52	posicao	vector[i] < vector[i+1]	vector[i] > vector[i+1]
vector[i]	10	7	1	15	20	52	0		vector[0] > vector[1]
vector[i]	7	10	1	15	20	52	1		vector[1] > vector[2]
vector[i]	7	1	10	15	20	52	2	vector[2] < vector[3]	
vector[i]	7	1	10	15	20	52	3	vector[3] < vector[4]	
vector[i]	7	1	10	15	20	52	4	vector[4] < vector[5]	
vector[i]	7	1	10	15	20	52			

Fase 5

i	0	1	2	3	4	5		não troca	troca
vector[i]	7	1	10	15	20	52	posicao	vector[i] < vector[i+1]	vector[i] > vector[i+1]
vector[i]	7	1	10	15	20	52	0		vector[0] > vector[1]
vector[i]	1	7	10	15	20	52	1	vector[1] < vector[2]	
vector[i]	1	7	10	15	20	52	2	vector[2] < vector[3]	
vector[i]	1	7	10	15	20	52	3	vector[3] < vector[4]	
vector[i]	1	7	10	15	20	52	4	vector[4] < vector[5]	
vector[i]	1	7	10	15	20	52			

Fase 6

i	0	1	2	3	4	5		não troca	troca
vector[i]	1	7	10	15	20	52	posicao	vector[i] < vector[i+1]	vector[i] > vector[i+1]
vector[i]	1	7	10	15	20	52	0	vector[0] < vector[1]	
vector[i]	1	7	10	15	20	52	1	vector[1] < vector[2]	
vector[i]	1	7	10	15	20	52	2	vector[2] < vector[3]	
vector[i]	1	7	10	15	20	52	3	vector[3] < vector[4]	
vector[i]	1	7	10	15	20	52	4	vector[4] < vector[5]	
vector[i]	1	7	10	15	20	52			

Tivemos 15 trocas e 30 comparações

CAPÍTULO 2: INSERÇÃO

2.1. Descrição do algoritmo

O algoritmo realiza busca sequencial nos n elementos não ordenado para depois inserir o elemento corretamente num seguimento ordenado, sendo assim o vetor ordenado aumenta e o vetor não ordenado diminui.

2.1.1. Características

- É fácil de implementar
- Em alguns casos chega a ser 2 vezes mais rápido que o algoritmo bubbleSort dependendo do tamanho do vetor.
- É recomendado o usar em tabelas muito pequenas.

2.1.2. Codificação em C#

```
static public void insertionsort(int[] vector)
{
    int temp = 0; // complexidade de espaço
    for (int i = 0; i < vector.Length - 1; i++) // n-1
    {
        for (int j = i + 1; j > 0; j--)
        {
            if (vector[j - 1] > vector[j])
            {
                temp = vector[j - 1];
                vector[j - 1] = vector[j];
            }
        }
    }
}
```

```

        vector[j] = temp;
    }
}
} // O((n-1)*(n-1)) --> n*n
}

```

2.1.5 Complexidade de tempo e de espaço

O número de comparações $\frac{(n-1)*n}{2}$


A complexidade em pior caso, caso médio e melhor caso e:

Tabela 2 Complexidade de tempo e de espaço. Algoritmo Inserção

Melhor caso	$2 * (n - 1)$	→	$O(n)$
Caso médio	$\frac{n*n}{4}$	→	$O(n^2)$
Pior caso	$n * n$	→	$O(n^2)$
Complexidade de espaços		→	$O(1)$

2.2. Melhor Caso

Fase 1: só ocorre comparação num vetor ordenado



i	0	1	2	3	4	5
vector[i]	1	7	10	15	20	52

i	j
0	1


1	7
---	---

vector[j-1] > vector[j] troca

vector[j-1] < vector[j] não troca

vector[0] > vector[1] não troca

Fase 2:



	0	1	2	3	4	5	i	j
vector[i]	1	7	10	15	20	52	1	2,1


1	7	10
---	---	----

1	7	10
---	---	----

vector[1] > vector[2] não troca

vector[0] > vector[1] não troca

Fase 3:



	0	1	2	3	4	5	i	j
vector[i]	1	7	10	15	20	52	2	3,2,1

1	7	10	15
---	---	----	----

1	7	10	15
---	---	----	----


1	7	10	15
---	---	----	----

vector[2] > vector[3] não troca

vector[1] > vector[2] não troca

vector[0] > vector[1] não troca

Fase 4:



	0	1	2	3	4	5	i	j
vector[i]	1	7	10	15	20	52	3	4,3,2,1

1	7	10	15	20
---	---	----	----	----

1	7	10	15	20
---	---	----	----	----

1	7	10	15	20
---	---	----	----	----

1	7	10	15	20
---	---	----	----	----


vector[3] > vector[4] não troca

vector[2] > vector[3] não troca

vector[1] > vector[2] não troca

vector[0] > vector[1] não troca

Fase 5:



	0	1	2	3	4	5	i	j
vector[i]	1	7	10	15	20	52	4	5,4,3,2,1

1	7	10	15	20	52
---	---	----	----	----	----

1	7	10	15	20	52
---	---	----	----	----	----

1	7	10	15	20	52
---	---	----	----	----	----

1	7	10	15	20	52
---	---	----	----	----	----

vector[4] > vector[5] não troca

vector[3] > vector[4] não troca

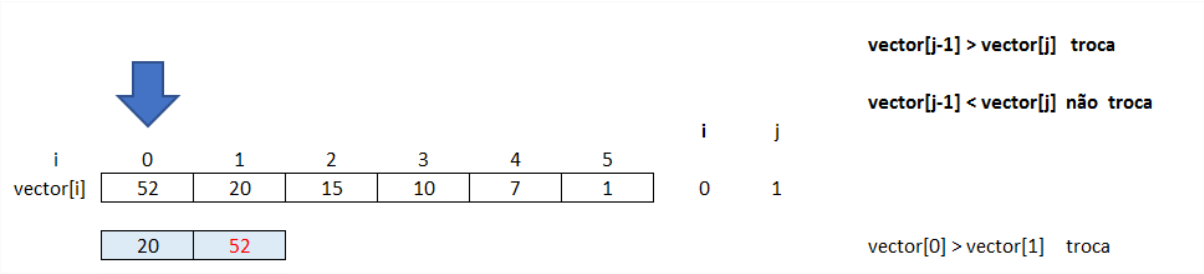
vector[2] > vector[3] não troca

vector[1] > vector[2] não troca

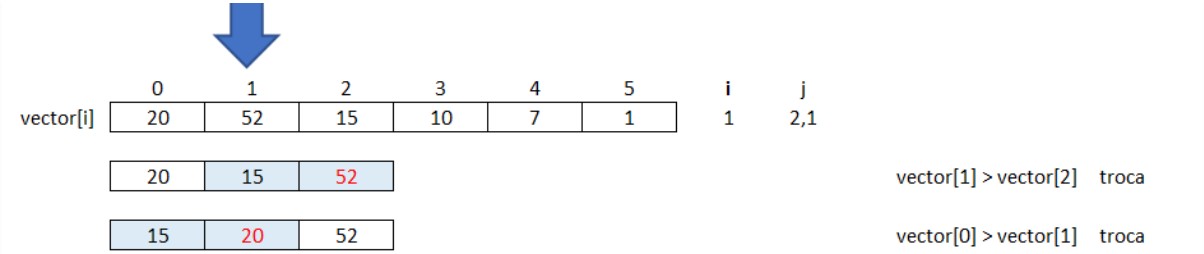
vector[0] > vector[1] não troca

2.3. Pior Caso

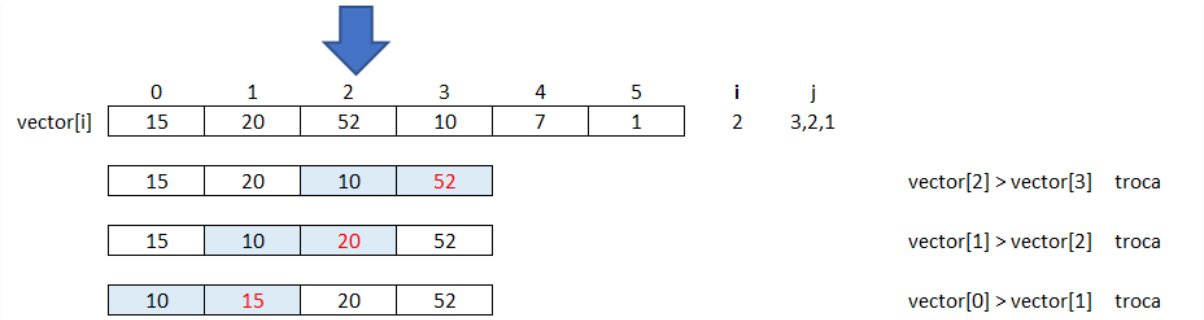
Fase 1: Toda compara ocorre troca de elemento num vetor




Fase 2:



Fase 3:




Fase 4:



	0	1	2	3	4	5	i	j
vector[i]	10	15	20	52	7	1	3	4,3,2,1

10	15	20	7	52				vector[3] > vector[4] troca
10	15	7	20	52				vector[2] > vector[3] troca
10	7	15	20	52				vector[1] > vector[2] troca
7	10	15	20	52				vector[0] > vector[1] troca

Fase 5:



	0	1	2	3	4	5	i	j
vector[i]	7	10	15	20	52	1	4	5,4,3,2,1

7	10	15	20	1	52			vector[4] > vector[5] troca
7	10	15	1	20	52			vector[3] > vector[4] troca
7	10	1	15	20	52			vector[2] > vector[3] troca
7	1	10	15	20	52			vector[1] > vector[2] troca
1	7	10	15	20	52			vector[0] > vector[1] troca

CAPÍTULO 3: SELEÇÃO

3.1. Descrição do algoritmo

A ordenação consiste em cada etapa selecionar o menor (ou o maior) elemento de um vetor e depois colocá-lo na posição correta.

3.1.1 Características

- É fácil de implementar
- Este é um exemplo de um método não estável, pois ele nem sempre deixa os registros com chaves iguais na mesma posição relativa.
- É rápidos e bem eficientes para problemas de dimensão média.

3.1.2 Codificação em C#

```
static public void selecao(int[] vector)
{
    int min, aux; // complexidade de espaço 2
    for (int i = 0; i < vector.Length - 1; i++) // n-1
    {
        min = i;
        for (int j = i + 1; j < vector.Length; j++) // n-1
        {
            if (vector[j] < vector[min])
            {
                min = j;
            }
        }
        if (min != i)
        {
            aux = vector[min];
            vector[min] = vector[i];
            vector[i] = aux;
        }
    } // (n-1)*(n-1)
}
```


3.1.3 Complexidade de tempo e de espaço

Tabela 3 Complexidade de tempo e de espaço. Algoritmo Seleção


O número de comparações	$\frac{(n-1)*n}{2}$	
A complexidade em pior caso, caso médio e melhor caso	$(n-1) + (n-2) + \dots + 2 + 1 = \frac{(n-1)*n}{2} \rightarrow O(n^2)$	
Complexidade de espaços		$\rightarrow O(1)$

3.2. Melhor Caso


Fase 1:

<div>  </div>									vector[j-1] > vector[j] troca vector[j-1] < vector[j] não troca	
i	0	1	2	3	4	5	minimo	i	j	
vector[i]	1	7	10	15	20	52				
	1	7					0	0	1	vector[0] < vector[1] não troca
	1	7	20				0	0	2	vector[0] < vector[2] não troca
	1	7	20	7			0	0	3	vector[0] < vector[3] não troca
	1	7	20	7	10		0	0	4	vector[0] < vector[4] não troca
	1	7	20	7	10	15	0	0	5	vector[0] < vector[5] não troca

Fase 2:

<div>  </div>										
i	0	1	2	3	4	5	minimo	i	j	
vector[i]	1	7	10	15	20	52				
	1	7	10				1	1	2	vector[1] < vector[2] não troca
	1	7	20	15			1	1	3	vector[1] < vector[3] não troca
	1	7	20	7	20		1	1	4	vector[1] < vector[4] não troca

Fase 3:

<div>  </div>										
i	0	1	2	3	4	5	minimo	i	j	
vector[i]	1	7	10	15	20	52				
	1	7	10	15			2	2	3	vector[2] < vector[3] não troca
	1	7	20	15	20		2	2	4	vector[2] < vector[4] não troca
	1	7	20	7	20	52	2	2	5	vector[2] < vector[5] não troca

Fase 4:

Diagram illustrating the selection sort algorithm. A blue arrow points to the element 15 at index 3 in the array [1, 7, 10, 15, 20, 52]. Below, two rows show comparisons: first, 15 is compared with 20 (index 4) and found to be smaller, so no swap occurs; second, 15 is compared with 52 (index 5) and found to be smaller, so no swap occurs. The labels 'minimo', 'i', and 'j' are shown above the comparison steps.

i	0	1	2	3	4	5
vector[i]	1	7	10	15	20	52


minimo	i	j	Condition
3	3	4	vector[3] < vector[4] não troca
3	3	5	vector[3] < vector[5] não troca

Diagram illustrating the selection sort algorithm. A blue arrow points to the element 20 at index 4 in the array [1, 7, 10, 15, 20, 52]. Below the array, the current state shows the element 20 at index 4 and 52 at index 5. The text "vector[4] < vector[5] não troca" indicates that since the element at index 4 is less than the element at index 5, no swap is performed.

3.3. Pior Caso


Diagram illustrating the first pass of bubble sort. A blue arrow points down to the initial array [52, 20, 15, 10, 7, 1]. Below it, five comparisons are shown between adjacent elements. Each comparison is represented by a row with a shaded box for the first element and an unshaded box for the second. The elements to be compared are highlighted in red. The results are: (20, 52) - troca; (15, 52) - troca; (10, 52) - troca; (7, 52) - troca; (1, 52) - troca. To the right, a table summarizes the comparisons with columns for 'minimo', 'i', and 'j', and the decision 'troca'.

minimo	i	j	troca
0	0	1	troca
0	0	2	troca
0	0	3	troca
0	0	4	troca
0	0	5	troca




i	0	1	2	3	4	5	minimo	i	j	
vector[i]	1	52	20	15	10	7				
	1	20	52				1	1	2	vector[1] < vector[2] troca
	1	15	52	20			1	1	3	vector[1] < vector[3] troca
	1	10	52	20	15		1	1	4	vector[1] < vector[4] troca
	1	7	52	20	15	10	1	1	5	vector[1] < vector[5] troca

Fase 3:




i	0	1	2	3	4	5	minimo	i	j	
vector[i]	1	7	52	20	15	10				
	1	7	20	52			2	2	3	vector[2] < vector[3] troca
	1	7	15	52	20		2	2	4	vector[2] < vector[4] troca
	1	7	10	52	20	15	2	2	5	vector[2] < vector[5] troca

Fase 4:



i	0	1	2	3	4	5	minimo	i	j	
vector[i]	1	7	10	52	20	15				
	1	7	10	20	52		3	3	4	vector[3] < vector[4] troca
	1	7	10	15	52	20	3	3	5	vector[3] < vector[5] troca

Fase 5:



i	0	1	2	3	4	5	minimo	i	j	
vector[i]	1	7	10	15	52	20				
	1	7	10	15	20	52	4	4	5	vector[4] < vector[5] troca

CAPÍTULO 4: QUICKSORT

4.1. Descrição do algoritmo

A ordenação consiste em dividir um problema maior de um conjunto com n itens em de subproblema menores, o método é conhecido com **dividir para conquistar**. O resultado final é a junções das partições menores já ordenado.

O pivô é o elemento de referência que vai separar a partição esquerda com elementos menores que o pivô e vai separar também a partição direita com elemento maiores que o pivô. Este processo se repete recursivamente até que o vetor esteja ordenado

4.1.1 Características

- Ótimo para ordenar vetores grandes.
- Os laços internos são simples o que resulta em ser mais rápido.
- Memória auxiliar para a pilha de recursão é pequena.
- Este é um exemplo de um método não estável, pois ele nem sempre deixa os registros com chaves iguais na mesma posição relativa.

4.1.2 Codificação em C#

```
static public void QuickSort(int[] vetor, int primeiro, int ultimo)
{
    int baixo, alto, meio, pivo, repositorio;
    baixo = primeiro;
    alto = ultimo;
    meio = (int)((baixo + alto) / 2);
    // complexidade de espaço 5
    pivo = vetor[meio];

    while (baixo <= alto)
    {
        while (vetor[baixo] < pivo)
            baixo++;
        while (vetor[alto] > pivo)
            alto--;
        if (baixo < alto)
        {
            repositorio = vetor[baixo];
            vetor[baixo++] = vetor[alto];
            vetor[alto--] = repositorio;
        }
        else
        {

```

```

        if (baixo == alto)
        {
            baixo++;
            alto--;
        }
    }

    if (alto > primeiro)
        QuickSort(vetor, primeiro, alto);
    if (baixo < ultimo)
        QuickSort(vetor, baixo, ultimo);
}

```

4.1.3 Complexidade de tempo e de espaço

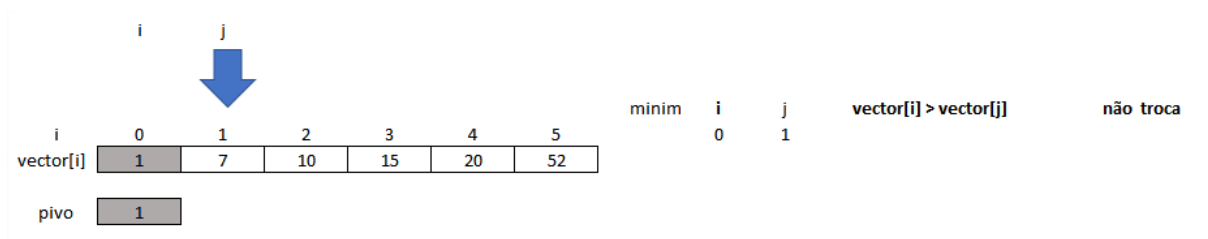
Tabela 4 Complexidade de tempo e de espaço. Algoritmo Quicksort

Caso médio	→ $O(n * \log n)$
Melhor caso	→ $O(n * \log n)$
Pior caso	→ $O(n^2)$
Complexidade de espaços	→ $O(1)$

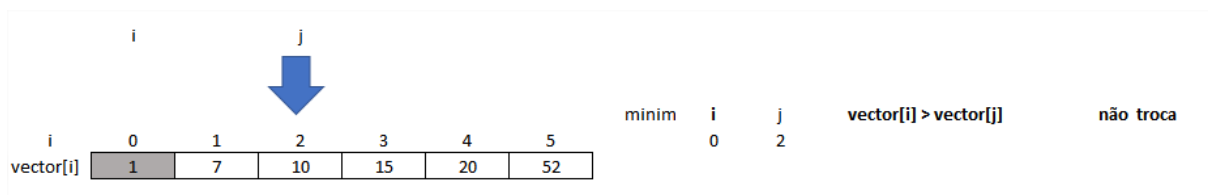
4.2. Melhor Caso

Melhor caso: ocorre quando o vetor já está ordenado

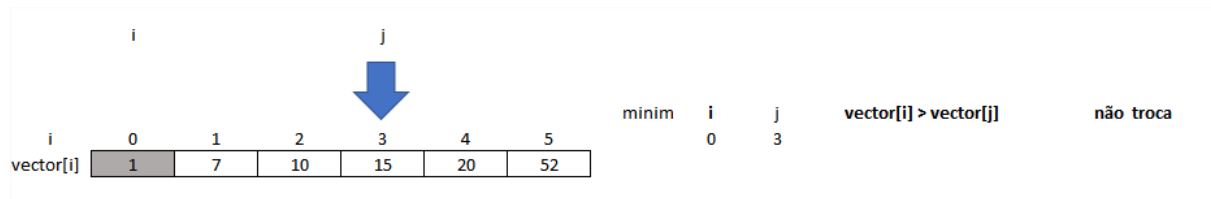
Iteração 1:



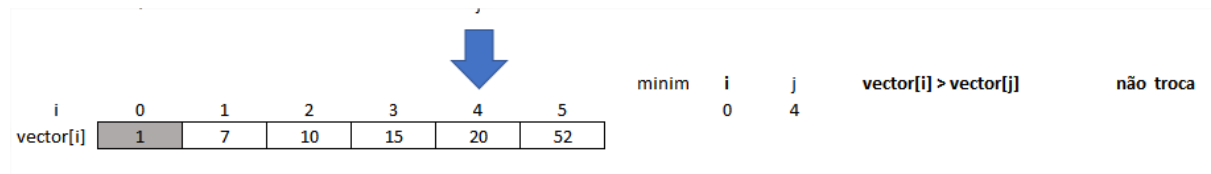
Iteração 2:



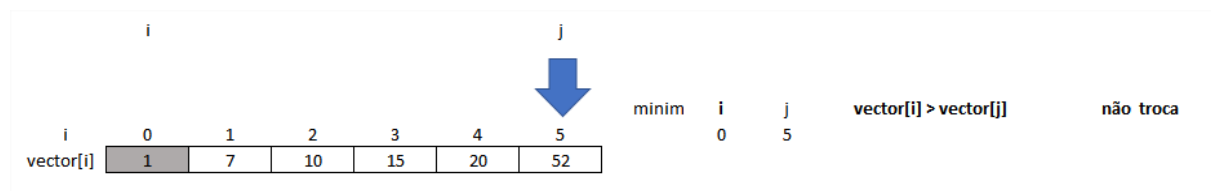
Iteração 3:



Iteração 5:



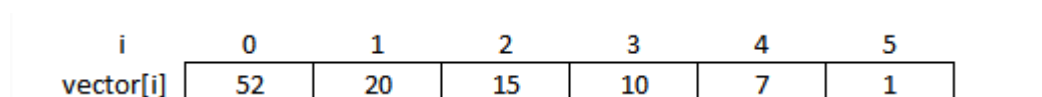
Iteração 6:



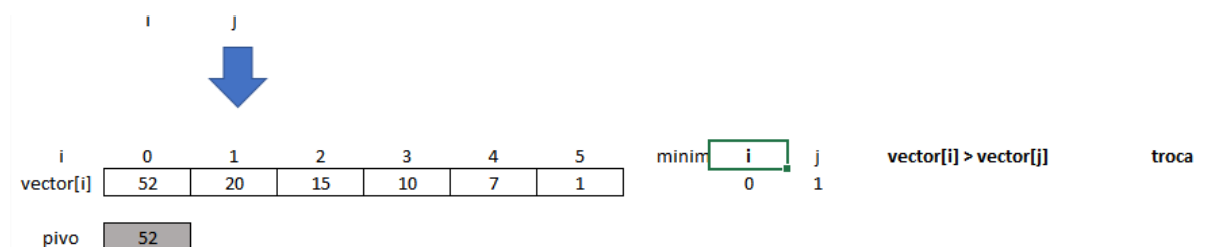
4.3. Pior Caso

Pior caso: ocorre quando o vetor está ordenado de forma decrescente.

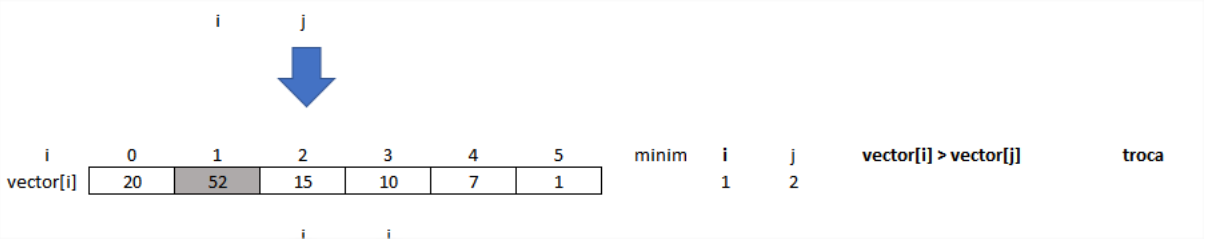
Iteração 1:



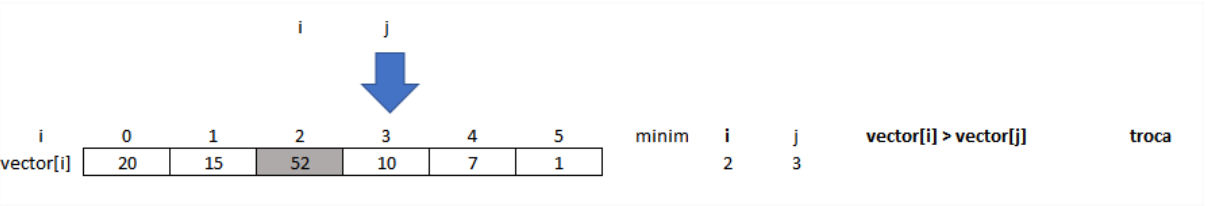
Iteração 2:



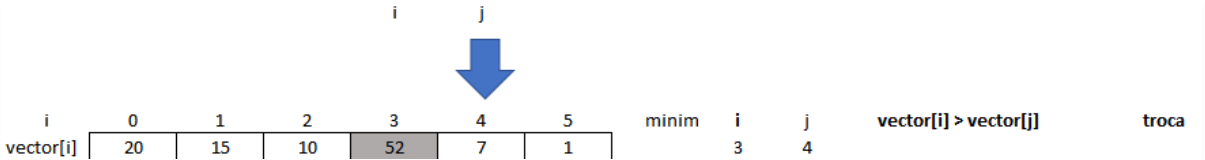
Iteração 3:



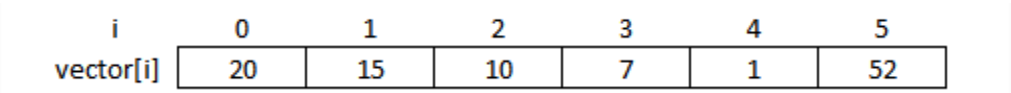
Iteração 4:



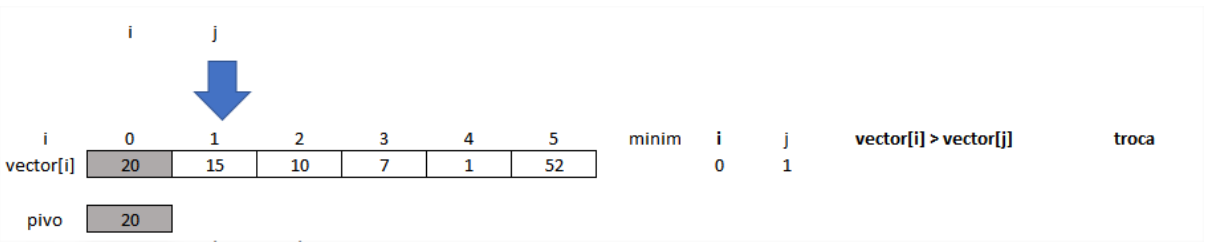
Iteração 5:



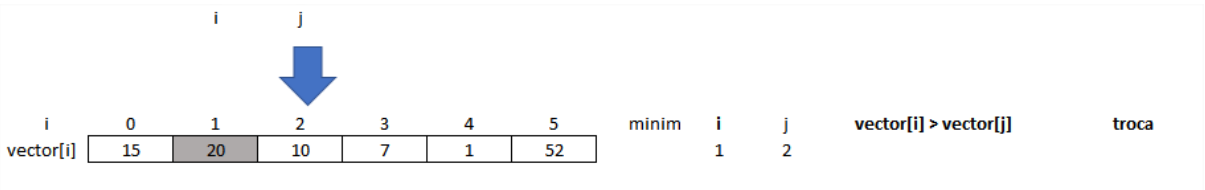
Iteração 6:



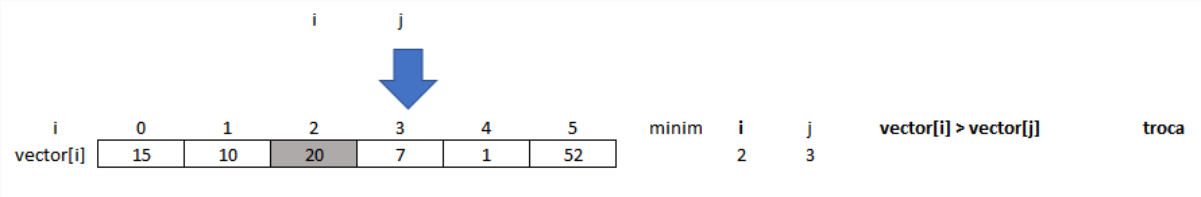
Iteração 7:



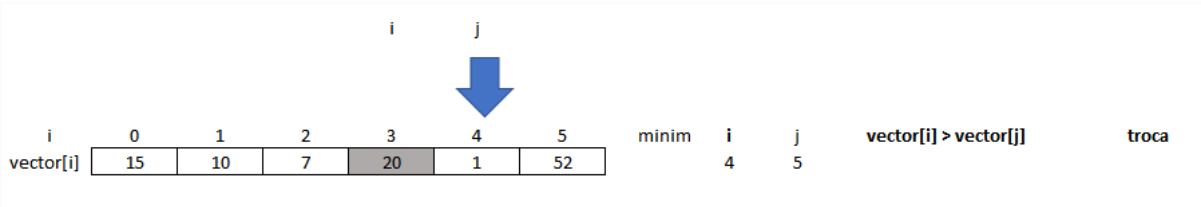
Iteração 8:



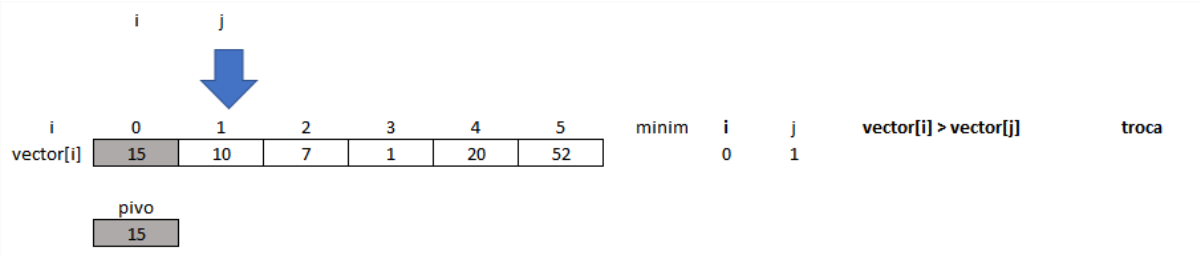
Iteração 9:



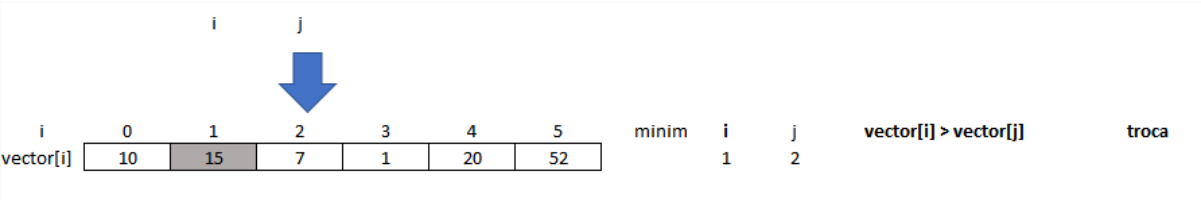
Iteração 10:



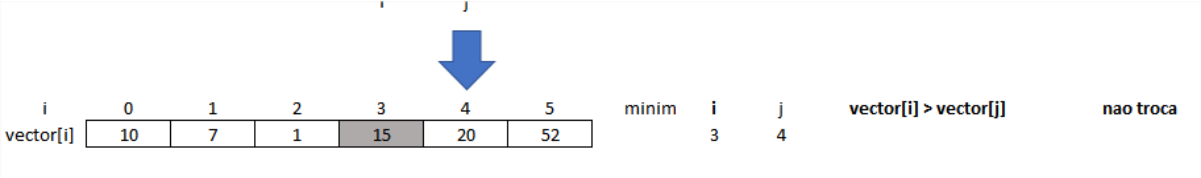
Iteração 11:



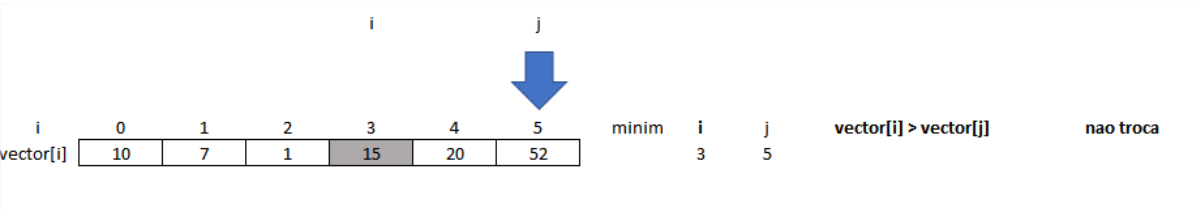
Iteração 12:



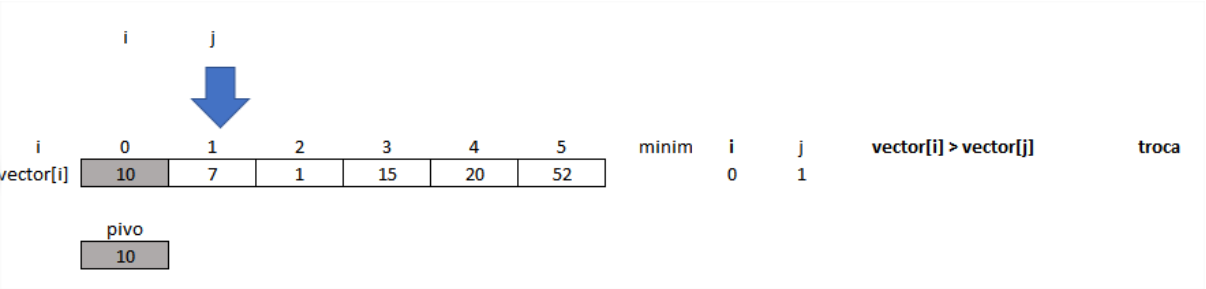
Iteração 13:



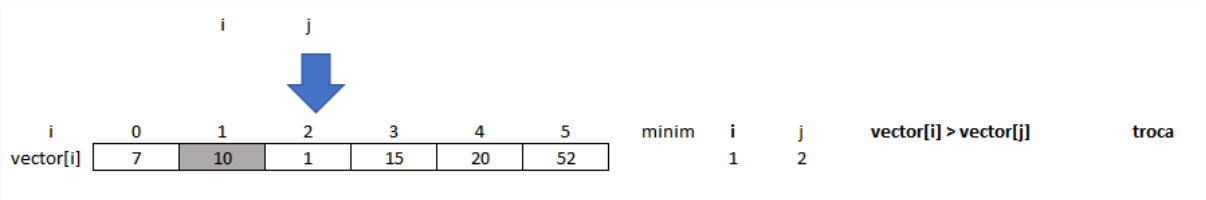
Iteração 14:



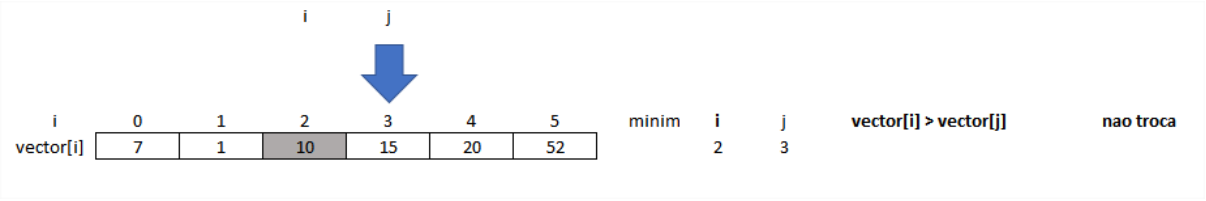
Iteração 15:



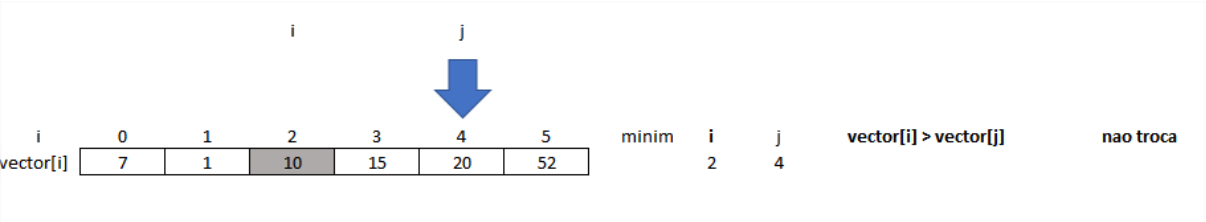
Iteração 16:



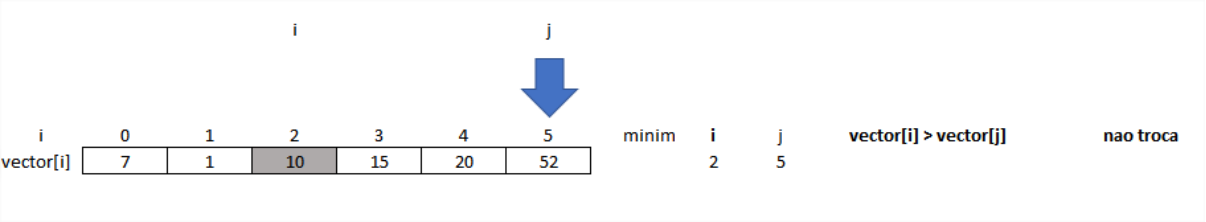
Iteração 17:



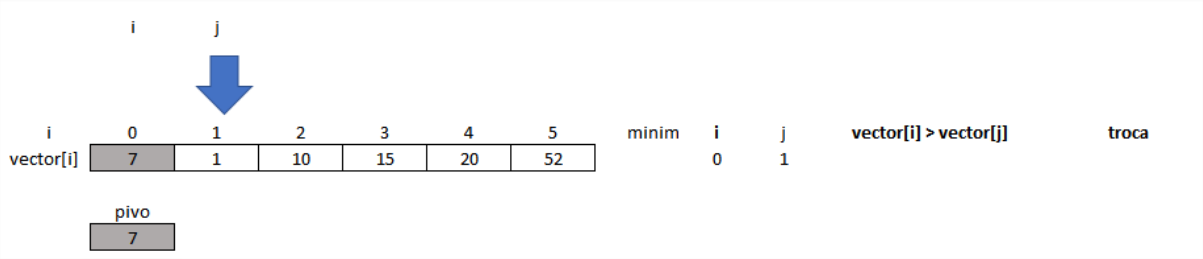
Iteração 18:



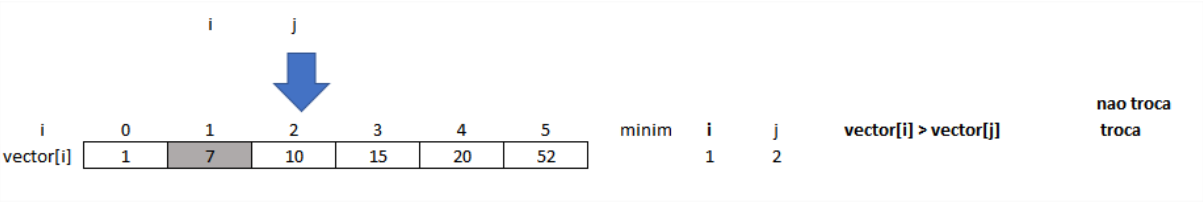
Iteração 19:



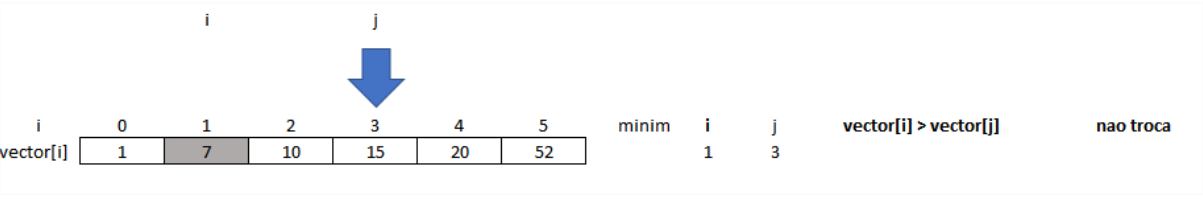
Iteração 20:



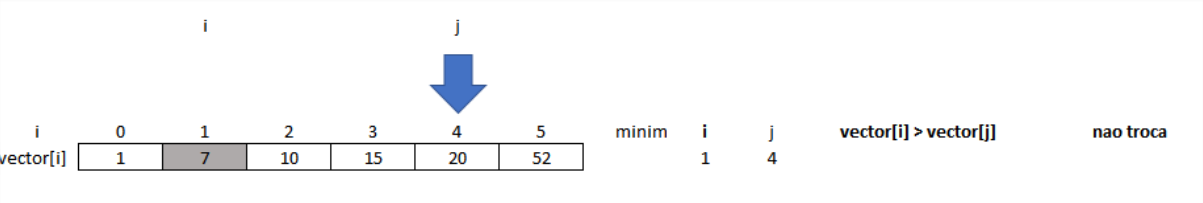
Iteração 21:



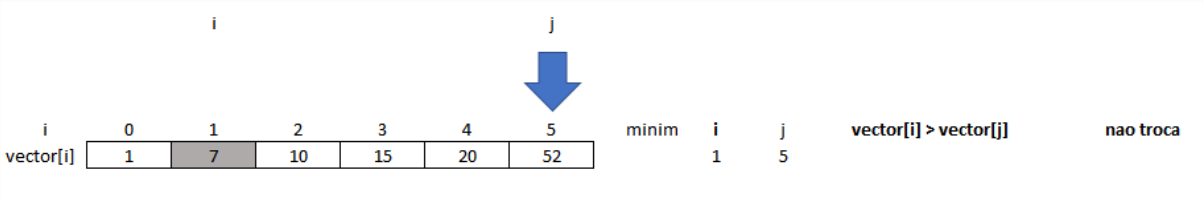
Iteração 22:



Iteração 23:



Iteração 24:



CAPÍTULO 5: HEAPSORT

5.1. Descrição do algoritmo

A ordenação funciona semelhante ao algoritmo de seleção, ocorre a seleção do menor item do vetor e em seguida é trocado pelo elemento da primeira posição. A heap é uma árvore binária que tem a propriedade de garantir que o valor de cada nó não é menor que os valores armazenados em cada nó filho. Os maiores elementos se mantêm na raiz.

5.1.1 Características

- Não é recomendado para vetores pequenos, devido à complexidade do heap.
- O algoritmo é mais eficiente com vetores maiores
- Vetor ordenado inversamente, o processo de reconstrução do heap leva zero movimentação para ser realizado.
- Desvantagem que não é estável, não é rápido comparável ao quicksort.

5.1.2 Codificação em C#

```
public static void HeapSort(int[] Vetor)
{
    //Max-Heap
    int tamanho_da_pilha = Vetor.Length;
    for (int p = (tamanho_da_pilha - 1) / 2; p >= 0; p--)
        MaxHeapify(Vetor, tamanho_da_pilha, p);
    for (int i = Vetor.Length - 1; i > 0; i--)
    {
        int temp = Vetor[i];
        Vetor[i] = Vetor[0];
        Vetor[0] = temp;
        tamanho_da_pilha--;
        MaxHeapify(Vetor, tamanho_da_pilha, 0);
    }
}

private static void MaxHeapify(int[] Vetor, int tamanho_da_pilha, int indice)
```

```

{
    int esquerda = (indice + 1) * 2 - 1;
    int direita = (indice + 1) * 2;
    int maior = 0;
    if (esquerda < tamanho_da_pilha && Vetor[esquerda] > Vetor[indice])
        maior = esquerda;
    else
        maior = indice;

    if (direita < tamanho_da_pilha && Vetor[direita] > Vetor[maior])
        maior = direita;

    if (maior != indice)
    {
        int temp = Vetor[indice];
        Vetor[indice] = Vetor[maior];
        Vetor[maior] = temp;
        MaxHeapify(Vetor, tamanho_da_pilha, maior);
    }
}

```

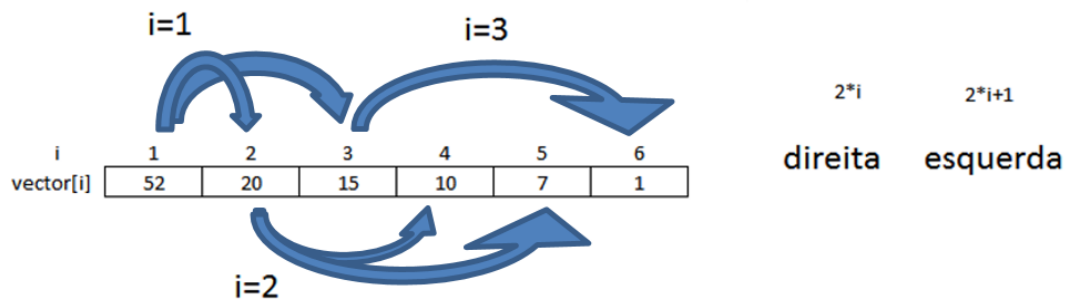
5.1.2 Complexidade de tempo e de espaço

Tabela 5 Complexidade de tempo e de espaço. Algoritmo Heapsort

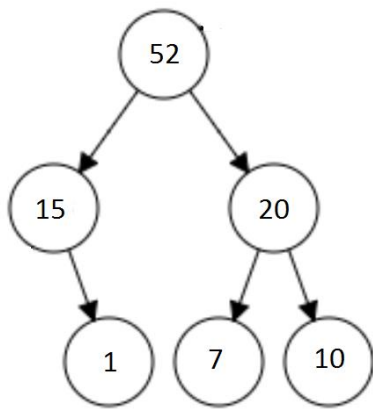
Caso médio	→ $O(n * \log n)$
Melhor caso	→ $O(n * \log n)$
Pior caso	→ $O(n * \log n)$
Complexidade de espaços	→ $O(1)$

5.2. Melhor Caso

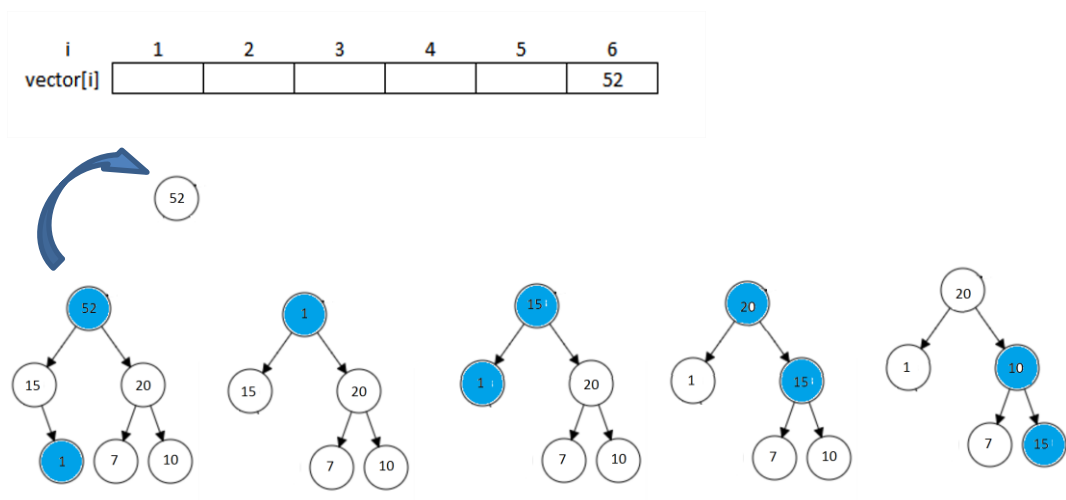
Iteração 1:



Iteração 2: primeira montagem verifica se a árvore esta ordenada

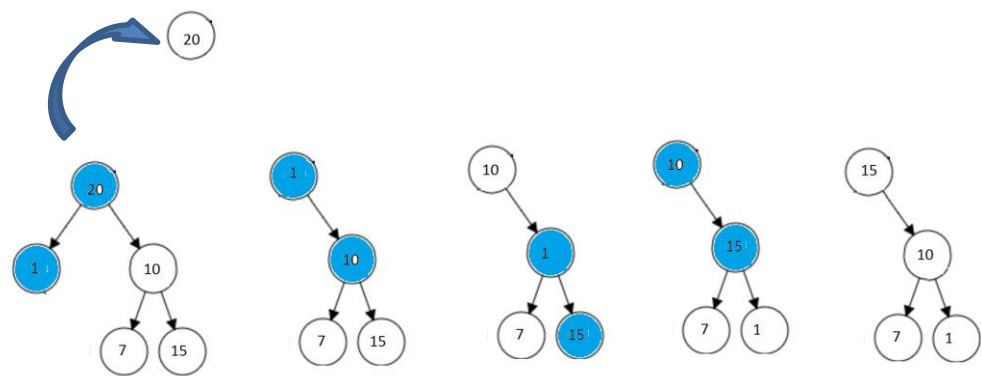


Iteração 3: Ocorre a retirada do elemento nó raiz (maior) e em seguida ocorre um novo balanceamento da árvore



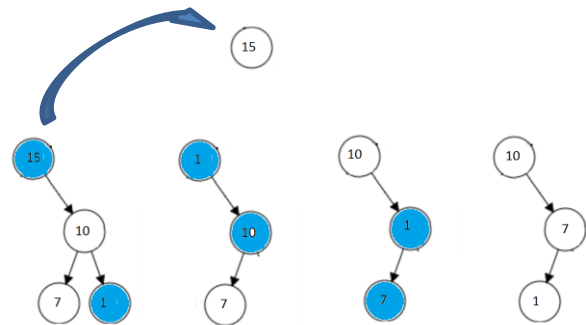
Iteração 4:

i	1	2	3	4	5	6
vector[i]					20	52



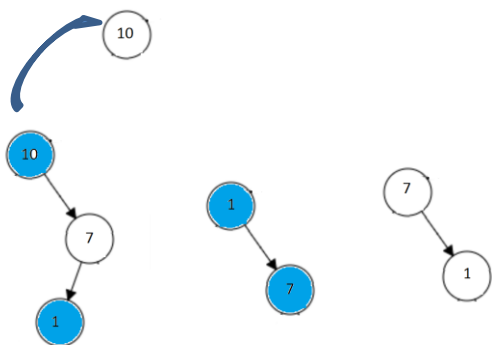
Iteração 5:

i	1	2	3	4	5	6
vector[i]				15	20	52



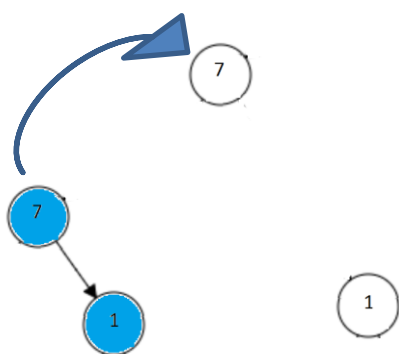
Iteração 6:

i	1	2	3	4	5	6
vector[i]			10	15	20	52



Iteração 7:

i	1	2	3	4	5	6
vector[i]		7	10	15	20	52

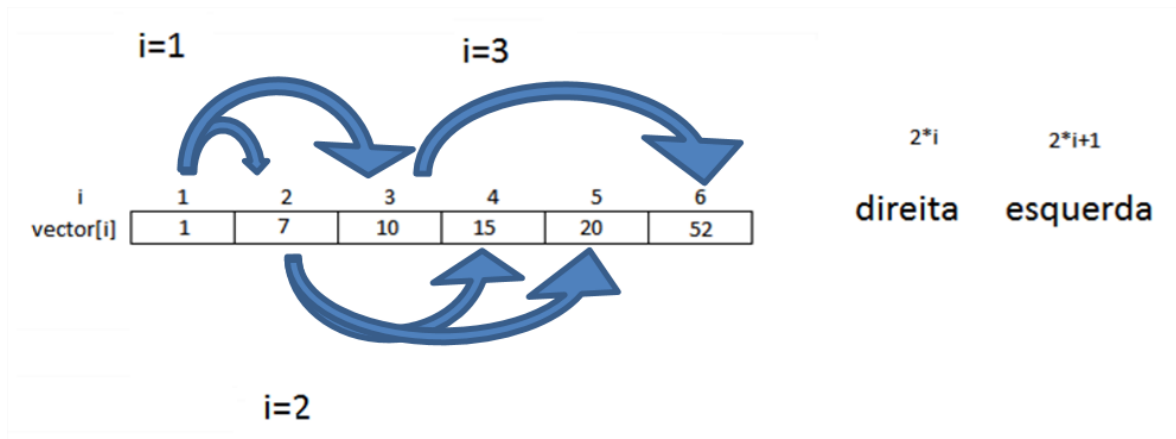


Resultado:

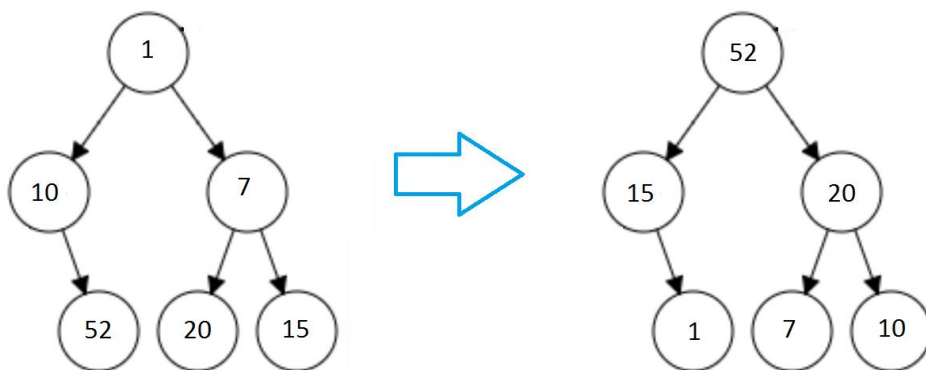
i	1	2	3	4	5	6
vector[i]	1	7	10	15	20	52

5.3. Pior Caso

Montagem da árvore



Iteração 1: ocorre o balanceamento para depois ocorrer a retirada dos elementos



Iteração 2: ocorre várias etapas para o novo balanceamento e depois segue os mesmos processos do melhor caso, ocorre retirada dos elementos e um novo balanceamento para manter o maior elemento na raiz da árvore.

Após estar balanceada a árvore, segue-se os mesmo processos de interações do melhor caso, as Iteração 1,25

CAPÍTULO 6: MERGESORT

6.1. Descrição do algoritmo

O mergesort é um algoritmo de ordenação recursiva, a ideia é dividir uma sequência original em pares de dados, gerar n divisões em seguida ordena-las e junta-la já ordenando.

O método usa a estratégia de divisão e conquista seguindo os seguintes passos:

Entrada → divisão → combinar (ordenar) → saída.

6.1.1 Características

- A ordenação é estável porque preserva a ordem das chaves iguais
- É eficiente com vetores muito grande

6.1.2 Codificação em C#

```
static public void DoMerge(int[] Vetor, int esquerda, int meio, int direita)
{
    int[] temp = new int[Vetor.Length]; // Complexidade de espaços n
    int i, esquerda_end, num_elements, tmp_pos; // espaços 3

    esquerda_end = (meio - 1);
    tmp_pos = esquerda;
    num_elements = (direita - esquerda + 1);

    while ((esquerda <= esquerda_end) && (meio <= direita))
    {
        if (Vetor[esquerda] <= Vetor[meio])
```



```

        temp[tmp_pos++] = Vetor[esquerda++];
    else
        temp[tmp_pos++] = Vetor[meio++];
}

while (esquerda <= esquerda_end)
    temp[tmp_pos++] = Vetor[esquerda++];

while (meio <= direita)
    temp[tmp_pos++] = Vetor[meio++];

for (i = 0; i < num_elements; i++)
{
    Vetor[direita] = temp[direita];
    direita--;
}
}

static public void MergeSort_Recursive(int[] Vetor, int esquerda, int direita)
{
    int meio; // espaços 1

    if (direita > esquerda)
    {
        meio = (direita + esquerda) / 2; // divide o vetor ao meio
        MergeSort_Recursive(Vetor, esquerda, meio);
        MergeSort_Recursive(Vetor, (meio + 1), direita);

        DoMerge(Vetor, esquerda, (meio + 1), direita);
    }
}

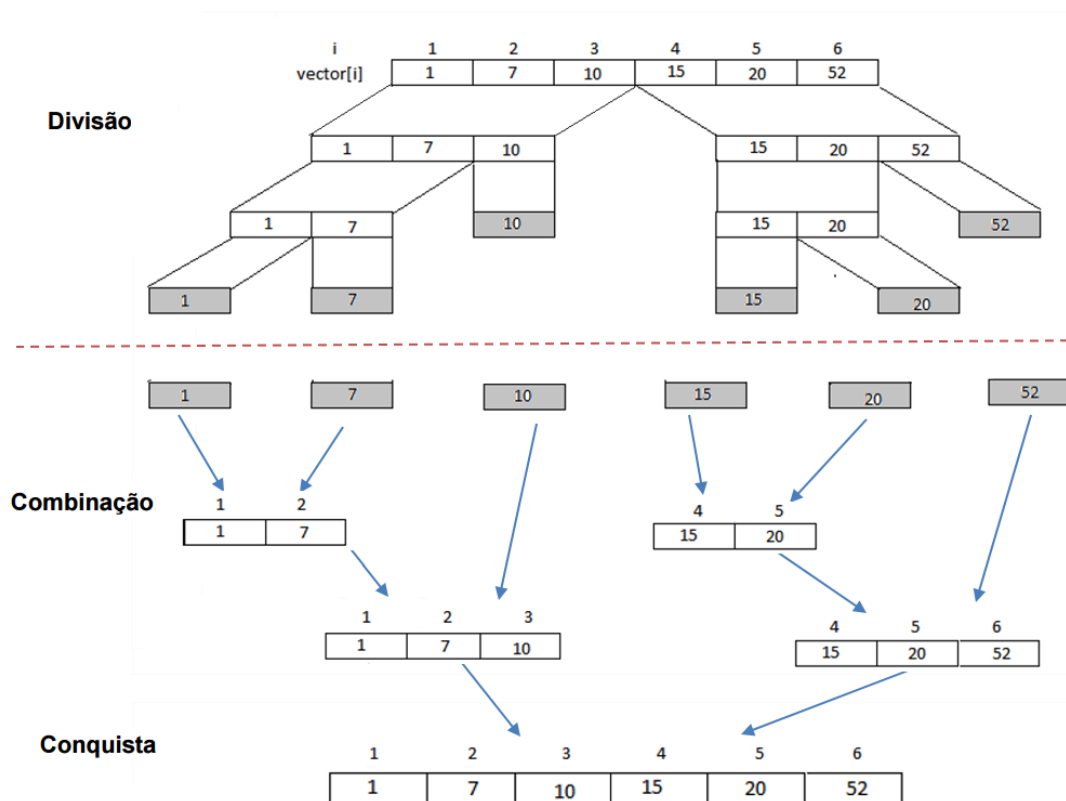
```

6.1.2 Complexidade de tempo e de espaço

Tabela 6 Complexidade de tempo e de espaço. Algoritmo Mergesort

Caso médio	→ $O(n * \log n)$
Melhor caso	→ $O(n * \log n)$
Pior caso	→ $O(n * \log n)$
Complexidade de espaços	→ $O(n)$

6.2. Melhor Caso



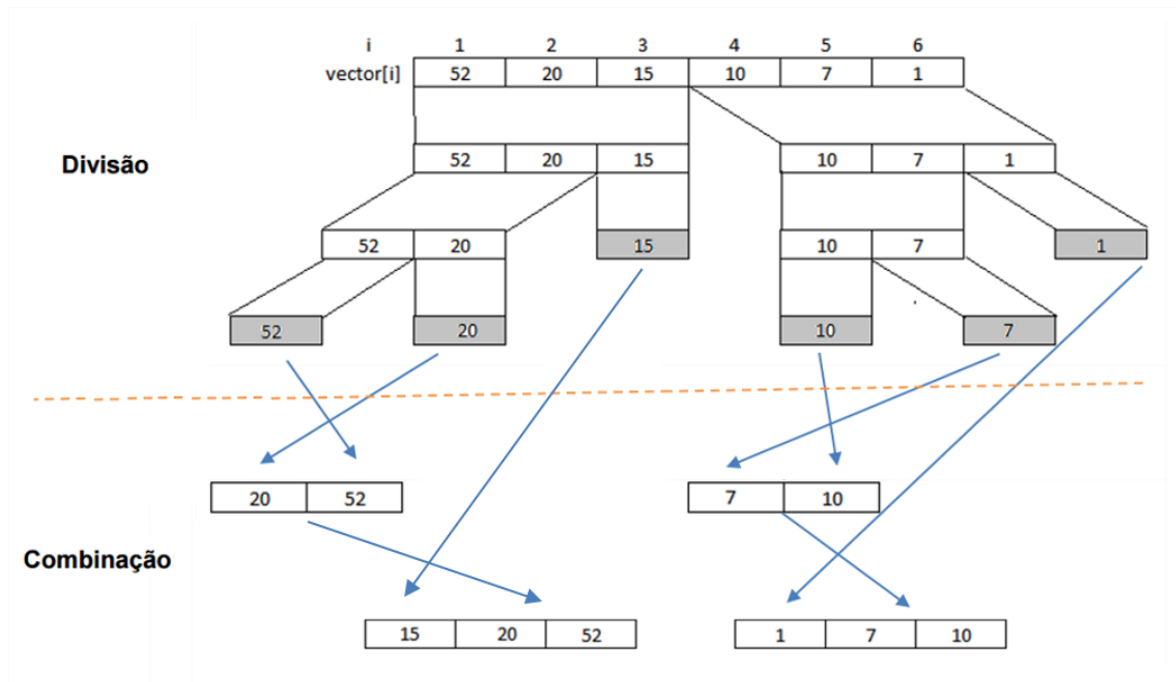
Divisão: O problema maior é dividido em vários subproblemas menores até chegarmos nas folhas da arvores onde há um único elemento.

Combinação: É resultado da combinação dos menores problemas que são ordenados até chegarmos na solução final.

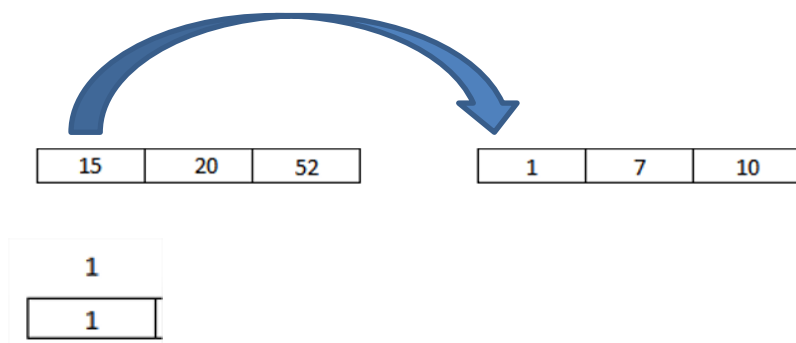
Conquista: Os problemas menores já foram ordenados e agrupados, esta etapa final concluir-se a ordenação.

6.3. Pior Caso

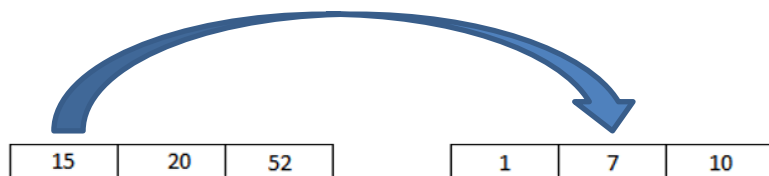
Iteração 1: ocorre a divisão



Iteração 2: combinação e ordenação

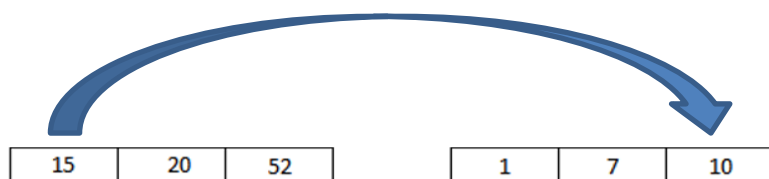


Iteração 3: combinação e ordenação



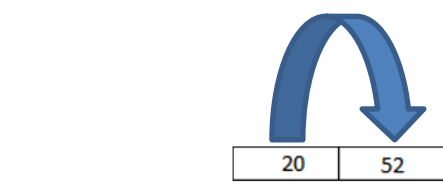
1	2
1	7

Iteração 4: combinação e ordenação



1	2	3	4
1	7	10	15

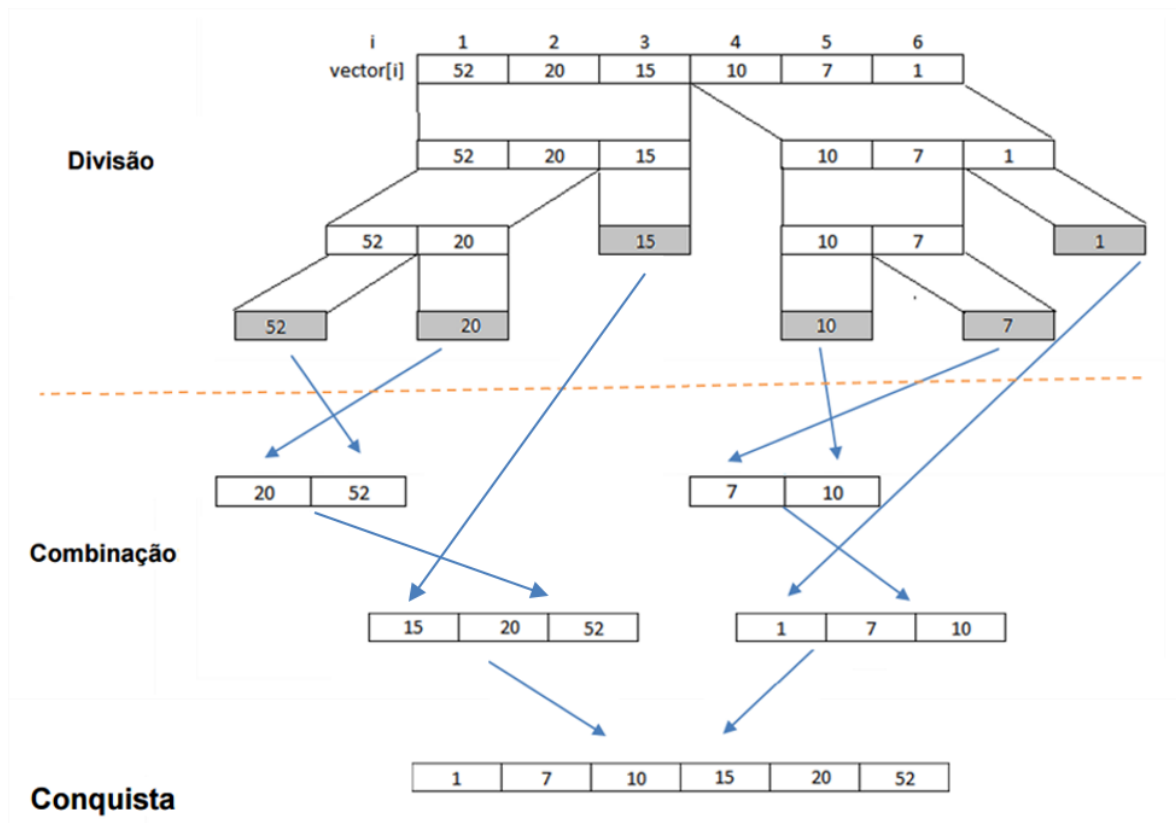
Iteração 5: combinação e ordenação



Conquista:

1	2	3	4	5	6
1	7	10	15	20	52

Resultado Final



CAPÍTULO 7: TESTES E ANALISE DE DESEMPENHO

Neste teste seguimos os seguintes procedimento:

- Geramos vetores de números aleatórios com uso de sementes, com tamanho variando a cada teste realizados.
- Ordenamos este vetor com quicksort o mais rápido.
- Criamos uma função para inverter (ordenação decrescente) para garantir o teste de desempenho dos piores casos
- Foi gerada 6 copias desses vetores e depois, usado para testar cada algoritmo de ordenação.
- Foi medido o tempo de processamento de cada um

Configuração do computador para realizar o teste:

- Sistema operacional: Windows 10 Education 64 bits
- Processador: Intel(R) Core(TM) i7-2600 CPU @ 3.40GHZ (8 CPUs),~3.4GHZ
- Memoria: 8192MB RAM

Tabela 7 Tabela de teste

	tempo em milisegundos																								
teste	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Bubblesort:	168	187	223	249	285	326	366	410	462	517	632	670	689	758	796	857	926	994	1067	1143	1220	1303	1387	1467	1563
Insertionsort:	105	135	139	161	183	209	236	264	354	356	362	420	468	470	513	552	594	639	684	734	788	823	886	942	1008
Selecao:	77	84	98	117	137	157	180	204	273	260	314	337	358	388	423	477	493	542	581	614	662	698	743	788	835
QuickSort:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
HeapSort:	2	2	2	2	2	2	3	3	3	4	3	4	4	4	4	4	5	5	5	5	6	6	6	6	6
MergeSort_Recursive:	11	9	10	12	14	16	18	20	22	28	27	29	31	33	36	39	42	45	48	52	54	57	61	65	68

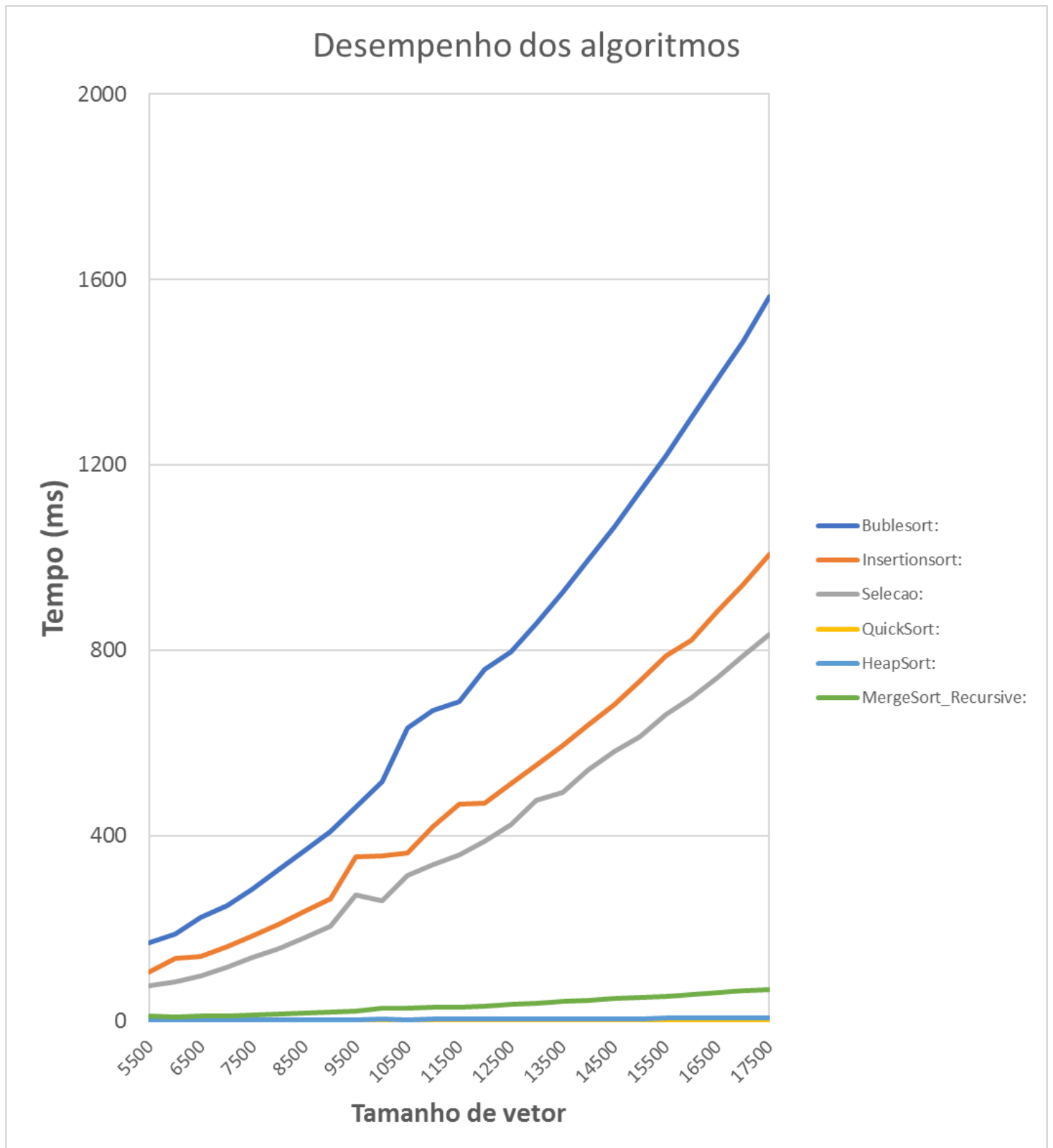


Figura 1: Comparação de desempenho dos algoritmos de ordenação

Conclusão:

Vetor de dados pequena e media

BUBBLESORT

- É recomendado o usar em pequena quantidade de dados.

INSERÇÃO

- É recomendado o usar em tabelas muito pequenas.

SELEÇÃO

- É rápido e bem eficiente para problemas de dimensão média.

Ótimo em todos os casos

QUICKSORT

- Ótimo para ordenar vetores grandes.

HEAPSORT

- O algoritmo é mais eficiente com vetores maiores

MERGESORT

- É eficiente com vetores muito grande

REFERÊNCIAS

T. H. Cormen, C. E. Leiserson, R. L. Rivest e C. Stein, Introduction to Algorithms, McGraw-Hill, 2001, second edition.

Anexo: PROJETO EM C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using static System.Console;
using System.Diagnostics;

namespace ConsoleApplication1
{
    class Program
    {

        static public void QuickSort(int[] vetor, int primeiro, int ultimo)
        {

            int baixo, alto, meio, pivo, repositorio;
            baixo = primeiro;
            alto = ultimo;
            meio = (int)((baixo + alto) / 2);

            pivo = vetor[meio];

            while (baixo <= alto)
            {
                while (vetor[baixo] < pivo)
                    baixo++;
                while (vetor[alto] > pivo)
                    alto--;
                if (baixo < alto)
                {
                    repositorio = vetor[baixo];
                    vetor[baixo++] = vetor[alto];
                    vetor[alto--] = repositorio;
                }
                else
                {
                    if (baixo == alto)
                    {

```

```

        baixo++;
        alto--;
    }
}

if (alto > primeiro)
    QuickSort(vetor, primeiro, alto);
if (baixo < ultimo)
    QuickSort(vetor, baixo, ultimo);
}

//-----
static public void DoMerge(int[] Vetor, int esquerda, int meio, int direita)
{
    int[] temp = new int[Vetor.Length];
    int i, esquerda_end, num_elements, tmp_pos;

    esquerda_end = (meio - 1);
    tmp_pos = esquerda;
    num_elements = (direita - esquerda + 1);

    while ((esquerda <= esquerda_end) && (meio <= direita))
    {
        if (Vetor[esquerda] <= Vetor[meio])
            temp[tmp_pos++] = Vetor[esquerda++];
        else
            temp[tmp_pos++] = Vetor[meio++];
    }

    while (esquerda <= esquerda_end)
        temp[tmp_pos++] = Vetor[esquerda++];

    while (meio <= direita)
        temp[tmp_pos++] = Vetor[meio++];

    for (i = 0; i < num_elements; i++)
    {
        Vetor[direita] = temp[direita];
        direita--;
    }
}

static public void MergeSort_Recursive(int[] Vetor, int esquerda, int direita)
{
    int meio;

    if (direita > esquerda)
    {
        meio = (direita + esquerda) / 2; // divide o vetor ao meio
        MergeSort_Recursive(Vetor, esquerda, meio);
        MergeSort_Recursive(Vetor, (meio + 1), direita);

        DoMerge(Vetor, esquerda, (meio + 1), direita);
    }
}

//-----
public static void HeapSort(int[] Vetor)
{

```

```

        int tamanho_da_pilha = Vetor.Length;
        for (int p = (tamanho_da_pilha - 1) / 2; p >= 0; p--)
            MaxHeapify(Vetor, tamanho_da_pilha, p);

        for (int i = Vetor.Length - 1; i > 0; i--)
        {
            int temp = Vetor[i];
            Vetor[i] = Vetor[0];
            Vetor[0] = temp;

            tamanho_da_pilha--;
            MaxHeapify(Vetor, tamanho_da_pilha, 0);
        }
    }

private static void MaxHeapify(int[] Vetor, int tamanho_da_pilha, int indice)
{
    int esquerda = (indice + 1) * 2 - 1; //calcula os filhos
    int direita = (indice + 1) * 2;
    int maior = 0; //busca se maior elemento está a direita ou esquerda

    if (esquerda < tamanho_da_pilha && Vetor[esquerda] > Vetor[indice])
        maior = esquerda;
    else
        maior = indice;

    if (direita < tamanho_da_pilha && Vetor[direita] > Vetor[maior])
        maior = direita;

    if (maior != indice)
    { //faz troca e chama MaxHeapify (Raiz sempre estará o maior elemento)
        int temp = Vetor[indice];
        Vetor[indice] = Vetor[maior];
        Vetor[maior] = temp;

        MaxHeapify(Vetor, tamanho_da_pilha, maior);
    }
}

//-----
static public void selecao(int[] vector)
{
    int min, aux, k;
    for (int i = 0; i < vector.Length - 1; i++) // n-1
    {
        min = i;
        for (int j = i + 1; j < vector.Length; j++) // n-1
        {
            if (vector[j] < vector[min])
            {
                min = j;
            }
        }
        if (min != i)
        {
            aux = vector[min];
            vector[min] = vector[i];
            vector[i] = aux;
        }
    } // (n-1)*(n-1)
}
//-----

```

```

static public void insertionsort(int[] vector)
{
    int temp = 0;
    for (int i = 0; i < vector.Length - 1; i++)// n-1
    {
        for (int j = i + 1; j > 0; j--) // (n-1) + (n-2) ... + (n-n)
        {
            if (vector[j - 1] > vector[j])
            {
                temp = vector[j - 1];
                vector[j - 1] = vector[j];
                vector[j] = temp;
            }
        }
    } // 0((n-1)*(n-1)) --> n*n
}

//-----
static public void bubblesort(int[] vector)
{
    int temp = 0;

    for (int i = 0; i < vector.Length; i++)// n
    {
        for (int percorre_vetor = 0; percorre_vetor < vector.Length - 1;
            percorre_vetor++)// n-1
        {
            if (vector[percorre_vetor] > vector[percorre_vetor + 1])// troca
            {
                temp = vector[percorre_vetor + 1];
                vector[percorre_vetor + 1] = vector[percorre_vetor];
                vector[percorre_vetor] = temp;
            }
            // nao tem troca
        }
    } // 0(n*(n-1)) --> 0 (n*n)
}

//-----
static public void RandomNumber(int range, int quantidades_de_numeros, int[] vector)
{
    Random teste = new Random(521);
    // semente usada para repetirmos o mesmo teste

    for (int i = 0; i < quantidades_de_numeros; i++)
    {
        vector[i] = teste.Next(0, range);
    }
}

static public void imprimir(int[] vector)
{
    for (int i = 0; i < vector.Length; i++)// n

```

```

        {
            Console.WriteLine(vector[i]);
        }
    }

static public void vetor_decrescente(int[] vector, int[] vetor_decrescente)
{
    int k = vector.Length-1;

    for (int i = 0; i < vector.Length; i++)// n/2 ou (n+1)/2
    {
        if(i<=k)
        {
            vetor_decrescente[i] = vector[k];
            vetor_decrescente[k] = vector[i];
            k--;
        }
        else // quando i e maior que k (cessa o processo)
        {
            i = vector.Length+1;
        }
    }
}

static void Main(string[] args)
{
    var stopwatch = new Stopwatch();

    int quantidade_de_numero = 5000; // valor inicial

    for (int i = 0; i < 25 ; i++)// teste para analise
    {
        int numeros_de_casa = 10000;
        // números de casas do números varia de 0 a 9999
        quantidade_de_numero = quantidade_de_numero+500;
        int[] vector0 = new int[quantidade_de_numero];
        int[] vector1 = new int[quantidade_de_numero];
        int[] vector2 = new int[quantidade_de_numero];
        int[] vector3 = new int[quantidade_de_numero];
        int[] vector4 = new int[quantidade_de_numero];
        int[] vector5 = new int[quantidade_de_numero];
        int[] vector6 = new int[quantidade_de_numero];
        int[] vector8 = new int[quantidade_de_numero];

        RandomNumber(numeros_de_casa, quantidade_de_numero, vector0);
        // gera os numeros randomicos
        QuickSort(vector0, 0, vector0.Length - 1); //ordena

        vetor_decrescente(vector0, vector1); // gera copias
        vetor_decrescente(vector0, vector2); // dos vetores
    }
}

```

```

vetor_decrescente(vector0, vector3); // vector0
vetor_decrescente(vector0, vector4);
vetor_decrescente(vector0, vector5);
vetor_decrescente(vector0, vector6);

WriteLine($"");
WriteLine($"Vetor de tamanho {quantidade_de_numero} teste {i + 1}");
WriteLine($"");

/////-----
stopwatch.Start(); // tempo inicio
bubblesort(vector1); // processamento
stopwatch.Stop(); // tempo final
WriteLine($"Tempo do bubblesort :{stopwatch.ElapsedMilliseconds}");
stopwatch.Reset();
/////-----
stopwatch.Start(); // tempo inicio
insertionsort(vector2);
stopwatch.Stop(); //tempo final
WriteLine($"Tempo do insertionsort: {stopwatch.ElapsedMilliseconds}");
stopwatch.Reset();
/////-----
stopwatch.Start(); // tempo inicio
selecao(vector3);
stopwatch.Stop(); //tempo final
WriteLine($"Tempo do selecao: {stopwatch.ElapsedMilliseconds}");
stopwatch.Reset();
/////-----
stopwatch.Start(); // tempo inicio
QuickSort(vector4, 0, vector4.Length - 1);
stopwatch.Stop(); //tempo final
WriteLine($"Tempo do QuickSort: {stopwatch.ElapsedMilliseconds}");
stopwatch.Reset();
/////-----
stopwatch.Start(); // tempo inicio
HeapSort(vector5);
stopwatch.Stop(); // tempo final
WriteLine($"Tempo do HeapSort: {stopwatch.ElapsedMilliseconds}");
stopwatch.Reset();
/////-----
stopwatch.Start(); // tempo inicio
MergeSort_Recursive(vector6, 0, vector6.Length - 1);
stopwatch.Stop(); //tempo final
WriteLine($"Tempo do MergeSort_Recursive: {stopwatch.ElapsedMilliseconds}");
stopwatch.Reset();

}

WriteLine($"");
WriteLine($" ---- Fim da analise ----- ");

Console.ReadKey();

}
}
}

```