UFRN
UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO

Disciplina: Algoritmos e Estruturas de Dados I (EDI)
Professor: Eduardo de Lucena Falcão
Exercício sobre Algoritmos de Ordenação
ALUNO: Teophilo Vitor de Carvalho Clemente
MATRÍCULA: 20190080555

1) Programe na linguagem C cada um dos seguintes algoritmos de ordenação: **(1.5)**

a) SelectionSort (in-place)

```
int* selectionSort(int* vetor, int tam){
	for(int i = 0; i < tam; i++){
		int menor = i;
		for(int j = i; j < tam; j++){
			if(vetor[j] < vetor[menor]){
				menor = j;
			}
		}
		//Realiza a troca do elemento
		int aux = vetor[i];
		vetor[i] = vetor[menor];
		vetor[menor] = aux;
	}
	return vetor;
}
```

b) BubbleSort

```
int* bubbleSort(int* vetor, int tam){
	for(int i = 0; i < tam-1; i++){
		for(int j = 0; j < tam-i-1; j++){
			if(vetor[j] > vetor[j+1]){
				int aux = vetor[j]; //swap
				vetor[j] = vetor[j+1];
				vetor[j+1] = aux;
			}
		}
	}
	return vetor;
}
```

c) InsertionSort (in-place)

```
int* insertionSort(int* vetor, int tam){
      for(int i = 1; i < tam; i++){
            for(int j = i; j>0 && vetor[j]<vetor[j-1]; j--){
                  int aux = vetor[j];
                  vetor[j] = vetor[j-1];
                  vetor[j-1] = aux;
            }
      }
      return vetor;
}
```

d) MergeSort

```
void merge(int* v, int tamv, int* v1, int tamv1, int* v2,
int tamv2){
    int indexv = 0;
    int indexv1 = 0;
    int indexv2 = 0;
    while(indexv1 < tamv1 && indexv2 < tamv2){
      int menor;
      if(v1[indexv1] <= v2[indexv2]){
            menor = v1[indexv1];
            indexv1++;
      }else{
            menor = v2[indexv2];
            indexv2++;
      }
      v[indexv] = menor;
      indexv++;
    }
       //pode haver elementos no vetor v1 que não foram
copiados para v
    while(indexv1 < tamv1){
        v[indexv] = v1[indexv1];
        indexv++;
        indexv1++;
    }
       //pode haver elementos no vetor v2 que não foram
copiados para v
    while(indexv2 < tamv2){
        v[indexv] = v2[indexv2];
        indexv++;
        indexv2++;
    }
}
void mergeSort(int* v, int tamv){
    if(tamv > 1){
```

```
        //primeiro quebramos o vetor em 2 vetores menores
        int meio = tamv/2;

        int tamv1 = meio;
        int* v1 = (int*)malloc(tamv1*sizeof(int));
        for(int i = 0; i < meio; i++){
            v1[i] = v[i];
        }

        int tamv2 = tamv-meio;
        int* v2 = (int*)malloc(tamv2*sizeof(int));
        for(int i = meio; i < tamv; i++){
            v2[i-meio] = v[i];
        }

        mergeSort(v1,tamv1);
        mergeSort(v2,tamv2);
        merge(v,tamv,v1,tamv1,v2,tamv2);

        free(v1);
        free(v2);
    }
}
```

e) QuickSort (com randomização de pivô)

```
void swap(int* v, int i, int j){
    int temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}

int particiona(int* v, int ini, int fim){
    int pIndex = ini;

    //a seguir parte do codigo que realiza a randomizacao
    //hora atual do sistema
    srand(time(NULL));
    //teremos uma posicao aleatoria entre inicio e fim
    int posAleatoria = ini+(rand()%(fim+1-ini));
    //realizamos o swap para que posAleatoria seja o fim
    swap(v, posAleatoria, fim);

    int pivo = v[fim];
    for(int i = ini; i < fim; i++){
        if(v[i] <= pivo){
            swap(v,i,pIndex);
            pIndex++;
        }
```

```
    }
    swap(v,pIndex,fim);
    return pIndex;
}

void quickSort(int* v, int ini, int fim){
    if(fim>ini){
        int indexPivo = particiona(v,ini,fim);
        quickSort(v,ini,indexPivo-1);
        quickSort(v,indexPivo+1,fim);
    }
}
```

f) CountingSort

```
int* countingSort (int* v, int tamv){
    int maior = v[0];
    for (int i = 0; i < tamv; i++){
        if (v[i] > maior)
            maior = v[i];
    }
    //Definimos o maior valor presente no vetor no for
    int tamVocorren = maior;
    //Definição do vetor preenchido com zeros
    int*vOcorren=(int*)calloc(tamVocorren,sizeof(int));
    for (int i = 0; i < tamv;i++){
        vOcorren[v[i]]++;
    }
    //Preenchimento do vetor de soma cumulativa
    for (int i = 1; i <= tamVocorren; i++){
        vOcorren[i] += vOcorren[i-1];
    }
    //Definição do vetor ordenado
    int* vOrdenado = (int*)malloc(tamVocorren*sizeof(int));
    //Preenchendo o vetor ordenado
    for (int i = tamv-1; i >= 0; i--) {
      vOrdenado[vOcorren[v[i]] - 1] = v[i];
      vOcorren[v[i]]--;
    }
    //preenchendo o vetor inicial de forma ordenada
    for (int i = 0; i < tamv; i++) {
        v[i] = vOrdenado[i];
    }
    return v;
}
```

2) Ilustre, em detalhes, o funcionamento dos seguintes algoritmos com os seguintes vetores. **(6.0)**
   - **aleatório** = [3, 6, 2, 5, 4, 3, 7, 1, 10^9]
   - **decrescente** = [7, 6, 5, 4, 3, 3, 2, 1]
   - **crescente**=[1,2,3,3,4,5,6,7]

a. SelectionSort (in-place)

i. vetores para ilustrar: **aleatório**

3	6	2	5	4	3	7	1	10^9
0	1	2	3	4	5	6	7	8

Para iniciarmos o algoritmo de ordenação temos que ter em mente como funciona o SelectionSort, ele ordena o vetor trocando a posição do menor elemento do vetor, de modo que a cada iteração os menores elementos estejam nas primeiras posições do vetor. Esse processo de troca é feito até que o penúltimo menor elemento do vetor esteja na penúltima posição, pois assim, necessariamente o maior valor estará na última posição, ou seja o vetor estará ordenado.

1º Iteração: Menor valor deve estar na posição 0, então varremos o vetor inteiro (posição 0 até 8) até encontrar o menor valor, seria o 1 e está na posição 7 e fazemos o swap entre posição 0 e 7, de modo que obtemos o seguinte vetor:

1	6	2	5	4	3	7	3	10^9

2º Iteração: Menor valor deve estar na posição 1, então varremos o vetor da posição 1 até 8 até encontrar o menor valor, seria o 2 e está na posição 2 e fazemos o swap entre posição 1 e 2, de modo que obtemos o seguinte vetor:

1	2	6	5	4	3	7	3	10^9

3º Iteração: Menor valor deve estar na posição 2, então varremos o vetor da posição 2 até 8 até encontrar o menor valor, seria o 3 e está na posição 5 e fazemos o swap entre posição 2 e 5, de modo que obtemos o seguinte vetor:

1	2	3	5	4	6	7	3	10^9

4º Iteração: Menor valor deve estar na posição 3, então varremos o vetor da posição 3 até 8 até encontrar o menor valor, seria o 3 e está na posição 7 e fazemos o swap entre posição 3 e 7, de modo que obtemos o seguinte vetor:

1	**2**	**3**	**3**	**4**	**6**	7	**5**	10^9

5º Iteração: Menor valor deve estar na posição 4, então varremos o vetor da posição 4 até 8 até encontrar o menor valor, seria o 4 e está na posição 4 então não fazemos swap, de modo que obtemos o seguinte vetor:

1	**2**	**3**	**3**	**4**	**6**	7	**5**	10^9

6º Iteração: Menor valor deve estar na posição 5, então varremos o vetor da posição 5 até 8 até encontrar o menor valor, seria o 5 e está na posição 7 então fazemos o swap entre 5 e 7, de modo que obtemos o seguinte vetor:

1	**2**	**3**	**3**	**4**	**5**	7	**6**	10^9

7º Iteração: Menor valor deve estar na posição 6, então varremos o vetor da posição 6 até 8 até encontrar o menor valor, seria o 6 e está na posição 7 então fazemos o swap entre 6 e 7, de modo que obtemos o seguinte vetor:

1	**2**	**3**	**3**	**4**	**5**	**6**	**7**	10^9

8º Iteração: Menor valor deve estar na posição 7, então varremos o vetor da posição 7 até 8 até encontrar o menor valor, seria o 7 está na posição 7 então não fazemos o swap, de modo que obtemos o seguinte vetor ordenado:

1	**2**	**3**	**3**	**4**	**5**	**6**	7	10^9

b. BubbleSort

i. vetores para ilustrar: **aleatório**

3	6	2	5	4	3	7	1	10^9
0	1	2	3	4	5	6	7	8

Para iniciarmos o algoritmo de ordenação temos que ter em mente como funciona o BubbleSort, ele ordena um vetor varrendo da esquerda para direita, em cada varredura dessa, dois elementos vizinhos são comparados, de modo que, se o da esquerda for maior do que o da direita, então eles trocaram de posição, isso acontece até que o maior valor esteja na última posição. Após é feita outra varredura que coloca o segundo maior elemento na penúltima posição e assim por diante até que o vetor esteja todo ordenado.

1º Passagem: O vetor será varrido de modo que o último elemento seja o maior:

3	6	2	5	4	3	7	1	10^9

Não faz nada ↑

3	6	2	5	4	3	7	1	10^9

Troca 6 com 2 ↑

3	2	6	5	4	3	7	1	10^9

Troca 6 com 5 ↑

3	2	5	6	4	3	7	1	10^9

Troca 6 com 4 ↑

3	2	5	4	6	3	7	1	10^9

Troca 6 com 3 ↑

3	2	5	4	3	6	7	1	10^9

Não faz nada↑

3	2	5	4	3	6	7	1	10^9

Troca 7 com 1↑

3	2	5	4	3	6	1	7	10^9

Não faz nada↑

3	2	5	4	3	6	1	7	10^9

Maior valor no fim↑

2º Passagem: Temos que o maior valor agora está na última posição, nessa segunda passagem vamos percorrer somente do primeiro elemento até o penúltimo, de modo que este seja o segundo maior.

3	2	5	4	3	6	1	7	10^9

Troca 3 com 2↑

2	3	5	4	3	6	1	7	10^9

Não faz nada↑

2	3	5	4	3	6	1	7	10^9

Troca 5 com 4↑

2	3	4	5	3	6	1	7	10^9

Troca 5 com 3↑

2	3	4	3	5	6	1	7	10^9

Não faz nada↑

2	3	4	3	5	6	1	7	10^9

Troca 6 e 1↑

2	3	4	3	5	1	6	7	10^9

Não faz nada↑

2	3	4	3	5	1	6	7	10^9

Segundo maior na penúltima posição↑

3º Passagem: Temos agora que o segundo maior valor está na penúltima posição, assim essa terceira passada vai percorrer somente do primeiro elemento até o antepenúltimo, de modo que este seja o terceiro maior.

2	3	4	3	5	1	6	7	10^9

Não faz nada↑

2	3	4	3	5	1	6	7	10^9

Não faz nada↑

2	3	4	3	5	1	6	7	10^9

Troca 4 com 3↑

2	3	3	4	5	1	6	7	10^9

Não faz nada↑

2	**3**	**3**	**4**	**5**	**1**	**6**	7	10^9

Troca 5 com 1↑

2	**3**	**3**	**4**	**1**	**5**	**6**	7	10^9

Não faz nada↑

2	**3**	**3**	**4**	**1**	**5**	**6**	7	10^9

Terceiro maior na antepenúltima posição↑

Esse mesmo processo se repetirá até a 8 passagem, seguindo a mesma lógica, assim os oito maiores valores estão ordenados da última para a segunda posição, desse modo o valor na primeira posição é o menor, então o vetor já está completamente ordenado. Assim obtemos o seguinte vetor ordenado:

1	**2**	**3**	**3**	**4**	**5**	**6**	7	10^9

c. InsertionSort (in-place, melhor versão)

i. vetores para ilustrar: **aleatório**

3	**6**	**2**	**5**	**4**	**3**	7	**1**	10^9
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

Para iniciarmos o algoritmo de ordenação temos que ter em mente como funciona o InsertionSort, ele ordena um vetor da seguinte forma, dizemos que o primeiro elemento do vetor passa a ser a parte ordenada dele, enquanto os outros elementos constituem a parte não ordenada do vetor. A partir disso, passamos o primeiro elemento da parte não ordenada para a outra parte, de modo que se não estiver ordenado, devemos ordenar. Assim, repetiremos o procedimento até que todos elementos estejam na parte ordenada do vetor.

1º time: Definimos o 3 como a parte ordenada (em azul) e o resto como a não ordenada.

3	**6**	**2**	**5**	**4**	**3**	7	**1**	10^9

2º time: Passamos o 6 para a parte ordenada (azul) e como 6 não é menor que 3 temos [3,6] ordenados.

3	**6**	**2**	**5**	**4**	**3**	7	**1**	10^9

3º time: Passamos o 2 para a parte ordenada (azul) e como 2 é menor que 6 trocamos eles e temos [3,2,6] .

3	**2**	**6**	**5**	**4**	**3**	7	**1**	10^9

4º time: Como 2 é menor que 3, trocamos eles e temos [2,3, 6] ordenados.

2	**3**	**6**	**5**	**4**	**3**	7	**1**	10^9

5º time: Passamos o 5 para a parte ordenada (azul) e como 5 é menor que 6 trocamos eles e temos [2,3,5,6] ordenados.

2	**3**	**5**	**6**	**4**	**3**	7	**1**	10^9

6º time: Passamos o 4 para a parte ordenada (azul) e como 4 é menor que 6 trocamos eles e temos [2,3,5,4,6].

2	**3**	**5**	**4**	**6**	**3**	7	**1**	10^9

7º time: Como 4 é menor que 5, trocamos eles e temos [2,3,4,5,6] ordenados.

2	**3**	**4**	**5**	**6**	**3**	7	**1**	10^9

8º, 9º e 10º time: Passamos o 3 para a parte ordenada (azul) e como 3 é menor que 6, 5 e 4, faremos as 3 trocas e teremos [2,3,3,4,5,6] ordenados.

2	**3**	**3**	**4**	**5**	**6**	7	**1**	10^9

11º time: Passamos o 7 para parte ordenada (azul) e como 7 não é menor que 6, não faremos trocas e teremos [2,3,3,4,5,6,7] ordenados.

2	**3**	**3**	**4**	**5**	**6**	**7**	**1**	10^9

12º até 18º time: Passamos 1 para parte ordenada (azul) e como 1 é menor que todos os elementos que estão na parte ordenada ele será

trocado com todos, então faremos 7 trocas e teremos [1,2,3,3,4,5,6,7] ordenados.

1	2	3	3	4	5	6	7	10^9

19º time: Passamos 10^9 para parte ordenada e como ele não é menor que 7 não faremos troca e assim temos nosso vetor ordenado.

1	2	3	3	4	5	6	7	10^9

d. MergeSort

i. vetores para ilustrar: **decrescente**

7	6	5	4	3	3	2	1
0	1	2	3	4	5	6	7

Para iniciarmos o algoritmo de ordenação temos que ter em mente como funciona o MergeSort, ele ordena o vetor primeiro dividindo ele em dois vetores, os dois vetores novos são divididos em mais dois, e assim por diante até que não seja mais possível dividir. Desse modo, teremos vários vetores de 1 elemento e que por si só estão ordenados, então começamos a parte da junção e ordenação desses vetores, comparamos os elementos dos vetores que serão unidos, assim o menor valor vai para a primeira posição de um vetor de duas posições e o maior vai para o final, como resultado disso, temos um vetor de duas posições já ordenado, faremos o mesmo processo novamente com dois vetores de uma posição, e com isso temos mais um vetor de duas posições já ordenado. Com isso, pegamos os dois vetores de duas posições e faço o mesmo procedimento de comparação e ordenação, assim vamos obter agora um vetor de 4 posições já ordenado. Esse procedimento é repetido até que tenhamos o vetor inteiramente ordenado e consequentemente junto como no início**.**

Abaixo segue a ilustração do processo feito no Paint, nela podemos notar as etapas 1, 2 e 3 de divisão do vetor até chegar em vetores de 1 elemento ordenados. Após temos as etapas 4, 5 e 6 de junção e ordenação dos vetores e como produto final o vetor ordenado.

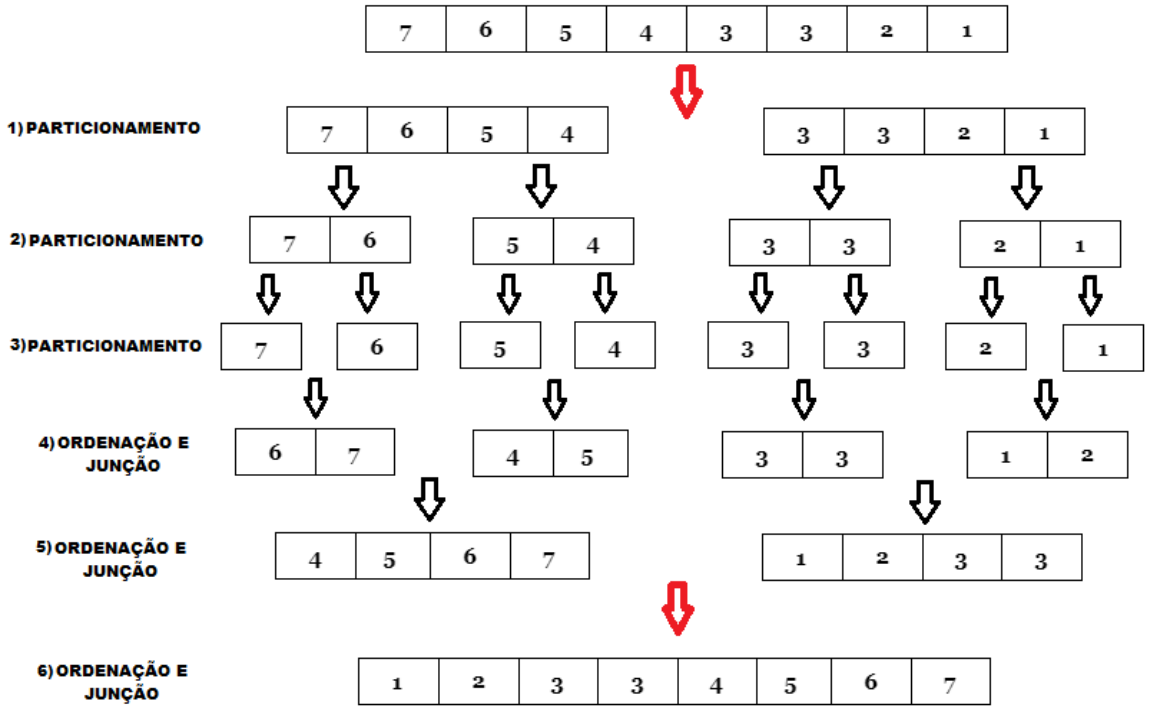

Figura 1 - MergeSort aplicado ao vetor.

e. QuickSort (**sem randomização de pivô**)

i. vetores para ilustrar: **crescente**

1	2	3	3	4	5	6	7
0	1	2	3	4	5	6	7

Para iniciarmos o algoritmo de ordenação temos que ter em mente como funciona o QuickSort sem randomização de pivô, ele ordena o vetor, primeiro escolhendo um elemento como pivô, nesse caso o último elemento do vetor, após isso, colocamos todos os elementos menores que o pivô à sua esquerda e todos os maiores à sua direita, com isso ficamos com dois subvetores e o pivô (lembrando que para isso não são criados novos vetores). Como os subvetores estão desordenados vamos aplicar o mesmo processo anterior de escolher um pivô e depois separar em dois subvetores. Desse modo, vemos que teremos diversos subvetores, então aplicamos recursivamente esse processo, até que todos os elementos estejam em ordem. Como nosso vetor já está ordenado veremos que ele somente formará subvetores a esquerda.

1º passo: Definimos o 7 como o pivô (vermelho) e separamos o subvetor de menores (verde).

1	2	3	3	4	5	6	7

2º passo: Definimos o 6 como o pivô (vermelho) do subvetor anterior [1,2,3,3,4,5,6] e separamos o subvetor de menores (verde).

1	2	3	3	4	5	6	7

3º passo: Definimos o 5 como o pivô (vermelho) do subvetor anterior [1,2,3,3,4,5]e separamos o subvetor de menores (verde).

1	2	3	3	4	5	6	7

4º passo: Definimos o 4 como o pivô (vermelho) do subvetor anterior [1,2,3,3,4] e separamos o subvetor de menores (verde).

1	2	3	3	4	5	6	7

5º passo: Definimos o 3 como o pivô (vermelho) do subvetor anterior [1,2,3,3] e separamos o subvetor de menores (verde).

1	2	3	3	4	5	6	7

6º passo: Definimos o 3 como o pivô (vermelho) do subvetor anterior [1,2,3]e separamos o subvetor de menores (verde).

1	2	3	3	4	5	6	7

7º passo: Definimos o 2 como o pivô (vermelho) do subvetor anterior [1,2]e separamos o subvetor de menores (verde).

1	2	3	3	4	5	6	7

8º passo: Temos como resultado final nosso vetor ordenado após a aplicação do método, como antes comentado, como esse vetor já estava ordenado de forma crescente, durante a aplicação do método não tivemos a formação do subvetor à direita (os valores maiores que o pivô), visto que, sempre que definimos o novo pivô todos os elementos do subvetor eram menores que ele.

1	2	3	3	4	5	6	7

f. CountingSort

i. vetores para ilustrar: **aleatório**

3	6	2	5	4	3	7	1	10^9
0	1	2	3	4	5	6	7	8

Para iniciarmos o algoritmo de ordenação temos que ter em mente como funciona o CountingSort, ele é um algoritmo que organizará os elementos de array pela contagem de ocorrências de cada elemento dentro do vetor, então a partir da frequência deles é baseado em vetor auxiliar que deve ser do tamanho n+1, onde n é maior elemento presente no vetor, esse vetor será preenchido com zeros inicialmente (calloc) e depois será preenchido com a respectiva quantidade de cada elemento na posição, por exemplo se no nosso vetor, temos 2 números 4, na posição quatro do vetor teremos o valor 2 e se não houver nenhum elemento da posição o valor será 0. Após isso faremos a soma cumulativa dos elementos e por fim, faremos o encontro do índice e assim deixar o vetor ordenado.

1º passo: Descobrimos o maior elemento que é 10^9 e criamos o nosso vetor auxiliar de tamanho 10^9+1.

0	0	0	0	0	0	0	0	...	0
0	1	2	3	4	5	6	7	...	10^9+1

2º passo: Vamos armazenar a quantidade de cada elemento em seu respectivo índice do vetor criado anteriormente.

0	1	1	2	1	1	1	1	...	1
0	1	2	3	4	5	6	7	...	10^9+1

3º passo: Agora vamos fazer a soma cumulativa dos elementos e armazenar na posição, ou seja, o elemento da frente armazenará o resultado da soma anterior somado ao valor que está na sua posição e assim sucessivamente.

0	1	2	4	5	6	7	8	...	9
0	1	2	3	4	5	6	7	...	10^9+1

4º passo: Agora vamos fazer o processo de encontro do índice correto de cada elemento e assim tê-lo ordenado. Para isso, vamos utilizar do vetor original e do que obtivemos anteriormente e será feita a busca como ilustrada abaixo.

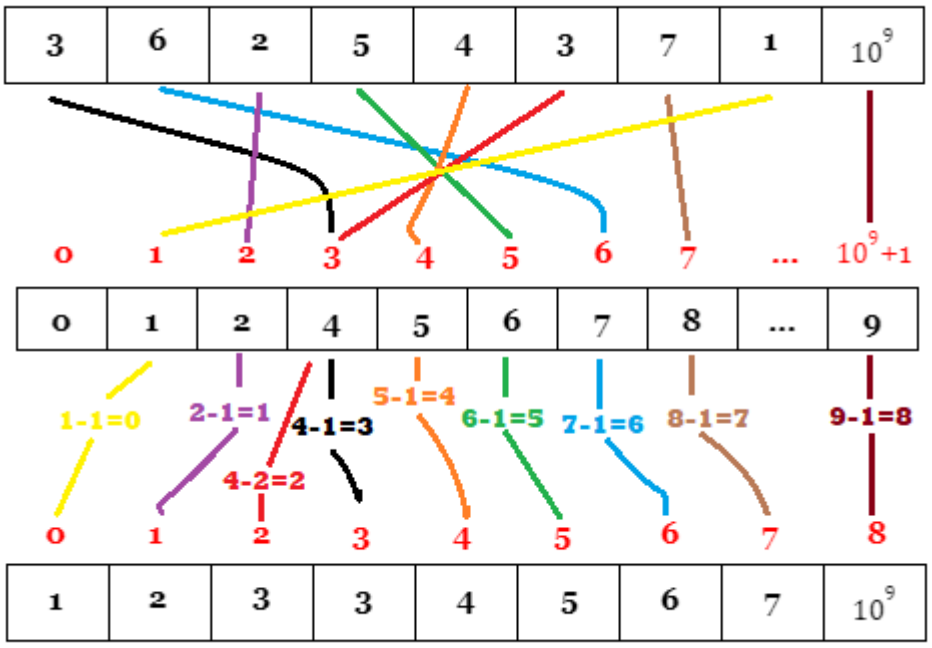

Figura 2 - Passo 5 do CountingSort.

Como busquei ilustrar na imagem, tomamos como referência o vetor original, no caso do primeiro elemento, o 3, ele irá apontar para o índice 3 do vetor de soma cumulativa, nele faremos o operação do elemento na posição - 1, assim 4-1=3, ou seja, a posição do elemento 3 no vetor ordenado deve ser o índice 3. O 6 irá apontar para o índice 6 do vetor de soma cumulativa, nele faremos o operação do elemento na posição - 1, assim 6-1=5, ou seja, a posição do elemento 6 no vetor ordenado deve ser o índice 5 e assim sucessivamente como estão demonstrados os cálculos de todas as posições na imagem, com uma ressalva para o segundo 3, (em vermelho) neste caso como ele já apareceu a operação fica posição - 2. Com isso, após aplicar o algoritmo para todas as posições do vetor ao final obteremos ele ordenado.

Além disso, vemos que as milhares de posições ocupadas com zero no vetor de soma cumulativa não interferiram no cálculo final, porém demandaram a criação de um vetor auxiliar de 10^9 índices o que prejudica o desempenho do algoritmo.

3) A seguir são apresentados 5 fatos sobre algoritmos de ordenação. Planeje, execute experimentos, e apresente resultados que evidenciem cada afirmação. **(2.5)**

A seguir serão apresentados os prints dos testes feitos para cada algoritmo:

Selection Sort com vetor de 1000↓

```
-------------------------------------------------------------------------------------------
Tempo de execucao c/ vetor ordenado = 0.010000 s
-------------------------------------------------------------------------------------------
Tempo de execucao c/ vetor aleatorio = 0.012000 s
-------------------------------------------------------------------------------------------
Tempo de execucao c/ com vetor ordenado decrescente = 0.013000 s
-------------------------------------------------------------------------------------------
PS C:\Users\bolt\Downloads>
```

Selection Sort com vetor de 10000↓

```
-------------------------------------------------------------------------------------------
Tempo de execucao c/ vetor ordenado = 1.150000 s
-------------------------------------------------------------------------------------------
Tempo de execucao c/ vetor aleatorio = 1.153000 s
-------------------------------------------------------------------------------------------
Tempo de execucao c/ com vetor ordenado decrescente = 2.102000 s
-------------------------------------------------------------------------------------------
PS C:\Users\bolt\Downloads>
```

Bubble Sort com vetor de 1000↓

```
-------------------------------------------------------------------------------------------
Tempo de execucao c/ vetor ordenado = 0.026000 s
-------------------------------------------------------------------------------------------
Tempo de execucao c/ vetor aleatorio = 0.060000 s
-------------------------------------------------------------------------------------------
Tempo de execucao c/ com vetor ordenado decrescente = 0.068000 s
-------------------------------------------------------------------------------------------
PS C:\Users\bolt\Downloads>
```

Bubble Sort com vetor de 10000↓

```
-------------------------------------------------------------------------------------------
Tempo de execucao c/ vetor ordenado = 2.500000 s
-------------------------------------------------------------------------------------------
Tempo de execucao c/ vetor aleatorio = 4.036000 s
-------------------------------------------------------------------------------------------
Tempo de execucao c/ com vetor ordenado decrescente = 4.260000 s
-------------------------------------------------------------------------------------------
PS C:\Users\bolt\Downloads>
```

Insertion Sort com vetor de 1000↓

```
--------------------------------------------------------------------------------
Tempo de execucao c/ vetor ordenado = 0.000000 s
--------------------------------------------------------------------------------
Tempo de execucao c/ vetor aleatorio = 0.032000 s
--------------------------------------------------------------------------------
Tempo de execucao c/ com vetor ordenado decrescente = 0.064000 s
--------------------------------------------------------------------------------
PS C:\Users\bolt\Downloads>
```

Insertion Sort com vetor de 10000↓

```
--------------------------------------------------------------------------------
Tempo de execucao c/ vetor ordenado = 0.001000 s
--------------------------------------------------------------------------------
Tempo de execucao c/ vetor aleatorio = 2.309000 s
--------------------------------------------------------------------------------
Tempo de execucao c/ com vetor ordenado decrescente = 4.394000 s
--------------------------------------------------------------------------------
PS C:\Users\bolt\Downloads>
```

Merge Sort com vetor de 1000↓

```
--------------------------------------------------------------------------------
Tempo de execucao c/ vetor ordenado = 0.002000 s
--------------------------------------------------------------------------------
Tempo de execucao c/ vetor aleatorio = 0.003000 s
--------------------------------------------------------------------------------
Tempo de execucao c/ com vetor ordenado decrescente = 0.003000 s
--------------------------------------------------------------------------------
PS C:\Users\bolt\Downloads>
```

Merge Sort com vetor de 10000↓

```
--------------------------------------------------------------------------------
Tempo de execucao c/ vetor ordenado = 0.019000 s
--------------------------------------------------------------------------------
Tempo de execucao c/ vetor aleatorio = 0.027000 s
--------------------------------------------------------------------------------
Tempo de execucao c/ com vetor ordenado decrescente = 0.022000 s
--------------------------------------------------------------------------------
PS C:\Users\bolt\Downloads>
```

Quick sort com vetor de 1000↓

```
--------------------------------------------------------------------------------
Tempo de execucao c/ vetor ordenado = 0.001000 s
--------------------------------------------------------------------------------
Tempo de execucao c/ vetor aleatorio = 0.001000 s
--------------------------------------------------------------------------------
Tempo de execucao c/ com vetor ordenado decrescente = 0.001000 s
--------------------------------------------------------------------------------
PS C:\Users\bolt\Downloads>
```

Quick Sort com vetor de 10000↓

```
----------------------------------------------------------------------------------------
Tempo de execucao c/ vetor ordenado = 0.031000 s
----------------------------------------------------------------------------------------
Tempo de execucao c/ vetor aleatorio = 0.057000 s
----------------------------------------------------------------------------------------
Tempo de execucao c/ com vetor ordenado decrescente = 0.041000 s
----------------------------------------------------------------------------------------
PS C:\Users\bolt\Downloads>
```

Couting Sort com vetor de 100↓

```
----------------------------------------------------------------------------------------
Tempo de execucao c/ vetor ordenado = 0.002000 s
----------------------------------------------------------------------------------------
Tempo de execucao c/ vetor aleatorio = 0.002000 s
----------------------------------------------------------------------------------------
Tempo de execucao c/ com vetor ordenado decrescente = 0.002000 s
----------------------------------------------------------------------------------------
PS C:\Users\bolt\Downloads>
```

Counting Sort com vetor de 1000↓

```
----------------------------------------------------------------------------------------
Tempo de execucao c/ vetor ordenado = 0.005000 s
----------------------------------------------------------------------------------------
Tempo de execucao c/ vetor aleatorio = 0.006000 s
----------------------------------------------------------------------------------------
Tempo de execucao c/ com vetor ordenado decrescente = 0.004000 s
----------------------------------------------------------------------------------------
PS C:\Users\bolt\Downloads>
```

A.

Pelos testes vemos que realmente para vetores pequeno, no qual usei vetores de 1000 índices todos os algoritmos apresentaram tempo de resposta na ordem entre 0.002... até 0.005..., que apesar de haver diferenças entre os casos de vetores aleatórios, ordenados e decrescentes, a diferença é mínima, de modo que, mesmo o ordenado decrescentemente quando utilizamos algum desses algoritmos de ordenação, que seria o pior caso, as varreduras foram entre 10^5 e 10^6 varreduras, o que pode ser considerado bem rápido o tempo de execução.

B.

Pelos teste, para vetores grandes, no qual usei 10000 índices, o algoritmo teve diferenças notáveis há depender da disposição dos elementos, os casos específicos comento abaixo:

Para o Selection Sort que é O(n²) para o melhor e pior caso, os testes mostraram que o melhor tempo de execução foi quando o vetor estava com os elementos ordenados, visto que, ele fará a varredura, porém sem fazer trocas. O segundo melhor desempenho foi quando o vetor estava com os elementos de forma aleatória, com um desempenho semelhante ao primeiro. O pior caso, foi o do vetor ordenado de forma decrescente, este caso levou bem mais tempo que os demais, isso acontece devido a precisarmos varrer o vetor inteiro e em todos os casos fazermos o swap o que demandou praticamente o dobro de varreduras dos outros casos.

Para o Bubble Sort que é O(n) para o melhor caso e O(n²) para o pior. O melhor caso também foi com o vetor ordenado, seu tempo de execução foi bastante rápido. O segundo melhor foi com o vetor ordenado de forma aleatória, porém com uma diferença grande em relação ao primeiro. O pior caso foi com o vetor ordenado de forma decrescente, com ainda mais diferença em relação ao melhor. Uma observação que podemos notar é que para o Bubble, ele apresenta um bom resultado somente com o vetor já ordenado, nos outros dois casos a sua performance fica entre as piores dos algoritmos testados.

Para o Insertion Sort que é O(n) para o melhor caso e O(n²) para o pior. Os testes mostraram o melhor desempenho quando o vetor está ordenado, como esperado. Foi observado o segundo melhor no vetor aleatório e pior no decrescente com uma diferença considerável ao melhor caso.

Para o Merge Sort que é O(nlog(n)) para ambos os casos. Seu desempenho se mostrou superior aos algoritmos anteriores, o que já era esperado devido a ser O(nlog(n)). Numa média ele foi entre 10/20 vezes mais rápido que os anteriores em seus casos de teste. Um ponto a ser observado é que ele é out-of-place, assim se faz necessária a criação de vetores auxiliares, já que ele só começa a ordenação após ter vários vetores unitários, neste caso, devido a isso ele necessita de bem mais memória para a criação de tais vetores.

Para o Quick Sort temos a possibilidade dele com randomização de pivô e sem, sendo ele O(nlog(n)) para o melhor caso e O(n²) para o pior caso. No segundo melhor caso ele também foi bastante rápido, semelhante ao desempenho do Merge Sort, porém ainda inferior. Para o terceiro o caso os resultados foram semelhantes e bons, estando junto com o Merge como um dos mais rápidos comparados aos anteriores, com isso atestamos o esperado, que a randomização otimiza o algoritmo de forma bem satisfatória.

Para o Counting Sort que tem complexidade O(n+k) para ambos os casos. Foi observado o melhor desempenho em vetores com valores menores, visto que, o Counting se utiliza da criação de um vetor com o tamanho do maior valor presente no vetor, como foi o caso da questão 2 que ele precisou criar um vetor de 1 bilhão de índices e isso afeta significativamente o desempenho. Tendo seu melhor caso o ordenado decrescente, o segundo o ordenado e o pior o vetor aleatório. Desse modo, apesar de sua complexidade ser a mesma para ambos os casos, a questão do tamanho do vetor interfere bastante em seu desempenho.

C. MergeSort tem sempre um desempenho muito bom, independente das disposição dos elementos no vetor.

O Merge Sort realmente apresentou um excelente desempenho em todos os casos nos testes feitos, a diferença entre eles foi muito pouca, realmente comprovando que ele é O(nlog(n)) como apresentei na explicação da questão anterior (B), sendo um dos mais eficientes entre os já apresentados.

D. O pior caso do Quicksort é com o vetor ordenado de forma crescente/decrescente. O Quicksort com randomização de pivô resolve esse mau desempenho.

Realmente os teste comprovaram que o uso da randomização melhorou bastante o desempenho do algoritmo, no pior caso, no Quick sem a randomização foi quase o dobro do tempo do melhor caso. Já quando aplicamos o código com randomização os três casos apresentaram resultados semelhantes, mostrando o seu rendimento superior.

E. Explique quando o CountingSort tem bom desempenho e quando tem mau desempenho mostrando os resultados através dos experimentos.

Essa questão também expliquei na letra B, os seus casos tiveram resultados próximos, tendo seu melhor caso o ordenado decrescente, o segundo o ordenado e o pior o vetor aleatório. Porém através do visto na questão 2 vemos que a maior problema do Counting é o vetor auxiliar que ele cria, pois o ele será do tamanho do maior valor do vetor, e em um caso que temos valores muito grandes, o vetor será enorme e o processo com ele também, consequentemente isso afetará o desempenho do algoritmo.