AVALIAÇÃO UNIDADE II

Aluno: Rychardson Ribeiro de Souza

Disciplina: Algoritmos e Estruturas de Dados I

Professor: Eduardo de Lucena Falcão

QUESTÃO 1

Algoritmos de Ordenação

ITEM A - Q1

```
#include <stdio.h>
#include <stdlib.h>


void selectionSort(int* v, int tamanho){
   int i, aux, j, menor;

   for(i=0;i<tamanho;i++){
      menor = v[i];
         for(j=i;j<tamanho;j++){
            if(v[j]<menor){
               aux=menor;
               menor=v[j];
               v[j]=aux;
            }
         }
   v[i]=menor;
   }
}

int main () {

   int tamanho, *v;
   printf("Entre com o tamanho: ");
   scanf("%d", &tamanho);
   v = (int*) malloc(tamanho*sizeof(int));
      for(int i=0;i<tamanho;i++){
            scanf("%d", &v[i]);
      }

      selectionSort(v, tamanho);
      printf("\nSeu vetor ordenado sera: [ ");
      for(int i=0;i<tamanho;i++){
```

```
            printf("%d ", v[i]);
        }
        printf("]");

return 0;
}
```

ITEM B - Q1

```
#include <stdio.h>
#include <stdlib.h>

void bubbleSort(int* v, int tamanho){
    int i, aux, j, cont=0;
     for(i=0;i<tamanho;i++){
            for(j=0;j<tamanho-1;j++){
                if(v[j]>v[j+1]){
                    aux=v[j+1];
                    v[j+1]=v[j];
                    v[j]=aux;
                    cont++;
                }
            }
            if(cont==0){
                break;
            }
            cont=0;
    }
}

int main () {

    int tamanho, *v;
    printf("Entre com o tamanho: ");
    scanf("%d", &tamanho);
    v = (int*) malloc(tamanho*sizeof(int));
        for(int i=0;i<tamanho;i++){
                scanf("%d", &v[i]);
        }

        bubbleSort(v, tamanho);
        printf("\nSeu vetor ordenado sera: [ ");
        for(int i=0;i<tamanho;i++){
            printf("%d ", v[i]);
        }
        printf("]\n");
```

```
return 0;
}
```

ITEM C - Q1

```
#include <stdio.h>
#include <stdlib.h>

void insertionSort(int* v, int tamanho){
   int i, aux, j;
   for(i=0;i<tamanho;i++){
         for(j=i;j>0;j--){
            if(v[j-1]>v[j]){
               aux=v[j];
               v[j]=v[j-1];
               v[j-1]=aux;
            }else{
               break;
            }
         }
   }
}

int main () {

   int tamanho, *v;
   printf("Entre com o tamanho: ");
   scanf("%d", &tamanho);
   if(tamanho<=1){
      return 0;
   }
   v = (int*) malloc(tamanho*sizeof(int));
      for(int i=0;i<tamanho;i++){
            scanf("%d", &v[i]);
      }

      insertionSort(v, tamanho);
      printf("\nSeu vetor ordenado sera: [ ");
      for(int i=0;i<tamanho;i++){
         printf("%d ", v[i]);
      }
      printf("]\n");

return 0;

}
```

ITEM D - Q1

```
#include <stdio.h>
#include <stdlib.h>

void merge(int *v, int inicio, int meio, int fim) {

    int ini1 = inicio, ini2 = meio+1, iniAux = 0, tamanho = fim-inicio+1;

    int *vAux;

    //alocar o tamanho para o meu vetor auxiliar

    vAux = (int*)malloc(tamanho * sizeof(int));

    //inicio dos laços de repetição para ordenar meu vetor
    while(ini1 <= meio && ini2 <= fim){
        if(v[ini1] < v[ini2]) {
            vAux[iniAux] = v[ini1];
            ini1++;
        } else {
            vAux[iniAux] = v[ini2];
            ini2++;
        }
        iniAux++;
    }

    //repetição para caso tenha sobrado elementos na primeira
    //metade do vetor
    while(ini1 <= meio){
        vAux[iniAux] = v[ini1];
        iniAux++;
        ini1++;
    }

    //repetição para caso tenha sobrado elementos na segunda
    //metade do vetor
    while(ini2 <= fim) {
        vAux[iniAux] = v[ini2];
        iniAux++;
        ini2++;
    }

    //repetição responsável por colocar os vetores de volta
    //no vetor principal que será impresso na Main
    for(iniAux = inicio; iniAux <= fim; iniAux++){
```

```
        v[iniAux] = vAux[iniAux-inicio];
    }

    free(vAux);
}

void mergeSort(int *v, int inicio, int fim){
    if (inicio < fim) {
        int meio = (fim+inicio)/2;

        mergeSort(v, inicio, meio);
        mergeSort(v, meio+1, fim);
        merge(v, inicio, meio, fim);
    }
}
int main() {
    int *vPrincipal, tamanho, aux=0;

    printf("Entre com o tamanho: ");
    scanf("%d", &tamanho);

    vPrincipal = (int*) malloc(tamanho*sizeof(int));
        for(int i=0;i<tamanho;i++){
                scanf("%d", &vPrincipal[i]);
        }

    mergeSort(vPrincipal, aux, tamanho-1);

    printf("\nSeu vetor ordenado sera: [ ");
        for(int i=0;i<tamanho;i++){
            printf("%d ", vPrincipal[i]);
        }
        printf("]");
}
```

ITEM E - Q1

```
#include <stdio.h>
#include <stdlib.h>

void quickSort(int *v, int esquerda, int direita);

int main(){
  int tamanho, *vet;
    printf("Entre com o tamanho: ");
    scanf("%d", &tamanho);
    if(tamanho<=1){
        return 0;
```

```
    }
    vet = (int*) malloc(tamanho*sizeof(int));
        for(int i=0;i<tamanho;i++){
                scanf("%d", &vet[i]);
        }

 quickSort(vet, 0, tamanho-1);
  printf("\nSeu vetor ordenado sera: [ ");
        for(int i=0;i<tamanho;i++){
            printf("%d ", vet[i]);
        }
        printf("]\n");


 return 0;
}

void quickSort(int *v, int esquerda, int direita) {
    int indiceI, indiceJ, aux, pivo;

    indiceI = esquerda;
    indiceJ = direita;

    pivo = v[(rand()%direita)];

    while(indiceI <= indiceJ) {
        while(v[indiceI] < pivo && indiceI < direita) {
            indiceI++;
        }

        while(v[indiceJ] > pivo && indiceJ > esquerda) {
            indiceJ--;
        }

        if(indiceI <= indiceJ) {
            aux = v[indiceI];
            v[indiceI] = v[indiceJ];
            v[indiceJ] = aux;
            indiceI++;
            indiceJ--;
        }
    }

    if(indiceJ > esquerda) {
        quickSort(v, esquerda, indiceJ);
    }
    if(indiceI < direita) {
        quickSort(v, indiceI, direita);
```

```
    }
}
```

QUESTÃO 02

Funcionamento dos algoritmos de ordenação

ITEM A - Q2

Como já possuímos o tamanho definido do vetor, basta alocar o espaço necessário para o array, com isso, nossas iterações ficarão da seguinte forma:

● aleatório = [3, 6, 2, 5, 4, 3, 7, 1, 10^9]

Vetor original

3	6	2	5	4	3	7	1	10^9

Iremos definir o menor valor como o primeiro elemento do vetor na primeira repetição do primeiro laço “for”

3	6	2	5	4	3	7	1	10^9

Iremos agora comparar o menor valor (1º elemento) com todos os outros elementos do vetor para encontrarmos qual é o menor elemento desse vetor

3	6	2	5	4	3	7	1	10^9

3<6? Verdadeiro, seguiremos com o 3 a próxima comparação

3	6

3<2? Falso, 2 será o novo menor valor
então iremos comparar o 2 com o os valores seguintes do vetor

3	2

2<5? Verdadeiro, seguiremos com o 2 a próxima comparação

2	5

2<4? Verdadeiro, seguiremos com o 2 a próxima comparação

2	4

2<3? Verdadeiro, seguiremos com o 2 a próxima comparação

2	3

2<7? Verdadeiro, seguiremos com o 2 a próxima comparação

2	7

2<4? Falso, 1 será o novo menor valor, então
iremos comparar o 1 com os valores seguintes do vetor

2	1

1<10^9? Verdadeiro, como chegamos ao fim do vetor e não temos elementos seguintes para fazer a comparação, no primeiro laço de

repetição do primeiro “for”, o 1 será o nosso menor valor

1	10^9

Feito isso, colocaremos o elemento “1” no primeiro laço de repetição “for” como elemento do índice 0 do vetor

1								

Como concluímos a primeira repetição do primeiro laço “for”, quando formos a segunda repetição do segundo laço for, faremos n-1 comparações, pois já temos o primeiro elemento do vetor ordenado definido, por isso começamos com o j=i, sabendo disso, faremos os mesmos passos feitos anteriormente e encontraremos o menor valor de cada repetição

Resultado para segundo laço de repetição do primeiro “for”

1	2							

Resultado para o terceiro laço de repetição do primeiro “for”

1	2	3						

Resultado para o quarto laço de repetição do primeiro “for”

1	2	3	3					

Resultado para o quinto laço de repetição do primeiro “for”

1	2	3	3	4				

Resultado para o sexto laço de repetição do primeiro “for”

1	2	3	3	4	5			

Resultado para o sétimo laço de repetição do primeiro “for”

1	2	3	3	4	5	6		

Resultado para o oitavo laço de repetição do primeiro “for”

1	2	3	3	4	5	6	7	

Resultado para o nono laço de repetição do primeiro “for”

Vetor Ordenado

1	2	3	3	4	5	6	7	10^9

ITEM B - Q2

Como já possuímos o tamanho definido do vetor, basta alocar o espaço necessário para o array, com isso, nossas iterações ficarão da seguinte forma:

● aleatório = [3, 6, 2, 5, 4, 3, 7, 1, 10^9]

Vetor Original

3	6	2	5	4	3	7	1	10^9

No bubbleSort nós iremos comparar o elemento da esquerda com o da sua direita, percorrendo todo o vetor, de maneira que se o elemento da esquerda for maior que o da sua direita, deslocaremos o elemento maior uma posição para a direita.

Como preciso percorrer todo o vetor, “n vezes”, adotei uma repetição aninhada com 2 “for”, primeiro irei ilustrar a repetição para o segundo for, que será responsável pelo deslocamento para a direita, vale ressaltar que como iremos fazer uma comparação, precisamos de 2 elementos no mínimo para realizá-lo, por isso sempre estaremos utilizando a comparação com “j+1” e indo até “tamanho-1”, se não na última repetição iria ser utilizado um “valor lixo” na comparação porque o vetor estaria encerrado e não teria um “j+1”.

Primeira repetição do segundo laço “for”, iremos comparar 3>6? Falso, logo o vetor não será mexido

3	6	2	5	4	3	7	1	10^9

Segunda repetição do segundo laço “for”, iremos comparar 6>2? Verdadeiro, logo o vetor será alterado, onde o 6 trocará de posição com o 2

3	6	2	5	4	3	7	1	10^9

Assim teremos:

3	2	6	5	4	3	7	1	10^9

Terceira repetição do segundo laço “for”, iremos comparar 6>5? Verdadeiro, logo o vetor será alterado, onde o 6 trocará de posição com o 5

3	2	6	5	4	3	7	1	10^9

Assim teremos:

3	2	5	6	4	3	7	1	10^9

Quarta repetição do segundo laço “for”, iremos comparar 6>4? Verdadeiro, logo o vetor será alterado, onde o 6 trocará de posição com o 4

3	2	5	6	4	3	7	1	10^9

Assim teremos:

3	2	5	4	6	3	7	1	10^9

Quinta repetição do segundo laço “for”, iremos comparar 6>3? Verdadeiro, logo o vetor será alterado, onde o 6 trocará de posição com o 3

3	2	5	4	6	3	7	1	10^9

Assim teremos:

3	2	5	4	3	6	7	1	10^9

Sexta repetição do segundo laço “for”, iremos comparar 6>7? Falso, logo o vetor não será mexido

3	2	5	4	3	6	7	1	10^9

Sétima repetição do segundo laço “for”, iremos comparar 7>1? Verdadeiro, logo o vetor será alterado, onde o 7 trocará de posição com o 1

3	2	5	4	3	6	7	1	10^9

Assim teremos:

3	2	5	4	3	6	1	7	10^9

Oitava repetição do segundo laço “for”, iremos comparar $7>10^9$? Falso, logo o vetor não será mexido

Com o fim da última repetição do segundo laço “for”, chega ao fim a primeira repetição do primeiro laço “for”, que ficará:

3	2	5	4	3	6	1	7	10^9

Agora irei ilustrar as demais repetições para o primeiro laço “for”, utilizando da mesma ideia dos passos anteriores, desta vez, indo de maneira bem mais rápida.

Operações feitas na segunda repetição: 3>2? (V) ; 3>5? (F) ; 5>4? (V) ; 5>3? (V) ; 5>6? (F) 6>1? (V) ; 6>7? (V)
Resultado para o fim da segunda repetição do primeiro laço “for”:

2	3	4	3	5	1	6	7	10^9

Visto isso, operações semelhantes irão ocorrer nas demais repetições do primeiro laço.

Resultado para o fim da terceira repetição do primeiro laço “for”:

2	3	3	4	1	5	6	7	10^9

Resultado para o fim da quarta repetição do primeiro laço “for”:

2	3	3	1	4	5	6	7	10^9

Resultado para o fim da quinta repetição do primeiro laço “for”:

2	3	1	3	4	5	6	7	10^9

Resultado para o fim da sexta repetição do primeiro laço “for”:

2	1	3	3	4	5	6	7	10^9

Resultado para o fim da sétima repetição do primeiro laço “for”:

1	2	3	3	4	5	6	7	10^9

Seguindo a sequência do código que implementei, o laço de repetição do primeiro “for” deveria ser passado mais duas vezes, sendo a oitava e nona repetição, mas como podemos notar, o vetor já está totalmente ordenado com apenas 7 repetições, isso se dá porque o código implementado foi pensado para situações que o vetor estivesse totalmente desordenado e precisasse realizar o máximo de iterações possíveis.

Para resolver isso, criei uma variável “cont=0” que iria aumentar em 1 unidade sempre que alguma troca de valores fosse realizada no vetor, caso toda a repetição do segundo “for” terminasse e o “cont” continuasse em 0, isso significa que o meu vetor está ordenado e eu posso parar a execução do programa.

Vale ressaltar que essa otimização só foi notada quando eu estava ilustrando para a “Questão 02” cada iteração do meu código.

ITEM C - Q2

● aleatório = [3, 6, 2, 5, 4, 3, 7, 1, 10^9]

Vetor Original

3	6	2	5	4	3	7	1	10^9

No InsertionSort nós iremos inserir ordenadamente um elemento “x” na sua posição correta no vetor, em outras palavras, guardaremos o “elemento da vez” e faremos comparações para saber a real posição dele, desta forma, nos lembra o BubbleSort, a grande diferença é que o nosso vetor fará menos operações, pois ele não precisará percorrer todo o tamanho do vetor original em cada repetição, já que será inserido apenas 1 novo elemento para comparação a cada repetição do “for” externo, desta forma, sendo mais eficiente.

Na primeira repetição do primeiro laço “for”, será simplesmente adicionado o primeiro valor, porque não temos um outro para comparar, nesta situação, pela confecção do meu código, montei o algoritmo visando ignorar o segundo laço de repetição (“for” interno) nessa situação

3								

Próximo elemento a ser adicionado: 6, com isso faremos a seguinte comparação v[j-1]>v[j]? 3>6? Falso, logo apenas adicionaremos o 6 no final do vetor:

3	6							

Com isso, o 3 e o 6, já estarão ordenados, agora iremos adicionar o próximo elemento: 2 Agora faremos uma comparação, v[j-1]>v[j]? 6>2? Verdadeiro, logo iremos trocá-los de posição

3	2	6						

Depois disso, será feito novamente v[j-1]>v[j]? Repare que como estamos decrescendo o valor de “j” e estamos na repetição seguinte do laço “for” interno, o elemento v[j-1] será o “3”, desta forma, 3>2? Verdadeiro, logo os trocamos de posição, ficando:

2	3	6						

Próximo elemento a ser adicionado: 5, então faremos v[j-1]>v[j]? 6>5? Verdadeiro, logo os trocamos de posição, gerando:

2	3	5	6					

É importante comentar que uma otimização que notei para evitar fazer muitas comparações, foi colocar um comando “break”, porque como estamos inserindo ordenadamente, quando formos a próxima repetição, se percebemos que “v[j-1]>v[j]” se provar sendo falso, apenas uma única vez, podemos encerrar o “loop do for interno”, pois significa que irá falhar para todas as outras repetições, que é o que acontece no passo acima, quando chegarmos na comparação 3>5, será falso, e como os elementos adicionados anteriormente já estavam ordenados, não há necessidade de continuar as comparações.

Próximo elemento a ser adicionado: 4, com isso faremos a seguinte comparação, 6>4? Verdadeiro, logo nosso vetor ficará assim:

2	3	5	4	6				

A próxima comparação será 5>4? Verdadeiro, logo nosso vetor fica:

2	3	4	5	6				

Encerramos no a repetição porque o “for” interno notou que deu falha na condição v[j-1]>v[j]

Próximo elemento a ser adicionado: 3, então iremos comparar 6>3? Verdadeiro, então trocamos suas posições, ficando:

2	3	4	5	3	6			

A próxima comparação será: 5>3? Verdadeiro, então trocamos suas posições, ficando:

2	3	4	3	5	6			

Próxima comparação: 4>3? Verdadeiro, então trocamos suas posições, ficando:

2	3	3	4	5	6			

Próxima comparação: 3>3? Falso, então iremos adicionar o próximo elemento, que será: 7, logo comparamos 6>7? Falso também, então a repetição encerra, o nosso vetor fica:

2	3	3	4	5	6	7		

Próximo elemento a ser adicionado: 1, então iremos comparar 7>1? Verdadeiro, então trocamos suas posições, ficando:

2	3	3	4	5	6	1	7	

Com isso, as próximas comparações a serem feitas, serão: 6>1 (V), 5>1? (V), 4>1 (V), 3>1 (V), 3>1 (V), 2>1? (V), todas as comparações se provaram verdadeiras, desta forma, sempre iremos realizar sucessivas trocas até o 1 ficar no início do vetor:

Comparação 6>1, vetor fica:

2	3	3	4	5	1	6	7	

Comparação 5>1, vetor fica:

2	3	3	4	1	5	6	7	

Comparação 4>1, vetor fica:

2	3	3	1	4	5	6	7	

Comparação 3>1, vetor fica:

2	3	1	3	4	5	6	7	

Comparação 3>1, vetor fica:

2	1	3	3	4	5	6	7	

Comparação 2>1, vetor fica:

1	2	3	3	4	5	6	7	

Próximo elemento a ser adicionado: 10^9, com isso faremos a seguinte comparação, $7>10^9$? Falso, logo nossa repetição se encerra e não possuímos mais elementos para adicionar, isso significa que temos o **Vetor Ordenado**:

1	2	3	3	4	5	6	7	10^9

Observação: é importante reparar que o meu código de ordenação está semelhante a implementação feita pelo professor, mesmo assim, esse foi o código que implementei por conta própria, tanto que para exemplificar isso, em versão anteriores eu estava utilizando uma relação errônea:

```
for(i=0;i<tamanho;i++){
        for(j=0;j<i;j++){
            if(v[j]>v[i]){
                aux=v[j];
                v[j]=v[i];
                v[i]=aux;
            }
        }
    }
```

Versão anterior do código antes das alterações

Neste código, tinham situações que eu estava desordenando o vetor para adicionar um elemento e depois precisava reordena-lo novamente, ou seja, muitas comparações desnecessárias, vale mencionar que eu só notei esses erros quando estava ilustrando iteração por iteração na questão 02C, desta forma, coloquei apenas a versão final melhorada do meu código, que como mencionei acima, é semelhante a realizada pelo professor.

ITEM D - Q2

● decrescente = [7, 6, 5, 4, 3, 3, 2, 1]

Vetor Original

7	6	5	4	3	3	2	1

Primeiramente, iremos dividir o vetor em 2 partes, e cada parte dessa, iremos dividir pela metade sucessivamente até que tenha apenas 1 elemento, essa é a nossa função “mergeSort”.

PRIMEIRA divisão do vetor original:

7	6	5	4

SEGUNDA divisão do vetor original:

3	3	2	1

Inicialmente, pegamos o vetor [7, 6, 5, 4] e iremos dividir até que sobre apenas 1 elemento, na próxima divisão, teremos como resultado:

7	6

5	4

Com isso, na próxima repetição, após outra divisão por 2, sobrará apenas um elemento, gerando assim 4 partes, cada uma com 1 elemento:
primeiro elemento:

7

segundo elemento:

6

terceiro elemento:

5

quarto elemento:

4

Agora partiremos para a **SEGUNDA divisão do vetor original**, que é:

3	3	2	1

Os processos a serem feitos, serão os mesmos feitos anteriormente, desta vez com os elementos [3, 3, 2, 1], desta forma, irei ilustrar mais rapidamente, já que a função “mergeSort” é responsável por sucessivas divisões para depois chamar a função de ordenação “merge”:

3	3

2	1

quinto elemento:

3

sexto elemento:

3

sétimo elemento:

2

oitavo elemento:

1

Devemos lembrar que o algoritmo de ordenação Merge Sort é do tipo de divisão e conquista, ou seja, quando não for mais divisível, sobrando apenas um elemento, será a hora de “conquistar”.

Com isso, faremos um caminho inverso, com a chamada da função “merge” responsável por ordenar todas as 8 partes do nosso vetor.

Iremos comparar o 1º elemento com o 2º elemento

Primeira repetição do primeiro laço while, 7<6? Falso

7	6

Assim na parte da “conquista” temos os 2 primeiros elementos ordenados:

6	7

Em seguida, comparamos o 3º elemento com 4º elemento, 5<4? Falso, :

5	4

Gerando:

4	5

Comparando os 4 elementos obtidos dos 2 passos anteriores, chegaremos a uma ordenação, assim temos que essa primeira parte do nosso vetor original está ordenada já:

4	5	6	7

Agora iremos comparar seu 5º elemento com o 6º elemento, assim faremos: 3<3? Falso

3	3

Ficando:

3	3

Depois, iremos comparar o 7º elemento com o 8º elemento, 2<1? Falso:

2	1

1	2

Desta forma, quando comparamos os elementos obtidos nos 2 passos anteriores, ficaremos com:

1	2	3	3

Resultado da ordenação para os **4 primeiros elementos**:

4	5	6	7

Resultado da ordenação para o **4 últimos elementos**:

1	2	3	3

Para finalizar o processo de ordenação, comparamos os resultados obtidos, o que nos dará finalmente o nosso **Vetor Ordenado**:

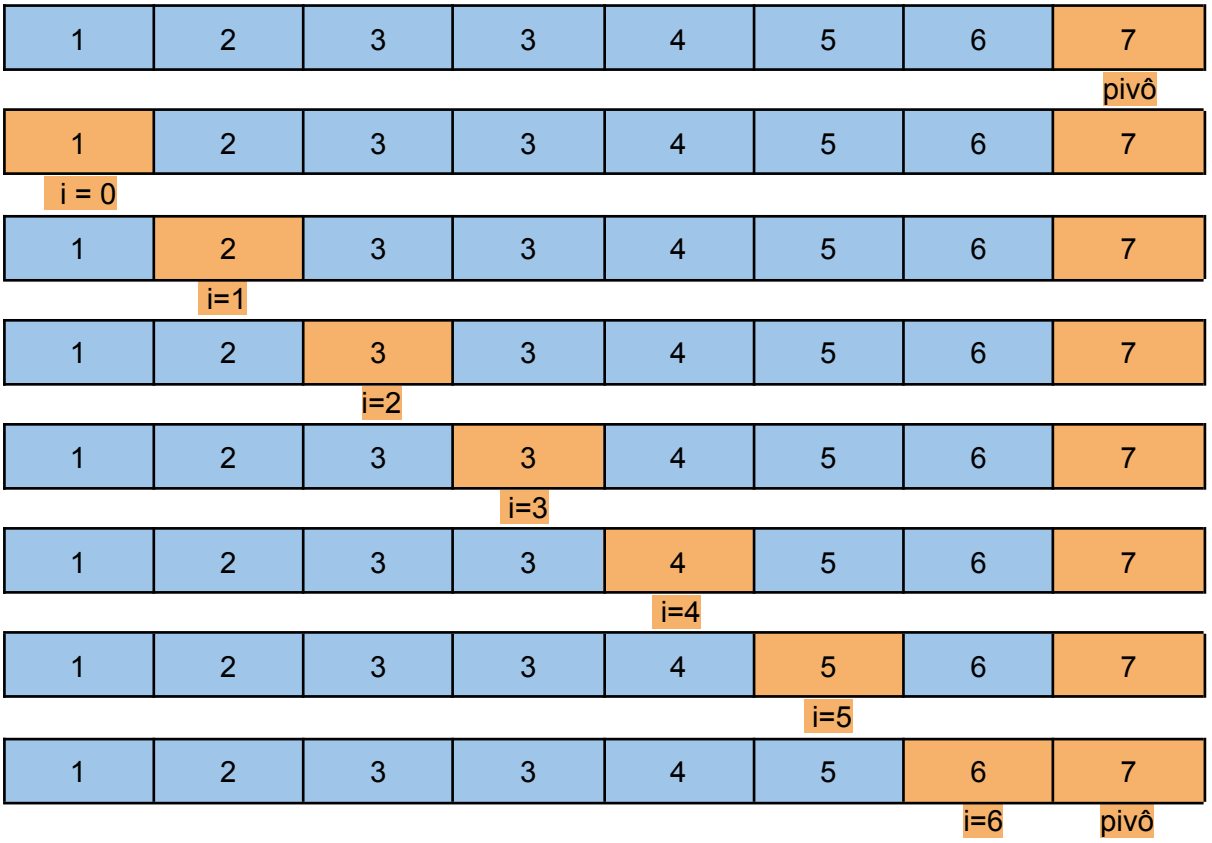
1	2	3	3	4	5	6	7

ITEM E - Q2

● crescente=[1,2,3,3,4,5,6,7]

1	2	3	3	4	5	6	7

Como nosso vetor está ordenado, se escolhermos o pivô como o último número, o nosso código começará com o índice “i” da esquerda e continuará enquanto ele for menor que o pivô, a fim de deixar todos os elementos menores que o pivô a sua esquerda e enquanto o índice “i” for menor que o pivô, ele será acrescido.



Como podemos notar, de acordo com o exemplo de nossa primeira repetição, é perceptível que os índices continuarão a se alterar, nosso algoritmo fará o particionamento, escolherá o pivô, mas o resultado final continuará sendo o vetor original, pois mesmo que ainda tenhamos que fazer as comparações, sempre que escolhermos um novo pivô e fizermos eventuais trocas para colocar os elementos menores a esquerda e maiores a direita, o próprio algoritmo irá “desfazer” posteriormente.

QUESTÃO 03

Experimentos com algoritmos de ordenação

ITEM A - Q3

Para realizar a experimentação do “item a”, iremos utilizar um tamanho pequeno para os vetores, nestes testes, usarei como base um tamanho de 10 para o array, para calcular o tempo, inclui a biblioteca “time.h” no meu código e marcar o tempo de execução antes da chamada da função (0) e o tempo após o término do ordenamento.

Tipo de Ordenação	Tempo (ms)
Selection Sort	0,000000
Bubble Sort	0,000000
Insertion Sort	0,000000
Merge Sort	0,000000
Quick Sort	0,000000

Como o vetor é muito pequeno, independente da disposição dos elementos, não influenciará no tempo, com todos os algoritmos tendo um desempenho semelhante.

ITEM B - Q3

No item B, verifiquei que para o tamanho de 10^3 os algoritmos permaneciam com desempenho muito semelhante, desta forma, passei a utilizar valores de 10^4 para uma melhor análise e de fato isso se constatou na tabela abaixo, onde notamos que dependendo do tamanho do vetor e da sua configuração de valores, o funcionamento será consideravelmente afetado, mas o principal destaque vai para o “Quick Sort”, que se manteve extremamente estável para os diferentes casos, tendo um desempenho excelente.

	Vetor Decrescente	Vetor Crescente	Vetor Aleatório
Tipo de Ordenação	Tempo (ms)	Tempo (ms)	Tempo (ms)
Selection Sort	140	133	249
Bubble Sort	335	0	350
Insertion Sort	234	0	124
Merge Sort	1	2	3
Quick Sort	0	0	0

ITEM C - Q3

Conforme a tabela do “item b - q3”, podemos reparar que o Merge Sort tem um desempenho bom para os diferentes casos, não é tão excepcional quanto o Quick Sort, mas podemos afirmar que sua eficiência é boa para valores menores que a casa de 10^5 independente da distribuição dos valores, dado que em diversas experimentações o Merge Sort se manteve bem.

ITEM D - Q3

Durante a experimentação, utilizando valores de 10^4 que gera a tabela mostrada no “item b - q3”, o desempenho se manteve estável, utilizando 10^5 o desempenho teve uma pequena variação, mas continuou se mantendo estável, sem variar tanto, as maiores variações só vieram quando utilizamos elementos da grandeza de 10^6, que enquanto davam tempos próximos de 0ms para 10^4, tempos próximo de 10ms para 10^5, para valores de 10^6 o tempo ficou próximo de 150ms.