```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "vetor_produtos.h"


Item *ptr_item;
int k=0;

/**Função de criação de repositório para inserçaõ de itens!
Apresenta complexidade também constante, O(1), pois se destina a, somente,
reservar espaço de alocação no ponteiro ptr_item, o qual estará sendo
utilizado como um vetor de índice n, pois a cada nova inserção este será
realocado.
*/
void create_repositorio (){
    ptr_item = (Item*) realloc(ptr_item, sizeof(Item));
}

/**Função de retorno de status de repositorio, retorna o tamanho do mesmo!
Apresenta complexidade constante, O(1), deverá unicamente retornar ao usuário
o tamanho do vetor, ou seja, o número de itens presente no repositório que
o usuário está preenchendo.
*/
int tamanho(){
    return k++;
}

/**Função que organiza os dados dentro do vetor!
O algoritmo "insertion_sort" trabalha percorrendo o vetor, no qual ele verifica cada índice
analisando se o seu valor está na posição correta. Inicia a verificação pelo primeiro valor
e compara este com todos os outros. Caso o valor de 'i' recaia no caso de ser maior, o algoritmo
entrará na condição dentro do for, onde, com a ajuda de uma variável temporária realizará a permuta
dos valores mal posicionados. A eficiêcia deste algoritmo varia conforme o estado inicial do vetor, pois
percorrerá todo o vetor para fazer as comparações, caso esteja ordenado fará uma única vez. Possui algumas
vantagens, pois pode ordenar os elemtnso a medida que se façam as inserções e não precisa ter todo o conjunto
de dados para colocar em ordem.
Apresenta complexidade quadrática, O(n²). Analisando pelo melhor caso, este algoritmo executa n
operações, onde n representa apenas o número de elementos do vetor. Já, se analisarmos o pior caso,
serão feitas n² operações. Dessa forma, a função não seria recomendada para
```

```
    programas que necessitem
41  de muita velocidade e que operem com um número considerável de dados. Ao analisar
    o algoritmo
42  implementado neste trabalho, fiz uma mudança em relação ao que apresentei na prova
    passada.
43  Pois, percebo que o primeiro for iniciava em 0 o que levava a comparar o indice 0
    com o proprio
44  índice quando da entrada do segundo laço, o que acaba consumido um ciclo
    desnecessário. A partir
45  do material postado pelo professor pude fazer essa correção e apresentar um
    algoritmo mais puro,
46  ou mais adequado.
47  */
48  void insertion_sort(){
49      int i, j;
50      Item temp;
51
52      for (i=1;i<=k;i++){
53          temp = ptr_item[i];
54          for (j=i;(j>0) && (temp.valor<ptr_item[j-1].valor);j--){
55              ptr_item[j] = ptr_item[j-1];
56          }
57          ptr_item[j] = temp;
58          }
59
60              /* Algoritmo utiliazado na Última prova.
61              for (int i=0;i<=k;i++){
62                  for (int j=i;j<=k;j++){
63                      if (ptr_item[i].valor > ptr_item[j].valor){
64                          temp = ptr_item[i];
65                          ptr_item[i] = ptr_item[j];
66                          ptr_item[j] = temp;
67                      }
68                  }
69              }*/
70  }
71
72  /**Função que organiza os dados dentro do vetor!
73  Neste algoritmo, ao passo que os laços vão acontecendo, ele vai escolhendo o
    melhor elemento para
74  ocupar determinada posição do array, que pode ser por maior ou menor elemento a
    depender da aplicação
75  da ordenação. Possui um desempenho, na prática, ligeiramente superior ao "bubble
    sort". Divide o array
76  em duas sessões: a primeira, ordenada, ficará a esquerda do elemento analisado, e
    o restante dos elementos
77  que ainda não foram organizados ficarão à direita do elemento analisado. Tem
    algumas vantagens a depender
78  da aplicação a que se destina, por exemplo nos trabalhos com memórias que se
```

```
    desgastam no processor de
79  escrita e leitura, mas uma vantagem em particular se destaca que é a estabilidade,
    pois não altera a ordem
80  dos dados que são iguais. Como esse algoritmo tem sua eficiência comprometida à
    medida em que se aumenta
81  o número de elementos, pois se torna muito custoso, ele não é indicado para
    aplicações com grandes quantidade de
82  dados. A complexidade deste algoritmo também é quadrática, O(n²), pois ele precisa
    realizar dois laços
83  em torno do vetor para conseguir finalizar o seu trabalho, isso no pior caso, é
    claro.
84  */
85  void selection_sort(){
86      int i,j,menor;
87      Item temp;
88
89      for(i=0;i<k;i++){
90          menor = i;
91          for (j=i+1;j<=k;j++){
92              if (ptr_item[j].valor < ptr_item[i].valor)
93                  menor = j;
94              }
95              if (menor != i){
96                  temp = ptr_item[i];
97                  ptr_item[i]=ptr_item[menor];
98                  ptr_item[menor]=temp;
99              }
100     }
101 }
102
103 /**Função que organiza os dados dentro do vetor!
104 Esse algoritmo é o, também conhecido, bolha. Ele é bastante sugerido na internet
    em sites como "stackoverflow"
105 pela facilidade de implementação, talvez. Possui esse nome em uma analogia a
    bolhas flutuando em um tanque de água
106 em direção a superfície até encontrar o seu próprio nível - por isso ordenação
    crescente. Compara pares de
107 valores vizinhos e os troca caso estejam na ordem errada. Ele atua de forma a
    mover, uma posição por vez,
108 o maior valor existente na posição não ordenada de um array para a sua respectiva
    posição no array ordenado. Toda a
109 atividade é repetida até que sejam sanadas as incorreções. Na primeira etapa, o
    do-while: encontra o maior valor e o
110 movimenta até a última posição. Na segunda o do-while encontra o segundo maior
    valor e o movimenta para a penúltima
111 posição do vetor. Isso continua até que não haja mais necessidade. Tem uma
    vantagem de ser de fácil entendimento e
112 construção, o que provavelmente contribua para ser um dos métodos mais difundidos
    de ordenação. Não é eficiente para
```

```
113 trabalhos com numero de dados elevado. Apresenta complexidade quadrática tanto no
    médio quando no pior caso, O(N²),
114 pois ainda que esteja parcialmente ordenado, este ainda passará pelo vetor
    realizando a leitura dos valores.
115 */
116 void bubble_sort(){
117     int i,passo = k, bolha;
118 
119     Item temp;
120 
121     do {
122         bolha = 0;
123         for(i=0;i<k;i++){
124             if (ptr_item[i].valor > ptr_item[i+1].valor){
125                 temp = ptr_item[i];
126                 ptr_item[i]=ptr_item[i+1];
127                 ptr_item[i+1]= temp;
128                 bolha=i;
129             }
130         }
131         passo--;
132     }while(bolha != 0);
133 }
134 
135 /**Função que organiza os dados dentro do vetor!
136 Esse algoritmo foi o que eu mais demorei a entender, pois ele é muito extenso e
    possui muitas condicionais o que acaba
137 confundido a análise além da recursividade que é outro processo ainda complexo
    para um novato na programação. Ele trabalha
138 com intercalação e com a filosofia "dividir para conquistar", em que faz bom
    proveito dessa ideia à medida que reduz o
139 tamanho de sua tarefa para ser mais eficiente. O algoritmo divide os dados em
    conjuntos cada vez menores para depois
140 ordená-los e combiná-los por meio da intercalação, daí o seu nome: merge. Para que
    eu conseguisse compreender e ajustar
141 o funcionamento do algoritmo ao meu código que até havia usado na prova de
    ED-MODII. Dividi em 3 etapas, a primeira da
142 chamada, a segunda a que realiza a divisão recursivamente e a terceira a que
    coloca os dados em ordem. Como é possível perceber,
143 ele vai dividindo o vetor até restar apenas um elemento, em seguida cria dois
    vetores combinando estes em um maior e ordenado.
144 Apresenta complexidade O(N log N) para quaisquer que sejam os casos, o que o torna
    extremamente estável, ainda, sua eficiência
145 não depende do estado inicial dos casos. No pior caso, realiza cerca de 39% menos
    comparações do que o quick sort no seu caso
146 médio. Tem uma desvantagem que é o fato de precisar de mais espaço de memória para
    a criação dos outros vetores.
147 */
148 void mergeSort(Item *v, int n) {
```

```
149     mergeSort_ordena(v, 0, n);
150 }
151 /* ordena o vetor v[esq..dir] */
152 void mergeSort_ordena(Item *v, int esq, int dir) {
153     if (esq == dir)
154     return;
155     int meio = (esq + dir) / 2;
156     mergeSort_ordena(v, esq, meio);
157     mergeSort_ordena(v, meio+1, dir);
158     mergeSort_intercala(v, esq, meio, dir);
159     return;
160 }
161 
162 /* intercala os vetores v[esq..meio] e v[meio+1..dir] */
163 void mergeSort_intercala(Item *v, int esq, int meio, int dir) {
164     int i, j, l, a_tam = meio-esq+1, b_tam = dir-meio;
165 
166     Item *a = (Item*) malloc(sizeof(Item) * a_tam);
167 
168     Item *b = (Item*) malloc(sizeof(Item) * b_tam);
169 
170     for (i = 0; i < a_tam; i++){
171         a[i] = v[i+esq];
172     }
173 
174     for (i = 0; i < b_tam; i++){
175         b[i] = v[i+meio+1];
176     }
177 
178     for (i = 0, j = 0, l = esq; l <= dir; l++) {
179         if (i == a_tam)
180             v[l] = b[j++];
181         else if (j == b_tam)
182             v[l] = a[i++];
183             else if (a[i].valor < b[j].valor)
184                 v[l] = a[i++];
185             else
186                 v[l] = b[j++];
187     }
188 
189 free(a); free(b);
190 
191 }
192 
193 /**Função que organiza os dados dentro do vetor!
194 Este algoritmo também é chamado de algoritmo de ordenação por partição. Assim como
    o MergeSort, este algoritmo também é
195 recursivo e também faz uso da filosofia "dividir para conquistar". Funciona
    dividindo o vetor a partir de um pivô, ficando,
```

```
196 por exemplo, os valores menores à esquerda do pivô e os maiores à direita. Então, um elemento é escolhido como pivô e,
197 recursivamente, assim como o merge, divide o vetor em partes até que se chegue a um único elemento. A função auxiliar chamada
198 de "particiona", ficará responsável por rearranjar os dados e calcular o pivô. Não é considerado um algoritmo estável por alguns
199 especialistas, pois ficam algumas incertezas em relação ao pivô: Como que se escolhe o pivô?
200 Perceba que um pivô, no pior caso, pode dividir o vetor em dois de forma que fiquem N-1 elementos em um lado e 0 no outro.
201 O que leva a uma outra percepção, o caso de o particionamento não ser balanceado, nesses casos o tempo de execução pode
202 chegar a O(N²). Sendo assim, essa é uma desvantagem do QuickSort, no caso de um particionamento não balanceado, o insertionsort
203 acaba se tornando tão eficiente quanto o QuickSort. Apesar de seu pior caso ser de ordem quadrática, este algoritmo costuma
204 ser a melhor opção prática para ordenação de grande números de dados. Isto pois, no melhor caso, quando o vetor não estar ordenado
205 em nenhum sentido, ou quando os valores não são iguais, o quicksort tem complexidade n log n, O(n (log n)).
206 */
207 void quick_sort(Item *v, int inicio, int fim){
208     int pivo;
209     if (fim > inicio){
210         pivo = particiona(v, inicio, fim);
211         quick_sort(v, inicio, pivo-1);
212         quick_sort(v, pivo+1, fim);
213     }
214 }
215
216 int particiona(Item *v, int inicio, int final){
217     int esq, dir;
218     Item temp, pivo;
219     esq = inicio;
220     dir = final;
221     pivo = v[inicio];
222     while (esq < dir){
223         while(esq <= final && v[esq].valor <= pivo.valor)
224             esq++;
225         while (dir >= 0 && v[dir].valor > pivo.valor)
226             dir--;
227
228         if(esq < dir){
229             temp = v[esq];
230             v[esq]=v[dir];
231             v[dir]=temp;
232         }
233     }
234     v[inicio] = v[dir];
```

```
235     v[dir]= pivo;
236     return dir;
237 }
238 
239 /**Função responśvel por adicionar valores no vetor!
240 Apresenta complexidade também quadrática, O(n²), pois para finalização do algoritmo
241 desta função, será necessário passar pela função "organizador", a qual tem este nível
242 de complexidade. Se analisado a função, desprezando o acesso a outra funçaõ, esta teria
243 complexidade constante, pois se obrigaria apenas de adicionar novos valores em um vetor.
244 */
245 void add(int codigo, float valor, int escolha){
246     ptr_item = (Item*) realloc(ptr_item, (k+1)*sizeof(Item));
247 
248     if (ptr_item != NULL){
249 
250         ptr_item[k].codigo = codigo;
251         ptr_item[k].valor = valor;
252 
253             switch(escolha){
254 
255                 case 1:
256                     insertion_sort();
257                     break;
258                 case 2:
259                     selection_sort();
260                     break;
261                 case 3:
262                      bubble_sort();
263                      break;
264                 case 4:
265                     mergeSort(ptr_item, k);
266                     break;
267                 case 5:
268                     quick_sort(ptr_item, 0, k);
269                     break;
270                 default:
271                     printf("\nEscolha inválida!\n");
272             }
273 
274         tamanho();
275     }else {
276         printf("Erro de memória!");
277         return;
278     }
279 }
280 
281 /**Função de busca de valores dentro do vetor e retorna a posição!
```

```
282  Apresenta complexidade logarítimica, O(log n(base binária)), ou seja, esta função
283  realiza sua tarefa quebrando-a pela metade, o que reduz bastante o tempo de realiza
284  ção da mesma. A depender da constante a se comparar, uma função logarítmica pode até
285  ser mais eficiente. Como os testes que realizei são com pequenas quantidade de dados,
286  a diferença não pôde ser notada.
287  */
288  int find_valor(float valor){
289      int esq = 0, dir=k;
290
291      while (esq <= dir){
292          int meio = (esq+dir) / 2;
293
294          if (ptr_item[meio].valor == valor){
295              return meio;
296          } else if (ptr_item[meio].valor > valor){
297              dir = meio -1;
298          } else {
299              esq = meio + 1;
300          }
301      }
302      return -1;
303  }
304
305  /**Função de busca de códigos dentro do vetor e que retorna a posição!
306  Apresenta complexidade logarítimica, O(log n(base binária)), ou seja, esta função
307  realiza sua tarefa quebrando-a pela metade, da mesma forma como a anterior. A única
308  diferença entre as duas é o tipo de parâmetro repassado na chamada.
309  */
310  int find_codigo(int cod){
311      int esq = 0, dir=k;
312
313      while (esq <= dir){
314          int meio = (esq+dir) / 2;
315
316          if (ptr_item[meio].codigo == cod){
317              return meio;
318          } else if (ptr_item[meio].codigo > cod){
319              dir = meio -1;
320          } else {
321              esq = meio + 1;
322          }
323      }
324      return -1;
325  }
326
327
328  /**Função responsável por imprimir os dados do vetor
329  Apresenta complexidade linear, O(n), pois o tempo de execução da tarefa deverá
330  ter relação com o tamanho do vetor, ou seja o número de itens inseridos no mesmo.
```

```
331 Desta forma, ela percorrerá todo o vetor para finalizar sua tarefa.
332 */
333 void imprime_repositorio(){
334     if (k>0){
335         for (int i=0;i<k;i++){
336             printf("\nItem %d - Codigo %i / Valor %.2f\n", i, ptr_item[i].codigo,
                ptr_item[i].valor);
337         }
338     } else
339         printf("\nRepositorio vazio!\n");
340 
341     printf("\n\n");
342 }
343 
344 /**Função responsável por "zerar" o vetor!
345 Apresenta complexidade constante, O(1), pois destina-se unicamente a liberar os
346 espaços de memória reservados pelo ponteiro. É certo que esses dados continuaram,
347 de certa forma, na memória. Mas como reseta o 'K', então automaticamente, o vetor
348 pode ser reutilizado.
349 */
350 void limpar_repositorio(){
351     k=0;
352     free(ptr_item);
353 }
354 
355 /**
356 6. Com base na complexidade computacional, qual você considerou como sendo o melhor
357 algoritmo de ordenação abordado? Justifique sua resposta em no mínimo 8 linhas.
358 
359 Diante das análises feitas durante o processo de implementação, percebi que o
    algoritmo MergeSort
360 tem uma lógica muito interessante, apesar de complexa de se analisar. Eu demorei
    consideravelmente
361 a compreender o seu funcionamento e cheguei até a congitar não fazer essa questão.
    Porém recorri ao
362 velho Google e, através de algumas postagens em fóruns, entender a lógica deste
    algoritmo. Se comparmos
363 ele a outros algoritmos com a lógica de divisão e conquista, como o Quicksort, o
    Merge apresenta a mesma
364 complexidade no pior e médio caso, mas se destaca no melhor caso. Além do que,
    existem implementações do
365 mergesort que permitem que ele seja melhorado em relação ao uso de memória, que é
    O(N) para O(N log N).
366 Apenas um fator me incomoda em relação ao QuickSort e MergeSort que é o fator
    recursividade. Pois acredito
367 que, a depender do compilador, a recursão acaba utilizando muita memória para
    realizar as tarefas, visto que
368 a recursividade só irá liberar memória após a saída final da função. Mas como
    temos memórias muito grandes
```

```
369  atualmente, acredito que isso acabe sendo minimizado, porém é válido resaltar. Por
     isso considero o MergeSort
370  o melhor algoritmo de ordenação, sobretudo por conta de sua estabilidade.
371  */
372
373
374
```