# BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

## *CS-F211: Data Structures and Algorithms*

## Lab 4: Insertion Sort & Merge Sort

### Introduction

Welcome to week 4 of Data Structures and Algorithms! Today we are going to get started with one of the most fundamental problems when dealing with collections of data — **Sorting**. Sorting is often required in itself or as a pre-requisite for other algorithms. For example, *Binary Search* requires the input array to be sorted. So, it becomes important to have efficient techniques to sort data. We would be looking at and implementing two important sorting algorithms - Insertion Sort and Merge Sort. We would also be comparing their performance and understanding the different use cases where one of them might be better than the other. Towards the end, we shall see how can we sort data stored in arbitrarily large files that don't even fit in memory.

### Topics to be covered in this labsheet

- Sorting
- Insertion Sort
- Merge Sort
    - Merge
        - Merge using an auxiliary function (Recursive)
        - Merge using an auxiliary function (Iterative)
        - Merge by insert
    - External Merge Sort

## Sorting

The sorting problem is defined as follows:

**Input:** A sequence of n numbers $(a_1, a_2, \ldots, a_n)$

**Output:** A permutation (reordering) $(a'_1, a'_2, \ldots, a'_n)$ of the input sequence such that $a'_1 \le a'_2 \le \ldots \le a'_n$.

The numbers to be sorted are also known as keys. Oftentimes, the members to be sorted would be records (or objects) having multiple fields. In these cases, they are sorted based on some particular field called the key. That is, the records are reordered such that the specific field of all instances are in the sorted order. A practical example of this would be sorting “student” records based on the CGPA field for PS allotment.

## Insertion Sort

Insertion sort is a simple sorting algorithm. It is the algorithm that many people end up using while sorting a hand of cards. You start with an empty hand and each time a card is dealt to you, you place it in the correct position in your hand. To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left, as illustrated in Figure 1. At all times, the cards held in the hand are sorted.

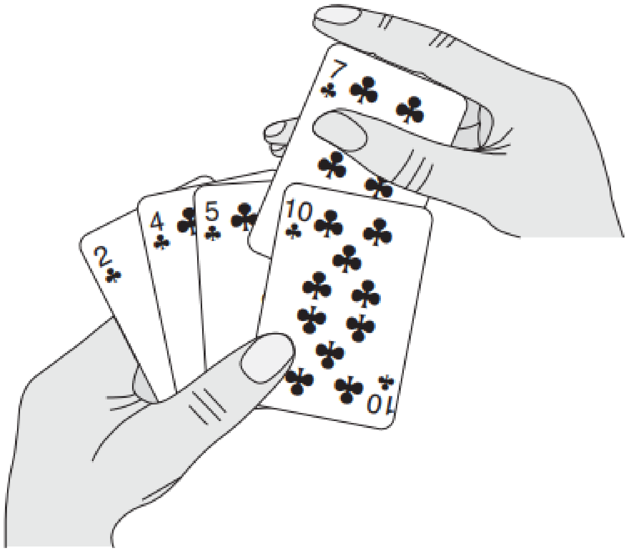

**Fig 1. Sorting a hand of cards using insertion sort**

Thus, we first place the first card in the hand, then we place the second card in the correct position relative to the first card, then we place the third card in the correct position relative to the first two cards, and so on. The algorithm is called insertion sort because we insert each card into its correct position in the hand.

"At each step, we 'insert in order' the newly encountered element into the already-sorted part of the array." Each insertion assumes that the elements to the left of it are already sorted. Thus, insertion sort is an application of the divide-and-conquer paradigm. In this paradigm, we divide the problem into subproblems, solve the subproblems, and then combine the solutions to the subproblems to solve the original problem. In insertion sort, the original problem is to sort an array of **n** elements. The subproblems are to sort the first **n-1** elements and to insert the **n**th element into the sorted array of the first **n-1** elements. The solution to the original problem is to solve the subproblems and then insert the **n**th element into the sorted array of the first **n-1** elements.

For this lab sheet, we would be implementing the iterative version of insertion sort as discussed in class. The algorithm (for an array of integer elements) is as follows:

```
void insertionSort(int A[], int n)
{
    for(int j = 1; j < n; j++)
    {
        insertInOrder(A[j], A, j);
    }
}
```

This code calls the subroutine insertInOrder() which is defined in the following manner:

```
// Pre-condition: (length(A) - 1 > last) & forall j: 0 <= j < last - 1: A[j] <= A[j+1]
void insertInOrder(int v, int A[], int last)
{
    int j = last - 1;
    while(j >= 0 && v < A[j])
    {
        A[j+1] = A[j];
        j--;
    }
    A[j+1] = v;
}
// Post-condition: forall j: 0 <= j <= last - 1: A[j] <= A[j+1]
```

Here, we start comparing the element to be inserted (int v) from the (last)$^{th}$ element (ie., A[j] where j = last - 1) down to 0. The elements are shifted right until an element lesser than or equal to v is found.

The working of insertion sort is illustrated in Fig 2.

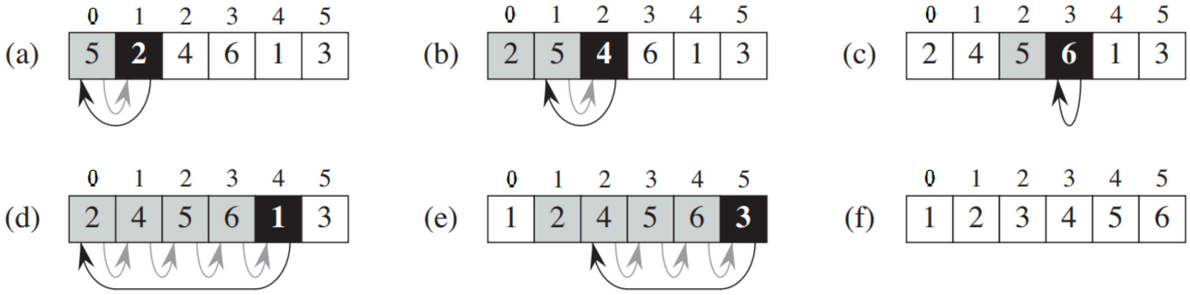

**Fig 2. Illustration of Insertion Sort**

Here, a run is shown on the array {5, 2, 4, 6, 1, 3}. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. (a)–(e) show the iterations of the for loop in insertionSort() function. In each iteration, the black rectangle holds the key taken from A[j] (ie v), which is compared with the values in shaded rectangles to its left in the test of the while loop in the function insertInOrder(). Shaded arrows show array values moved one position to the right in the body of the while loop, and black arrows indicate where the key moves to in the last line of the function insertInOrder(). (f) shows the final sorted array.

Insertion sort's time complexity is O(n^2) in the average and worst case, and O(n) in the best case (already-sorted array).

Task 1: Consider the following structure for storing the details of a person as defined below:

```
struct person
{
    int id;
    char *name;
    int age;
    int height;
    int weight;
};
```

"Write an implementation of insertion sort to sort an array of `person` structures. Use the `height` field as the sorting key..

For example:
Consider an array of persons containing the following data:

| *id* | *name* | *age* | *height* | *weight* |
|---|---|---|---|---|
| 1 | Sokka | 15 | 150 | 45 |
| 2 | Aang | 112 | 137 | 35 |
| 3 | Zuko | 16 | 160 | 50 |
| 4 | Katara | 14 | 145 | 38 |
| 5 | Toph | 12 | 113 | 30 |

**Table 1: Input Data**

The sorted result should look like:

| *id* | *name* | *age* | *height* | *weight* |
|---|---|---|---|---|
| 5 | Toph | 12 | 113 | 30 |
| 2 | Aang | 112 | 137 | 35 |
| 4 | Katara | 14 | 145 | 38 |
| 1 | Sokka | 15 | 150 | 45 |
| 3 | Zuko | 16 | 160 | 50 |

**Table 2: Sorted Data**

Also write an appropriate main() function that initialises an array with the values of Table 1, passes it as the input to your implementation of insertion sort and prints the array post sorting.

The expected output on stdout is:

```
5 Toph 12 113 30
2 Aang 112 137 35
4 Katara 14 145 38
```

```
1 Sokka 15 150 45
3 Zuko 16 160 50
```

Now we would try to inspect the complexity of this algorithm. We know from our lecture sections that the insertInOrder() function has a time complexity of O(n) and it is called n times in the insertionSort() function. Thus, insertion sort has an asymptotic time complexity of $O(n^2)$.

Let us try to see whether this holds empirically:

Task 2: You have been provided with a set of files having file names *datX.csv* where *X* stands for the input size. These files contain comma-separated entries for the details of the students. Write a program to read the data from these files and store them in a dynamically allocated array of `struct person` as defined in Task 1. You may take in the input file as a command line argument. Now, run the insertion sort algorithm on this array of `struct person` to sort the data in ascending order of the height of the students. Measure the time taken by the algorithm for sorting each of the input-sized arrays individually. Plot a graph of the time taken by the algorithm against the input size. (You can use any spreadsheet software like Microsoft Excel or plotting software to plot the graph. Interested students may also try using the python library matplotlib; you may refer to this tutorial: https://matplotlib.org/stable/tutorials/introductory/pyplot.html)

Our plot should reflect an approximately quadratic relationship between the time taken and the size of the array. We can also see that insertion-sort is **in-place**. In other words, it does not require any significant additional space over the memory required to store the original array itself. The sorted array ends up at the same locations as the original array and the elements are just moved around.

Home Exercise 1: Write a program to sort elements of an array of struct person (as defined in Task 1) in lexicographical order of the name of the students. Measure the time taken by the algorithm for each of the input sizes. Plot a graph of the time taken by the algorithm against the input size. (You can use any spreadsheet software or plotting software to plot the graph.) [Hint: You can use the strcmp() function of the string.h library to compare two strings.]

## Merge Sort

Merge sort also uses the divide-and-conquer paradigm. Here, the problem of sorting the sequence is reduced to sorting sub-sequences followed by the merging of the sub-sequences.

To sort a sequence **S** with **n** elements using the three divide-and-conquer steps, the merge-sort algorithm proceeds as follows:

1. **Divide**: If S has zero or one element, return S immediately; it is already sorted. Otherwise (S has at least two elements), remove all the elements from S and put them into two sequences, S1 and S2, each containing about half of the elements of S; that is, S1 contains the first ⌊n/2⌋ elements of S, and S2 contains the remaining ⌈n/2⌉ elements. (Recall that the notation ⌊x⌋ indicates the floor of x, that is, the largest integer k, such that k ≤ x. Similarly, the notation ⌈x⌉ indicates the ceiling of x, that is, the smallest integer m, such that x ≤ m.)
2. **Conquer**: Recursively sort sequences S1 and S2.
3. **Combine**: Put the elements back into S by merging the sorted sequences S1 and S2 into a sorted sequence.

It is intuitive to think of recursion when specifying the implementation. The algorithm for the mergeSort() function is given as follows:

```
// Precondition: A is an array indexed from st to en
void mergeSort(int A[], int st, int en)
{
    if (en - st < 1)
        return;
    int mid = (st + en) / 2;    // mid is the floor of (st+en)/2
    mergeSort(A, st, mid);      // sort the first half
    mergeSort(A, mid + 1, en); // sort the second half
    merge(A, st, mid, en);      // merge the two halves
}
// Postcondition: forall j: st<=j<en-1 --> A[j]<=A[j+1]
```

Here, mergeSort() recursively calls itself on the two halves of the array and the two sorted halves are then merged to form one sorted array through the merge() routine. An illustration of merge sort is provided in Fig 3.

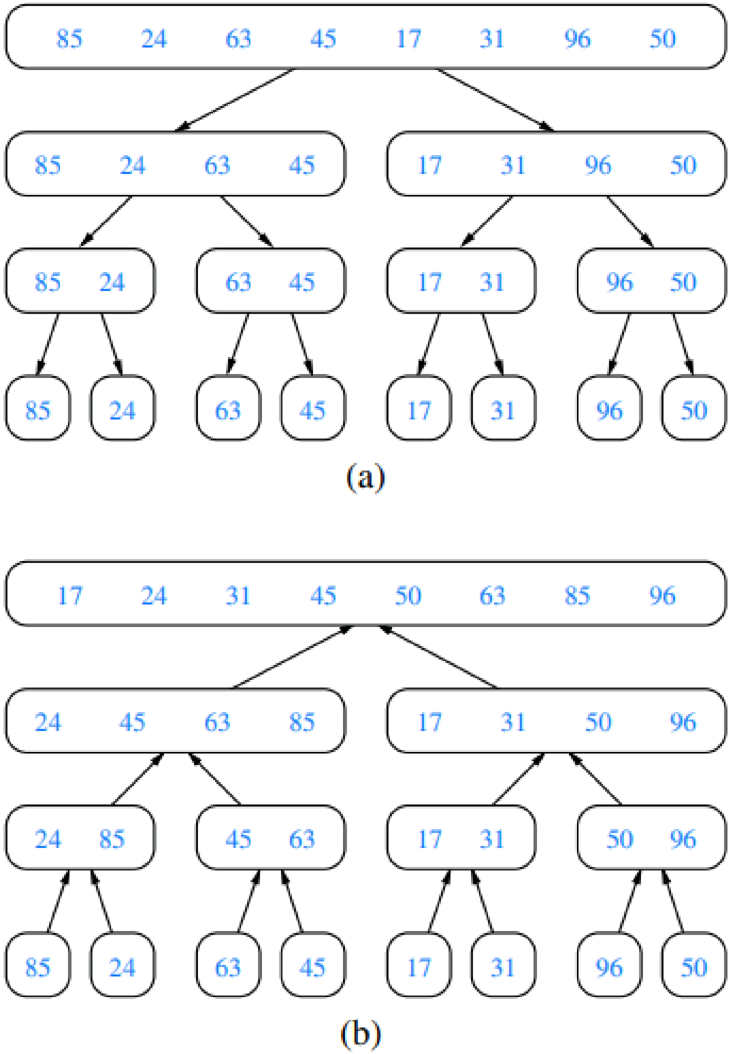

**Fig 2. Illustration of Merge Sort**

Here in Fig 2. The merge-sort tree T for an execution of the merge-sort algorithm on a sequence with 8 elements is shown. (a) shows the input sequences processed at each node of T, while (b) shows the output sequences generated at each node of T

Let us now look at the merge operation in detail.

### Merge

The merge function combines two sorted arrays into a single array. In this algorithm, it is used to combine the two recursively sorted subarrays into a merged array. The merge routine is also known as the “two-pointer technique”, because we are moving ahead two pointers and “merging” from the appropriate one according to the comparison between the two elements.

In other words, the merge operation reads/deletes elements from the front of two sorted sub-lists and adds them onto the rear of a new sorted list.
This is FIFO (i.e First-in-First-out) order i.e. the behaviour of merge can be described as follows:

1. reading from two FIFO lists (where FIFO order matches sorted order)
2. writing into a new FIFO list (resulting in a sorted order).

In Fig 3 (b), you can observe the merge operation being performed at each step.

We can make use of an auxiliary function here that takes two sorted arrays **L1** and **L2** indexed from **s1** to **e1** and **s2** to **e2** respectively and store the sorted result in an array **L3** indexed from **s3** to **e3**.

Thus the mergeAux() function has the signature:

```
void mergeAux(int L1[], int s1, int e1, int L2[], int s2, int e2, int
L3[], int s3, int e3);
```

Given an implementation of mergeAux(), merge() can simply act as a wrapper in the following manner:

```
void merge(int A[], int s, int mid, int e)
{
    int *C = (int *)malloc(sizeof(int) * (e - s + 1));
    mergeAux(A, s, mid, A, mid + 1, e, C, 0, e-s);
    for(int i = 0; i < e - s + 1; i++)
    {
        A[s + i] = C[i];
    }
    free(C);
}
```

**Merge using an auxiliary function (Recursive)**

Here, merge declares a new array to store the result of merging, calls the merge auxiliary function and copies the sorted array back into the original location. The mergeAux() function here is recursive in nature. When neither L1 nor L2 is empty, the logic followed here is:

```
if (L1[s1] > L2[s2])
{
    L3[s3] = L2[s2];
    mergeAux(L1, s1, e1, L2, s2 + 1, e2, L3, s3 + 1, e3);
}
else
{
    L3[s3] = L1[s1];
    mergeAux(L1, s1 + 1, e1, L2, s2, e2, L3, s3 + 1, e3);
}
```

When L1 or L2 or both are empty, we need to handle the corner cases appropriately. Try to do this yourself! :)

**Merge using an auxiliary function (Iterative)**

Here mergeAux() is implemented in an iterative manner. We shall see next week in great detail how iterative algorithms have an inherent advantage over their recursive versions. Can you figure out why?

```
void mergeAux (int L1[], int s1, int e1, int L2[], int s2, int e2, int L3[], int s3, int e3)
{
    int i,j,k;
    // Traverse both arrays
    i=s1; j=s2; k=s3;
    while (i <= e1 && j <= e2) {
        // Check if current element of first array is smaller
        // than current element of second array
        // If yes, store first array element and increment first
        // array index. Otherwise do same with second array
        if (L1[i] < L2[j])
            L3[k++] = L1[i++];
        else
            L3[k++] = L2[j++];
    }
    // Store remaining elements of first array
    while (i <= e1)
        L3[k++] = L1[i++];

    // Store remaining elements of second array
    while (j <= e2)
        L3[k++] = L2[j++];
}
```

Read the above implementation and try to understand its working.

**Merge by insert**

Now, in both versions of mergeAux() that we saw, we had to allocate an additional array of size O(n). Can this be avoided?

We can modify our merging logic by now inserting the elements of L1 into L2 (similar to insertion sort). This comes at a greater time complexity but demands lesser space. You will learn more about this space-time tradeoff in week 7's labsheet.

Task 3: Implement merge sort as described above for sorting an array of integers. Define the main() and the mergeSort() functions in the file *intMergeSort.c*. Define the signatures for merge() and mergeAux() in the files *intMerge.h* and *intMergeAux.h* respectively. In the file *intMerge.c* write the implementation of merge() using mergeAux(). Write the iterative and recursive implementations in the files intMergeAuxIter() and intMergeAuxRec() respectively. Also implement the merge() by insert in the file *intMergeByInsert.c*. You have been provided a file *balances.txt* that contains the bank account balances of the customers of a given branch. Compare the time taken and peak heap space used in these three implementations for sorting the input given in the file *balances.txt*.

Task 4: Similar to Task 1, implement merge sort (using the mergeAux() by iteration) for sorting elements of `struct person` based on **height** as the key field. Now as described in Task 2, measure the time taken in sorting the files of the form *datX.csv* that have been provided to you. Plot a graph of the time taken by the algorithm against the input size. (You can use any spreadsheet software or plotting software to plot the graph.) Compare this plot with that obtained for insertion sort and try to justify the difference observed.

Home Exercise 2: Given an integer array A, find if an integer p exists in the array such that the number of integers greater than p in the array equals to p. Print the integer if it exists, else print "No such integer found". While this problem can be solved directly, you are encouraged to use your implementation of merge sort in your solution.

***Sample Input and Output:***

*Input 1:*

A = [3, 2, 1, 3]

Output:

2

Explanation:
For integer 2, there are exactly 2 elements in the array greater than 2. Hence, 2 satisfies the condition, and we print **2**.

*Input 2:*
A = [1, 1, 3, 3]
Output:
No such integer found

Home Exercise 3: Given an integer array `nums` of size N, print all the triplets `[nums[i], nums[j], nums[k]]` such that `i != j, i != k, and j != k`, and `nums[i] + nums[j] + nums[k] == 0`. Your output should not contain duplicates. The order of printing the triplets does not matter. Your solution should have a time complexity of $O(N^2)$ and a space complexity of `O(1)`.

***Sample Input and Output:***

*Example 1:*
Input: nums = [-1,0,1,2,-1,-4]
Output:
(-1, -1, 2)
(-1, 0, 1)
Explanation:
nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.
nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.
nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.
The distinct triplets are [-1,0,1] and [-1,-1,2].

*Example 2:*
Input: nums = [0,1,1]
Output: []
Explanation: The only possible triplet does not sum up to 0.

**External Merge Sort**

In all the sorting implementations we saw, we read the files into memory in a statically or dynamically allocated array. However, this is not always possible. Many a time, the data to be sorted is so large that it might not fit into memory. In this case, we use an approach known as external merge sort.

We are going to be applying our merge-sort algorithm, but treat the input as a file rather than an array. The external merge sort algorithm is described as follows:

1. Split into chunks small enough to sort in memory (“runs”).
2. Merge pairs (or groups) of runs using the external merge algorithm
3. Keep merging the resulting runs (each time = a “pass”) until left with one sorted file!

Let us say we have a huge file, for example, say input size 10 M records that need to be sorted and only up to 1 M entries can fit in memory at any given time.

Here we would start by populating an array of size 1 M with the first 1 M entries of the file as we do with normal merge-sort. Now, we would run merge sort on this array and store the sorted result in a new file (called sorted1.csv). In this way we would continue reading, sorting and storing the 10 chunks of size 1 M each in 10 files (sorted1.csv, …, sorted10.csv).

Now, we can merge these files 2 at a time to form the final sorted output file. [Note that while merging we would need to read only one record each from the two files and store the smaller one in the final sorted file. Thus only two records need to be in memory at the same time.]

In this way, the approach of external merge sort can be used to sort files having an arbitrarily large input size.

Task 5: Implement external merge sort as described above and sort the file *dat7578440.csv* which has approximately 7.5 M entries using chunks of size 1M. [You may use a smaller chunk size if 1 M does not fit in memory]

Home Exercise 4: You are the captain of the 99th Precinct of the NYPD. You have been given a list of criminals and their crimes.

Crimes can be of the following types:

- ARSON
- ASSAULT
- BURGLARY

- CRIMINAL MISCHIEF
- GRAND LARCENY
- GRAND THEFT AUTO
- HOMICIDE
- BREAKING AND ENTERING
- ROBBERY

Each criminal has a name, age and number of counts of each crime. You have to create a structure to store the criminals and their crimes. You may consider using an enum for the crimes.

The input is in the following files:

***- criminal_database.txt***

This file contains the list of criminals, their age and ID.
The format of the file is as follows:
The first line contains the number of criminals (`n`)
The next n lines contain the name, age, and ID of the criminal in the following format:
`name,age,ID` [Note that the names contain space characters in the input file.]

***- crimes.txt***

This file contains the list of the crimes, the year it was committed, and the ID of the criminal who committed them.
The format of the file is as follows:
The first line contains the number of crimes (`m`)
The next m lines contain the crime, year, and ID of the criminal in the following format:
`crime,year,ID`

Now the Police Department has decided to calculate a score for each criminal called **criminality**. The **criminality** is calculated as follows:

First, each crime is assigned a **crime coefficient** as enumerated below:

| | |
|---|---|
| - Arson: | 10 |
| - Assault: | 5 |
| - Burglary: | 5 |
| - Criminal Mischief: | 5 |
| - Grand Larceny: | 10 |
| - Grand Theft Auto: | 10 |
| - Homicide: | 20 |
| - Breaking and Entering: | 5 |

- Robbery: 10

The **criminality** of a criminal is the sum of the **crime coefficients** for each count of crime committed by the criminal.

If the criminal is less than 18 years old while committing the crime, the **crime coefficient** for that crime is multiplied by **0.5**. **The current year is 2023.**

For example, if a criminal named Doug Judy is 25 years old and has committed 2 crimes:
- Arson in 2010
- Grand Theft Auto in 2020

His year of birth is 2023 - 25 = 1998.
He committed Arson in 2010. His age while committing the crime is 2010 - 1998 = 12.
He committed Grand Theft Auto in 2020. His age while committing the crime is 2020 - 1998 = 22.

So the **criminality** of Doug Judy is 10 * 0.5 + 10 = 15 since he was a minor while committing Arson and he was an adult while committing Grand Theft Auto.

You have to create a function to calculate the **criminality** of each criminal. Create an array of elements of the structure you defined and store the required details of the criminals and the crimes committed by them in it. **Sort the array of structures in the order of the criminality of the criminals in *descending* order and store the sorted array in a file called *sorted_criminals.txt.***

***Sample Input:***

***criminal_database.txt:***

```
3
Doug Judy,52,200
Melanie Hawkins,61,3
James Dylan Borden,78,125
```

***crimes.txt:***

```
10
GRAND THEFT AUTO,1980,200
GRAND THEFT AUTO,1990,200
ARSON,2022,200
ARSON,2010,3
BREAKING AND ENTERING,2000,3
BREAKING AND ENTERING,1960,125
```

```
ROBBERY,1960,125
HOMICIDE,1975,3
HOMICIDE,1990,3
HOMICIDE,2000,125
```

***OUTPUT:***

***sorted_criminals.txt:***

```
Melanie Hawkins,61,3,45.000000
James Dylan Borden,78,125,27.500000
Doug Judy,52,200,25.000000
```

***Explanation:***

The crimes committed by Melanie Hawkins are:
ARSON,2010,3
BREAKING AND ENTERING,2000,3
HOMICIDE,1975,3
HOMICIDE,1990,3

Now, she was a minor when she committed the homicide of 1975, but was an adult during the other three crimes

So her criminality is calculated as follows:
ARSON,2010 = 10
BREAKING AND ENTERING,2000 = 5
HOMICIDE,1975 = 20 * 0.5 = 10
HOMICIDE,1990 = 20
Final criminality score = 45

Similarly, the crimes committed by James Dylan Borden are:
BREAKING AND ENTERING,1960,125 -> minor -> 5 * 0.5 = 2.5
ROBBERY,1960,125 -> minor -> 10 * 0.5 = 5
HOMICIDE,2000,125 -> adult -> 20
Final criminality = 27.5

And, the crimes committed by Doug Judy are:
GRAND THEFT AUTO,1980,200 -> minor -> 10 * 0.5 = 5
GRAND THEFT AUTO,1990,200 -> adult -> 10
ARSON,2022,200 -> adult -> 10
Final criminality = 25