Computer Science I Program 4: Games Rank Lists (Sorting)

Please Check Webcourses for the Due Date

Please read the whole assignment before you start coding

Objective

Give practice with implementing either Merge Sort or Quick Sort.
Give practice designing a sorting program that allows for several keys by which to sort.
Give practice with functions in C.
Give practice with creating a design for a program without a given list of functions or structs.

Games Rank Lists Problem

There are many people who play the NY Times Daily Games. For this program, you'll read in information about several people who play the games, including their scores on each of six games, and then be asked to sort the data based on one of seven items from greatest score to least score: any one of the six games, or the sum of scores of a player over all six games. To make the task more efficient, you'll be required to write a single sort function that will take in a parameter indicating which of the seven keys to sort by.

Since this program is a bit shorter (and easier) than some of the others, you will be asked to implement it in two versions: one using **Merge Sort** and another using **Quick Sort**. In both cases, you must use an **optimized hybrid approach**, where the algorithm switches to one of the three $O(n^2)$ sorting methods discussed in class once the array size becomes small enough during sorting—based on the threshold specified in the implementation restrictions below.

Input

The first line of input contains a single positive integer, n ($1 \le n \le 10^5$), representing the number of players for the input.

Each of the next n lines indicates information about a single player. The format of each of these lines is as follows:

```
name wScore sbScore crScore coScore stScore lbScore
```

where `name` is the name of the player (in between 1 and 12 lowercase letters), `wScore` is the player's Wordle score, `sbScore` is the player's Spelling Bee score, `crScore` is the player's Crossword score, `coScore` is the player's Connections score, `stScore` is the player's Strands score, and `lbScore` is the player's Letter Box score. Each score will be an integer in between 0 and 10^8, inclusive.

The final line of input has an integer, g ($0 \le g \le 6$), representing the game for which you want the rank list. 0 represents Wordle, 1 represents Spelling Bee, 2 represents Crossword, 3 represents Connections, 4 represents Strands, 5 represents Letter Box and 6 represents the total sum of scores over all the games.

Output

The first line of output will have the format:

```
Game Ranklist
```

where `Game` is one of the following constant strings stored in this array that will be required in your code:

```
const char GAMES[NUMGAMES][MAXSIZE+1] = {"Wordle", "Spelling Bee",
"Crossword", "Connections", "Strands", "Letter Boxed", "Total"};
```

based on the ranking criteria The indexes to this array correspond to the previously described codes for each of the games and the total score of a player.

The following ***n*** lines should be a ranked list of all the players sorted by the key value described in the input. If two players are tied with respect to the key value, then the tie should be broken by name, in alphabetical order. Each of these lines should have the following format:

```
x. name score
```

where x is the 1-based rank of the player according to that metric, name is the player's name, and score is the player's score in the associated game (or total). In addition, make sure that the name is printed with a field width of 15 characters, left justified. Formally, please use the following printf to print one line, changing the variable names as necessary:

```
printf("%d. %-15s %d\n", rank, name, score);
```

Sample Input	**Sample Output**
`7`	`Connections Ranklist`
`james 200 100 50 300 500 700`	`1. zach            800`
`marnie 500 50 50 50 700 300`	`2. james           300`
`abbie 100 100 100 100 100 100`	`3. abbie           100`
`zach 0 200 500 800 300 100`	`4. dave            100`
`emery 400 300 200 100 100 0`	`5. emery           100`
`mabel 50 40 30 80 70 90`	`6. mabel           80`
`dave 600 200 300 100 0 0`	`7. marnie          50`
`3`	
`7`	`Total Ranklist`
`james 200 100 50 300 500 700`	`1. zach            1900`
`marnie 500 50 50 50 700 300`	`2. james           1850`
`abbie 100 100 100 100 100 100`	`3. marnie          1650`
`zach 0 200 500 800 300 100`	`4. dave            1200`
`emery 400 300 200 100 100 0`	`5. emery           1100`
`mabel 50 40 30 80 70 90`	`6. abbie           600`
`dave 600 200 300 100 0 0`	`7. mabel           360`
`6`	

Implementation Requirements/Run Time Requirements

Required Constants, Struct and Constant Array

Please use the following constants, struct and constant array:

```
#define MAXSIZE 12
#define NUMGAMES 7
#define TOTAL 6
#define BASECASESIZE 30

const char GAMES[NUMGAMES][MAXSIZE+1] = {"Wordle", "Spelling Bee",
"Crossword", "Connections", "Strands", "Letter Boxed", "Total"};

typedef struct Player {
    char* name;
    int scores[NUMGAMES];
} Player;
```

Constant explanation: MAXSIZE is the maximum size of all the strings in the problem, both the names of the players and any of the game names. NUMGAMES is the total number of possible keys for sorting, so the number of games plus 1 (for the total). TOTAL is the index into the scores array of the struct Player indicating the total score for the player. BASECASESIZE is the size of the array that the base case should be triggered for the recursive sorts. For the base case, any of the three $O(n^2)$ sorts taught in class can be applied.

Sorting Requirements

1. Your main function must store an array of pointers to Players (so type Player**). Each index in this array will have a pointer which points to a single struct Player.

2. You must implement the following compare function:

```
// Returns a negative integer if the player pointed to by ptrP1
// "comes before" the player pointed to by ptrP2 on the ranklist
// sorted by the game indicated by the integer key, breaking ties
// by the player's name in alphabetical order. Returns 0 is the
// two players are identical and returns a positive integer if the
// player pointed to by ptrP2 comes before the player pointed to
// by ptrP1.
int compare(Player* ptrP1, Player* ptrP2, int key);
```

3. For the Merge Sort version of your program, your wrapper function must have the following prototype:

```
// Merge Sorts the array list of size n according to the game
// indicated by the integer key.
void mergeSort(Player** list, int n, int key);
```

4. For the Merge Sort version of your program, your wrapper function must have the following prototype:

```
// Performs a Merge Sort on list[low..high] according to the game
// indicated by the integer key.
void mergeSortRec(Player** list, int low, int high, int key);
```

5. For the Quick Sort version of your program, your wrapper function must have the following prototype:

```
// Quick Sorts the array list of size n according to the game
// indicated by the integer key.
void quickSort(Player** list, int n, int key);
```

6. For the Quick Sort version of your program, your wrapper function must have the following prototype:

```
// Performs a Quick Sort on list[low..high] according to the game
// indicated by the integer key.
void quickSortRec(Player** list, int low, int high, int key);
```

7. For your Quick Sort implementation, you must use the median of 3 strategy in your Partition function to choose the pivot item.

8. For both Merge Sort and Quick Sort, you must use a base case for a subarray of size 30 or smaller. Solve the base case via Bubble Sort, Insertion Sort or Selection Sort.

9. Your code must compile and execute on the Eustis system. The C compiler on this system is the one that the graders will be using to grade/evaluate your submissions.

Deliverables

1. Please submit a source file `ranklist_ms.c`, via Webcourses, for your solution to the problem that uses Merge Sort.

2. Please submit a source file `ranklist_qs.c`, via Webcourses, for your solution to the problem that uses Quick Sort.