CS 2414 – Data Structures – Fall 2025 – Project Four
Due 11:59 PM, November 30, 2025

Description:

You will implement **table sorting** on integer tables using three algorithms:

1. **Quick Sort** (row-wise, in-place, unstable)
2. **Heap Sort** (row-wise, in-place, unstable)
3. **LexSort (LexPass)** — a **stable, multi-pass** algorithm we specify below

For each table read from the input, your program will:

a) run Quick Sort and print the **number of comparisons** and **number of exchanges**, then print the **sorted table**
b) run Heap Sort and print the same metrics and sorted table
c) run LexSort and print the same metrics and sorted table

There are **no commands**: you sort every table with all three algorithms, in that order.

Goals

- Implement/compare multiple sorting algorithms on tabular rows.
- Correctly count and report comparisons and exchanges.
- Use a generic, reusable **multi-key table sort pattern** (provided) with a **row comparator**.
- Implement the **stable LexSort** algorithm exactly as described.

Input Format

```
T
R1 C1
<row 0 of table 1 (C1 ints)>
<row 1 of table 1>
...
<row R1-1 of table 1>

R2 C2
<row 0 of table 2>
...
<row R2-1 of table 2>

...
RT CT
<row 0 of table T>
...
<row RT-1 of table T>
```

- T = number of tables
- For table i: Ri rows, Ci columns
- Every row has exactly Ci integers separated by spaces

Output Format (strict)

For each table `t` (1-based), produce **three blocks** in this order: Quick, Heap, Lex. Each block prints metrics first, then the sorted table.

```
Table t (R=<rows>, C=<cols>) — Quick
Comparisons=<compQ> Exchanges=<exchQ>
<sorted rows...>

Table t (R=<rows>, C=<cols>) — Heap
Comparisons=<compH> Exchanges=<exchH>
<sorted rows...>

Table t (R=<rows>, C=<cols>) — Lex
Comparisons=<compL> Exchanges=<exchL>
<sorted rows...>
```

Print each row on its own line, entries separated by a single space. Insert a **blank line** between the three blocks for readability (although it is not required if your autograder prefers none—keep it consistent).

Counting Rules (very important)

We will grade both the counts and the order.

1) What is a "comparison"?
Count one comparison for each scalar cell comparison performed while ordering rows.

- For Quick/Heap (multi-key comparator):
  When comparing two rows, you may compare column 0; if equal, compare column 1; etc.
  Count 1 per column comparison you actually perform. If you break at column 2, that was 3 comparisons.
- For LexSort:
  Each pass compares rows by a single column.
  Count 1 per single-column comparison (in insertion or merge steps).

2) What is an "exchange"?
Count row-level moves (not per-cell copies).

- If you swap two rows (e.g., in Quick/Heap partitioning or heapifying), that is 1 exchange.
  (A swap of two *indices*, if you implement index-based sorting, also counts as 1 exchange.)
- In LexSort – Stable Insertion:
  Each shift of a row one position to the right counts 1 exchange; the final write of the saved "temp row" back into the gap counts 1 exchange.
- In LexSort – Stable Merge:
  Each write of a row from temporary buffer back to the main array during the merge counts 1 exchange. (Reading from the main array into temp is not counted.)

We consistently count row moves across all methods, ensuring the metrics are comparable. Do not count individual integer copies within a row.

INPUT

```
1
4 3
7 2 9
1 2 3
1 1 3
7 2 1
```

Output (illustrative; your counts may vary by pivot/heap details)

```
Table 1 (R=4, C=3) — Quick
Comparisons=?? Exchanges=??
1 1 3
1 2 3
7 2 1
7 2 9

Table 1 (R=4, C=3) — Heap
Comparisons=?? Exchanges=??
1 1 3
1 2 3
7 2 1
7 2 9

Table 1 (R=4, C=3) — Lex
Comparisons=?? Exchanges=??
1 1 3
1 2 3
7 2 1
7 2 9
```

Implementation Details

Data Representation (YOU MUST FOLLOW THIS PRESENTATION

- Store each table in a contiguous 1D array of int of size R*C (row-major).
  Access (r,c) as data[r*C + c].
- Implement a rowSwap(i, j) that exchanges all C entries of rows i and j.
  (Alternatively, keep a row-index array and swap indices; then physically reorder once, or print by index. The counting rules above still apply.)

Reusable Multi-Key Comparator (Ascending on all columns) (VERY USEFUL)

```
int compareRows(const int* A, const int* B, int C,
                /* counters by reference */ long long& cmpCount) {
    for (int col = 0; col < C; ++col) {
        ++cmpCount;                          // 1 per scalar comparison
        if (A[col] < B[col])  return -1;
        if (A[col] > B[col])  return +1;
    }
    return 0;
}
```

High-Level "Generic Table Sort" Pattern (students reuse this)

This pattern is the same for Quick, Heap, and Lex. You're giving them this *high-level* algorithm (not the internals of Quick/Heap).

```
GENERIC_TABLE_SORT(Table, R, C, ALG,
                   /* out */ long long& comparisons,
                   /* out */ long long& exchanges):
    comparisons = 0
    exchanges  = 0

    if ALG == QUICK:
        QUICKSORT_ROWS(Table, R, C, comparisons, exchanges)
    else if ALG == HEAP:
        HEAPSORT_ROWS(Table, R, C, comparisons, exchanges)
    else if ALG == LEX:
        LEXSORT_LEXPASS(Table, R, C, comparisons, exchanges)
    else:
        error
```

- All algorithms must use the same compareRows routine (or equivalent), incrementing comparisons.
- Each algorithm increments exchanges according to the Counting Rules.

Algorithms You Must Implement

1) **Quick Sort** (you implement; unstable)
- Any standard partition scheme (Lomuto/Hoare) and pivot policy (e.g., last, median-of-three).
- On each row comparison, call compareRows to increment comparisons.
- On each row swap, increment exchanges by 1.

2) **Heap Sort** (you implement; unstable)
- Build a max-heap of rows (or row indices).
- Use the comparator for ordering; increment comparisons wherever you compare heap elements.
- On each swap during heapify or extraction, increment exchanges by 1.

3) **LexSort (LexPass)** — the provided stable algorithm (implement exactly)
Idea: To sort lexicographically by columns 0..C-1 ascending, perform C stable passes: sort by column C-1, then C-2, …, finishing with column 0. Stability ensures earlier columns dominate.

You can implement each pass using either Stable Insertion Sort (the simplest option) or Stable Merge Sort (the faster option). Both must obey the counting rules.

LexSort-LexPass (overall)

```
LEXSORT_LEXPASS(Table, R, C, /*io*/ comparisons, /*io*/ exchanges):
    for col from C-1 down to 0:
        STABLE_SORT_BY_SINGLE_COLUMN(Table, R, C, col, comparisons, exchanges)
Stable Insertion Sort (single column)

STABLE_SORT_BY_SINGLE_COLUMN(Table, R, C, col, /*io*/ comp, /*io*/ exch):

    for i in 1 .. R-1:
        // Save row i
        copy row i to TEMP
        j ← i - 1
        // Move larger rows one step to the right (stable)
        while j ≥ 0:
            ++comp                  // compare Table[j][col] vs TEMP[col]
            if Table[j][col] <= TEMP[col]:     // ascending stable condition
                break
            move row j -> row j+1               // shift counts as 1 exchange
            ++exch
            --j
        write TEMP -> row j+1                // final placement counts as 1 exchange
        ++exch
```

- Comparisons: 1 per loop check where you compare Table[j][col] to TEMP[col].
- Exchanges: each row shift = 1, and the final write of TEMP = 1.

Program Flow (driver; no commands)

```
read T
for t in 1..T:
    read R, C
    read table into A[R*C]
    make a working copy W[R*C]

    // Quick
    copy A -> W
    run GENERIC_TABLE_SORT(W, R, C, QUICK, compQ, exchQ)
    print block "Table t — Quick", metrics, and W

    // Heap
    copy A -> W
    run GENERIC_TABLE_SORT(W, R, C, HEAP, compH, exchH)
    print block "Table t — Heap", metrics, and W

    // Lex
    copy A -> W
    run GENERIC_TABLE_SORT(W, R, C, LEX, compL, exchL)
    print block "Table t — Lex", metrics, and W
```

Redirected Input: Redirected input provides you a way to send a file to the standard input of a program without typing it using the keyboard. To use redirected input in Visual Studio environment, follow these steps: After you have opened or created a new project, on the menu go to project, project properties, expand configuration properties until you see Debugging, on the right you will see a set of options, and in the command arguments type "< **input filename**". The < sign is for redirected input and the **input filename** is the name of the input file (including the path if not in the working directory). A simple program that reads a matrix can be found below.

```
#include <iostream>
using namespace std;

int main () {

     int r,c,nsv;
     cin >> r >> c >> cv;
               cout << r << c << cv << endl;
     for (int i=0; i < nsv; i++) {
        cin >> rn >> cn >> val;
        cout << rn << cn << val << endl;
     }
     return 0;
}
```

Constraints

1. Use only #include <iostream> and #include <string> (and optionally #include <iomanip>).
2. Do **not** use std::sort / std::stable_sort or STL containers to hold rows. Use raw arrays.
3. No file I/O; read from std::cin, write to std::cout.
4. Your output must **exactly** match the specified format (including capitalization/spelling of tags).
5. All integers fit in 32-bit signed range.
6. None of the projects is a group project. Consulting with other members of this class our seeking coding solutions from other sources including the web on programming projects is strictly not allowed and plagiarism charges will be imposed on students who do not follow this.

Project Submission Requirements: Table Sorting (Quick, Heap, LexSort)

1. Code Development (75%)

You must write a C++ program that sorts one or more tables of integers using three algorithms — Quick Sort, Heap Sort, and LexSort (LexPass) — and prints the number of comparisons, number of exchanges, and the sorted table after each algorithm.

2. LLM and GitHub Copilot Usage Documentation (15%)

If you use AI tools, submit a short PDF report that includes:

Prompts and Suggestions

- List the actual prompts you used.
    - Examples:
        - "Write a heapify (sift-down) for heap sort over row indices using a custom comparator and counters."
        - "Stable insertion sort over rows by a single column with comparison/exchange counters."
        - "Implement Hoare partition quicksort that calls a multi-key comparator and counts swaps."
- Include notable snippets or ideas you adopted/modified.

Rationale for AI Usage

- Why you used certain prompts.
- How AI suggestions influenced your design (e.g., index-based vs row-swap), and what you changed.

Incremental Development

- Show how your implementation evolved: early stub → working version → refined counting correctness.
- If you rejected AI-generated parts, explain why.

3. Debugging and Testing Plan (10%)

Provide a detailed plan (PDF or .txt) that demonstrates correctness and validates counters.

a) Specific Test Cases

- Tiny cases: R=0, R=1, C=1.
- Duplicates: many identical rows; rows equal on column 0 but differing later (tests lexicographic tie-breaks).
- Already sorted / reverse sorted tables.
- Wide rows: larger C to stress row movement.
- Stability check: construct a table where rows tie on early columns and differ later; show that:
    - Lex preserves original order among equals (stable),
    - Quick/Heap may not (document observed behavior).

b) Issues and Resolutions

- Off-by-one errors in partitioning and heap boundaries.
- Comparison counting mistakes (e.g., forgetting to count the last equal-column check).
- Exchange counting consistency (index-swap vs row-swap; insertion shifts; merge writes).
- Pivot pathologies (Quicksort worst-case); recursion depth.
- Temporary buffer handling in merge; avoiding memory leaks.

c) Verification

- Write a helper IS_SORTED using the same comparator to assert order after each algorithm.

- Cross-check that all three algorithms yield the same final order (for a strict total order).
- Validate that comparison counts increase with harder distributions (e.g., reverse order).
- Manually compute expected counts for very small examples (e.g., R=3) to confirm your counters.

Example Submission Structure

1. Code Implementation (tablesort.cpp)
    - Contains:
        - I/O loop that reads T, then for each table reads R C and the rows.
        - A reusable COMPARE_ROWS and rowSwap (or index strategy with final materialization/printing).
        - TABLE_SORT dispatcher.
        - QUICKSORT_ROWS, HEAPSORT_ROWS, LEXSORT_LEXPASS (+ stable pass).
        - Exact printing per spec.
2. LLM Usage Documentation (llm_usage.pdf)
    - Prompts, rationale, incremental development notes, and what you kept/changed.
3. Testing Plan (testing.pdf or testing.txt)
    - Test cases, known issues & fixes, and verification steps (including stability demonstration for Lex).

Notes & Hints

- Using row indices can greatly reduce per-exchange data movement for wide rows while keeping counting fair (you still count index swaps/writes as exchanges).
- For Quicksort, consider median-of-three to reduce worst-case behavior.
- For LexSort, stable insertion is simpler to code; stable merge is faster on large R. Either is acceptable if you follow the counting rules.