Name: Brandon Mountan

Q1	**There is no significant time accumulated by any functions during the profiling run. This could be due to short runtime, I/O-bound behavior, or insufficient sampling resolution**
Q2	**Flat Profile Analysis** **Key Metrics** • **Total Runtime**: 3.12 seconds. • **Sampling Interval**: Each sample counts as 0.01 seconds. **Top Time-Consuming Functions** 1. **std::operator==<char>**: ○ **% Time**: 33.52% ○ **Cumulative Time**: 1.04 seconds ○ **Calls**: 647,482,750 ○ **Time per Call**: 0.00 seconds ○ **Analysis**: This function is the most time-consuming, likely due to its high number of calls. It is used for comparing std::string objects, which suggests the program performs a significant amount of string comparisons. 2. **std::operator< <char>**: ○ **% Time**: 15.47% ○ **Cumulative Time**: 1.52 seconds ○ **Calls**: 379,465,206 ○ **Time per Call**: 0.00 seconds ○ **Analysis**: This function is the second most time-consuming, indicating that the program performs many string comparisons for ordering (e.g., sorting or searching). 3. **std::operator!=<char>**: ○ **% Time**: 14.83% ○ **Cumulative Time**: 1.98 seconds ○ **Calls**: 647,482,750 ○ **Time per Call**: 0.00 seconds ○ **Analysis**: Similar to std::operator==, this function is heavily used for string comparisons, contributing significantly to the runtime. 4. **search1**: ○ **% Time**: 14.83% ○ **Cumulative Time**: 2.45 seconds ○ **Calls**: 38,948 ○ **Time per Call**: 0.00 seconds

   - **Analysis**: This function is a custom search function that operates on an array of std::string objects. Its high percentage of runtime suggests it is a critical part of the program's logic.

5. **sort1**:
   - **% Time**: 13.86%
   - **Cumulative Time**: 2.88 seconds
   - **Calls**: 2
   - **Time per Call**: 0.22 seconds
   - **Analysis**: This function is called only twice but consumes a significant portion of the runtime, indicating that it performs a computationally expensive operation (likely sorting a large dataset).
6. **std::char_traits<char>::compare**:
   - **% Time**: 7.57%
   - **Cumulative Time**: 3.11 seconds
   - **Calls**: 108,224,639
   - **Time per Call**: 0.00 seconds
   - **Analysis**: This low-level function is used for comparing characters within strings. Its high number of calls suggests it is a building block for many string operations.

Other Functions

- **find_print_add_records**:
  - **% Time**: 0.00%
  - **Cumulative Time**: 3.12 seconds
  - **Calls**: 2
  - **Time per Call**: 0.00 seconds
  - **Analysis**: This function is called twice but does not contribute significantly to the runtime.
- **readFile**:
  - **% Time**: 0.00%
  - **Cumulative Time**: 3.12 seconds
  - **Calls**: 2
  - **Time per Call**: 0.00 seconds
  - **Analysis**: This function is responsible for reading data from a file but does not significantly impact the runtime.
- **Initialization Functions**:
  - Functions like _GLOBAL__sub_I_* and __static_initialization_and_destruction_0 are called once during program startup and have negligible runtime impact.

Summary of Findings

1. **String Operations Dominate Runtime**:
    - The program spends most of its time performing string comparisons (std::operator==, std::operator!=, std::operator<, and std::char_traits<char>::compare).
    - These operations are called hundreds of millions of times, indicating that the program processes a large amount of string data.
2. **Custom Functions**:
    - The search1 and sort1 functions are critical to the program's performance. While search1 is called frequently, sort1 is called only twice but consumes a significant portion of the runtime.
3. **I/O Operations**:
    - Functions like readFile and find_print_add_records do not significantly impact the runtime, suggesting that the program is CPU-bound rather than I/O-bound.

Recommendations for Optimization

1. **Optimize String Comparisons**:
    - Reduce the number of string comparisons by using more efficient data structures (e.g., hash tables for lookups).
    - Consider using std::string_view instead of std::string to avoid unnecessary copies and improve comparison performance.
2. **Improve Search and Sort Algorithms**:
    - Optimize the search1 function to reduce its reliance on expensive string comparisons.
    - Use a more efficient sorting algorithm or data structure (e.g., std::set or std::unordered_set) if applicable.
3. **Profile with Larger Inputs**:
    - Run the profiler with larger datasets to identify potential scalability issues.
4. **Parallelize Computations**:
    - If the program processes independent data, consider parallelizing the search1 or sort1 functions using multithreading or GPU acceleration.

Q3

The function that consumes the highest percentage of the program's execution time is:

- **std::operator==<char>** (used for string equality comparison).
- **Percentage**: **33.52%**

Q4	The bottleneck in the program is: • **search1**. o It consumes **14.83%** of the total runtime directly. o However, it calls **std::operator!=** and **std::operator==**, which together account for **88.3%** of the runtime (33.52% + 55.8%). o The high number of calls to these functions (647,482,750) indicates that the search1 function is heavily reliant on string comparisons, making it the primary bottleneck.
Q5	For the **search1** function: • **Self Seconds per Call**: **0.00 seconds**. o This is the time spent executing the search1 function itself, excluding the time spent in its child functions. o The value is very small because most of the work is done in the child functions (e.g., std::operator!= and std::operator==). • **Total Seconds per Call**: **0.056 milliseconds** (calculated as total time / calls = 1.74 seconds / 38,948 calls). o This includes the time spent in search1 and all its child functions. o The higher total time per call indicates that the function's performance is heavily influenced by the child functions it calls.
Q6	For the **find_print_add_records** function: • **Self Seconds per Call**: **0.00 seconds**. o This is the time spent executing the find_print_add_records function itself, excluding the time spent in its child functions. o The value is very small because most of the work is done in the child functions (e.g., search1). • **Total Seconds per Call**: **1.10 seconds** (calculated as total time / calls = 2.20 seconds / 2 calls). o This includes the time spent in find_print_add_records and all its child functions. o The high total time per call indicates that the function's performance is heavily influenced by the child functions it calls, particularly search1.
Q7	The child function contributing most to the time of the **main** function is: • **find_print_add_records**. o It propagates **2.20 seconds** to main.

	○ This is the total time spent in find_print_add_records and all its child functions.
Q9	⍰ **Optimization Level 0 (-O0)**: • The program is unoptimized, and the runtime is dominated by low-level string operations (std::operator==, std::operator!=, std::operator<, and std::char_traits<char>::compare). • These functions account for **69.17%** of the runtime, indicating that the program spends most of its time performing string comparisons. ⍰ **Optimization Level 1 (-O1)**: • The runtime is significantly reduced (from 3.22 seconds to 1.69 seconds). • The sort1 function becomes the dominant function, accounting for **65.33%** of the runtime. • The search1 function also becomes more prominent, accounting for **32.81%** of the runtime. • Low-level string operations are no longer visible in the profile, indicating that the compiler has optimized them or inlined them. ⍰ **Optimization Level 2 (-O2)**: • The runtime increases slightly to 1.80 seconds, which is unusual and may be due to variability in profiling or system load. • The sort1 and search1 functions remain the dominant functions, with **63.47%**and **36.75%** of the runtime, respectively. ⍰ **Optimization Level 3 (-O3)**: • The runtime is further reduced to 1.41 seconds. • The sort1 and search1 functions remain the dominant functions, with **59.69%**and **40.50%** of the runtime, respectively. • The compiler's aggressive optimizations (e.g., loop unrolling, vectorization) have further improved performance.
Q10	The best optimization level for minimizing execution time is **-O3**: • It achieves the lowest total runtime (**1.41 seconds**). • It aggressively optimizes the code, reducing the overhead of low-level operations and improving the performance of critical functions like sort1 and search1.
Q12	The best-performing combination of functions is: • **Sorting Algorithm**: **sort3 (Merge Sort)**. • **Search Algorithm**: **search2 (Binary Search)**. • **Execution Time**: **0.01 seconds**.

This combination achieves the lowest execution time, making it the most efficient.

Q13

To calculate the program enhancement percentage, we compare the execution time of the **worst-performing combination** (baseline) with the **best-performing combination**(optimized).

1. **Baseline (Worst-Performing Combination)**:
   - **Sorting Algorithm**: sort2 (Bubble Sort).
   - **Search Algorithm**: search1 (Linear Search).
   - **Execution Time**: **1.89 seconds**.
2. **Optimized (Best-Performing Combination)**:
   - **Sorting Algorithm**: sort3 (Merge Sort).
   - **Search Algorithm**: search2 (Binary Search).
   - **Execution Time**: **0.01 seconds**.
3. **Enhancement Percentage**:
   The formula for enhancement percentage is:

   Enhancement Percentage = (Old Runtime − New Runtime) / Old Runtime × 100

   Substituting the values:

   Enhancement Percentage = (1.89 − 0.01) / 1.89 × 100 = 1.88 / 1.89 × 100 ≈ 99.47% Enhancement Percentage

Q8:

Optimization Level	**Total execution time**
O0 (default)	3.22 s
O1	1.69 s
O2	1.80 s
O3	1.41 s

Q11:

Sort function	**Search function**	**Total execution time**
Sort1	Search1	1.41 s
Sort2	Search1	1.89 s
Sort3	Search1	0.61 s
Sort1	Search2	0.70 s
Sort2	Search2	1.48 s
Sort3	Search2	0.01 s