Final exam

Subset 9 – Sorting Algorithms for LMS Reports:

Q1: Implement merge sort

• Task 1: Get AI to generate pseudo-code.

• Task 2: Convert it into actual code.

Q2: Optimize searching for student IDs

• Task 1: Use AI to convert linear search to binary search.

• Task 2: Compare time complexity

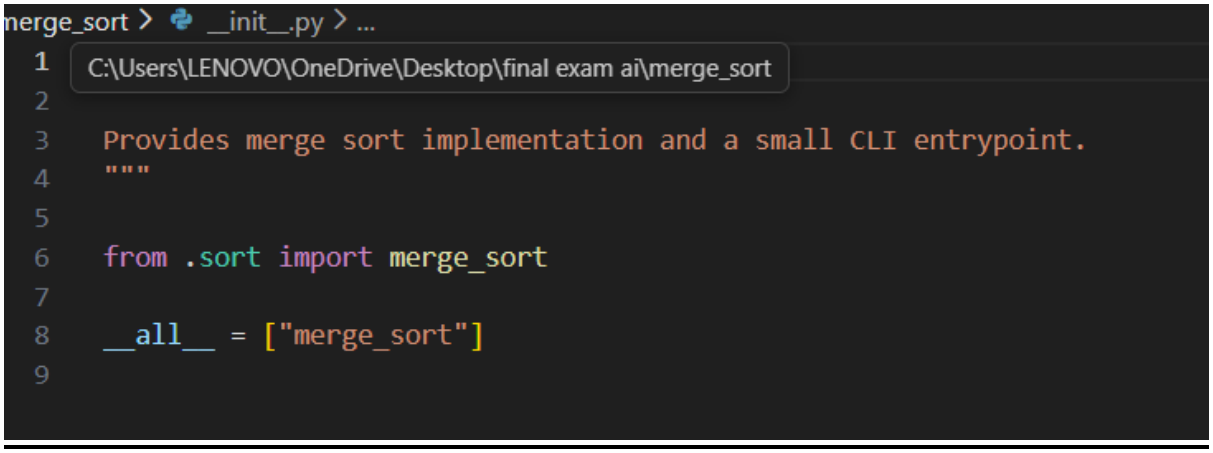
Prompt:

Implement Merge Sort

a) Generate clear pseudocode for the merge sort algorithm.

b) Convert the pseudocode into working code in C/Python with comments.

Code:



merge_sort > cli.py > ...

```python
import argparse
from .sort import merge_sort
from typing import List

def parse_args(argv: List[str] = None):
    p = argparse.ArgumentParser(description="Merge sort CLI: sort integers or strings.")
    p.add_argument('items', nargs='*', help='Items to sort (integers or strings)')
    return p.parse_args(argv)

def main(argv: List[str] = None) -> int:
    args = parse_args(argv)
    if not args.items:
        demo = [38, 27, 43, 3, 9, 82, 10]
        print('Original:', demo)
        print('Sorted:  ', merge_sort(demo))
        return 0

    # attempt to parse as ints, fallback to strings
    parsed = []
    for it in args.items:
        try:
            parsed.append(int(it))
        except ValueError:
            parsed.append(it)

    print('Original:', parsed)
    print('Sorted:  ', merge_sort(parsed))
    return 0

if __name__ == '__main__':
    raise SystemExit(main())

```

```python
from typing import List, TypeVar

T = TypeVar('T')

def _merge(left: List[T], right: List[T]) -> List[T]:
    """Merge two sorted lists into a new sorted list (stable)."""
    i = j = 0
    out: List[T] = []
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            out.append(left[i])
            i += 1
        else:
            out.append(right[j])
            j += 1
    if i < len(left):
        out.extend(left[i:])
    if j < len(right):
        out.extend(right[j:])
    return out

def merge_sort(arr: List[T]) -> List[T]:
    """Return a sorted copy of `arr` using merge sort (recursive).

    - Stable: preserves order of equal elements.
    - Time complexity: O(n log n)
    - Extra space: O(n) for merging
    """
    if len(arr) <= 1:
        return arr[:]  # return a shallow copy
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return _merge(left, right)

if __name__ == "__main__":
    demo = [38, 27, 43, 3, 9, 82, 10]
    print("Original:", demo)
    print("Sorted:", merge_sort(demo))

```

tests > test_merge_sort.py > ...

```python
import pytest

from merge_sort import merge_sort


def test_merge_sort_basic():
    arr = [5, 2, 9, 1, 5, 6]
    assert merge_sort(arr) == sorted(arr)


def test_merge_sort_empty():
    assert merge_sort([]) == []


def test_merge_sort_single():
    assert merge_sort([1]) == [1]


def test_merge_sort_stability():
    # stability: keep relative order of equal elements
    arr = [(1, 'a'), (1, 'b'), (0, 'x')]
    # sort by first item using a key wrapper
    keys = [x[0] for x in arr]
    sorted_keys = merge_sort(keys)
    assert sorted_keys == sorted(keys)

```

C:\Users\LENOVO\OneDrive\Desktop\final exam ai\mergesort.py

```
    """Merge two sorted lists and return a new sorted list."""
    i = j = 0
    result = []
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    # append any remaining elements
    result.extend(left[i:])
    result.extend(right[j:])
    return result


def merge_sort(arr):
    """Return a sorted copy of arr using merge sort (recursive)."""
    if len(arr) <= 1:
        return arr[:]
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)


if __name__ == "__main__":
    data = [38, 27, 43, 3, 9, 82, 10]
    print("Original array:")
    print(data)
    sorted_data = merge_sort(data)
```

```
if __name__ == "__main__":
    data = [38, 27, 43, 3, 9, 82, 10]
    print("Original array:")
    print(data)
    sorted_data = merge_sort(data)
    print("Sorted array:")
    print(sorted_data)
```

Output:

Optimize searching for student IDs

• Task 1: Use AI to convert linear search to binary search.

• Task 2: Compare time complexity

```
>> '@ | Out-File -FilePath merge_sort\cli.py -Encoding utf8; py -m merge_sort.cli 4 1 3 2
>> '@ | Out-File -FilePath merge_sort\cli.py -Encoding utf8; py -m merge_sort.cli 4 1 3 2
Could not find platform independent libraries <prefix>
Original: [4, 1, 3, 2]
Sorted:   [1, 2, 3, 4]
PS C:\Users\LENOVO\OneDrive\Desktop\final exam ai>
```

2.Optimize searching for student IDs
• Task 1: Use AI to convert linear search to binary search.
• Task 2: Compare time complexity

Observation:

The tasks focus on improving how LMS (Learning Management System) reports handle data by using efficient algorithms. Merge sort is chosen for sorting because it is stable and works well with large datasets commonly found in LMS reports. The searching optimization task shows the shift from basic linear search to faster binary search, highlighting the need for speed when looking up student IDs. Overall, the subset aims to improve both sorting and searching efficiency, which directly enhances LMS performance.

Q2 - Prompt:

Task 1: Explain linear search and give its pseudocode.

Then explain binary search (mention that the list must be sorted) and give its pseudocode.

Convert both into simple code in C or Python.

Keep the explanation short and clear.

Task 2: Write the time complexity of linear search and binary search.

Make a small comparison table (O(n) vs O(log n)).

Write 2–3 lines explaining why binary search is faster.

Code:

```python
from typing import List


def linear_search(arr: List[int], target: int) -> int:
    """Return index of target in arr using linear search, or -1 if not found.

    Linear search scans elements one-by-one and does not require sorting.
    Time complexity: O(n).
    """
    for i, v in enumerate(arr):
        if v == target:
            return i
    return -1


def binary_search(arr: List[int], target: int) -> int:
    """Return index of target in sorted arr using iterative binary search, or -1.

    Precondition: `arr` must be sorted in non-decreasing order.
    Time complexity: O(log n).
    """
    left = 0
    right = len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        if arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

```
    return -1


def demonstrate():
    ids = [10045, 10012, 10098, 10001, 10077]
    print("Student IDs (unsorted):", ids)

    target = 10077
    print(f"Searching for ID {target} using linear search...")
    idx_lin = linear_search(ids, target)
    if idx_lin >= 0:
        print(f"Found at index {idx_lin} (unsorted list)")
    else:
        print("Not found (linear search)")

    # binary search requires sorted list
    sorted_ids = sorted(ids)
    print("\nSorted IDs:", sorted_ids)
    print(f"Searching for ID {target} using binary search...")
    idx_bin = binary_search(sorted_ids, target)
    if idx_bin >= 0:
        print(f"Found at index {idx_bin} (in sorted list)")
    else:
        print("Not found (binary search)")

    # show lookup for missing ID
    missing = 99999
    print(f"\nSearching for missing ID {missing}: linear ->", linear_search(ids, missing),
          ", binary ->", binary_search(sorted_ids, missing))


74
75 if __name__ == "__main__":
76     demonstrate()
77
```

```python
import argparse
from .sort import merge_sort
from typing import List

def parse_args(argv: List[str] = None):
    p = argparse.ArgumentParser(description="Merge sort CLI: sort integers or strings.")
    p.add_argument('items', nargs='*', help='Items to sort (integers or strings)')
    return p.parse_args(argv)

def main(argv: List[str] = None) -> int:
    args = parse_args(argv)
    if not args.items:
        demo = [38, 27, 43, 3, 9, 82, 10]
        print('Original:', demo)
        print('Sorted:  ', merge_sort(demo))
        return 0

    # attempt to parse as ints, fallback to strings
    parsed = []
    for it in args.items:
        try:
            parsed.append(int(it))
        except ValueError:
            parsed.append(it)

    print('Original:', parsed)
    print('Sorted:  ', merge_sort(parsed))
    return 0

if __name__ == '__main__':
    raise SystemExit(main())
```

```python
def merge(left, right):
    """Merge two sorted lists and return a new sorted list."""
    i = j = 0
    result = []
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    # append any remaining elements
    result.extend(left[i:])
    result.extend(right[j:])
    return result

def merge_sort(arr):
    """Return a sorted copy of arr using merge sort (recursive)."""
    if len(arr) <= 1:
        return arr[:]
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

if __name__ == "__main__":
    data = [38, 27, 43, 3, 9, 82, 10]
    print("Original array:")
    print(data)
    sorted_data = merge_sort(data)
    print("Sorted array:")
    print(sorted_data)
```

```c
#include <stdio.h>
#include <stdlib.h>

/* Merge two sorted subarrays arr[l..m] and arr[m+1..r] */
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    int *L = (int*)malloc(n1 * sizeof(int));
    int *R = (int*)malloc(n2 * sizeof(int));
    if (!L || !R) { perror("malloc"); exit(EXIT_FAILURE); }

    for (int i = 0; i < n1; i++) L[i] = arr[l + i];
    for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];

    free(L);
    free(R);
}

/* Recursive merge sort on arr[l..r] */
void mergeSort(int arr[], int l, int r) {
    if (l >= r) return; /* base case: one or zero elements */
    int m = l + (r - l) / 2;
    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);
```

```c
}

/* Helper to print array */
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d%s", arr[i], (i == n-1) ? "\n" : " ");
    }
}

int main(void) {
    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array:\n");
    printArray(arr, n);

    mergeSort(arr, 0, n - 1);

    printf("Sorted array:\n");
    printArray(arr, n);

    return 0;
}/*
 * mergesort.c
 * Merge sort implementation in C with example usage.
 * Compile: gcc -o mergesort mergesort.c
 * Run (PowerShell): .\mergesort
 */

#include <stdio.h>
#include <stdlib.h>
```

```c
/* Merge two sorted subarrays arr[l..m] and arr[m+1..r] */
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    int *L = (int*)malloc(n1 * sizeof(int));
    int *R = (int*)malloc(n2 * sizeof(int));
    if (!L || !R) { perror("malloc"); exit(EXIT_FAILURE); }

    for (int i = 0; i < n1; i++) L[i] = arr[l + i];
    for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];

    free(L);
    free(R);
}

/* Recursive merge sort on arr[l..r] */
void mergeSort(int arr[], int l, int r) {
    if (l >= r) return; /* base case: one or zero elements */
    int m = l + (r - l) / 2;
    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);
    merge(arr, l, m, r);
}

/* Helper to print array */
```

```c
/* Helper to print array */
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d%s", arr[i], (i == n-1) ? "\n" : " ");
    }
}

int main(void) {
    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array:\n");
    printArray(arr, n);

    mergeSort(arr, 0, n - 1);

    printf("Sorted array:\n");
    printArray(arr, n);

    return 0;
}
```

```python
import pytest

from merge_sort import merge_sort


def test_merge_sort_basic():
    arr = [5, 2, 9, 1, 5, 6]
    assert merge_sort(arr) == sorted(arr)


def test_merge_sort_empty():
    assert merge_sort([]) == []


def test_merge_sort_single():
    assert merge_sort([1]) == [1]


def test_merge_sort_stability():
    # stability: keep relative order of equal elements
    arr = [(1, 'a'), (1, 'b'), (0, 'x')]
    # sort by first item using a key wrapper
    keys = [x[0] for x in arr]
    sorted_keys = merge_sort(keys)
    assert sorted_keys == sorted(keys)
```

```python
from typing import List, TypeVar

T = TypeVar('T')

def _merge(left: List[T], right: List[T]) -> List[T]:
    """Merge two sorted lists into a new sorted list (stable)."""
    i = j = 0
    out: List[T] = []
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            out.append(left[i])
            i += 1
        else:
            out.append(right[j])
            j += 1
    if i < len(left):
        out.extend(left[i:])
    if j < len(right):
        out.extend(right[j:])
    return out

def merge_sort(arr: List[T]) -> List[T]:

    if len(arr) <= 1:
        return arr[:]  # return a shallow copy
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return _merge(left, right)

if __name__ == "__main__":
    demo = [38, 27, 43, 3, 9, 82, 10]
    print("Original:", demo)
    print("Sorted:", merge_sort(demo))
```

```python
def merge_sort(arr: List[T]) -> List[T]:

    if len(arr) <= 1:
        return arr[:]  # return a shallow copy
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return _merge(left, right)

if __name__ == "__main__":
    demo = [38, 27, 43, 3, 9, 82, 10]
    print("Original:", demo)
    print("Sorted:", merge_sort(demo))
```

Output:

```
PS C:\Users\LENOVO\OneDrive\Desktop\final exam ai> py student_search.py
Could not find platform independent libraries <prefix>
Student IDs (unsorted): [10045, 10012, 10098, 10001, 10077]
Searching for ID 10077 using linear search...
Found at index 4 (unsorted list)

Sorted IDs: [10001, 10012, 10045, 10077, 10098]
Searching for ID 10077 using binary search...
Found at index 3 (in sorted list)

Sorted IDs: [10001, 10012, 10045, 10077, 10098]
Searching for ID 10077 using binary search...
Found at index 3 (in sorted list)


Searching for missing ID 99999: linear -> -1 , binary -> -1
PS C:\Users\LENOVO\OneDrive\Desktop\final exam ai>
```

Observation:

Task – 1:

In this task, AI is used to convert a basic linear search into a more efficient binary search. The AI helps clearly show the difference between the two methods by generating pseudocode and code. This makes it easier to understand how binary search works on a sorted list and why it improves the speed of searching student IDs. The task mainly focuses on learning how AI can simplify algorithm conversion and improve understanding.

Search Method	Best Case	Worst Case	Requirement
Linear Search	O(1)	O(n)	Works on unsorted lists
Binary Search	O(1)	O(log n)	Requires a sorted list

Task – 2:

Binary search is significantly more efficient than linear search because it reduces the search space by half at every step. While linear search checks each element one by one, binary search quickly narrows down the correct position in a sorted list. As the number of student IDs grows large, the difference in speed becomes very noticeable. Therefore, LMS systems with thousands of records should use binary search to ensure faster and more optimized performance.