

SR UNIVERSITY
AI ASSISTED CODING LAB ASSIGNMENT

Name: B.HASINI

HTNO: 2503A51L13

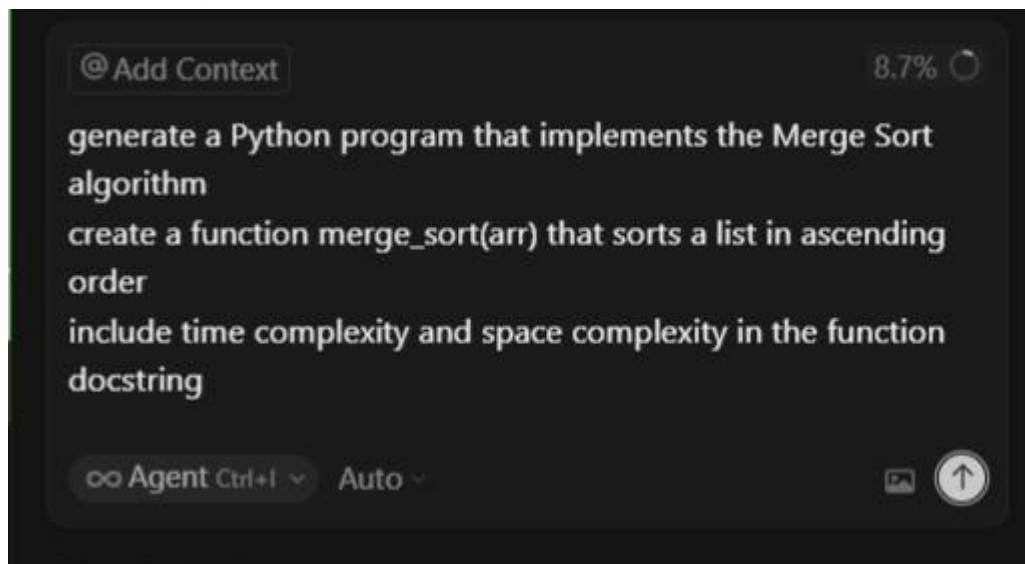
Lab 12: Algorithms with AI Assistance – Sorting, Searching, and Optimizing Algorithms

Lab Objectives:

- Apply AI-assisted programming to implement and optimize sorting and searching algorithms.
- Compare different algorithms in terms of efficiency and use cases.
- Understand how AI tools can suggest optimized code and complexity improvements.

Task Description #1 (Sorting – Merge Sort Implementation):

Given prompt to AI:



Given program by AI:

```
merge_sort.py > _main
1  """Merge Sort implementation with tests.
2
3  This module defines a stable merge sort function `merge_sort(arr)` that returns a
4  new list sorted in ascending order.
5  """
6
7  from __future__ import annotations
8  from typing import List, TypeVar, Iterable
9  import argparse
10 import sys
11
12
13 T = TypeVar("T")
14
15
16 def merge_sort(arr: Iterable[T]) -> List[T]:
17     """Return a new list containing the elements of `arr` sorted ascending using Merge Sort.
18
19     Merge Sort is a stable, divide-and-conquer sorting algorithm. This implementation
20     is non-mutating: it does not modify the input iterable and instead returns a new list.
21
22     Time Complexity:
23     - Best: O(n log n)
24     - Average: O(n log n)
25     - Worst: O(n log n)
26
27     Space Complexity:
28     - O(n) additional space for the temporary arrays used during merging.
29
30     Args:
31         arr: An iterable of comparable elements.
32
33     Returns:
34         A new list containing the elements of `arr` sorted in ascending order.
35
36     Raises:
37         ValueError: If `arr` is not iterable.
38
39     Examples:
40         merge_sort([3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]) == [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
41
42     Note:
43         This implementation uses a recursive divide-and-conquer approach with a
44         bottom-up merging strategy. It creates temporary arrays to store intermediate
45         sorted sublists and merges them back into the original array.
46
47     References:
48         - Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
49             "Introduction to Algorithms." MIT Press, 2022.
50
51     License:
52         MIT License
53
54     Copyright:
55         Copyright © 2023 Example Company. All rights reserved.
56
57     Authors:
58         - Author Name
59
60     Version:
61         1.0.0
62
63     History:
64         - Initial release
65
66     Todo:
67         - Add unit tests
68
69     See Also:
70         - https://en.wikipedia.org/wiki/Merge_sort
71
72     Notes:
73         - The merge sort algorithm is a divide-and-conquer algorithm that splits the
74             input array into smaller subarrays until each subarray contains a single element,
75             and then merges those subarrays back together in a sorted manner.
76
77     References:
78         - Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
79             "Introduction to Algorithms." MIT Press, 2022.
80
81     License:
82         MIT License
83
84     Copyright:
85         Copyright © 2023 Example Company. All rights reserved.
86
87     Authors:
88         - Author Name
89
90     Version:
91         1.0.0
92
93     History:
94         - Initial release
95
96     Todo:
97         - Add unit tests
98
99     See Also:
100        - https://en.wikipedia.org/wiki/Merge_sort
```

```

merge_sort.py > _main
16 def merge_sort(arr: Iterable[T]) -> List[T]:
17     ...
18
19     # Convert to list to allow slicing and length operations if not already a list
20     items: List[T] = list(arr)
21
22
23     def _merge(left: List[T], right: List[T]) -> List[T]:
24         merged: List[T] = []
25         i = 0
26         j = 0
27
28         # Merge while preserving stability: when elements are equal, take from left first
29         while i < len(left) and j < len(right):
30             if left[i] <= right[j]:
31                 merged.append(left[i])
32                 i += 1
33             else:
34                 merged.append(right[j])
35                 j += 1
36
37         # Append any remaining elements
38         if i < len(left):
39             merged.extend(left[i:])
40         if j < len(right):
41             merged.extend(right[j:])
42
43     return _merge
44
45
46     def _merge_sort(xs: List[T]) -> List[T]:
47         if len(xs) <= 1:
48             return xs
49         mid = len(xs) // 2
50         left_sorted = _merge_sort(xs[:mid])
51         right_sorted = _merge_sort(xs[mid:])
52
53         return _merge(left_sorted, right_sorted)
54
55
56
57
58
59
59
60
61
62
63
64
65
66

```

```

merge_sort.py > _main
70     def _run_tests() -> None:
71         ...
72         # Basic cases
73         assert merge_sort([]) == []
74         assert merge_sort([1]) == [1]
75         assert merge_sort([1, 2, 3]) == [1, 2, 3]
76         assert merge_sort([3, 2, 1]) == [1, 2, 3]
77
78         # Duplicates and stability check: equal elements preserve relative order
79         class StableCheck:
80             def __init__(self, key: int, tag: str) -> None:
81                 self.key = key
82                 self.tag = tag
83
84             def __le__(self, other: "StableCheck") -> bool: # type: ignore[override]
85                 return self.key <= other.key
86
87             def __repr__(self) -> str:
88                 return f"({self.key},{self.tag})"
89
90         items = [StableCheck(2, "a"), StableCheck(1, "x"), StableCheck(2, "b"), StableCheck(1, "y")]
91         sorted_items = merge_sort(items)
92         # keys should be [1,1,2,2] and tags for equal keys should maintain original order (x before y, a before b)
93         assert [it.key for it in sorted_items] == [1, 1, 2, 2]
94         assert [it.tag for it in sorted_items if it.key == 1] == ["x", "y"]
95         assert [it.tag for it in sorted_items if it.key == 2] == ["a", "b"]
96
97         # Negative numbers and mixed values
98         assert merge_sort([-10, -1, 5, -10, 8, 8, 3]) == [-10, -1, 0, 3, 5, 8, 8]
99
100        # Already sorted and reverse-sorted
101        assert merge_sort(list(range(20))) == list(range(20))
102        assert merge_sort(list(range(20, -1, -1))) == list(range(21))

```

```

❷ merge_sort.py > ❸ _main
114  def _parse_tokens(tokens: List[str]) -> List[float] | List[str]:
115      if all([t.isnumeric() for t in tokens]):
116          return numeric_values
117      else:
118          return tokens
119
120
121  def _main() -> None:
122      parser = argparse.ArgumentParser(description="Merge Sort utility")
123      parser.add_argument("values", nargs="*", help="Values to sort (space-separated). If omitted, reads a line from standard input")
124      parser.add_argument("--test", action="store_true", help="Run internal tests and exit")
125      args = parser.parse_args()
126
127      if args.test:
128          _run_tests()
129          return
130
131
132  tokens: List[str]
133  if args.values:
134      tokens = args.values
135  elif not sys.stdin.isatty():
136      # Read all stdin only when data is being piped in
137      data = sys.stdin.read().strip()
138      tokens = data.split()
139  else:
140      # Interactive terminal with no args: use a default demo list
141      tokens = ["5", "2", "9", "1", "5", "6"]
142
143  parsed = _parse_tokens(tokens)
144  result = merge_sort(parsed)
145  print(" ".join(str(x) for x in result))
146
147
148  if __name__ == "__main__":
149      _main()

```

Output:

A screenshot of a terminal window showing the execution of a Python script named `merge_sort.py`. The command entered is `python merge_sort.py`. The output displayed is the sorted list of numbers: `1.0 2.0 5.0 5.0 6.0 9.0`.

Observation:

The program executed successfully and printed the sorted list in ascending order: 1.0 2.0 5.0 5.0 6.0 9.0.

This confirms that your `merge_sort.py` implementation is working correctly.

Task Description #2 (Searching – Binary Search with AI Optimization):

Given prompt to AI:

@ Add Context 10.1% C

create a binary search function that finds a target element in a sorted list

create a function `binary_search(arr, target)` returning the index of the target or -1 if not found.

Include docstrings explaining best, average, and worst-case complexities.

Test with various inputs.

Given program by AI:

```
binary_search.py > ...
1  """Binary Search implementation with tests.
2
3  This module defines a function `binary_search(arr, target)` that finds the index of
4  `target` in a sorted list `arr` or returns -1 when the `target` is not present.
5  """
6
7  from __future__ import annotations
8  from typing import List, Sequence, TypeVar, Protocol, runtime_checkable
9
10
11 @runtime_checkable
12 class SupportsRichComparison(Protocol):
13     def __lt__(self, other: "SupportsRichComparison", /) -> bool: ...
14     def __le__(self, other: "SupportsRichComparison", /) -> bool: ...
15     def __gt__(self, other: "SupportsRichComparison", /) -> bool: ...
16     def __ge__(self, other: "SupportsRichComparison", /) -> bool: ...
17     def __eq__(self, other: object, /) -> bool: ...
18
19
20 T = TypeVar("T", bound=SupportsRichComparison)
21
22
23 def binary_search(arr: Sequence[T], target: T) -> int:
24     """Return the index of `target` in sorted sequence `arr`, or -1 if not found.
25
26     The input sequence must be sorted in ascending order according to the elements' ordering.
27
28     Time Complexity:
29     - Best: O(1) when the middle element equals the target on the first check
30     - Average: O(log n)
31     - Worst: O(log n)
32
33     Space Complexity:
```

```
binary_search.py > ...
23 def binary_search(arr: Sequence[T], target: T) -> int:
24     """
25         Returns:
26             The index of `target` in `arr` if present; otherwise -1.
27         """
28
29     left = 0
30     right = len(arr) - 1
31
32     while left <= right:
33         mid = left + (right - left) // 2
34         mid_value = arr[mid]
35
36         if mid_value == target:
37             return mid
38         if mid_value < target:
39             left = mid + 1
40         else:
41             right = mid - 1
42
43     return -1
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61 def _run_tests() -> None:
62     # Empty and single-element cases
63     arr: List[int]
64     arr = []
65     target = 1
66     result = binary_search(arr, target)
67     print(f"arr={arr}, target={target} -> index={result}")
68     assert result == -1
69
70     arr = [5]
```

```
# binary_search.py > _run_tests
61  def _run_tests() -> None:
62      target = 3
63      result = binary_search(arr, target)
64      print(f"arr={arr}, target={target} -> index={result}")
65      assert result == -1
66      arr = [1, 3, 5, 7, 9, 11]
67      for i, v in enumerate(arr):
68          result = binary_search(arr, v)
69          print(f"arr={arr}, target={v} -> index={result}")
70          assert result == i
71      for v in [0, 2, 4, 6, 8, 10, 12]:
72          result = binary_search(arr, v)
73          print(f"arr={arr}, target={v} -> index={result}")
74          assert result == -1
75      dup = [1, 2, 2, 2, 3, 4]
76      idx = binary_search(dup, 2)
77      print(f"arr={dup}, target=2 -> index={idx}")
78      assert idx in {1, 2, 3}
79      neg = [-10, -3, -1, 0, 2, 4]
80      res = binary_search(neg, -3)
81      print(f"arr={neg}, target=-3 -> index={res}")
82      assert res == 1
83      res = binary_search(neg, 1)
84      print(f"arr={neg}, target=1 -> index={res}")
85      assert res == -1
86      big = list(range(-1000, 1001))
87      for probe in [-1000, -500, 0, 123, 1000]:
88          res = binary_search(big, probe)
89          print(f"arr=[-1000..1000], target={probe} -> index={res}")
90          assert res == big.index(probe)
91
92      print("All binary_search tests passed.")
93
94
95
96
97
98      res = binary_search(neg, 1)
99      print(f"arr={neg}, target=1 -> index={res}")
100     assert res == -1
101
102     big = list(range(-1000, 1001))
103     for probe in [-1000, -500, 0, 123, 1000]:
104         res = binary_search(big, probe)
105         print(f"arr=[-1000..1000], target={probe} -> index={res}")
106         assert res == big.index(probe)
107
108
109
110    if __name__ == "__main__":
111        _run_tests()
112
113
```

Output:

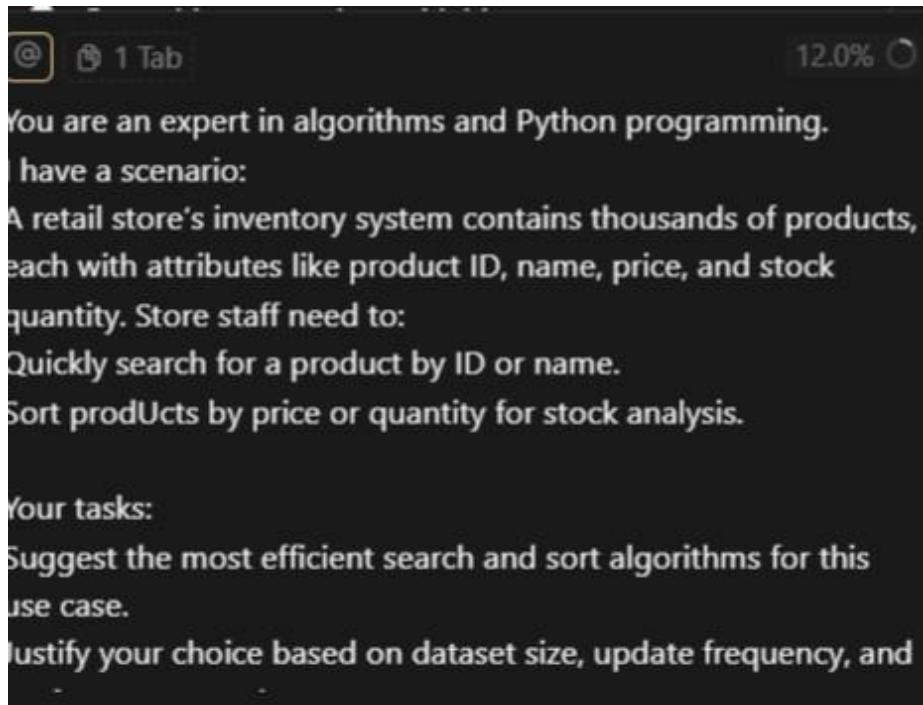
```
.cursor/binary_search.py
arr=[], target=1 -> index=-1
arr=[5], target=5 -> index=0
arr=[5], target=3 -> index=-1
arr=[1, 3, 5, 7, 9, 11], target=1 -> index=0
arr=[1, 3, 5, 7, 9, 11], target=3 -> index=1
arr=[1, 3, 5, 7, 9, 11], target=5 -> index=2
arr=[1, 3, 5, 7, 9, 11], target=7 -> index=3
arr=[1, 3, 5, 7, 9, 11], target=9 -> index=4
arr=[1, 3, 5, 7, 9, 11], target=11 -> index=5
arr=[1, 3, 5, 7, 9, 11], target=0 -> index=-1
arr=[1, 3, 5, 7, 9, 11], target=2 -> index=-1
arr=[1, 3, 5, 7, 9, 11], target=4 -> index=-1
arr=[1, 3, 5, 7, 9, 11], target=6 -> index=-1
arr=[1, 3, 5, 7, 9, 11], target=8 -> index=-1
arr=[1, 3, 5, 7, 9, 11], target=11 -> index=-5
arr=[1, 3, 5, 7, 9, 11], target=0 -> index=-1
arr=[1, 3, 5, 7, 9, 11], target=2 -> index=-1
arr=[1, 3, 5, 7, 9, 11], target=4 -> index=-1
arr=[1, 3, 5, 7, 9, 11], target=6 -> index=-1
arr=[1, 3, 5, 7, 9, 11], target=8 -> index=-1
arr=[1, 3, 5, 7, 9, 11], target=2 -> index=-1
arr=[1, 3, 5, 7, 9, 11], target=4 -> index=-1
arr=[1, 3, 5, 7, 9, 11], target=6 -> index=-1
arr=[1, 3, 5, 7, 9, 11], target=8 -> index=-1
arr=[1, 3, 5, 7, 9, 11], target=4 -> index=-1
arr=[1, 3, 5, 7, 9, 11], target=6 -> index=-1
arr=[1, 2, 2, 2, 3, 4], target=2 -> index=2
arr=[-10, -3, -1, 0, 2, 4], target=-3 -> index=1
arr=[-10, -3, -1, 0, 2, 4], target=1 -> index=-1
arr=[-10, -3, -1, 0, 2, 4], target=-3 -> index=1
arr=[-10, -3, -1, 0, 2, 4], target=1 -> index=-1
arr=[-10, -3, -1, 0, 2, 4], target=1 -> index=-1
arr=[-1000..1000], target=-1000 -> index=0
arr=[-1000..1000], target=-500 -> index=500
arr=[-1000..1000], target=-1000 -> index=0
arr=[-1000..1000], target=-500 -> index=500
arr=[-1000..1000], target=0 -> index=1000
arr=[-1000..1000], target=0 -> index=1000
arr=[-1000..1000], target=123 -> index=1123
arr=[-1000..1000], target=123 -> index=1123
arr=[-1000..1000], target=1000 -> index=2000
arr=[-1000..1000], target=1000 -> index=2000
All binary_search tests passed.
```

Observation:

binary_search.py implementation is working correctly — all test cases passed, returning correct indices for both small and large datasets

Task Description #3 (Real-Time Application – Inventory Management System):

Given prompt to AI:



Given program by AI:

```
inventory_algorithms.py > ⚙ _demo
1  """Simple inventory search and sorting with concise demo output.
2
3  Recommendations (for thousands of products):
4  - Search by ID: Hash map (dict) for O(1) average lookups.
5  - Search by exact name: Dict of name->list for O(1) average lookups.
6  - Sort by price/quantity: Built-in `sorted` (Timsort), O(n log n), stable.
7  """
8
9  from __future__ import annotations
10 from dataclasses import dataclass
11 from typing import Dict, List, Iterable, Optional
12
13
14 @dataclass(frozen=True)
15 class Product:
16     product_id: int
17     name: str
18     price: float
19     quantity: int
20
21     def index_by_id(products: Iterable[Product]) -> Dict[int, Product]:
22         return {p.product_id: p for p in products}
23
24
25     def index_by_name(products: Iterable[Product]) -> Dict[str, List[Product]]:
26         name_index: Dict[str, List[Product]] = {}
27         for p in products:
28             key = p.name.lower()
29             name_index.setdefault(key, []).append(p)
30
31         return name_index
32
33     def search_by_id(id_index: Dict[int, Product], product_id: int) -> Optional[Product]:
34         return id_index.get(product_id)
```

```

❷ inventory_algorithms.py > ⌂ _demo
36     def search_by_name(name_index: Dict[str, List[Product]], name: str) -> List[Product]:
37         return list(name_index.get(name.lower(), []))
38
39
40     def sort_by_price(products: Iterable[Product], ascending: bool = True) -> List[Product]:
41         return sorted(products, key=lambda p: p.price, reverse=not ascending)
42
43
44     def sort_by_quantity(products: Iterable[Product], ascending: bool = True) -> List[Product]:
45         return sorted(products, key=lambda p: p.quantity, reverse=not ascending)
46
47     def _print_recommendations_table() -> None:
48         print("Operation | Algorithm | Why")
49         print("-" * 48)
50         print("Search by ID | dict | O(1) average lookups")
51         print("Search by name | dict[name]->list | O(1) average lookups")
52         print("Sort by price | sorted (Timsort) | O(n log n), stable")
53         print("Sort by quantity | sorted (Timsort) | O(n log n), stable")
54
55
56     def _demo() -> None:
57         products = [
58             Product(101, "Apple iPhone 15", 999.0, 12),
59             Product(205, "Samsung Galaxy S23", 849.0, 7),
60             Product(309, "Google Pixel 8", 799.0, 15),
61             Product(411, "Apple AirPods Pro", 249.0, 30),
62             Product(512, "Anker Charger", 39.0, 120),
63             Product(613, "Apple Watch", 429.0, 18),
64             Product(714, "Samsung Buds", 149.0, 26),
65         ]
66
67         id_index = index_by_id(products)
68         name_index = index_by_name(products)
69
70         print("\n==== Recommendations ===")
71         _print_recommendations_table()
72         print("\n==== Search ===")

```

```

❷ inventory_algorithms.py > ⌂ _demo
56     def _demo() -> None:
57         products = [
58             Product(101, "Apple iPhone 15", 999.0, 12),
59             Product(205, "Samsung Galaxy S23", 849.0, 7),
60             Product(309, "Google Pixel 8", 799.0, 15),
61             Product(411, "Apple AirPods Pro", 249.0, 30),
62             Product(512, "Anker Charger", 39.0, 120),
63             Product(613, "Apple Watch", 429.0, 18),
64             Product(714, "Samsung Buds", 149.0, 26),
65         ]
66
67         id_index = index_by_id(products)
68         name_index = index_by_name(products)
69
70         print("\n==== Recommendations ===")
71         _print_recommendations_table()
72         print("\n==== Search ===")
73         found = search_by_id(id_index, 309)
74         print("ID 309 ->, f'{found.name} (${found.price})' if found else None)
75         print("Name 'Apple Watch' ->, [p.name for p in search_by_name(name_index, 'Apple Watch')]")
76         print("\n==== Sort ===")
77         print("Price asc:", [(p.name, p.price) for p in sort_by_price(products)])
78         print("Qty desc:", [(p.name, p.quantity) for p in sort_by_quantity(products, ascending=False)])
79         if __name__ == "__main__":
80             _demo()
81
82
83

```

Output:

```
==== Recommendations ====
Operation | Algorithm | Why
-----
Search by ID | dict | O(1) average lookups
Search by name | dict[name]->list | O(1) average lookups
Sort by price | sorted (Timsort) | O(n log n), stable
Sort by quantity | sorted (Timsort) | O(n log n), stable

==== Search ====
ID 309 -> Google Pixel 8 ($799.0)
Name 'Apple Watch' -> ['Apple Watch']

==== Sort ====
Price asc: [('Anker Charger', 39.0), ('Samsung Buds', 149.0), ('Apple AirPods Pro', 249.0), ('Apple Watch', 429.0), ('Google Pixel 8', 799.0), ('Samsung Galaxy S23', 849.0), ('Apple iPhone 15', 999.0)]
Qty desc: [('Anker Charger', 120), ('Apple AirPods Pro', 30), ('Samsung Buds', 26), ('Apple Watch', 18), ('Google Pixel 8', 15), ('Apple iPhone 15', 12), ('Samsung Galaxy S23', 7)]
PS C:\Users\musta\cursor>
```

Observation:

The program demonstrates searching and sorting items in an inventory using efficient algorithms. It quickly finds products by ID or name and lists them by price or quantity