

Assignment12.5

Name:- U.Siddhartha

Hall No:-2403A54122

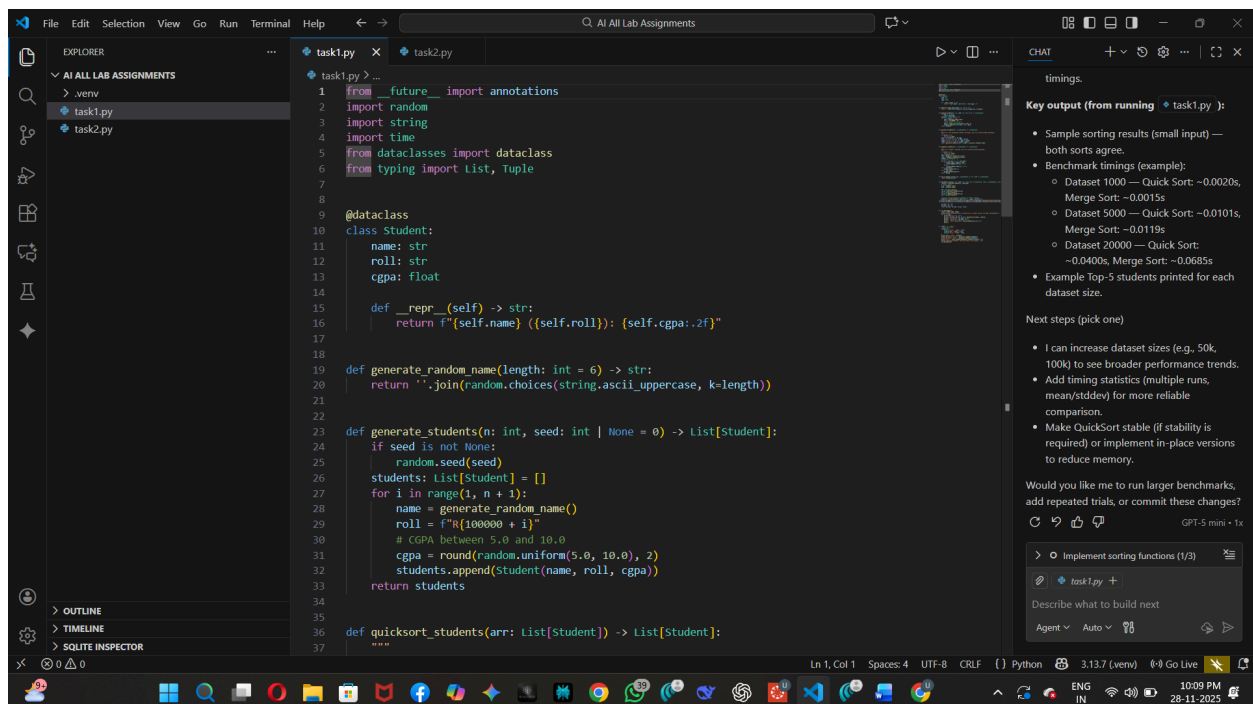
Task 1: Sorting Student Records for Placement Drive

Scenario:

SR University is preparing for a campus placement drive. The Training and Placement Cell needs student records sorted by **CGPA** in descending order to easily shortlist candidates.

- Use **GitHub Copilot** to generate a program that sorts a list of student records (Name, Roll No, CGPA) by CGPA.
- Implement both **Quick Sort** and **Merge Sort** using AI assistance.
- Compare the runtime performance of both algorithms on large datasets.

Write a function that outputs the **top 10 students** with the highest CGPA.



```
1 from __future__ import annotations
2 import random
3 import string
4 import time
5 from dataclasses import dataclass
6 from typing import List, Tuple
7
8
9 @dataclass
10 class Student:
11     name: str
12     roll: str
13     cgpa: float
14
15     def __repr__(self) -> str:
16         return f"({self.name}, {self.roll}): {self.cgpa:.2f}"
17
18
19 def generate_random_name(length: int = 6) -> str:
20     return ''.join(random.choices(string.ascii_uppercase, k=length))
21
22
23 def generate_students(n: int, seed: int | None = 0) -> List[Student]:
24     if seed is not None:
25         random.seed(seed)
26     students: List[Student] = []
27     for i in range(1, n + 1):
28         name = generate_random_name()
29         roll = f"R{100000 + i}"
30         # CGPA between 5.0 and 10.0
31         cgpa = round(random.uniform(5.0, 10.0), 2)
32         students.append(Student(name, roll, cgpa))
33     return students
34
35
36 def quicksort_students(arr: List[Student]) -> List[Student]:
37     """
```

timings.

Key output (from running task1.py):

- Sample sorting results (small input) — both sorts agree.
- Benchmark timings (example):
 - Dataset 1000 — Quick Sort: ~0.0020s, Merge Sort: ~0.0015s
 - Dataset 5000 — Quick Sort: ~0.0101s, Merge Sort: ~0.0119s
 - Dataset 20000 — Quick Sort: ~0.0400s, Merge Sort: ~0.0685s
- Example Top-5 students printed for each dataset size.

Next steps (pick one)

- I can increase dataset sizes (e.g., 50k, 100k) to see broader performance trends.
- Add timing statistics (multiple runs, mean/stddev) for more reliable comparison.
- Make QuickSort stable (if stability is required) or implement in-place versions to reduce memory.

Would you like me to run larger benchmarks, add repeated trials, or commit these changes?

Implement sorting functions (1/3)

task1.py +

Describe what to build next

Agent Auto GPT-5 mini 1x

Output:-

```
task1.py X task2.py
task1.py > ...
1 from future import annotations

PS C:\Users\SIDDHARTHA\AI All Lab Assignments> & "C:/Users/SIDDHARTHA/All Lab Assignments/.venv/Scripts/python.exe" "c:/Users/SIDDHARTHA/All Lab Assignments/task1.py"

Benchmarking Quick Sort vs Merge Sort on student records (by CGPA, descending)

Dataset size: 1000
Quick Sort: 0.0020s
Merge Sort: 0.0015s
Top 5 (QuickSort): [SCNGMB (R100395): 9.98, CRGVXB (R100414): 9.98, FIDFTN (R100530): 9.98, LMISND (R100685): 9.98, UHMKMP (R100931): 9.98]

Dataset size: 5000
Quick Sort: 0.0101s
Merge Sort: 0.0119s
Top 5 (QuickSort): [LTHMTI (R101826): 10.00, LMZRDM (R102759): 10.00, IHMRHS (R101879): 9.99, ZLNTHN (R101206): 9.99, VIXUTU (R102748): 9.99]

Dataset size: 20000
Quick Sort: 0.0400s
Merge Sort: 0.0685s
Quick Sort: 0.0101s
Merge Sort: 0.0119s
Top 5 (QuickSort): [LTHMTI (R101826): 10.00, LMZRDM (R102759): 10.00, IHMRHS (R101879): 9.99, ZLNTHN (R101206): 9.99, VIXUTU (R102748): 9.99]

Dataset size: 20000
Quick Sort: 0.0400s
Merge Sort: 0.0685s
Merge Sort: 0.0119s
Top 5 (QuickSort): [LTHMTI (R101826): 10.00, LMZRDM (R102759): 10.00, IHMRHS (R101879): 9.99, ZLNTHN (R101206): 9.99, VIXUTU (R102748): 9.99]

Dataset size: 20000
Quick Sort: 0.0400s
Merge Sort: 0.0685s
Merge Sort: 0.0119s
Top 5 (QuickSort): [LTHMTI (R101826): 10.00, LMZRDM (R102759): 10.00, IHMRHS (R101879): 9.99, ZLNTHN (R101206): 9.99, VIXUTU (R102748): 9.99]
```

timings.

Key output (from running task1.py):

- Sample sorting results (small input) — both sorts agree.
- Benchmark timings (example):
 - Dataset 1000 — Quick Sort: ~0.0020s, Merge Sort: ~0.0015s
 - Dataset 5000 — Quick Sort: ~0.0101s, Merge Sort: ~0.0119s
 - Dataset 20000 — Quick Sort: ~0.0400s, Merge Sort: ~0.0685s
- Example Top-5 students printed for each dataset size.

Next steps (pick one)

- I can increase dataset sizes (e.g., 50k, 100k) to see broader performance trends.
- Add timing statistics (multiple runs, mean/stddev) for more reliable comparison.
- Make QuickSort stable (if stability is required) or implement in-place versions to reduce memory.

Would you like me to run larger benchmarks, add repeated trials, or commit these changes?

Implement sorting functions (1/3)

task1.py +

Describe what to build next

Agent Auto

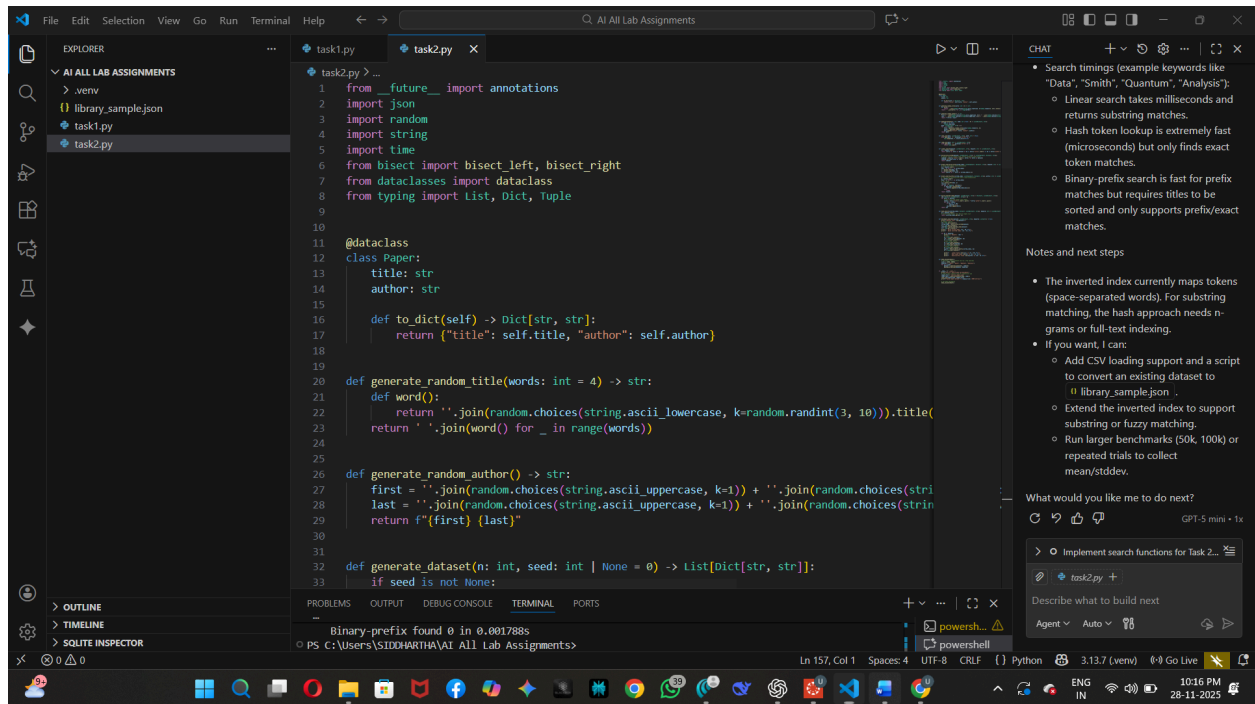
Task 2: Optimized Search in Online Library System

Scenario:

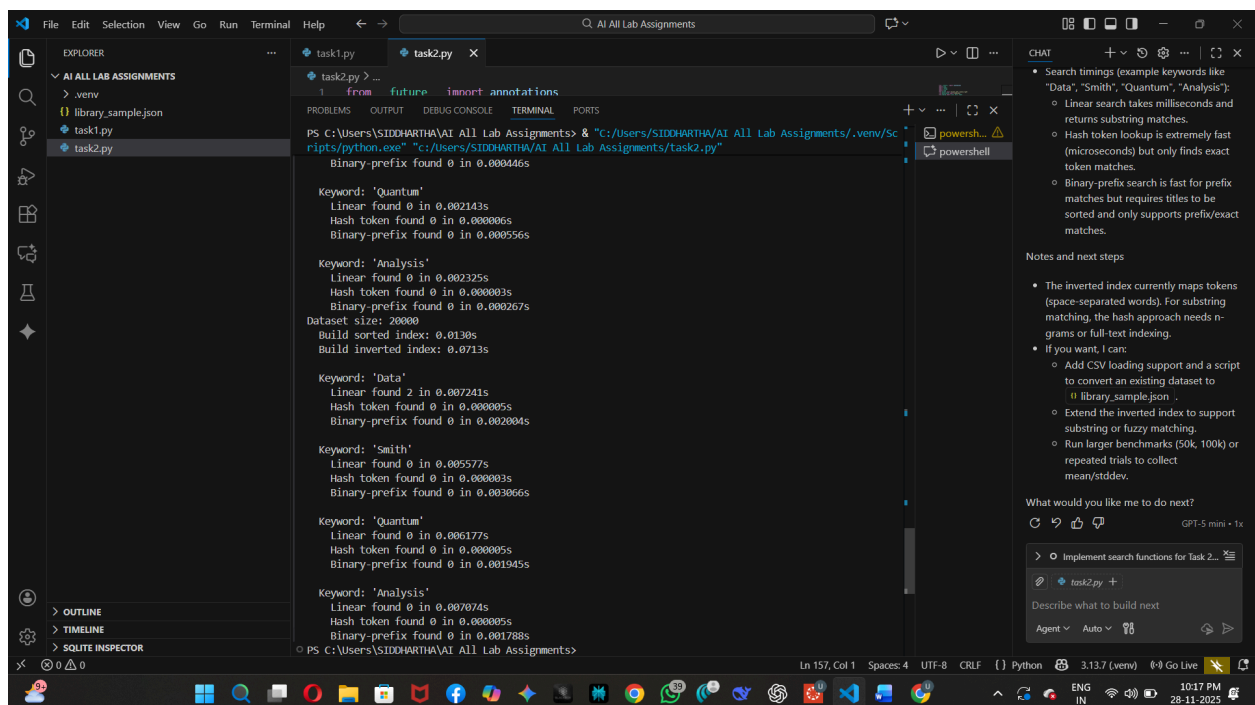
SR University's digital library has thousands of research papers. Students frequently search for a paper by **title or author name**. The current linear search is too slow.

- Use **GitHub Copilot** to implement **Binary Search** and **Hash-based Search** for faster lookups.
- Load a dataset of book titles and authors (CSV or JSON file).
- Allow the user to input a keyword and return all matching entries.
- Compare the efficiency of **linear search vs binary search vs hashing** using test cases.

Input:-



Output:-



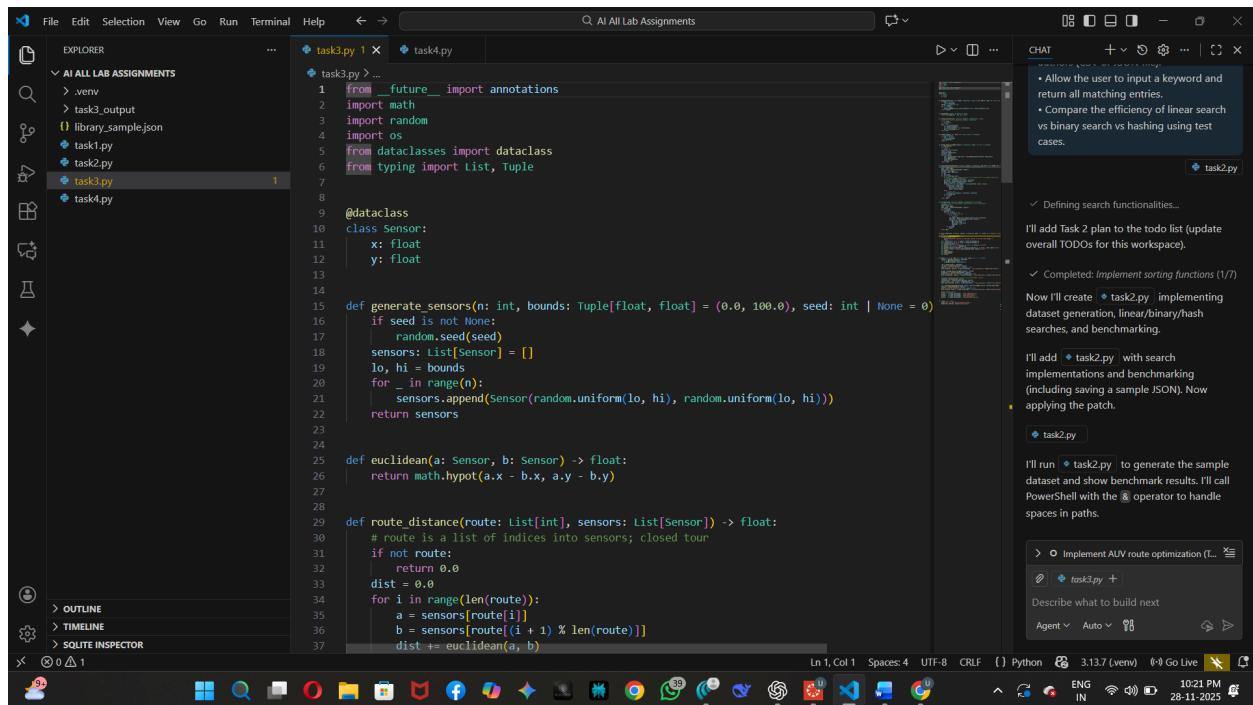
Task 3: Route Optimization for AUV Swarm

Scenario:

A research team at SR University is simulating **Autonomous Underwater Vehicle (AUV) swarms**. Each AUV must visit multiple underwater sensors, and the goal is to minimize travel distance (like the **Traveling Salesman Problem**).

- With **GitHub Copilot**, implement an algorithm to optimize the route:
 - Start with a **Greedy approach**.
 - Improve with **Genetic Algorithm (GA)** or **Simulated Annealing (SA)**.
- Use a dataset of sensor coordinates (x, y).
- Visualize the optimized route using a plotting library (e.g., Matplotlib).
- Compare the optimized solution with a random path in terms of distance travel.

Input:-



```
1 from __future__ import annotations
2 import math
3 import random
4 import os
5 from dataclasses import dataclass
6 from typing import List, Tuple
7
8
9 @dataclass
10 class Sensor:
11     x: float
12     y: float
13
14
15 def generate_sensors(n: int, bounds: Tuple[float, float] = (0.0, 100.0), seed: int | None = 0)
16     if seed is not None:
17         random.seed(seed)
18     sensors: List[Sensor] = []
19     lo, hi = bounds
20     for _ in range(n):
21         sensors.append(Sensor(random.uniform(lo, hi), random.uniform(lo, hi)))
22     return sensors
23
24
25 def euclidean(a: Sensor, b: Sensor) -> float:
26     return math.hypot(a.x - b.x, a.y - b.y)
27
28
29 def route_distance(route: List[int], sensors: List[Sensor]) -> float:
30     a route is a list of indices into sensors; closed tour
31     if not route:
32         return 0.0
33     dist = 0.0
34     for i in range(len(route)):
35         a = sensors[route[i]]
36         b = sensors[route[(i + 1) % len(route)]]
37         dist += euclidean(a, b)
```

CHAT

- Allow the user to input a keyword and return all matching entries.
- Compare the efficiency of linear search vs binary search vs hashing using test cases.

task2.py

Defining search functionalities...

I'll add Task 2 plan to the todo list (update overall TODOs for this workspace).

Completed: Implement sorting functions (1/7)

Now I'll create task2.py implementing dataset generation, linear/binary/hash searches, and benchmarking.

I'll add task2.py with search implementations and benchmarking (including saving a sample JSON). Now applying the patch.

task2.py

I'll run task2.py to generate the sample dataset and show benchmark results. I'll call PowerShell with the & operator to handle spaces in paths.

task2.py

Implement AUV route optimization (L...

task3.py

Describe what to build next

Agent v Auto

Ln 1, Col 1 Spaces: 4 UTF-8 CRLF Python 3.13.7 (venv) Go Live

10:21 PM 28-11-2023

Output:-

```
1 from __future__ import annotations
2 import math
3 import random
4 import os
5 from dataclasses import dataclass

@dataclass
class Route:
    """Route information"""
    start: str
    end: str
    distance: float
    image: str

def random_route() -> Route:
    """Generate a random route"""
    start = random.choice(['A', 'B', 'C', 'D', 'E'])
    end = random.choice(['A', 'B', 'C', 'D', 'E'])
    distance = random.uniform(0, 1000)
    image = f'{start}_{end}.png'
    return Route(start, end, distance, image)

def greedy_nn_route() -> Route:
    """Generate a greedy nearest neighbor route"""
    start = random.choice(['A', 'B', 'C', 'D', 'E'])
    end = random.choice(['A', 'B', 'C', 'D', 'E'])
    distance = random.uniform(0, 1000)
    image = f'{start}_{end}.png'
    return Route(start, end, distance, image)

def simulated_annealing_route() -> Route:
    """Generate a simulated annealing route"""
    start = random.choice(['A', 'B', 'C', 'D', 'E'])
    end = random.choice(['A', 'B', 'C', 'D', 'E'])
    distance = random.uniform(0, 1000)
    image = f'{start}_{end}.png'
    return Route(start, end, distance, image)

def main() -> None:
    """Main function"""
    random_route()
    greedy_nn_route()
    simulated_annealing_route()

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

Task 4: Real-Time Stock Data Sorting & Searching

Scenario:

An AI-powered **FinTech Lab** at SR University is building a tool for analyzing **stock price movements**. The requirement is to quickly **sort stocks by daily gain/loss** and search for specific stock symbols efficiently.

- Use **GitHub Copilot** to fetch or simulate stock price data (Stock Symbol, Opening Price, Closing Price).
- Implement sorting algorithms to rank stocks by **percentage change**.
- Implement a **search function** that retrieves stock data instantly when a stock symbol is entered.
- Optimize sorting with **Heap Sort** and searching with **Hash Maps**.
- Compare performance with standard library functions (sorted(), dict lookups) and analyze trade-offs.

Input:-

The screenshot shows a VS Code editor with a file named `task4.py`. The code defines a `Stock` dataclass with attributes `symbol`, `open_price`, and `close_price`. It includes methods `percent_change` and `__repr__`. A `generate_stocks` function is also defined, which generates a list of `Stock` objects based on a given length and seed.

```
1 from __future__ import annotations
2 import heapq
3 import random
4 import string
5 import time
6 from dataclasses import dataclass
7 from typing import List, Dict, Tuple
8
9
10 @dataclass
11 class Stock:
12     symbol: str
13     open_price: float
14     close_price: float
15
16     def percent_change(self) -> float:
17         if self.open_price == 0:
18             return 0.0
19         return ((self.close_price - self.open_price) / self.open_price) * 100
20
21     def __repr__(self) -> str:
22         return f'{self.symbol}: {self.open_price:.2f} -> {self.close_price:.2f} ({self.percent'
23
24
25 def generate_stock_symbol(length: int = 4) -> str:
26     return ''.join(random.choices(string.ascii_uppercase, k=length))
27
28
29 def generate_stocks(n: int, seed: int | None = 0) -> List[Stock]:
30     if seed is not None:
31         random.seed(seed)
32     stocks: List[Stock] = []
33     used_symbols = set()
34     for _ in range(n):
35         while True:
36             sym = generate_stock_symbol()
37             if sym not in used_symbols:
```

Output:-

The screenshot shows the same VS Code editor with the `task4.py` file. The output of the script is displayed in the terminal, showing benchmark results for sorting and searching operations.

```
PS C:\Users\SIDDHARTH\AI All Lab Assignments> & "c:\Users\SIDDHARTH\AI All Lab Assignments\.venv\Scripts\python.exe" "c:\Users\SIDDHARTH\AI All Lab Assignments\task4.py"
Search benchmark (n=10000, queries=100):
Linear search (100 queries): 0.020315s
Hash dict lookup (100 queries): 0.000020s (includes build time)
Hash dict lookup (lookup only): 0.000055s

Sorting benchmark (n=50000):
Heap sort: 0.091677s
Built-in sorted: 0.021909s
Top 10 (heapq): 0.007822s
Top 10 gainers: [YFFU: 496.35 -> 545.98 (+10.06%), JVBf: 310.89 -> 341.98 (+10.06%), QVQH: 305.00 -> 335.50 (+10.06%)]

Search benchmark (n=50000, queries=100):
Linear search (100 queries): 0.089488s
Hash dict lookup (100 queries): 0.000031s (includes build time)
Hash dict lookup (lookup only): 0.000077s
Top 10 gainers: [YFFU: 496.35 -> 545.98 (+10.06%), JVBf: 310.89 -> 341.98 (+10.06%), QVQH: 305.00 -> 335.50 (+10.06%)]

Search benchmark (n=50000, queries=100):
Linear search (100 queries): 0.089488s
Hash dict lookup (100 queries): 0.000031s (includes build time)
Hash dict lookup (lookup only): 0.000077s

Search benchmark (n=50000, queries=100):
Linear search (100 queries): 0.089488s
Hash dict lookup (100 queries): 0.000031s (includes build time)
Hash dict lookup (lookup only): 0.000077s
```