

main.py

```
1 # 8-puzzle A* solver (Manhattan heuristic)
2 # State: tuple of 9 ints, 0 = blank, row-major order
3 # Goal: (1,2,3,4,5,6,7,8,0)
4
5 import heapq
6
7 GOAL = (1, 2, 3, 4, 5, 6, 7, 8, 0)
8 GOAL_POS = {val: (i // 3, i % 3) for i, val in enumerate(GOAL)}
9
10 def manhattan(state):
11     total = 0
12     for idx, val in enumerate(state):
13         if val == 0:
14             continue
15         r, c = divmod(idx, 3)
16         gr, gc = GOAL_POS[val]
17         total += abs(r - gr) + abs(c - gc)
18     return total
19
20 def neighbors(state):
21     z = state.index(0)
22     nz, nc = divmod(z, 3)
```

Run

Output

Start state:
1 6 5
7 3 8
_ 4 2

Solved in 24 moves; expanded 3248 nodes; solution cost 24

Step 0:
1 6 5
7 3 8
_ 4 2

Move: Up

Step 1:
1 6 5
_ 3 8
7 4 2

Move: Up

Step 2:
_ 6 5
1 3 8

Clear

The screenshot shows a Python code editor and a terminal window. The code editor contains a file named 'main.py' with the following content:

```
main.py
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
```

```
def move_disk(source, destination):
    return

# Move n-1 disks from source → auxiliary
hanoi(n-1, source, destination, auxiliary)

# Move last disk from source → destination
print(f"Move disk {n} from {source} → {destination}")

# Move n-1 disks from auxiliary → destination
hanoi(n-1, auxiliary, source, destination)

# -----
# MAIN PROGRAM
#
if __name__ == "__main__":
    n = int(input("Enter number of disks: "))

    print("\nSequence of moves:")
    hanoi(n, 'A', 'B', 'C')
```

The terminal window shows the output of running the code with 3 disks:

```
Enter number of disks: 3
Sequence of moves:
Move disk 1 from A → C
Move disk 2 from A → B
Move disk 1 from C → B
Move disk 3 from A → C
Move disk 1 from B → A
Move disk 2 from B → C
Move disk 1 from A → C
== Code Execution Successful ==
```

Programiz
Python Online Compiler

Programiz PRO
Python course where you not just learn concept, but see it run line-by-line visually.

Try Now

main.py

```
1 from collections import deque
2
3 def bfs(graph, start):
4     visited = set()
5     queue = deque([start])
6
7     print("BFS Traversal:", end=" ")
8
9     while queue:
10         node = queue.popleft()
11
12         if node not in visited:
13             print(node, end=" ")
14             visited.add(node)
15
16             # Add all neighbors
17             for neigh in graph[node]:
18                 if neigh not in visited:
19                     queue.append(neigh)
20
21 # -----
22 # -----
```

Run

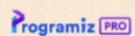
Output

Clear

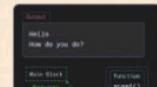
Graph: {0: [2, 1, 3], 1: [], 2: [4], 3: [], 4: []}
BFS Traversal: 0 2 1 3 4
== Code Execution Successful ==



Python Online Compiler



Understand Python
better by watching your
code unfold visually.



Try Now



main.py



Run

Clear

```
1 # Monkey and Banana Problem
2
3+ class Monkey:
4+     def __init__(self):
5         self.monkey_pos = "door"
6         self.box_pos = "window"
7         self.banana_pos = "middle"
8         self.on_box = False
9
10+    def move(self, place):
11        print(f"Monkey moves from {self.monkey_pos} to {place}.")
12        self.monkey_pos = place
13
14+    def push_box(self, place):
15        print(f"Monkey pushes box from {self.box_pos} to {place}.")
16        self.box_pos = place
17        self.monkey_pos = place
18
19+    def climb_box(self):
20        if self.monkey_pos == self.box_pos:
21            self.on_box = True
22            print("Monkey climbs onto the box. 🌟")
```

Output

Monkey moves from door to window.
Monkey pushes box from window to middle.
Monkey climbs onto the box.
Monkey takes the banana! 🌟

==== Code Execution Successful ===



① ⓘ
TAC Apply

main.py

1 # Eight Queens Problem using Backtracking
2
3 board_size = 8
4 solution = []
5
6 def is_safe(row, col, solution):
7 for r, c in solution:
8 if c == col or r == row or abs(r - row) == abs(c - col):
9 return False
10 return True
11
12 def solve(row=0):
13 if row == board_size:
14 return True
15
16 for col in range(board_size):
17 if is_safe(row, col, solution):
18 solution.append((row, col))
19 if solve(row + 1):
20 return True
21 solution.pop()
22

Run

Output

Solution for 8 Queens:

Q
.	.	.	Q
.	Q	.
.	.	.	.	Q	.	.	.
.	.	Q
.	Q	.	.
.	Q
.	.	.	Q

==== Code Execution Successful ===

Multiply your assets in minutes

Google Ads

main.py

```
1 # Depth First Search (DFS) on a Graph
2
3 graph = {
4     1: [2, 3],
5     2: [4, 8],
6     3: [5, 7],
7     4: [],
8     5: [],
9     7: [],
10    8: []
11 }
12
13 visited = set()
14
15 def dfs(node):
16     if node not in visited:
17         print(node, end=" ")
18         visited.add(node)
19         for neighbor in graph[node]:
20             dfs(neighbor)
21
22 DFS Traversal starting from node 1:
```

Run

Output

Clear

File

Copy

Share

Run

Output

Clear

Multiply your assets in minutes



Google Ads



main.py



Run

Output

Clear

```
1 # Water Jug Problem (4-gallon and 3-gallon)
2
3 from collections import deque
4
5 def water_jug():
6     # state = (jug4, jug3)
7     start = (0, 0)
8     goal = (2, 0)
9     visited = set([start])
10    queue = deque([(start, [])])
11
12    while queue:
13        (a, b), path = queue.popleft()
14
15        if (a, b) == goal:
16            print("Steps to reach 2 gallons in 4-gallon jug:\n")
17            for step in path:
18                print(step)
19            print(f"\nFinal state: {a} gallons in 4-gallon jug, {b}
in 3-gallon jug.")
20
21    return
```

Steps to reach 2 gallons in 4-gallon jug:

Fill 3-gallon jug => (0, 3)
Pour from 3-gallon to 4-gallon => (3, 0)
Fill 3-gallon jug => (3, 3)
Pour from 3-gallon to 4-gallon => (4, 2)
Empty 4-gallon jug => (0, 2)
Pour from 3-gallon to 4-gallon => (2, 0)

Final state: 2 gallons in 4-gallon jug, 0 in 3-gallon jug.

==== Code Execution Successful ===

https://programiz.pro/offer/black-november-2025?utm_source=programiz_compiler&utm_medium=referral&utm_campaign=offer-black-november-sale&utm_content=interests_free_gifts&utm_term=sticky_banner_lau