BMI3 Integrated Reflection

Yuchen SHEN

2026-01-03

▶ Things to Check when Encountering Errors

- Misusing == and =?
- Misspelling variable names?
- Not sorting when needed?
- Variable name conflicts?
- Misreading the problem statement?
- Using a variable name as an index by mistake?
- Forgetting to import something (e.g., json, sys)?
- Writing data = input(json.loads()) instead of data = json.loads(input())?
- If you get a "memory limit exceeded" error, try submitting a few more times and see if it passes

▶ Examined Points Summary

Examined point	Where
Time complexity assessment	1. **F-1124-Q2** (While loop dependence) 2. **F-1-Q2** (Recursive function) 3. **F-2-Q3** (While loop entry & dependence) 4. **Bilibili** ($O(n)$ notes)
Combinatorial pattern matching	1. **PTA** (Rabin Karp) 2. **Leaked** (Rabin Karp)
Divide and conquer	1. **F-1124-Q1** (BLAST table sorting) 2. **F-2-Q1** (Inversion counting) 3. **F-3-Q4** (Peak element) 4. **PTA** (Merge sort) 5. **PTA** (Closest pair of points)
Graph algorithms	1. **F-1124-Q4** (Cycle detection in DAG) 2. **F-1-Q1** (Longest path in DAG) 3. **F-2-Q2** (Shortest path in unweighted UAG) 4. **PTA** (Shortest path in weighted UAG) 5. **PTA** (Connected components)
Dynamic programming	1. **F-1124-Q3** (LCS in two sequences) 2. **F-1-Q3** (Edit distance) 3. **F-2-Q4** (Cashier coin changing) 4. **Lecture** (Longest path in DAG) 5. **Lecture** (Sequence alignment) 6. **PTA** (Longest path in an alignment graph) 7. **PTA** (Rock game) 8. **PTA** (Weighted interval scheduling)

	9. **PTA** (Exon chaining) 10. **YouTube** (LIS in a sequence)

▶ Time Complexity Assessment

- Constant change is irrelevant:
  $x < n / 2 = x < n + 100 = x < n$
- Low-level is irrelevant:
  $n > x^2 + 2x + 1 = n > x^2$
- Use the worst case:

Outer	Inner
Linear (n)	Linear (n), log, sqrt
log	log

- Altering: j = j - i
- Function: j = f(i)
- Masking: j = i, i = n

Dependent

- Multiply directly

Independent

CORE: TOTAL RUN COUNT

*Outer loop value*determines inner loop run count*

Self-call

- Master's Theorem
- While-loop rewrite

**Some times not, but mostly yes*

▶ Divide and Conquer

Insight:

- *Divide-and-Conquer* is not *Divide-and-Merge*. *Merging* is a required step for sorting-based problems, but not an essential element for *Divide-and-Conquer*.
- What makes *Divide-and-Conquer Divide-and-Conquer* is that it divide a large problem into a base case.
- The workflow for sequence-based problem: $Divide \rightarrow Conquer \rightarrow Combine$.

DIVIDE AND MERGE

Base case	if len(input) <= 1: return
Divide	Left string; Right string
Conquer	Sort left; Sort right
Combine	while if-else; while; while | merged.append() + pointer += 1

DIVIDE AND CONQUER WITHOUT MERGING

1. Set base-case solution
2. Get insights
   - Find a middle line
   - Special to each problem

▶ GRAPH ALGORITHM

Graph algorithm problems can be solved in multiple ways. **(1) Greedy way** like Dijkstra based on the principle that if a node is visited then the shortest path is set; **(2) Dynamic programming**; **(3) DFS**; **(4) BFS**.

SCENARIO

- For finding **connected components** and **cycle detection**: DFS - Stack *using set()*
- **Longest path in directed acyclic graph (DAG)**
  - Weighted: Dynamic programming – dynamic programming
- **Shortest path in undirected acyclic graph (UAG)**
  - Weighted: DFS - Priority queue *using heapq*
  - Unweighted: BFS - Queue *using queue*

GENERAL STEPS

1. Construct graph as adjacency list (if the graph is directed, calculate topological order)
2. Initialize key players: distance - previous, visited (set)
3. Choose an appropriate traversal method

stack (set)	object.add()	object.remove()
queue (list)	queue.pop()	queue.append()
heapq	heapq.heappop()	heapq.heappush()

```
def BFS(start_node):
    mark start_node as visited
    create a queue and enqueue start_node

    while queue is not empty:
        current_node = dequeue from queue
        for each neighbor of current_node:
            if neighbor is not visited:
                mark neighbor as visited
                enqueue neighbor
```

```
def DFS(node):
    mark node as visited
    for each neighbor of node:
        if neighbor is not visited:
            DFS(neighbor)
    return
```

4. Update key players during traversal until the termination condition is hit

ADJACENCY LIST OPERATIONS

for **neighbor** in **graph.get(current, []):**
for **neighbor, weight** in **graph[current]:**
for **node** in **graph:**

TOPOLOGICAL SORTING

```
in_degree = {}

for v in vertices:
    in_degree[v] = 0

for _, row in edges.iterrows():
    v = row['to']
    in_degree[v] += 1

queue = []
for v in vertices:
    if in_degree[v] == 0:
        queue.append(v)
```

```
topo_order = []
while len(queue) > 0:
    u = queue.pop(0)
    topo_order.append(u)
    for v, w in adj[u]:
        in_degree[v] -= 1
        if in_degree[v] == 0:
            queue.append(v)
```

NODE COLLECTION

```
vertices = set()
for _, row in edges.iterrows():
    vertices.add(row['from'])
    vertices.add(row['to'])
```

KEY PLAYER INITIALIZATION

```
dist = {}
for v in vertices:
    dist[v] = float('-inf')
dist[source] = 0
# or
dist = []
for i in range(n):
    dist.append([float('inf'),0])
dist[source] = [0.0,-1]

queue = [source]
visited = {source}
# or
visited = [False] * n # use it like if
visited[pos] = True
```

```
parent = {source: -1}
```

BACKTRACING

```
path = []
node = target
while node is not None:
    path.append(node)
    node = parent[node]
path.reverse()
print(len(path) - 1, path)
```

▶ DYNAMIC PROGRAMMING

MATRIX CONSTRUCTION

- Edit distance
- LCS in two sequences
- Sequence alignment
- Rock game

GENERAL STEPS

Note: DP is NOT a method that “updates something only when it is better than the current solution”, but the type of “updates based on the former state”.

1. Construct a empty 2D array / 2D list
   - For sequence-sequence comparison `dp = [[0]*(n+1) for _ in range(m+1)]`
   - For a grid `dp = [[0] * cols for _ in range(rows)]`
2. Base case construction ((0,0), x-axis, y-axis; not necessarily), see (i in *the appendix*)
3. Initialize key players for DP (sometimes you think you should have one, but in fact you don’t)
4. Fill the DP table (if-else or dynamically, see ii in *the appendix*)

```
for i in range(1, m + 1):
 for j in range(1, n + 1):
  if
  else
```

$$dp[i][j] = \max(\text{diagonal},\ \text{up},\ \text{left})$$

4. Check if the case exists (see iii in *the appendix*)
5. Backtracking: while if-else (see iii in *the appendix*)

NON-MATRIX CONSTRUCTION

- Cashier coin changing
- Weighted interval scheduling
- Exon chaining
- Longest path in DAG
- Longest path in an alignment graph

- LIS in two sequences

The key idea is to break the problem down into the present and the past. It is either adding all (Cashier coin changing; LIS in two sequences) or choosing to skip the present (Weighted interval scheduling; Longest path in DAG; Longest path in an alignment graph).

Typically, backtracking is needed for these type of DP problems. Before backtracking, it is necessary to check if there's a special case.

APPENDIX

(i) Base case construction

a

```
for i in range(1, n + 1):
  rocks[i, 0] = not rocks[i - 1, 0]
for j in range(1, m + 1):
  rocks[0, j] = not rocks[0, j - 1]
```

b

```
for i in range(n + 1):
  dp[i][0] = -gap * i
for j in range(m + 1):
  dp[0][j] = -gap * j
```

c

```
dp[0][0] = grid[0][0]
for j in range(1, cols):
  dp[0][j] = dp[0][j-1] + grid[0][j]
for i in range(1, rows):
  dp[i][0] = dp[i-1][0] + grid[i][0]
```

(ii) Fill the DP

A

```
for i in range(1, m + 1):
  for j in range(1, n + 1):
    if s1[i - 1] == s2[j - 1]:
      dp[i][j] = dp[i - 1][j - 1]
    else:
      dp[i][j] = 1 + min(
        dp[i - 1][j - 1],
        dp[i - 1][j],
        dp[i][j - 1]
      )
```

B

```
for i in range(1, m + 1):
  for j in range(1, n + 1):
    if seq1[i-1] == seq2[j-1]:
      dp[i][j] = dp[i-1][j-1] + 1
      if dp[i][j] > max_len:
        max_len = dp[i][j]
        end_positions = [i]
      elif dp[i][j] == max_len:
        end_positions.append(i)
```

C

```
for i in range(1, n + 1):
  for j in range(1, m + 1):
    if all((rocks[i - 1, j - 1], rocks[i, j - 1], rocks[i - 1, j])):
      rocks[i, j] = False
    else:
      rocks[i, j] = True
```

D

```
for i in range(1, n + 1):
  for j in range(1, m + 1):
    diagonal = dp[i-1][j-1] + score(seq1[i-1], seq2[j-1])
    up = dp[i-1][j] - gap
    left = dp[i][j-1] - gap
    dp[i][j] = max(diagonal, up, left)
```

E

```
for i in range(1, rows):
  for j in range(1, cols):
    dp[i][j] = grid[i][j] + min(dp[i-1][j], dp[i][j-1])
```

F

```
for node in sorted(nodes):
  if dist[node] > -np.inf: # can reach
    for neighbor, weight in adj[node]:
      dist[neighbor] = max(dist[neighbor], dist[node] + weight)
```

(iii) Check edge case and backtracking

A

```
if max_len == 0:
  return []
```

```
substrings = set()
for end in end_positions:
    substrings.add(seq1[end - max_len : end])
```

B

```
path = []
i, j = rows - 1, cols - 1
path.append((i, j))

while i > 0 or j > 0:
    if i == 0:
        j -= 1
    elif j == 0:
        i -= 1
    else:
        if dp[i-1][j] < dp[i][j-1]:
            i -= 1
        else:
            j -= 1
    path.append((i, j))

path.reverse()

minimum_cost = dp[rows-1][cols-1]
```