Optimal Merge Pattern: Greedy Algorithm

Varun Kumar

July 8, 2025

Contents

1 Introduction

The **Optimal Merge Pattern** is a problem where we are given multiple sorted files and our task is to merge them all into a single file in such a way that the total cost of merging is minimized.

Key Concept: Each merge of two files costs the sum of their sizes. Merge the two smallest files first — a greedy strategy.

2 Problem Statement

Given n files with sizes $f_1, f_2, \ldots, f_n$, merge them into one single file with the minimum total cost. Merging two files of sizes a and b costs $a + b$.

3 Approach: Greedy Algorithm using Min-Heap

- Insert all file sizes into a min-heap.
- While more than one file remains:
  - Extract two smallest files.
  - Merge them: cost = sum of sizes.
  - Add merged file back to heap.
  - Accumulate the cost.
- When one file remains, return total cost.

3.1 Example

Given file sizes: [20, 30, 10, 5]

Step-by-Step Merging

1. Merge 5 + 10 = 15 $\rightarrow$ Cost = 15
2. Merge 15 + 20 = 35 $\rightarrow$ Cost = 35

3. Merge $30 + 35 = 65 \rightarrow$ Cost $= 65$

Total Cost $= 15 + 35 + 65 =$ **115**

3.2 2-Way Merging in Optimal Merge Pattern

Concept

In the Optimal Merge Pattern, each merge operation is a standard **2-way merge** between two sorted files. The cost of merging is the sum of their sizes. The goal is to minimize total merge cost by merging smallest files first.

Example:

Given file sizes: $[10, 20, 30, 40]$

Step	Files	Merged	Cost	Total
1	[10, 20, 30, 40]	$10 + 20 = 30$	30	30
2	[30, 30, 40]	$30 + 30 = 60$	60	90
3	[40, 60]	$40 + 60 = 100$	100	190

Optimal Merge Order

- Merge (10, 20) $\rightarrow$ 30
- Merge (30, 30) $\rightarrow$ 60
- Merge (40, 60) $\rightarrow$ 100

Number of Comparisons

For 2 sorted files of size m and n:

- **Best case:** $\min(m, n)$ comparisons
- **Worst case:** $m + n - 1$ comparisons

Example: Merging [1, 2, 3] and [4, 5, 6] $\rightarrow$ Best case = 3 comparisons
Merging [1, 3, 5] and [2, 4, 6] $\rightarrow$ Worst case = 5 comparisons

Conclusion

2-way merging is the core operation in Optimal Merge Pattern. Using a greedy strategy (merge smallest files first) ensures minimal total merge cost. The number of comparisons depends on data distribution, but the merge cost depends only on file sizes.

4 Theoretical Logic Behind Optimal Merge Pattern Code Implementation

The Optimal Merge Pattern problem is a classic example of the **Greedy Algorithm** paradigm. It involves merging multiple sorted files (or datasets) into a single file while minimizing the total cost of all merge operations.

Problem Insight

Each merge operation incurs a cost equal to the sum of the sizes of the two files being merged. Thus, merging larger files early would result in higher cumulative costs. To minimize the total cost, it is optimal to:

- Always merge the **two smallest files first**.
- Repeat the process until only one file remains.

Why Greedy Works

The greedy strategy guarantees an optimal solution because:

- **Greedy choice property:** Choosing the two smallest files at each step minimizes the immediate cost, and repeating this ensures the overall cost is minimized.
- **Optimal substructure:** The structure of the problem allows the optimal solution to be built from optimal solutions to its subproblems.

Algorithm Components

- A **min-heap** is used to efficiently access the two smallest files at each step.
- After merging, the resulting file (with updated size) is reinserted into the heap.
- The process continues until one final file remains.

Step-by-Step Working

For example, given file sizes: $[20, 30, 10, 5]$

1. Insert all sizes into a min-heap: $[5, 10, 20, 30]$
2. Merge $5 + 10 = 15 \rightarrow$ Total cost $= 15$
3. Heap: $[15, 20, 30]$
4. Merge $15 + 20 = 35 \rightarrow$ Total cost $+= 35 = 50$
5. Heap: $[30, 35]$
6. Merge $30 + 35 = 65 \rightarrow$ Total cost $+= 65 =$ **115**

Time and Space Complexity

- **Time Complexity:** $O(n \log n)$ due to $n - 1$ heap insertions and extractions.
- **Space Complexity:** $O(n)$ for storing the heap.

Practical Justification

This pattern arises in various real-world domains:

- **File merging:** Compiler and OS design where merging of sorted segments/files is frequent.
- **Data compression:** Forms the basis for Huffman encoding tree construction.
- **Rope cutting problems:** Similar logic applies in minimizing the cost of combining ropes.

Conclusion

The Optimal Merge Pattern utilizes the greedy approach to minimize the cost of merging operations. By always merging the two smallest files, the algorithm ensures that expensive merges are delayed as much as possible, resulting in a minimal total cost.

4.1 Pseudocode

Algorithm 1 Optimal Merge Pattern

```
1:  procedure OptimalMerge(files)
2:      Insert all file sizes into min-heap H
3:      totalCost ← 0
4:      while H.size > 1 do
5:          a ← H.extractMin()
6:          b ← H.extractMin()
7:          mergeCost ← a + b
8:          totalCost ← totalCost + mergeCost
9:          H.insert(mergeCost)
10:     end while
11:     return totalCost
12: end procedure
```

4.2 Python Implementation

```python
import heapq

def optimal_merge(files):
    heapq.heapify(files)
    total_cost = 0
    merge_pattern = []

    while len(files) > 1:
        a = heapq.heappop(files)
        b = heapq.heappop(files)
        cost = a + b
        total_cost += cost
        heapq.heappush(files, cost)

        # Track the pattern
        merge_pattern.append((a, b, cost))

    return total_cost, merge_pattern

# Example usage
files = [20, 30, 10, 5]
cost, pattern = optimal_merge(files)

print("Total Merge Cost:", cost)
print("Merge Pattern:")
for step, (a, b, c) in enumerate(pattern, 1):
    print(f" Step {step}: Merge {a} + {b} = {c}")
```

Listing 1: Optimal Merge Pattern in Python

```
Total Merge Cost: 115
Merge Pattern:
    Step 1: Merge 5 + 10 = 15
    Step 2: Merge 15 + 20 = 35
    Step 3: Merge 30 + 35 = 65
```

4.3 C++ Implementation

```cpp
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

int optimalMerge(vector<int>& files) {
    priority_queue<int, vector<int>, greater<int>> pq(files.
       begin(), files.end());
    int total_cost = 0;

    while (pq.size() > 1) {
        int a = pq.top(); pq.pop();
        int b = pq.top(); pq.pop();
        int cost = a + b;
        total_cost += cost;
        pq.push(cost);
    }

    return total_cost;
}

int main() {
    vector<int> files = {20, 30, 10, 5};
    cout << optimalMerge(files) << endl; // Output: 115
    return 0;
}
```

Listing 2: Optimal Merge Pattern in C++

5 Time and Space Complexity

- **Time:** $O(n \log n)$ due to heap operations
- **Space:** $O(n)$ for the heap

6 Applications

- File merging in compilers
- Data compression algorithms (e.g., Huffman Coding)
- Efficient merging of sorted lists
- Rope cutting and combining problems

7 Conclusion

The Optimal Merge Pattern is an elegant greedy solution where always merging the smallest available files first leads to the minimum total cost. This approach underlies several efficient systems in data compression and file organization.