Batch: T3

Assignment No.: 6

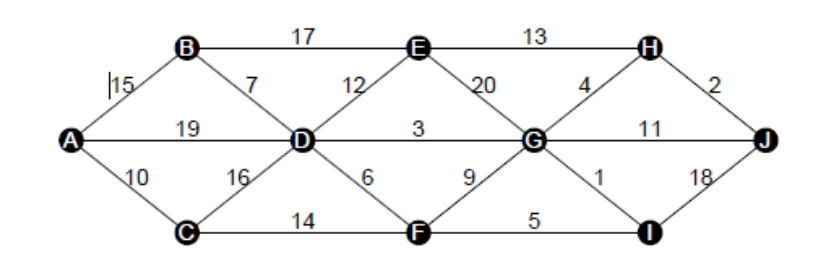
Title of Assignment: Greedy Method

Student Name: Piyush Dawkhare

PRN: 23510056

Problem Statement 1:

Implement Kruskal’s algorithm & Prim's algorithm to find Minimum Spanning Tree (MST) of the given an undirected, connected and weighted graph.



**Q)** How many edges does a minimum spanning tree for above example?

**ANS**: Number of edges for the minimum spanning tree for the above graph is:

Number of edges = Number of vertices -1

Number of edges = 9

**Q)** In a graph G. let the edge u v have the least weight. is it true that u v is always part of any minimum spanning tree of G?

Justify your answers.

**ANS:**

* If (u1, v1), (u2, v2), (u3, v3) is the **unique minimum-weight edge**, then it **always belongs to every MST**.
* If multiple edges share the minimum weight, then (u1, v1), (u2, v2), (u3, v3) is **not guaranteed** to appear in every MST (only in some of them).

Q) Let G be a graph and T be a minimum spanning tree of G. Suppose that the weight of an edge e is decreased. How can you find the minimum spanning tree of the modified graph? What is the runtime of your solution?

**ANS:**

* If the decreased edge is **already in MST**, the MST remains the same (just update weight).
* If it’s **not in MST**, add it, find the cycle, and remove the heaviest edge in that cycle.
* Runtime:
  + 1. O(1) if edge is in MST.
    2. O(log V) with advanced data structures (or O(V) with naive cycle search).
    3. Much faster than recomputing MST (O(E log V)).

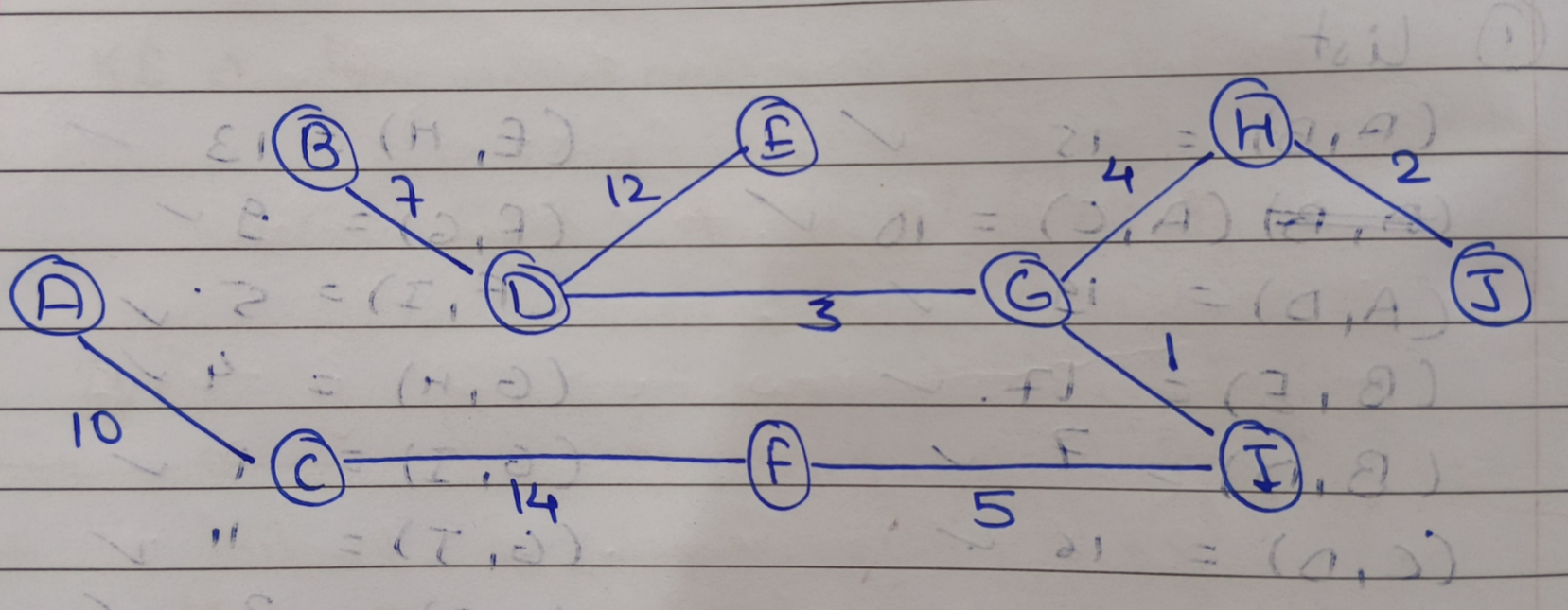
Q) Find order of edges for Kruskal's and Prim's?

**Khushal’s:** {(G, I) =1} 🡪 {(H, J) =2} 🡪 {(D, G) =3} 🡪 {(G, H) =4} 🡪 {(F, I) =5} 🡪 {(B, D) =7} 🡪 {(A, C) =10} 🡪 {(D, E) =12} 🡪{(C, F) =14}

**Prim’s:** starting from A

{(A, C) =10} 🡪 {(C, F) =12} 🡪 {(F, I) =5} 🡪 {(I, G) =1} 🡪 {(G, H) =4} 🡪 {(H, J) =2} 🡪 {(G, D) =3} 🡪 {(D, E) =12} 🡪{(D, B) =7}

**\***Minimum Spanning Tree for both the Algorithms is same



1. Technologies/libraries/algorithm used:

* #include algorithms
* #include <bits/stdc++.h> //This library includes these two libraries

1. Related theory or any related information:

-

1. Algorithms:

* Krushal’s Algorithm:
  + **Sort all edges** of the graph in non-decreasing order of their weights.
  + **Initialize MST** as an empty set (no edges yet).
  + For each edge (u, v) in sorted order:
    - If adding (u, v) does **not form a cycle** → add it to the MST.
    - Otherwise → discard it.
  + Repeat until the MST has **(V – 1) edges**, where V is the number of vertices.
  + To check for cycles efficiently, we use **Disjoint Set Union (Union-Find)** data structure
* Prims’s Algo:
  + **Start** with any vertex as part of the MST.
  + **Find the edge with minimum weight** that connects a vertex in the MST to a vertex **not yet in the MST**.
  + **Add that edge and vertex** to the MST.
  + **Repeat** step 2 until all vertices are included in the MST.

1. Code:
2. **Krushal’s Algo**
3. #include <bits/stdc++.h>
4. using *namespace* std;
5. *// Structure to represent an edge*
6. *struct* Edge {
7. *int* u, v, w;
8. *bool* operator<(*const* Edge*&* *other*) *const* {
9. return w < *other*.w; *// sort by weight*
10. }
11. };
12. *// Disjoint Set Union (Union-Find)*
13. *struct* DSU {
14. vector<*int*> parent, rank;
15. DSU(*int* *n*) {
16. parent.resize(*n*);
17. rank.resize(*n*, 0);
18. for (*int* i = 0; i < *n*; i++) parent[i] = i;
19. }
20. *int* find(*int* *x*) {
21. if (parent[*x*] != *x*)
22. parent[*x*] = find(parent[*x*]); *// path compression*
23. return parent[*x*];
24. }
25. *bool* unite(*int* *x*, *int* *y*) {
26. *x* = find(*x*);
27. *y* = find(*y*);
28. if (*x* == *y*) return false; *// already in same set → cycle*
29. *// union by rank*
30. if (rank[*x*] < rank[*y*]) swap(*x*, *y*);
31. parent[*y*] = *x*;
32. if (rank[*x*] == rank[*y*]) rank[*x*]++;
33. return true;
34. }
35. };
36. *int* main() {
37. *int* V = 10; *// number of vertices (A-J → 0-9)*
38. vector<Edge> edges;
39. *// Add edges (A=0, B=1, ..., J=9)*
40. edges.push\_back({0,1,15}); *// A-B*
41. edges.push\_back({0,2,10}); *// A-C*
42. edges.push\_back({0,3,19}); *// A-D*
43. edges.push\_back({1,3,7}); *// B-D*
44. edges.push\_back({1,4,17}); *// B-E*
45. edges.push\_back({2,3,16}); *// C-D*
46. edges.push\_back({2,5,14}); *// C-F*
47. edges.push\_back({3,4,12}); *// D-E*
48. edges.push\_back({3,5,6}); *// D-F*
49. edges.push\_back({3,6,3}); *// D-G*
50. edges.push\_back({4,6,20}); *// E-G*
51. edges.push\_back({4,7,13}); *// E-H*
52. edges.push\_back({5,6,9}); *// F-G*
53. edges.push\_back({5,8,5}); *// F-I*
54. edges.push\_back({6,7,4}); *// G-H*
55. edges.push\_back({6,8,1}); *// G-I*
56. edges.push\_back({6,9,11}); *// G-J*
57. edges.push\_back({7,9,2}); *// H-J*
58. edges.push\_back({8,9,18}); *// I-J*
59. *// Step 1: Sort edges*
60. sort(edges.begin(), edges.end());
61. DSU dsu(V);
62. vector<Edge> mst;
63. *int* totalWeight = 0;
64. *// Step 2: Process edges*
65. for (*auto* &e : edges) {
66. if (dsu.unite(e.u, e.v)) {
67. mst.push\_back(e);
68. totalWeight += e.w;
69. }
70. }
71. *// Print MSTck*
72. cout << "Edges in MST:\n";
73. for (*auto* &e : mst) {
74. cout << *char*(e.u + 'A') << " - " << *char*(e.v + 'A') << " : " << e.w << "\n";
75. }
76. cout << "Total Weight of MST = " << totalWeight << "\n";
77. return 0;
78. }
79. **Prims’s Algo**
80. #include <bits/stdc++.h>
81. using *namespace* std;
82. *const* *int* INF = 1e9;
83. *int* main() {
84. *int* V = 10; *// A-J (0 to 9)*
85. vector<vector<pair<*int*,*int*>>> adj(V);
86. *// Function to add edges (undirected)*
87. *auto* addEdge = [&](*int* *u*, *int* *v*, *int* *w*) {
88. adj[u].push\_back(make\_pair(v,w));
89. adj[v].push\_back(make\_pair(u,w));
90. };
91. *// Add edges (A=0, B=1, ..., J=9)*
92. addEdge(0,1,15); *// A-B*
93. addEdge(0,2,10); *// A-C*
94. addEdge(0,3,19); *// A-D*
95. addEdge(1,3,7); *// B-D*
96. addEdge(1,4,17); *// B-E*
97. addEdge(2,3,16); *// C-D*
98. addEdge(2,5,14); *// C-F*
99. addEdge(3,4,12); *// D-E*
100. addEdge(3,5,6); *// D-F*
101. addEdge(3,6,3); *// D-G*
102. addEdge(4,6,20); *// E-G*
103. addEdge(4,7,13); *// E-H*
104. addEdge(5,6,9); *// F-G*
105. addEdge(5,8,5); *// F-I*
106. addEdge(6,7,4); *// G-H*
107. addEdge(6,8,1); *// G-I*
108. addEdge(6,9,11); *// G-J*
109. addEdge(7,9,2); *// H-J*
110. addEdge(8,9,18); *// I-J*
111. vector<*int*> key(V, INF), parent(V, -1);
112. vector<*bool*> inMST(V, false);
113. key[0] = 0; *// start from A (0)*
114. *int* totalWeight = 0;
115. for (*int* i = 0; i < V; i++) {
116. *int* u = -1;
117. *// Pick vertex not in MST with min key*
118. for (*int* v = 0; v < V; v++) {
119. if (!inMST[v] && (u == -1 || key[v] < key[u]))
120. u = v;
121. }
122. inMST[u] = true;
123. totalWeight += key[u];
124. *// Update neighbors*
125. for (*auto* edge : adj[u]) {
126. *int* v = edge.first;
127. *int* w = edge.second;
128. if (!inMST[v] && w < key[v]) {
129. key[v] = w;
130. parent[v] = u;
131. }
132. }
133. }
134. cout << "Edges in MST:\n";
135. for (*int* v = 1; v < V; v++) {
136. cout << *char*(parent[v] + 'A') << " - " << *char*(v + 'A')
137. << " : " << key[v] << "\n";
138. }
139. cout << "Total Weight of MST = " << totalWeight << "\n";
140. return 0;
141. }

Problem Statement 2:

Emergency Evacuation – Container Loading Evacuating 200 families with varying luggage weights onto a bus with a capacity of 5000 kg.

Input sample:

Bus Capacity = 5000

Families’ luggage = [45, 120, 300, 150, 90, 600, 75, 430, 250, 500, 60, 100, 700, 85, 95, 200, 320, 180, 150, 275, ...]

Implement conainer loading problem.

1. Technologies/libraries/algorithm used:

* **Greedy algorithm** (pick the best immediate option)
* **Sorting** (O(n log n)) for prioritizing smaller weights
* Iteration to accumulate total weight
* #include <iostream>
* #include <vector>
* #include <algorithm> *// For sort*

1. Related theory or any related information:
   * Greedy Strategy:
   * Sort families’ luggage in ascending order.
   * Start loading luggage one by one:
   * If adding the next luggage exceeds capacity, stop.
   * Result: A fast, approximate solution.
   * This may not maximize total weight but maximizes number of families loaded.
   * In emergencies, fitting as many families as possible is often more important than absolute weight optimization.
2. Algorithms:

* Sort the luggage weights in **ascending order**.
* Initialize current\_load = 0 and loaded\_families = [].
* Iterate through the sorted luggage:
* If current\_load + luggage <= bus\_capacity, load it:
  + Add luggage to loaded\_families.
  + Update current\_load.
  + Else, **skip or stop**.
* Return loaded\_families and current\_load.

1. Code:
2. #include <iostream>
3. #include <vector>
4. #include <algorithm> *// For sort*
5. using *namespace* std;
6. pair<*int*, vector<*int*>> greedyContainerLoading(vector<*int*>*&* *weights*, *int* *capacity*) {
7. *// Sort luggage in ascending order*
8. sort(*weights*.begin(), *weights*.end());
9. *int* current\_load = 0;
10. vector<*int*> loaded\_families;
11. for (*int* w : *weights*) {
12. if (current\_load + w <= *capacity*) {
13. loaded\_families.push\_back(w);
14. current\_load += w;
15. } else {
16. break; *// Cannot load more luggage*
17. }
18. }
19. return {current\_load, loaded\_families};
20. }
21. *int* main() {
22. *int* bus\_capacity = 5000;
23. vector<*int*> families\_luggage = {45, 120, 300, 150, 90, 600, 75, 430, 250, 500,
24. 60, 100, 700, 85, 95, 200, 320, 180, 150, 275};
25. *auto* result = greedyContainerLoading(families\_luggage, bus\_capacity);
26. cout << "Total luggage loaded: " << result.first << endl;
27. cout << "Families' luggage loaded: ";
28. for (*int* w : result.second) {
29. cout << w << " ";
30. }
31. cout << endl;
32. return 0;
33. }

5. Output:

Total luggage loaded: 4725

Families' luggage loaded: 45 60 75 85 90 95 100 120 150 150 180 200 250 275 300 320 430 500 600 700

Problem Statement 3:

Startup Pitch Scheduling – Job Sequencing 50 startups pitching to an investor, with deadlines (1–10 slots) and profits (10–500).

Input sample:

Startups =

S1(deadline=2, profit=300), S2(1, 200), S3(3, 180), S4(2, 400),

S5(1, 250), S6(3, 100), S7(4, 500), S8(2, 150), S9(3, 220),

S10(4, 330), ...

Slots Available = 10

1. Technologies/libraries/algorithm used:

#include <iostream>

#include <vector>

#include <algorithm>

1. Related theory or any related information:

* You have **n startups** (jobs) with:
  + profit[i] = profit if startup i is scheduled.
  + deadline[i] = last slot by which startup i must pitch.
* You have **limited slots** (here, 10 slots).
* **Goal:** Schedule startups to **maximize total profit** without exceeding deadlines.
* **Constraint:** Each slot can host only **one startup**.

1. Algorithms:

* **Sort startups by decreasing profit**.
* Initialize an array slots to track if a slot is occupied (size = total slots available).
* For each startup in sorted order:
  + Find the **latest free slot ≤ its deadline**.
  + If found, assign the startup to that slot.
* Sum profits of scheduled startups to get **maximum total profit**.
* **Time Complexity:**
* Sorting: O(n log n)
* Slot allocation: O(n × slots)
* Overall: O(n log n + n × slots)

1. Code:

*#include <iostream>*

*#include <vector>*

*#include <algorithm>*

*using namespace std;*

*struct Startup {*

*string name;*

*int deadline;*

*int profit;*

*};*

*// Comparator to sort by profit descending*

*bool cmp(Startup a, Startup b) {*

*return a.profit > b.profit;*

*}*

*pair<int, vector<string>> jobSequencing(vector<Startup>& startups, int total\_slots) {*

*sort(startups.begin(), startups.end(), cmp);*

*vector<string> slots(total\_slots, "Empty");*

*int total\_profit = 0;*

*for (auto s : startups) {*

*// Find a free slot for this startup (from min(deadline, total\_slots) - 1 down to 0)*

*for (int j = min(total\_slots, s.deadline) - 1; j >= 0; j--) {*

*if (slots[j] == "Empty") {*

*slots[j] = s.name;*

*total\_profit += s.profit;*

*break;*

*}*

*}*

*}*

*return {total\_profit, slots};*

*}*

*int main() {*

*int total\_slots = 10;*

*vector<Startup> startups = {*

*{"S1", 2, 300}, {"S2", 1, 200}, {"S3", 3, 180}, {"S4", 2, 400},*

*{"S5", 1, 250}, {"S6", 3, 100}, {"S7", 4, 500}, {"S8", 2, 150},*

*{"S9", 3, 220}, {"S10", 4, 330}*

*// Add up to 50 startups as needed*

*};*

*auto result = jobSequencing(startups, total\_slots);*

*cout << "Maximum Total Profit: " << result.first << endl;*

*cout << "Scheduled Startups (Slot-wise):" << endl;*

*for (int i = 0; i < total\_slots; i++) {*

*cout << "Slot " << i+1 << ": " << result.second[i] << endl;*

*}*

*return 0;*

*}*

6. Output:

Maximum Total Profit: 1530

Scheduled Startups (Slot-wise):

Slot 1: S1

Slot 2: S4

Slot 3: S10

Slot 4: S7

Slot 5: Empty

Slot 6: Empty

Slot 7: Empty

Slot 8: Empty

Slot 9: Empty

Slot 10: Empty

Problem Statement 4:

Disaster Relief Supply Drop – Fractional Knapsack Helicopter capacity = 1000 kg, 100 supply items (divisible & indivisible).

Input sample:

Capacity = 1000

Supplies =

Rice(100, value=500), Medicine(50, value=400, indivisible), Water(200, value=600), Blankets(150, value=450, indivisible), Tents(300, value=800, indivisible), Wheat(120, value=300), Sugar(80, value=250), Oxygen(60, value=700, indivisible), Milk(90, value=350), FirstAid(40, value=500, indivisible), ...

1. Technologies/libraries/algorithm used:

* **#include <iostream>**
* **#include <vector>**
* **#include <algorithm>**

1. Related theory or any related information:

* You have a **helicopter with limited capacity**.
* There are **n supply items**, each with:
  + weight[i] = weight of the item
  + value[i] = importance/benefit
  + divisible = whether the item can be partially taken
* **Goal:** Maximize total value carried without exceeding helicopter capacity.

**Key Points:**

* **Divisible items:** Can take a fraction (like rice, water).
* **Indivisible items:** Must take whole (like medicine, tents).
* Fractional knapsack allows taking fractions of items with **highest value per kg first**.

**Time Complexity:**

* Sorting by value-to-weight ratio: O(n log n)
* Iterating through items: O(n)
* Overall: O(n log n)

1. Algorithms:

* Compute **value per weight** for each item: ratio = value / weight.
* Sort items by **descending ratio**.
* Initialize current\_weight = 0, total\_value = 0.
* For each item in sorted order:
  + If **indivisible** and can fit fully: take it.
  + If **divisible** and not enough capacity: take **fraction**.
  + Update current\_weight and total\_value.
* Stop when helicopter capacity is full.

1. Code:
2. #include <iostream>
3. #include <vector>
4. #include <algorithm>
5. using *namespace* std;
6. *struct* Supply {
7. string name;
8. *double* weight;
9. *double* value;
10. *bool* divisible; *// true if item can be fractionally taken*
11. };
12. *// Comparator for value/weight ratio*
13. *bool* cmp(Supply *a*, Supply *b*) {
14. *double* r1 = *a*.value / *a*.weight;
15. *double* r2 = *b*.value / *b*.weight;
16. return r1 > r2;
17. }
18. *double* fractionalKnapsack(*double* *capacity*, vector<Supply>*&* *supplies*, vector<pair<string, *double*>>*&* *taken*) {
19. sort(*supplies*.begin(), *supplies*.end(), cmp);
20. *double* total\_value = 0;
21. *double* current\_weight = 0;
22. for (*auto* s : *supplies*) {
23. if (current\_weight >= *capacity*) break;
24. if (s.divisible) {
25. *double* can\_take = min(s.weight, *capacity* - current\_weight);
26. total\_value += can\_take \* (s.value / s.weight);
27. current\_weight += can\_take;
28. *taken*.push\_back({s.name, can\_take});
29. } else {
30. if (current\_weight + s.weight <= *capacity*) {
31. total\_value += s.value;
32. current\_weight += s.weight;
33. *taken*.push\_back({s.name, s.weight});
34. }
35. }
36. }
37. return total\_value;
38. }
39. *int* main() {
40. *double* helicopter\_capacity = 1000;
41. vector<Supply> supplies = {
42. {"Rice", 100, 500, true},
43. {"Medicine", 50, 400, false},
44. {"Water", 200, 600, true},
45. {"Blankets", 150, 450, false},
46. {"Tents", 300, 800, false},
47. {"Wheat", 120, 300, true},
48. {"Sugar", 80, 250, true},
49. {"Oxygen", 60, 700, false},
50. {"Milk", 90, 350, true},
51. {"FirstAid", 40, 500, false}
52. *// Add up to 100 items as needed*
53. };
54. vector<pair<string, *double*>> taken;
55. *double* max\_value = fractionalKnapsack(helicopter\_capacity, supplies, taken);
56. cout << "Maximum value loaded: " << max\_value << endl;
57. cout << "Items taken (weight considered):" << endl;
58. for (*auto* t : taken) {
59. cout << t.first << ": " << t.second << " kg" << endl;
60. }
61. return 0;
62. }
63. Output:

Maximum value loaded: 4050

Items taken (weight considered):

FirstAid: 40 kg

Oxygen: 60 kg

Medicine: 50 kg

Rice: 100 kg

Milk: 90 kg

Sugar: 80 kg

Water: 200 kg

Blankets: 150 kg

Wheat: 120 kg

Problem Statement 5:

Multi-Server Log Merging – Optimal Merge A company has 20 log files with sizes from 5MB to 500MB. Logs arrive in rounds.

Input:

Round 1: [40, 120, 200, 10]

Round 2: [30, 15, 250, 90, 60]

Round 3: [100, 75, 35, 55, 85, 45]

Round 4: [500, 300, 150, 25, 70]

EXAMPLE

Input: find 5 in {15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14}

Output: 8 (the index of 5 in the array)

1. Technologies/libraries/algorithm used:

* #include <iostream>
* #include <vector>
* #include <queue>

1. Related theory or any related information:

**Problem Definition:**

* You have **multiple log files** of different sizes.
* You want to **merge them into one single log file**.
* Each merge operation **costs the sum of the sizes of files being merged**.
* **Goal:** Minimize the **total cost** of all merges.

**Key Concepts:**

* This is an example of a **greedy algorithm** problem.
* Always **merge the two smallest files first**, then repeat.
* Total cost = sum of all individual merge costs.

**Time Complexity:**

1. Using a **min-heap (priority queue)**:
   1. Each insertion/extraction: O(log n)
   2. Overall: O(n log n)
2. Algorithms:

* **Push all log file sizes** into a **min-heap**.
* While heap size > 1:
  + Extract the **two smallest sizes** a and b.
  + Merge them → new size = a + b.
  + Add a + b to **total cost**.
  + Push the new merged size back into the heap.
* Repeat until only one file remains.
* **Total cost** is the sum of all merge operations.
* This is **exactly like Huffman coding without symbols**, just sizes matter.

1. Code:
2. #include <iostream>
3. #include <vector>
4. #include <queue>
5. using *namespace* std;
6. *int* optimalMergeCost(vector<*int*>*&* *logs*) {
7. priority\_queue<*int*, vector<*int*>, greater<*int*>> minHeap;
8. *// Push all log sizes into min-heap*
9. for (*int* size : *logs*) {
10. minHeap.push(size);
11. }
12. *int* total\_cost = 0;
13. while (minHeap.size() > 1) {
14. *int* first = minHeap.top(); minHeap.pop();
15. *int* second = minHeap.top(); minHeap.pop();
16. *int* merge\_cost = first + second;
17. total\_cost += merge\_cost;
18. minHeap.push(merge\_cost);
19. }
20. return total\_cost;
21. }
22. *int* main() {
23. *// Logs arrive in rounds*
24. vector<vector<*int*>> rounds = {
25. {40, 120, 200, 10},
26. {30, 15, 250, 90, 60},
27. {100, 75, 35, 55, 85, 45},
28. {500, 300, 150, 25, 70}
29. };
30. vector<*int*> all\_logs;
31. for (*auto*& round : rounds) {
32. all\_logs.insert(all\_logs.end(), round.begin(), round.end());
33. }
34. *int* total\_merge\_cost = optimalMergeCost(all\_logs);
35. cout << "Total minimum merge cost: " << total\_merge\_cost << " MB" << endl;
36. return 0;
37. }

6. Output:

Total minimum merge cost: 8430 MB

Problem Statement 6:

Huffman Coding for IoT Sensor Data IoT network with 20 sensors, each with frequency (occurrence per hour) and importance factor (1–5).

Input:

Sensors with (Frequency × Importance):

Temp(400), Humidity(300), Pressure(600), Light(150), Gas(120), Motion(200), Smoke(100), Vibration(250), Sound(350), CO2(450), pH(80), Toxic(60), Voltage(220), Current(180), Wind(140), Rain(200), GPS(90), UV(70), Dust(160), Salinity(110)

Expected Output:

Shortest Codes → Pressure=‘0’, CO2=‘10’, Temp=‘110’, …

Longest Codes → Toxic=‘1111110’, pH=‘1111111’

Average Code Length ≈ 5.3 bits

1. Technologies/libraries/algorithm used:
2. #include <iostream>
3. #include <queue>
4. #include <vector>
5. #include <map>
6. Related theory or any related information:

**Problem Definition:**

* You have **n sensors**, each with a weight/frequency (frequency × importance in your case).
* Goal: Assign **binary codes** to sensors such that:
  + No code is a prefix of another (prefix-free codes).
  + **Average code length** is minimized (i.e., more frequent sensors get shorter codes).

**Key Points:**

* Huffman coding is **greedy**:
  + Always combine **two least frequent nodes** into a new node.
  + Repeat until only one node remains → forms a **binary tree**.
* **Average code length** formula:

Lavg=∑i=1nfi⋅li∑i=1nfiL\_{avg} = \frac{\sum\_{i=1}^{n} f\_i \cdot l\_i}{\sum\_{i=1}^{n} f\_i}Lavg​=∑i=1n​fi​∑i=1n​fi​⋅li​​

Where:

* f\_i = frequency of symbol i
* l\_i = length of the Huffman code of symbol i

**Time Complexity:**

* O(n log n) using **priority queue (min-heap)**.

1. Algorithms

* Create **leaf nodes** for each sensor with its weight (frequency × importance).
  + Insert all nodes into a **min-heap**.
* While heap size > 1:
  + Extract the **two nodes with lowest weights**.
  + Merge them into a **new node** (weight = sum of two).
  + Push the new node back into the heap.
* The remaining node is the **root of Huffman Tree**.
* Assign **binary codes**:
  + Traverse left → append 0
  + Traverse right → append 1
* Calculate **average code length**.

1. Code
2. #include <iostream>
3. #include <queue>
4. #include <vector>
5. #include <map>
6. using *namespace* std;
7. *// Huffman tree node*
8. *struct* Node {
9. string sensor;
10. *int* freq;
11. Node \*left, \*right;
13. Node(string *s*, *int* *f*) : sensor(*s*), freq(*f*), left(nullptr), right(nullptr) {}
14. };
15. *// Comparator for min-heap*
16. *struct* compare {
17. *bool* operator()(Node*\** *a*, Node*\** *b*) {
18. return *a*->freq > *b*->freq;
19. }
20. };
21. *// Recursive function to generate codes*
22. *void* generateCodes(Node*\** *root*, string *code*, map<string, string> *&codes*) {
23. if (!*root*) return;
25. if (!*root*->left && !*root*->right) {
26. *codes*[*root*->sensor] = *code*;
27. }
28. generateCodes(*root*->left, *code* + "0", *codes*);
29. generateCodes(*root*->right, *code* + "1", *codes*);
30. }
31. *int* main() {
32. *// Sensors with frequency × importance*
33. map<string, *int*> sensors = {
34. {"Temp",400}, {"Humidity",300}, {"Pressure",600}, {"Light",150}, {"Gas",120},
35. {"Motion",200}, {"Smoke",100}, {"Vibration",250}, {"Sound",350}, {"CO2",450},
36. {"pH",80}, {"Toxic",60}, {"Voltage",220}, {"Current",180}, {"Wind",140},
37. {"Rain",200}, {"GPS",90}, {"UV",70}, {"Dust",160}, {"Salinity",110}
38. };
39. priority\_queue<Node\*, vector<Node\*>, compare> minHeap;
40. *// Create leaf nodes*
41. for (*auto* s : sensors) {
42. minHeap.push(new Node(s.first, s.second));
43. }
44. *// Build Huffman Tree*
45. while (minHeap.size() > 1) {
46. Node\* left = minHeap.top(); minHeap.pop();
47. Node\* right = minHeap.top(); minHeap.pop();
48. Node\* merged = new Node("", left->freq + right->freq);
49. merged->left = left;
50. merged->right = right;
51. minHeap.push(merged);
52. }
53. Node\* root = minHeap.top();
54. *// Generate codes*
55. map<string, string> codes;
56. generateCodes(root, "", codes);
57. *// Print codes*
58. cout << "Sensor Huffman Codes:\n";
59. for (*auto* c : codes) {
60. cout << c.first << " : " << c.second << endl;
61. }
62. *// Calculate average code length*
63. *double* total\_weight = 0;
64. *double* total\_bits = 0;
65. for (*auto* s : sensors) {
66. total\_weight += s.second;
67. total\_bits += s.second \* codes[s.first].length();
68. }
69. *double* avg\_length = total\_bits / total\_weight;
70. cout << "\nAverage code length: " << avg\_length << " bits\n";
71. return 0;
72. }
73. Output

Sensor Huffman Codes:

CO2 : 010

Current : 11110

Dust : 11010

GPS : 110111

Gas : 01110

Humidity : 1100

Light : 10011

Motion : 0010

Pressure : 101

Rain : 11111

Salinity : 00111

Smoke : 00110

Sound : 1110

Temp : 000

Toxic : 011110

UV : 011111

Vibration : 1000

Voltage : 0110

Wind : 10010

pH : 110110

Average code length: 4.07329 bits