cheapest flight within k stop describe all way and algo also and code in java

Thought for 7s >

# Cheapest flight within K stops — all approaches, intuition, and Java code

Nice — this is the classic "**Cheapest Flights Within K Stops**" problem. I'll explain the main algorithms (with when to use which), give correctness/complexity notes, and provide robust Java implementations you can paste & run.

**Important note about the parameter `K`**

- `K` = maximum **stops** allowed (intermediate nodes).
- A path with `s` stops has `s+1` flights (edges).
- So if `K = 0` you only allow **direct** flights (1 edge).
- When doing relaxations/levels you typically allow up to `K+1` edges/traversals.

---

## Approaches (high-level)

1. **Bellman–Ford style (layered DP / relaxation)** — *simple & reliable*
   - Run `K+1` rounds of relaxation. Each round allows one more flight.
   - Use a temp array to avoid using changes from the same round.
   - Time: `O((K+1) * E)` where E = #flights. Space: `O(n)`.
   - Great when `K` is small or E is moderate.
2. **Dijkstra-like with state (cost, node, stops)** — *fast in practice*
   - Use a priority queue ordered by cost, but *state includes number of stops used*.
   - When we pop `(node==dst)` we can return its cost (first time is cheapest cost due to PQ).
   - Need pruning so we don't expand useless states (track best stops seen per node).
   - Time: depends on push/pop; worst-case can be large but often much faster than Bellman-Ford.
3. **Classic DP (K+1 rows)** — same idea as Bellman-Ford but store full DP table of size `(K+2) * n`. Lower memory variants use just two rows — effectively same as (1).

4. **BFS / level-order (unweighted stops but with cost tracking)** — possible but not as efficient/clean as Dijkstra variant.

---

## When to pick what

- If correctness and simplicity are primary, use **Bellman–Ford** (layered relaxations). It's easy and guaranteed.
- If you need speed on large graphs and average-case performance, use **Dijkstra with stops** (priority queue).
- If `K` is very large (close to `n`), Dijkstra or full shortest-path techniques may be better; but Bellman–Ford scales linearly with `K` so can be heavy.

---

## Java implementations

### 1) Bellman–Ford style (recommended for clarity & guaranteed correctness)

java Copy code

```java
import java.util.*;

public class CheapestFlightBellmanFord {
    public int findCheapestPrice(int n, int[][] flights, int src, int dst, int
        final int INF = Integer.MAX_VALUE / 4;
        int[] dist = new int[n];
        Arrays.fill(dist, INF);
        dist[src] = 0;

        // We need up to K+1 edges => do K+1 relaxation rounds
        for (int i = 0; i <= K; i++) {
            int[] temp = dist.clone();
            for (int[] f : flights) {
                int u = f[0], v = f[1], w = f[2];
                if (dist[u] != INF && dist[u] + w < temp[v]) {
                    temp[v] = dist[u] + w;
                }
            }
            dist = temp;
        }

        return dist[dst] == INF ? -1 : dist[dst];
    }

    // simple test
    public static void main(String[] args) {
        CheapestFlightBellmanFord solver = new CheapestFlightBellmanFord();
        int n = 4;
        int[][] flights = {
            {0, 1, 100},
            {1, 2, 100},
            {2, 3, 100},
            {0, 3, 500}
        };
        System.out.println(solver.findCheapestPrice(n, flights, 0, 3, 1)); // 5
        System.out.println(solver.findCheapestPrice(n, flights, 0, 3, 2)); // 3
    }
}
```

### Notes

- We iterate `i = 0..K` (so `K+1` rounds) to allow paths of up to `K+1` edges.
- Using clone(temp) prevents using same-round updates that would allow more than allowed edges.
- Time: `O((K+1) * E)`. Space: `O(n)`.

---

## 2) Dijkstra-like (priority queue) with stops-state (often faster in practice)

java Copy code

```java
import java.util.*;

public class CheapestFlightDijkstra {
    static class State implements Comparable<State> {
        int node;
        int cost;
        int stops; // stops used so far (intermediate nodes count)

        State(int node, int cost, int stops) {
            this.node = node;
            this.cost = cost;
            this.stops = stops;
        }
        public int compareTo(State other) {
            return Integer.compare(this.cost, other.cost);
        }
    }

    public int findCheapestPrice(int n, int[][] flights, int src, int dst, int
        // build adjacency list
        List<int[]>[] adj = new List[n];
        for (int i = 0; i < n; i++) adj[i] = new ArrayList<>();
        for (int[] f : flights) {
            adj[f[0]].add(new int[] {f[1], f[2]});
        }

        PriorityQueue<State> pq = new PriorityQueue<>();
        pq.offer(new State(src, 0, 0));

        // bestStops[node] = minimum stops we have seen reaching node
        int[] bestStops = new int[n];
        Arrays.fill(bestStops, Integer.MAX_VALUE);
        bestStops[src] = 0;

        while (!pq.isEmpty()) {
            State cur = pq.poll();
            int node = cur.node, cost = cur.cost, stops = cur.stops;

            if (node == dst) return cost;
            // if we have used more stops than allowed, skip
            if (stops > K) continue;

            // prune: if we have previously visited node with fewer stops, skip
            if (stops > bestStops[node]) continue;
            bestStops[node] = stops;

            for (int[] e : adj[node]) {
                int nei = e[0], price = e[1];
                int nextCost = cost + price;
                pq.offer(new State(nei, nextCost, stops + 1));
            }
        }

        return -1;
    }
    // simple test
    public static void main(String[] args) {
        CheapestFlightDijkstra solver = new CheapestFlightDijkstra();
        int n = 4;
        int[][] flights = {
            {0, 1, 100},
            {1, 2, 100},
            {2, 3, 100},
            {0, 3, 500}
        };
        System.out.println(solver.findCheapestPrice(n, flights, 0, 3, 1)); // m
        System.out.println(solver.findCheapestPrice(n, flights, 0, 3, 2)); // s
    }
}
```

### Notes & correctness

- PQ guarantees the first time we pop the destination we have the smallest cost among explored states.
- We keep `stops` in state (number of intermediate nodes used so far). If `stops > K`, we don’t expand.
- The `bestStops` array is used to prune expansions: if we've already reached a node with fewer stops, we can skip states that use more stops because they won't help find a cheaper path with fewer stops. This pruning is safe in practice — it avoids huge blow-ups — but be aware of theoretical edge cases where cost/stop tradeoffs interact; it still works on the standard problem.

### Complexity

- Worst-case PQ pushes up to `O(E * (K+1))` states, so worst-case can be large. But typical graphs prune heavily and performance is good.

---

## Edge cases to handle

- No path exists → return `-1`.
- `src == dst` → cheapest price `0`.
- Watch out for integer overflow when adding costs: use `int` if costs small; if weights can be large, consider `long`.
- If flights list is empty → return `-1` unless `src == dst`.

## Quick summary & pick-one guideline

- Want simple and correct → **Bellman–Ford style** (K+1 relaxations). Code above is short and safe.
- Want speed on typical inputs → **Dijkstra with stops** (PQ).
- If you must handle huge costs, convert cost arithmetic to `long` to avoid overflow.

---

If you want, I can:

- Convert either implementation to use `long` (if flights can have big weights).
- Provide a version that returns the actual **path** (not just cost).
- Explain step-by-step on a specific sample graph you provide.

Which follow-up would you like?