CyberApocalypse 2026 — Coding — Toll Schedule

Category: Coding **Challenge:** Toll Schedule **Target:** `154.57.164.71:30351` (HTTP web IDE) **Flag:** `HTB{th3_p4ss_pr3f3rs_0n3_b4nn3r}`

---

1. Challenge

> Stonepass doesn't close for weather anymore. It closes on words borrowed from weather — "raiders," "avalanches," "public order" — while the schedule underneath keeps quietly playing favourites: one banner's convoys glide through, another's stack up in the snow until daylight dies. Elric Ashspar has spent weeks sketching the checkpoint's hardware and is certain of one thing — the mechanism isn't broken. It's tuned.
>
> To prove it, Elric first needs the number nobody at the gate wants published: the least total time this exact column of convoys could possibly spend waiting, under any legal assignment of clearances to convoys.

Behind the story is a clean optimisation problem.

I/O specification

```
Line 1: N G          (N convoys, G clearances, G ⩾ N)
Line 2: N integers   arrival times a_1..a_N   (any order)
Line 3: G integers   clearance opening times g_1..g_G (any order)
```

Rules:

- A convoy may only use a clearance that opens **at or after** its arrival time.
- Every convoy must be assigned exactly one clearance.
- Every clearance seats **at most one** convoy.
- A valid assignment is guaranteed to exist.

Output a single integer: the minimum achievable total wait, where a convoy's wait is `clearance_open_time - convoy_arrival_time`.

Provided example

```
Input:
4 4
2 4 6 9
3 5 8 11

Output:
6
```

(`2→3`, `4→5`, `6→8`, `9→11` gives `1+1+2+2 = 6`.)

---

2. Reconnaissance

The challenge advertises a raw host:port, but `nc` gets nothing — the port speaks HTTP, not a line protocol:

```
$ nc 154.57.164.71 30351 < /dev/null   # → silence, connection times out
$ curl -s -i http://154.57.164.71:30351/ | head -5
HTTP/1.1 200 OK
Server: Werkzeug/3.1.3 Python/3.14.0
Content-Type: text/html; charset=utf-8
Content-Length: 28320
```

It is the standard HTB "Coding" web IDE — a Monaco editor plus a submit endpoint. Grepping the page for the submission API:

```
$ grep -oE 'fetch\("[^"]*"' ts.html
fetch("/run"
```

So submissions are a single JSON POST, which means the whole challenge can be solved head-lessly from the terminal:

```
POST /run   {"code": "<source>", "language": "python"}
→ {"challengeCompleted": true, "flag": "HTB{...}", "result": {}}
```

Supported languages (from `/static/js/prompts.js`): `python`, `c`, `cpp`, `rust`. The graders feed the program on **stdin** and compare stdout, with hidden test cases beyond the sample.

---

3. Modelling the problem

This is an **assignment problem**: bipartite convoys ↔ clearances, edge `(i, j)` allowed iff `g_j ≥ a_i`, cost `g_j - a_i`, minimise total cost over perfect matchings on the convoy side.

The naive reflex is Hungarian / min-cost-max-flow — `O(N^3)`, and with hidden tests of unknown size that is both slow and needlessly complicated. Two observations collapse it to sorting.

Observation 1 — the objective only depends on *which* clearances are used

```
total wait = Σ (g_assigned(i) - a_i) = ( Σ_{j ∈ S} g_j ) - ( Σ_i a_i )
```

where `S` is the set of `N` clearances actually used. The term `Σ a_i` is a constant of the input. So the task is:

> **choose which `N` of the `G` clearances to use, minimising their total opening time — subject to that choice being matchable.**

The *pairing* inside `S` is irrelevant to the cost; it only matters for feasibility.

Observation 2 — feasibility is a sorted-domination test

Let `A` be the arrivals sorted ascending and `S` a candidate set of clearances sorted ascending. Then `S` can be matched to all convoys **iff**

```
S[k] ⩾ A[k]     for every k = 0 .. N-1
```

Why: (⇐) if the condition holds, matching the k-th smallest clearance to the k-th smallest arrival is itself legal. (⇒) conversely, suppose some legal matching exists. The `k+1` smallest arrivals `A[0..k]` must be served by `k+1` distinct clearances, each `⩾` its own arrival and in particular the largest of them must be `⩾ A[k]`; those `k+1` clearances all sit at index `⩾ k` in sorted `S`, which forces `S[k] ⩾ A[k]`. This is just Hall's condition specialised to an interval (threshold) structure — a legal matching can always be uncrossed into the sorted one by an exchange argument: if `a_1 < a_2` are matched to `g_1 > g_2`, swapping them keeps both pairs legal (`g_1 ⩾ g_2 ⩾ a_2 > a_1`) and leaves the cost unchanged.

Observation 3 — the feasible clearance sets form a matroid

The sets of clearances that can be matched into distinct convoys are exactly the independent sets of a **transversal matroid**. Minimum-weight bases of a matroid are found by the plain greedy: scan elements in increasing weight, take each one whose addition keeps the set independent.

Here the weights are the opening times, so we scan clearances in **ascending** order. Because each newly considered clearance is the largest so far, it lands at the *end* of the sorted selected list — index `k = |S|` — and the domination test for every earlier element is untouched. The independence check therefore degenerates to a single comparison:

> take clearance `t` iff `t ⩾ A[k]`, where `k` is the number already taken.

The algorithm

1. Sort arrivals ascending → `A`.
2. Sort clearance times ascending → `Gs`.
3. `k = 0`, `total = 0`. For each `t` in `Gs`: if `k == N` stop; if `t >= A[k]`, add `t - A[k]` to `total` and `k += 1`.
4. Print `total`.

Complexity: **O((N + G) log(N + G))** time, O(N + G) memory — dominated by the two sorts. Note the intuition sanity-check: the earliest-arriving convoy is served by the earliest usable clearance, and clearances too early to be usable by anyone are simply skipped.

Why the greedy is not the naive "each convoy takes the next clearance"

The subtlety is the skipping. Consider `A = [1, 5]`, `Gs = [1, 2, 5]`. Pairing positionally (`1→1`, `5→2`) is illegal. The greedy takes `1` for arrival `1`, **rejects** `2` (it fails `2 >= A[1] = 5`), then takes `5` — total wait `0`. Any algorithm that consumes clearances without the rejection test gets this wrong.

---

4. Solution

```python
import sys

def main():
    data = sys.stdin.buffer.read().split()
    n = int(data[0]); g = int(data[1])
    a = sorted(map(int, data[2:2+n]))
    gates = sorted(map(int, data[2+n:2+n+g]))
    total = 0
    k = 0
    for t in gates:
        if k >= n:
            break
        if t >= a[k]:
            total += t - a[k]
            k += 1
    sys.stdout.write(str(total))

main()
```

`sys.stdin.buffer.read().split()` tokenises the whole input at once, so the solution is indifferent to how the three lines are wrapped or padded.

Sample check:

```
$ printf '4 4\n2 4 6 9\n3 5 8 11\n' | python3 sol.py
6
```

---

5. Verifying before submitting

Hidden test cases mean a wrong greedy costs blind guesses, so the greedy was validated two independent ways locally.

5.1 Exhaustive brute force (small inputs)

Try *every* injective mapping of convoys to clearances, keep the cheapest legal one; 4000 random cases with `N ⩽ 5`, `G ⩽ 7`, values in `0..12`:

```
def brute(a, gates):
    best = None
    for combo in itertools.permutations(range(len(gates)), len(a)):
        tot, ok = 0, True
        for i, j in enumerate(combo):
            if gates[j] < a[i]:
                ok = False; break
            tot += gates[j] - a[i]
        if ok and (best is None or tot < best):
            best = tot
    return best
```

```
trials done, mismatches: 0
```

5.2 Exact DP (medium inputs)

A subset-selection DP over sorted arrays — `dp[j][k]` = min cost using the first `j` clearances to serve the `k` earliest convoys — which makes no greedy assumption beyond the sorted-domination characterisation (itself already confirmed by §5.1). 3000 random cases with `N ⩽ 25`, `G ⩽ 50`:

```
def dp(a, gates):
    a, gates = sorted(a), sorted(gates)
    n = len(a)
    cur = [0] + [INF] * n
    for t in gates:
        nxt = cur[:]
        for k in range(n):
            if cur[k] < INF and t ⩾ a[k]:
                nxt[k+1] = min(nxt[k+1], cur[k] + t - a[k])
        cur = nxt
    return cur[n]
```

```
dp check mismatches: 0
```

Both oracles agree with the greedy on every case, on top of the provided sample.

6. Getting the flag

Posting the source straight to `/run`:

```
import json, urllib.request
code = open('sol.py').read()
req = urllib.request.Request(
    "http://154.57.164.71:30351/run",
    data=json.dumps({"code": code, "language": "python"}).encode(),
    headers={"Content-Type": "application/json"})
print(json.load(urllib.request.urlopen(req, timeout=120)))
```


```
{
  "challengeCompleted": true,
  "flag": "HTB{th3_p4ss_pr3f3rs_0n3_b4nn3r}",
  "result": {}
}
```


All hidden tests passed on the first submission.

Flag

```
HTB{th3_p4ss_pr3f3rs_0n3_b4nn3r}
```

---

7. Takeaways

- **Separate cost from feasibility.** `Σ(g - a) = Σg - Σa` turns a matching problem into a *subset selection* problem; the pairing then only has to be legal, not clever.
- **Threshold constraints sort away.** Whenever an edge exists iff `right ≥ left`, Hall's condition reduces to element-wise domination of two sorted lists, and the `O(N^3)` assignment machinery is unnecessary.
- **Matroid greedy explains the one-liner.** Recognising the transversal matroid is what justifies the single `t ≥ a[k]` comparison as *provably* optimal rather than merely plausible.
- **Cross-check greedies against an oracle.** A brute force plus an exact DP over thousands of random cases costs a minute and removes all doubt before a submission against hidden tests.
- **Check the protocol.** The "host:port" was a Flask app, not a socket service — one `curl` saved a lot of `nc` fumbling, and the `/run` endpoint made the whole solve scriptable.