CyberApocalypse 2026 — Coding — Ash Record

- **Category:** Coding
- **Challenge:** Ash Record
- **Endpoint:** `http://154.57.164.83:30124`
- **Flag:** `HTB{th3_h4ml3t_w4s_k3pt_n0t_burn3d}`

---

1. Challenge overview

Aeron's scouts find a riverside hamlet abandoned mid-meal — ovens still warm, stew fresh, chairs pushed back. Elowen Ashglass reads the scene as a staged procession rather than a raid: the hamlet was *preserved*, not burned.

Mechanically, the story is a dressing for a sequence-matching problem. Elowen has a set of residues (each with a timestamp and a material type) and a suspected *extraction sequence* of materials. She wants the longest **prefix** of that sequence she can confirm as a subsequence of the residues, in chronological order, where any two consecutively matched residues are at least `min_gap` apart in time.

Input format

```
N P min_gap
<P space-separated material types — the extraction sequence>
<timestamp> <material>      (N lines, NOT sorted by timestamp)
```

Output

A single integer: the maximum number of consecutive steps, counted from the start of the extraction sequence, that can be confirmed.

Provided example

```
5 4 3
ash rope oil ash
1 ash
4 rope
7 oil
10 ash
11 rope
```

Expected output: `4`

Steps: `ash@1 → rope@4 → oil@7 → ash@10`, each gap exactly `3 ⩾ 3`. The `rope@11` residue is unused.

---

2. Reconnaissance

The challenge address does not speak a raw TCP protocol — connecting with `nc` yields no banner. It is an HTTP service:

```
$ curl -sS -i http://154.57.164.83:30124/ | head -5
HTTP/1.1 200 OK
Server: Werkzeug/3.1.3 Python/3.14.0
Content-Type: text/html; charset=utf-8
```

It is the standard HTB "Coding" web judge: a Monaco editor page that POSTs your source to `/run` and returns either a failing test case or the flag.

```
fetch("/run", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ code: code, language: "python" }),
})
// → { challengeCompleted: true, flag: "..." }  |  { result: <failed test case> }
```

So the whole task is to write a correct, fast solution and POST it.

---

3. Solving the problem

3.1 Restating it cleanly

Sort the residues chronologically. We then walk the pattern stage by stage. For each stage we must pick a residue of the required material that comes *after* the previously picked one and whose timestamp is at least `min_gap` later. We stop at the first stage we cannot satisfy, and the answer is the number of stages satisfied.

3.2 The greedy argument

The natural approach is greedy: **at each stage, pick the earliest residue that is still legal.**

This is optimal by a standard exchange argument. The only state that constrains the future is the timestamp (and position) of the last matched residue — a later stage is feasible iff there exists a matching residue at time `⩾ last_ts + min_gap`. That feasibility condition is monotone: the smaller `last_ts` is, the strictly larger the set of future options. So if any solution confirms `k` stages, the greedy — which by induction always holds a `last_ts` no larger than that solution's at every stage — also confirms `k`. Hence greedy achieves the maximum, and taking the first stage it fails at gives the longest confirmable prefix.

Note the problem asks for a *prefix*, not the best matching anywhere in the pattern, which is what makes this simple greedy sufficient — no dynamic programming is needed.

3.3 Making each stage O(log N)

A linear scan per stage would be O(N·P), too slow at the input sizes the judge uses. Instead, bucket the residues by material type, each bucket holding `(timestamp, position)` pairs sorted ascending. Answering a stage becomes a single binary search for the first entry in that material's bucket strictly greater than `(last_ts + min_gap - epsilon)`.

Total complexity: **O(N log N + P log N)** time, **O(N)** memory.

3.4 The subtle bug: `min_gap = 0`

My first version tracked only the timestamp and used `bisect_left(times, last_ts + gap)`. It passed the provided example, but differential fuzzing against an exhaustive brute force immediately found this counterexample:

```
2 3 0
oil oil rope
1 oil
1 rope
```

Greedy returned `3`, brute force returned `1`.

With `min_gap = 0`, the required next timestamp equals the current one, so `bisect_left` re-found *the very same residue* and happily matched `oil` twice using a single physical residue. A residue may only be consumed once, so the constraint is not just "timestamp far enough ahead" but also "strictly later in the sequence".

The fix is to carry the position alongside the timestamp and binary search on the tuple:

```
j = bisect_right(entries, (last_ts + gap, last_pos))
```

Because the global array is sorted by timestamp, positions increase with timestamps, so `bisect_right` on `(ts, pos)` yields exactly the right element: either a strictly later timestamp, or the same timestamp at a strictly later position (legal only when `gap == 0`). This also fixes ordering for duplicate timestamps in general.

---

4. Final solution

```python
import sys
from bisect import bisect_right


def main():
    data = sys.stdin.buffer.read().split()
    if not data:
        print(0)
        return
    i = 0
    n = int(data[i]); i += 1
    p = int(data[i]); i += 1
    gap = int(data[i]); i += 1

    pattern = data[i:i + p]
    i += p

    residues = []
    for _ in range(n):
        ts = int(data[i]); i += 1
        residues.append((ts, data[i])); i += 1

    # chronological order; ties keep input order
    residues.sort(key=lambda r: r[0])

    # per material type: (timestamp, position) sorted ascending
    buckets = {}
    for pos, (ts, mat) in enumerate(residues):
        buckets.setdefault(mat, []).append((ts, pos))

    # greedy: each stage takes the earliest residue that is far enough
    # ahead of the previous match, which keeps later stages maximally free
    count = 0
    last_ts = None
    last_pos = -1
    for mat in pattern:
        entries = buckets.get(mat)
        if not entries:
            break
        if last_ts is None:
            j = 0
        else:
            j = bisect_right(entries, (last_ts + gap, last_pos))
        if j >= len(entries):
            break
        last_ts, last_pos = entries[j]
        count += 1

    sys.stdout.write(str(count) + "\n")


main()
```

Implementation details worth noting:

- Input is read as one raw token stream (`sys.stdin.buffer.read().split()`), so the "residues are not necessarily given in timestamp order" caveat and any whitespace irregularity are handled uniformly.
- Material types are kept as `bytes` tokens and never decoded — they are only ever compared and used as dict keys, which avoids `N` pointless UTF-8 decodes.
- A material absent from the residues (`buckets.get(mat)` returning `None`) terminates the prefix immediately.

---

5. Verification

Provided example:

```
$ printf '5 4 3\nash rope oil ash\n1 ash\n4 rope\n7 oil\n10 ash\n11 rope\n' | python3 sol.py
4
```

Regression for the `min_gap = 0` bug:

```
$ printf '2 3 0\noil oil rope\n1 oil\n1 rope\n' | python3 sol.py
1
```

Differential fuzzing. I wrote an exhaustive brute force (recursive search over every valid subsequence) and compared against the greedy on 1000 random small inputs, with alphabets of 1–4 materials, `N` up to 11, `P` up to 7, and `min_gap` drawn from `{0, 1, 2, 3, 7}` to keep collisions and zero-gap cases frequent:

```
$ python3 fuzz.py && python3 fuzz2.py
all ok
all ok
```

Performance. Worst-case-shaped input with `N = P = 200000`:

```
$ time python3 sol.py < big.txt
40033
python3 sol.py < big.txt  0.14s user 0.01s system 97% cpu 0.157 total
```

Comfortably inside any judge limit.

---

6. Getting the flag

```
$ python3 -c "
import json,urllib.request
code=open('sol.py').read()
req=urllib.request.Request('http://154.57.164.83:30124/run',
    data=json.dumps({'code':code,'language':'python'}).encode(),
    headers={'Content-Type':'application/json'})
print(urllib.request.urlopen(req,timeout=120).read().decode())
"
{"challengeCompleted":true,"flag":"HTB{th3_h4ml3t_w4s_k3pt_n0t_burn3d}","result":{}}
```

Flag

```
HTB{th3_h4ml3t_w4s_k3pt_n0t_burn3d}
```

7. Takeaways

- The narrative hides a plain "longest confirmable prefix of a pattern as a time-gapped subsequence" problem. Because it asks for a **prefix**, earliest-match greedy is provably optimal and no DP is required.
- Bucketing by symbol plus binary search turns the per-stage scan into `O(log N)`.
- The real trap is `min_gap = 0` combined with duplicate timestamps, where a timestamp-only greedy silently reuses one residue for several stages. Tracking `(timestamp, position)` and using `bisect_right` fixes it. Differential fuzzing against a brute force found this in seconds — the provided example never would have.