Kai Heng Gan

1. Explanation Document

1.1. Overview

This document explains how the provided C++ program:

1. **Parses** an input file containing station definitions and charger availability reports.
2. **Validates** the input format and ensures consistency.
3. **Calculates** per-station uptime, where *uptime* is defined as the percentage of total reporting coverage during which *at least one charger* is reported "up."
4. **Outputs** each station's ID and its integer (floor) uptime percentage.

If at any point the input file is invalid (formatting errors, missing data, unexpected IDs, etc.), the program prints `ERROR` and exits.

1.2. Requirements Recap

- A text file contains two sections:
    1. **[Stations]**
       Each line has a station ID followed by one or more charger IDs.
    2. **[Charger Availability Reports]**
       Each line specifies a charger ID, a start time (nanos), an end time (nanos), and a boolean ("true" / "false") for up/down.
- **Station IDs** are distinct `uint32_t` (though we store them in `unsigned long long`).
- **Charger IDs** are distinct across the entire file. No charger may appear in two different stations.
- All times fit in `uint64_t` (stored in `unsigned long long`).
- If a station has multiple chargers, the station is considered *up* at a given time if *any* of its chargers is up at that time.
- Uptime is the fraction:

$$\frac{station\_up\_time}{station\_coverage\_time} * 100$$

  then **floored** (rounded down).

- **Coverage** for a station is determined by unioning the coverage intervals [min_start,max_end] of *each charger* in the station.

1.3. Implementation Details

1.3.1. Data Structures

1. **`charger_to_station`**:

A map (`std::unordered_map<unsigned long long, unsigned long long>`) mapping `charger_id` → `station_id`. Populated during the `[Stations]` parsing.

2. **`station_ids`**:

A `std::set<unsigned long long>` storing all unique station IDs found in `[Stations]`.

3. **`charger_data`**:

A map (`std::unordered_map<unsigned long long, ChargerInfo>`) that, for each charger, holds:

   - `min_start` / `max_end`: the earliest/largest times we have any report for that charger.
   - `up_intervals`: a list of `(start, end)` intervals where the charger is reported "true."
   - `initialized`: a boolean tracking if we've seen any availability line for this charger.

4. **`ChargerInfo`** Struct:

```
struct ChargerInfo {
    unsigned long long min_start;
    unsigned long long max_end;
    std::vector<std::pair<unsigned long long, unsigned long long>>
up_intervals;
    bool initialized = false;
};
```

5. **`station_coverage`** and **`station_up`**:

For each station, we collect lists of coverage intervals (i.e., `[min_start, max_end]` for each charger) and up intervals (the union of each charger's up intervals).

```
std::map<unsigned long long, std::vector<std::pair<unsigned long long,
unsigned long long>>> station_coverage;
std::map<unsigned long long, std::vector<std::pair<unsigned long long,
unsigned long long>>> station_up;
```

1.3.2. Parsing the Input

1. **Find `[Stations]`**
   The program scans lines until it sees a line exactly matching `[Stations]`. If it never finds one, it prints `ERROR` and exits.
2. **Read station lines**
   After `[Stations]`, each non-empty line is split into tokens.
   - The **first token** is the station ID (must be all digits).
   - The **remaining tokens** are charger IDs (also must be all digits).
   - We store each `(charger_id → station_id)` in `charger_to_station`. We also record the `station_id` in `station_ids`.
3. **Find `[Charger Availability Reports]`**
   The program continues scanning until a line matches `[Charger Availability Reports]`. If not found, print `ERROR`.
4. **Read Charger Availability lines**
   - Each line should have exactly 4 tokens:
     1. Charger ID (digits)
     2. Start time (digits)
     3. End time (digits)
     4. "true" or "false"
   - If the charger ID was *not* defined in the `[Stations]` section, print `ERROR`.
   - Update `ChargerInfo` for that charger.
     - `min_start/max_end` are updated to track the earliest/largest coverage times.
     - If `up == true`, push this `(start, end)` interval into `up_intervals`.

1.3.3. Interval Merging

The function `unify_intervals` merges overlapping or adjacent intervals (e.g., `(0,50)` and `(50,100)` unify to `(0,100)`). This is necessary to accurately compute total up time and coverage length.

1.3.4. Coverage vs. Up

- **Coverage** for a single charger: `[min_start, max_end]`. Any gaps are implicitly downtime, but they still count in the charger's coverage window.
- **Station Coverage**: For a station with multiple chargers, coverage is the union of coverage intervals from each charger.
- **Station Up**: The station is considered up if *any* charger is up. So for each charger, we unify its "up intervals" across the station. Then we merge all those intervals across all chargers to get the station's up intervals.

1.3.5. Calculating Uptime

For each station:

1. Merge all coverage intervals → `coverage_merged`.
2. Merge all "up" intervals → `up_merged`.
3. `coverage_length = sum(cover.end - cover.start for cover in coverage_merged).`
4. `up_length = sum(up.end - up.start for up in up_merged).`
5. `uptime_percent = floor( (up_length / coverage_length) * 100 )` if `coverage_length > 0`; otherwise `0`.

1.3.6. Output

- If any errors were detected, we printed `ERROR` and exited.
- Otherwise, we print each station ID and its uptime percentage in ascending station ID order:
- `<stationID> <uptimePercent>`

1.4. Complexity Analysis

- **Parsing**: Linear in the number of lines.
- **Interval Merging**: Each charger merges its up intervals in O(M log M) time (M = number of intervals for a charger).
- **Station Coverage Merging**: Summed across all chargers. Each station merges N intervals (N = sum of intervals from all chargers at that station) in O (N log N).
- Typically, this is efficient enough for moderate-sized inputs.

1.5. Edge Cases

1. **No coverage for a station**: If no chargers for a station have any availability lines, coverage length is 0 → uptime is 0%.
2. **Charger ID repeated in two stations**: The code prints `ERROR` if it detects a charger assigned to more than one station.
3. **Overlapping intervals**: Merged properly in `unify_intervals`.
4. **Contiguous intervals**: E.g., `(0,50)` and `(50,100)` are treated as a single merged interval `(0,100)`.
5. **Invalid lines or missing sections**: The code prints `ERROR`.

2. Unit & Integration Tests

This section provides sample **test input files** (integration tests) and **unit tests** (targeted function tests).

2.1. Integration Test Files

Below are several minimal input files to test various conditions. Each test has an **Expected Output** shown below.

2.1.1. Test 1: Minimal Valid Input

File: `test_input_1.txt`

```
[Stations]
0 1001

[Charger Availability Reports]
1001 0 100000 true
```

- **Explanation**:
  - Station `0` has one charger `1001`.
  - Coverage `[0,100000]`, always up → uptime = 100%.

Expected Output:

```
0 100
```

2.1.2. Test 2: Missing `[Stations]` Section

File: `test_input_2.txt`

```
[Charger Availability Reports]
1001 0 100000 true
```

- **Explanation**: We never see `[Stations]`, so the code should print `ERROR`.

Expected Output:

```
ERROR
```

2.1.3. Test 3: Missing `[Charger Availability Reports]` Section

File: `test_input_3.txt`

```
[Stations]
0 1001
```

- **Explanation**: We have stations but no `[Charger Availability Reports]` → no coverage intervals → coverage = 0 → uptime = 0.

However, the specification says we must have the `[Charger Availability Reports]` header. Because it is *completely missing*, the code will detect that and print `ERROR`.

Expected Output:

```
ERROR
```

2.1.4. Test 4: Charger Assigned to Multiple Stations

File: `test_input_4.txt`

```
[Stations]
0 1001
1 1001

[Charger Availability Reports]
1001 0 100000 true
```

- **Explanation**: Charger `1001` is listed under both station `0` and station `1`. This violates the uniqueness constraint.

Expected Output:

```
ERROR
```

2.1.5. Test 5: Charger in Reports Not in Stations

File: `test_input_5.txt`

```
[Stations]
0 1001

[Charger Availability Reports]
1002 0 100000 true
```

- **Explanation**: Charger `1002` was never mentioned in `[Stations]`, so we print `ERROR`.

Expected Output:

```
ERROR
```

2.1.6. Test 6: The Provided Example (with partial downtime)

File: `test_input_6.txt` (the original example)

```
[Stations]
```

```
0 1001 1002
1 1003
2 1004

[Charger Availability Reports]
1001 0 50000 true
1001 50000 100000 true
1002 50000 100000 true
1003 25000 75000 false
1004 0 50000 true
1004 100000 200000 true
```

Expected Output:

```
0 100
1 0
2 75
```

2.2. Building and Running Integration Tests

Compile (with Makefile):

1. `Run 'make'`

Run:

2. `./station_uptime <test_input_x.txt>`

Output:

3. Expected Output can be found in `input_x_expected_stdout.txt`

2.3. Unit Tests

Because the challenge states *"avoid non-standard dependencies"*, we can implement minimal "in-code" tests rather than linking a test framework. Below is an example of a **test_main.cpp** that exercises critical utility functions, such as `unify_intervals` and `intervals_length`.

2.3.1. test_main.cpp

```
#include <iostream>
#include <vector>
#include <utility>
#include <cassert>
#include "functions.h"

// Suppose we've extracted unify_intervals() and intervals_length() into
"functions.h"
// so that we can test them separately from the main program.
```

```
void test_unify_intervals() {
    using Interval = std::pair<unsigned long long, unsigned long long>;

    // 1) Overlapping intervals
    std::vector<Interval> intervals1 = {
        {0ULL, 50ULL}, {20ULL, 100ULL}, {100ULL, 200ULL}
    };
    // Expect merge => (0,200)
    auto merged1 = unify_intervals(intervals1);
    assert(merged1.size() == 1);
    assert(merged1[0].first == 0ULL && merged1[0].second == 200ULL);

    // 2) Adjacent intervals
    std::vector<Interval> intervals2 = {
        {0ULL, 50ULL}, {50ULL, 100ULL}, {200ULL, 300ULL}
    };
    // Expect => (0,100), (200,300)
    auto merged2 = unify_intervals(intervals2);
    assert(merged2.size() == 2);
    assert(merged2[0].first == 0ULL && merged2[0].second == 100ULL);
    assert(merged2[1].first == 200ULL && merged2[1].second == 300ULL);

    // 3) No intervals
    std::vector<Interval> intervals3;
    auto merged3 = unify_intervals(intervals3);
    assert(merged3.empty());

    std::cout << "test_unify_intervals() passed." << std::endl;
}

void test_intervals_length() {
    using Interval = std::pair<unsigned long long, unsigned long long>;

    std::vector<Interval> intervals = {
        {0ULL, 50ULL}, {50ULL, 100ULL}, {200ULL, 300ULL}
    };
    // length = (50-0)+(100-50)+(300-200) = 50 + 50 + 100 = 200
    unsigned long long length = intervals_length(intervals);
    assert(length == 200ULL);

    std::cout << "test_intervals_length() passed." << std::endl;
}

int main() {
    test_unify_intervals();
    test_intervals_length();

    std::cout << "All unit tests passed!" << std::endl;
    return 0;
}
```

2.3.2. `functions.h` (Extracted from main code)

Below is a snippet showing how I might extract the key functions to reuse in both the main program and the test harness.

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

#include <vector>
#include <algorithm>

// Merges overlapping or adjacent intervals
inline std::vector<std::pair<unsigned long long, unsigned long long>>
unify_intervals(std::vector<std::pair<unsigned long long, unsigned long
long>>& intervals)
{
    if (intervals.empty()) {
        return {};
    }
    std::sort(intervals.begin(), intervals.end(),
        [](auto &a, auto &b){ return a.first < b.first; });

    std::vector<std::pair<unsigned long long, unsigned long long>> merged;
    merged.reserve(intervals.size());

    auto current_start = intervals[0].first;
    auto current_end   = intervals[0].second;

    for (size_t i = 1; i < intervals.size(); ++i) {
        auto s = intervals[i].first;
        auto e = intervals[i].second;
        if (s <= current_end) {
            // Extend
            if (e > current_end) {
                current_end = e;
            }
        } else {
            merged.push_back({current_start, current_end});
            current_start = s;
            current_end   = e;
        }
    }
    merged.push_back({current_start, current_end});
    return merged;
}

// Returns sum of (end - start) for each interval
inline unsigned long long
intervals_length(const std::vector<std::pair<unsigned long long, unsigned
long long>>& intervals)
{
    unsigned long long total = 0ULL;
    for (auto &iv : intervals) {
        total += (iv.second - iv.first);
    }
    return total;
}

#endif // FUNCTIONS_H
```

2.3.3. Building & Running the Unit Tests

Unit Tests

Compile (with Makefile):

1. `Run 'make'`

Run:

2. `./test_main`

Output:

```
test_unify_intervals() passed.

test_intervals_length() passed.

All unit tests passed!
```

3. Summary

- The **main** application (`main.cpp`) handles end-to-end parsing, validation, interval merging, and final output.
- **Integration tests** involve feeding different input files, ensuring the console output matches expectations.
- **Unit tests** focus on internal logic like interval merging and length calculation.

My submission:

1. `main.cpp` – the primary source code.
2. `functions.h` – extracted utility functions
3. `test_main.cpp` – unit tests.
4. Several `test_input_*.txt` and `input_*_expected_stdout.txt` files
5. A README and an Explanation Document describing usage, design, and testing.