Explanation

Peak Vs Down

The difference in when we increment `peak` vs `down` is due to how we're tracking which element we're currently processing and what candy count it should get.

Let me break this down:

Peak (Ascending) Case:

```
while i < n and ratings[i] > ratings[i-1]:
    peak += 1# First increment
    total_candy += peak# Then add to total
    i += 1
```

Here, `i` points to the current element we're processing. Since `ratings[i] > ratings[i-1]`, the element at index `i` should get more candies than the previous element. We increment `peak` first because we want the current element (`ratings[i]`) to get `peak` candies.

Example: If we have ratings `[1, 2, 3]`

- At `i=1`: `peak=2`, element gets 2 candies
- At `i=2`: `peak=3`, element gets 3 candies

Down (Descending) Case:

```
while i < n and ratings[i] < ratings[i-1]:
    total_candy += down# First add to total
    down += 1# Then increment
    i += 1
```

Here, `i` still points to the current element, but since `ratings[i] < ratings[i-1]`, the current element should get fewer candies. However, we're building the descending sequence and need to know how many candies each element gets. We use `down` first (which represents candies for current element), then increment it for the next element in the descending sequence.

Example: If we have ratings `[3, 2, 1]` (starting from the peak)

- At `i=1`: current element gets `down=1` candy, then `down=2`
- At `i=2`: current element gets `down=2` candies

Why the difference?

The key insight is that in the ascending case, we're directly assigning candies to the current element. In the descending case, we're working backwards from the peak - each element in the descending sequence gets 1 more candy than the next element, so we need to track this incrementally.

The `down > peak` adjustment at the end handles the case where the descending sequence is longer than the ascending sequence, ensuring the peak element gets enough candies to satisfy both constraints.

Example

Walk through the algorithm step by step with the array `[4,3,2,1,2,3,2,2,1]`.

Initial Setup

- `n = 9`
- `total_candy = 1` (first child at index 0 gets 1 candy)
- `i = 1` (starting from second child)

Step-by-Step Execution

i = 1: ratings[1] = 3, ratings[0] = 4

- `3 < 4`, so we enter the **descending (down) phase**

- `peak = 1` (from previous iteration)
- `down = 1`

Down phase loop:

- **i = 1**: `ratings[1] < ratings[0]` (3 < 4)
  - `total_candy += down` → `total_candy = 1 + 1 = 2`
  - `down += 1` → `down = 2`
  - `i += 1` → `i = 2`
- **i = 2**: `ratings[2] < ratings[1]` (2 < 3)
  - `total_candy += down` → `total_candy = 2 + 2 = 4`
  - `down += 1` → `down = 3`
  - `i += 1` → `i = 3`
- **i = 3**: `ratings[3] < ratings[2]` (1 < 2)
  - `total_candy += down` → `total_candy = 4 + 3 = 7`
  - `down += 1` → `down = 4`
  - `i += 1` → `i = 4`
- **i = 4**: `ratings[4] > ratings[3]` (2 > 1), exit down loop

Peak adjustment check:

- `down = 4`, `peak = 1`
- `down > peak`, so adjust: `total_candy += down - peak` → `total_candy = 7 + (4-1) = 10`

Current candy distribution: `[4, 3, 2, 1, ?, ?, ?, ?, ?]`

i = 4: ratings[4] = 2, ratings[3] = 1

- `2 > 1`, so we enter the **ascending (peak) phase**
- `peak = 1`

Peak phase loop:

- **i = 4**: `ratings[4] > ratings[3]` (2 > 1)

- `peak += 1` → `peak = 2`
- `total_candy += peak` → `total_candy = 10 + 2 = 12`
- `i += 1` → `i = 5`

- **i = 5**: `ratings[5] > ratings[4]` (3 > 2)
  - `peak += 1` → `peak = 3`
  - `total_candy += peak` → `total_candy = 12 + 3 = 15`
  - `i += 1` → `i = 6`
- **i = 6**: `ratings[6] < ratings[5]` (2 < 3), exit peak loop

i = 6: ratings[6] = 2, ratings[5] = 3

- `2 < 3`, so we enter the **descending (down) phase**
- `peak = 3`
- `down = 1`

Down phase loop:

- **i = 6**: `ratings[6] < ratings[5]` (2 < 3)
  - `total_candy += down` → `total_candy = 15 + 1 = 16`
  - `down += 1` → `down = 2`
  - `i += 1` → `i = 7`
- **i = 7**: `ratings[7] = ratings[6]` (2 = 2), exit down loop

Peak adjustment check:

- `down = 2`, `peak = 3`
- `down ≤ peak`, no adjustment needed

i = 7: ratings[7] = 2, ratings[6] = 2

- Equal ratings, so add 1 candy and continue
- `total_candy += 1` → `total_candy = 16 + 1 = 17`
- `i += 1` → `i = 8`

i = 8: ratings[8] = 1, ratings[7] = 2

- `1 < 2`, so we enter the **descending (down) phase**
- `peak = 1`
- `down = 1`

Down phase loop:

- **i = 8**: `ratings[8] < ratings[7]` (1 < 2)
  - `total_candy += down` → `total_candy = 17 + 1 = 18`
  - `down += 1` → `down = 2`
  - `i += 1` → `i = 9`
- **i = 9**: `i >= n`, exit loop

Peak adjustment check:

- `down = 2`, `peak = 1`
- `down > peak`, so adjust: `total_candy += down - peak` → `total_candy = 18 + (2-1) = 19`

Final Result

- **Total candies needed:** `19`
- **Candy distribution:** `[4, 3, 2, 1, 2, 3, 1, 2, 1]`

The algorithm efficiently handles the ascending and descending sequences, ensuring each child gets the minimum required candies while satisfying the rating constraints.

Greatest Peak

The `down > peak` adjustment is crucial for handling cases where the descending sequence is longer than the ascending sequence. Let me illustrate this with your example `[1,2,3,4,5,4,3,2,1,0,1,2]`.

The Problem

When we have a peak followed by a long descending sequence, the peak element needs enough candies to satisfy **both constraints**:

1. More candies than the element before it (from ascending)
2. More candies than the element after it (from descending)

Step-by-Step with [1,2,3,4,5,4,3,2,1,0,1,2]

Initial Setup

- `total_candy = 1` (first element gets 1 candy)
- `i = 1`

Ascending Phase (indices 1-4)

```
i=1: ratings[1]=2 > ratings[0]=1, peak=2, total_candy = 1+2 = 3
i=2: ratings[2]=3 > ratings[1]=2, peak=3, total_candy = 3+3 = 6
i=3: ratings[3]=4 > ratings[2]=3, peak=4, total_candy = 6+4 = 10
i=4: ratings[4]=5 > ratings[3]=4, peak=5, total_candy = 10+5 = 15
```

At this point: `peak = 5` and our distribution looks like:

```
Ratings: [1, 2, 3, 4, 5, ?, ?, ?, ?, ?, ?, ?]
Candies: [1, 2, 3, 4, 5, ?, ?, ?, ?, ?, ?, ?]
```

Descending Phase (indices 5-9)

```
i=5: ratings[5]=4 < ratings[4]=5, down=1, total_candy = 15+1 = 16
i=6: ratings[6]=3 < ratings[5]=4, down=2, total_candy = 16+2 = 18
i=7: ratings[7]=2 < ratings[6]=3, down=3, total_candy = 18+3 = 21
```

```
i=8: ratings[8]=1 < ratings[7]=2, down=4, total_candy = 21+4 = 25
i=9: ratings[9]=0 < ratings[8]=1, down=5, total_candy = 25+5 = 30
```

At this point: `down = 6` (after incrementing), and our **tentative** distribution:

```
Ratings: [1, 2, 3, 4, 5, 4, 3, 2, 1, 0, ?, ?]
Candies: [1, 2, 3, 4, 5, 1, 2, 3, 4, 5, ?, ?]
```

The Problem is Here!

Look at index 4 (rating=5):

- It has 5 candies
- The element after it (index 5, rating=4) has 1 candy ✓
- The element before it (index 3, rating=4) has 4 candies ✓

But wait! In our descending sequence, we assigned candies as `[1,2,3,4,5]` to indices `[5,6,7,8,9]`. This means the peak element (index 4) only has 5 candies, but the last element in the descending sequence (index 9) has 5 candies too!

The **real constraint** is that each element in the descending sequence must have 1 fewer candy than the previous element, starting from the peak. Since our descending sequence has length 5, we need:

- Index 9: 1 candy
- Index 8: 2 candies
- Index 7: 3 candies
- Index 6: 4 candies
- Index 5: 5 candies
- **Index 4 (peak): must have 6 candies!** (more than index 5's 5 candies)

The Fix: `down > peak` Adjustment

- `down = 6` (length of descending sequence + 1)

- `peak = 5` (candies assigned to peak during ascending phase)
- Since `down > peak`, we need: `total_candy += down - peak = 6 - 5 = 1`

This adds 1 extra candy to account for the peak needing 6 candies instead of 5.

Corrected Distribution

```
Ratings: [1, 2, 3, 4, 5, 4, 3, 2, 1, 0, ?, ?]
Candies: [1, 2, 3, 4, 6, 5, 4, 3, 2, 1, ?, ?]
```

Continuing with the Rest

```
i=10: ratings[10]=1 > ratings[9]=0, peak=2, total_candy = 31+2 = 33
i=11: ratings[11]=2 > ratings[10]=1, peak=3, total_candy = 33+3 = 36
```

Final Result

```
Ratings: [1, 2, 3, 4, 5, 4, 3, 2, 1, 0, 1, 2]
Candies: [1, 2, 3, 4, 6, 5, 4, 3, 2, 1, 2, 3]
Total: 36 candies
```

The key insight: The `down - peak` adjustment ensures the peak element has enough candies to be greater than both its neighbors when the descending sequence is longer than the ascending sequence.