香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

T26G3 - Annoying Advertisement II

Isaac Wong {WongChun1234}

2026-03-28

Background

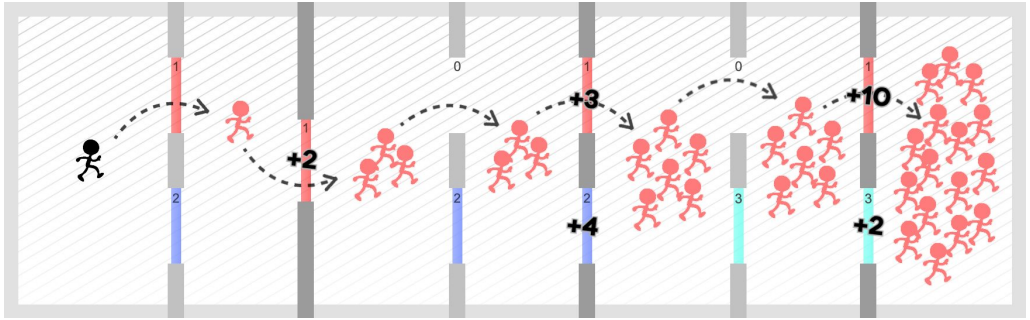
Problem Idea by `WongChun1234`

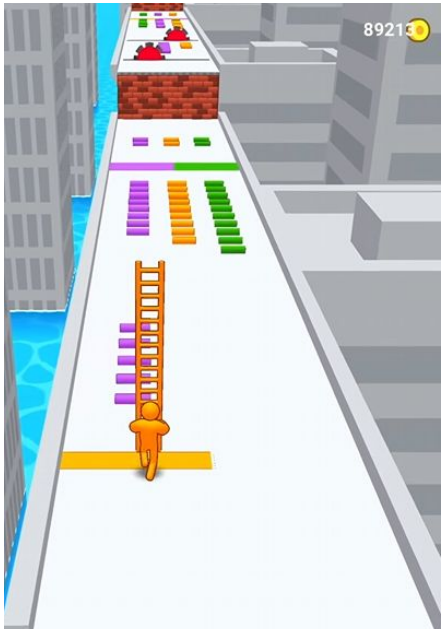
Preparation by `ITO` and `mlwong`

Illustrations by `bedrockfake`(statement) and `WongChun1234`(editorial)

Problem Statement

- Input gives you **N** rounds of gates.
- In odd rounds (coloring rounds), you can recolor all runners (if $A_{i,j} \neq 0$) or keep their color unchanged (if $A_{i,j} = 0$).
- In even rounds (reward rounds), you get $X_{i,j}$ runners if their color is currently $B_{i,j}$
- Maximize the number of runners in the end

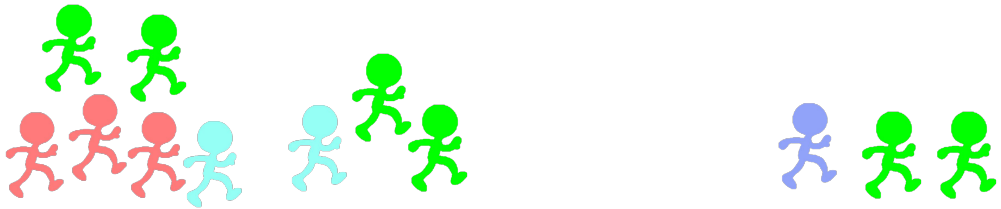




[Inspiration]

Statistics

???



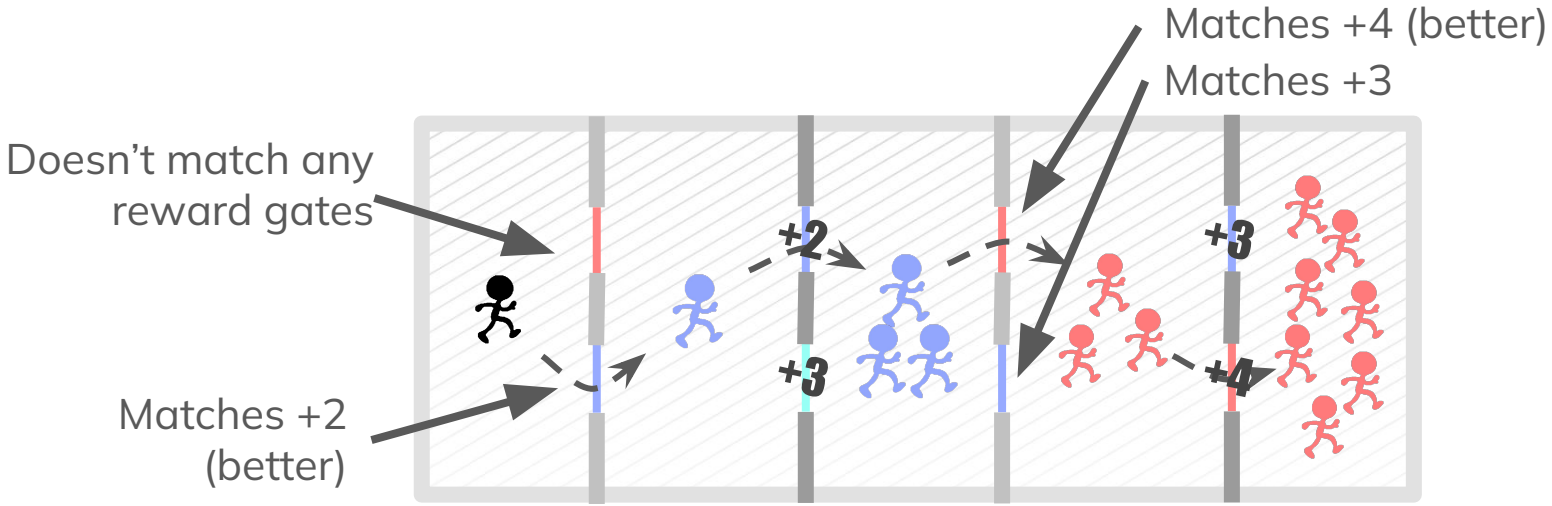
Subtask	1	2	3	4	5

Statistics

???

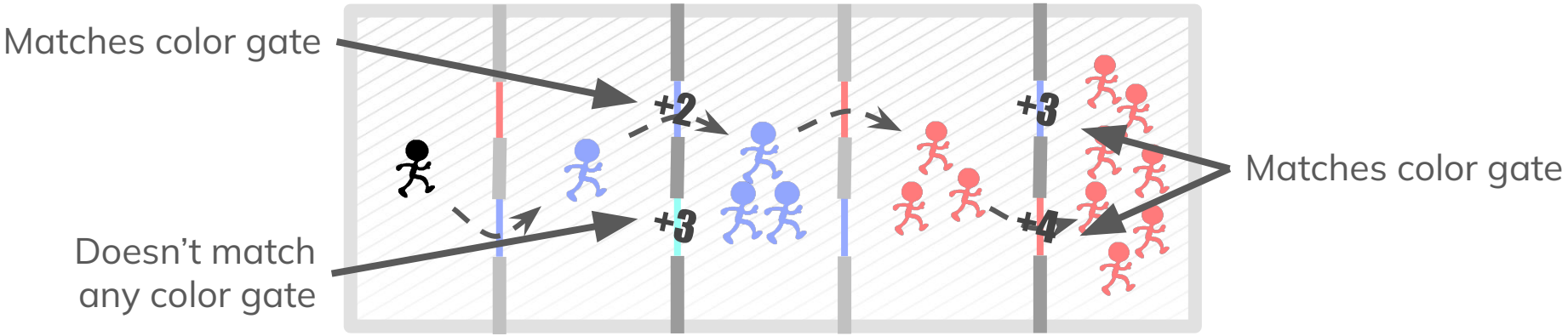
Subtask 1 (19%): $A_{i,j} \neq 0, \sum C_i \leq 1000$

- You must recolor your runners to one of the $A_{i,j}$ in each coloring round
- For each possible $A_{i,j}$ in round i, check if color $A_{i,j}$ exist in the next reward round as $B_{i+1,k}$, and select the maximum $X_{i+1,k}$



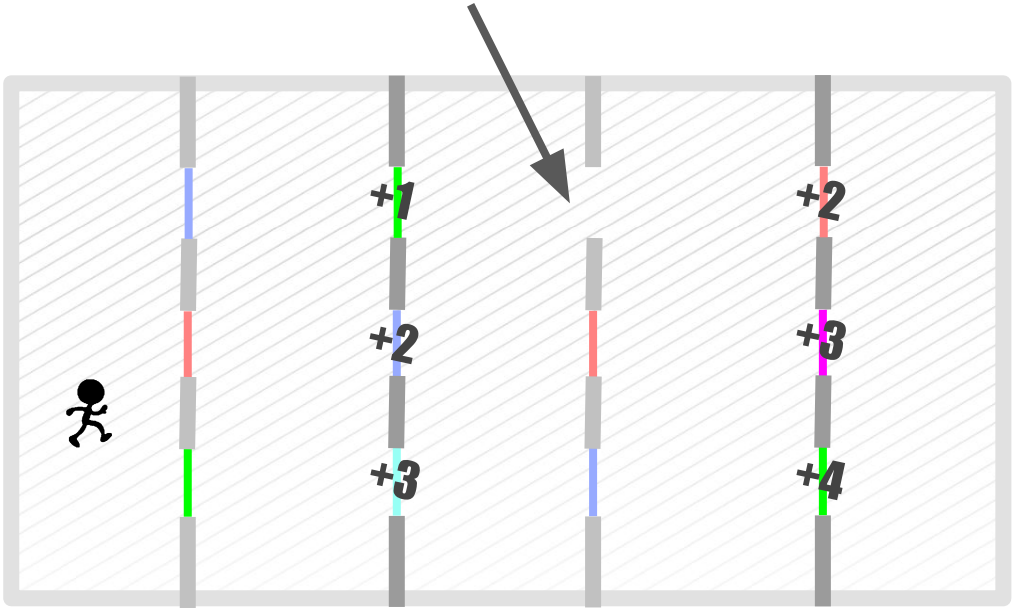
Subtask 2 (12%): $A_{i,j} \neq 0$

- You must recolor your runners to one of the $A_{i,j}$ in each coloring round
- For each possible $A_{i,j}$ in round i, check if color $A_{i,j}$ exist in the next reward round as $B_{i+1,k}$, and select the maximum $X_{i+1,k}$ Slowest part!
- Use binary search to check instead
- Note: looping through $B_{i,j}$ is easier to implement as you already know $X_{i,j}$.



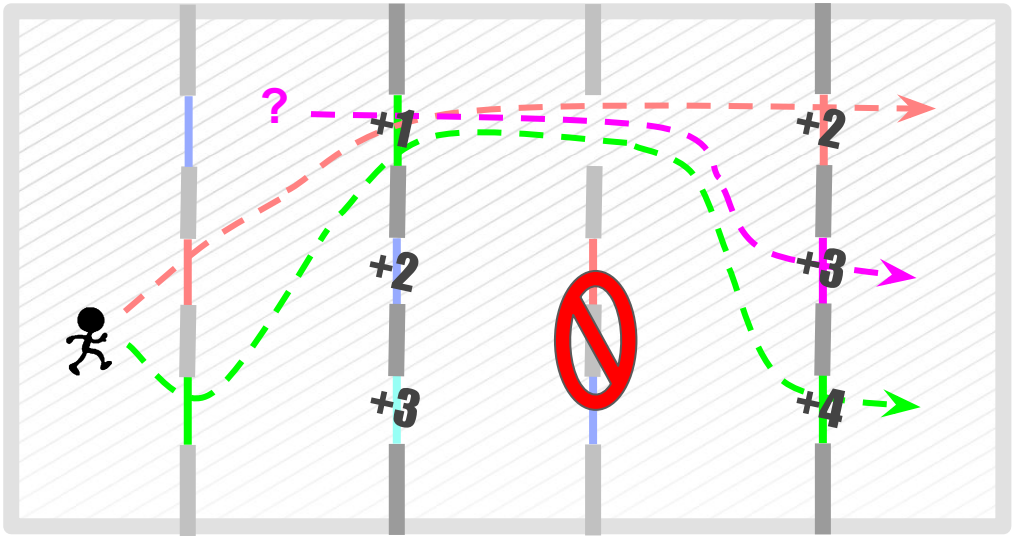
Subtask 3 (20%): $A_{i,j} \neq 0$ for all $i \equiv 1 \bmod 4$

- The game looks like this
- Two cases to handle: color the runners or not



Subtask 3 (20%): $A_{i,j} \neq 0$ for all $i \equiv 1 \bmod 4$

- Case 1: we use the uncolored gate
- We check if the color exist in the previous coloring round and reward round



Total: +2

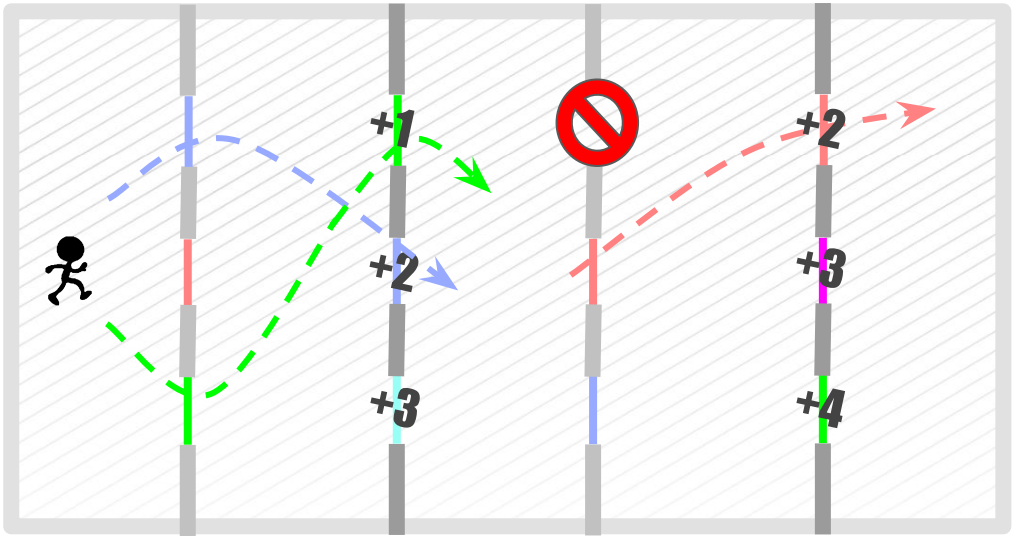
Invalid color

Total: +5

Subtask 3 (20%): $A_{i,j} \neq 0$ for all $i \equiv 1 \bmod 4$

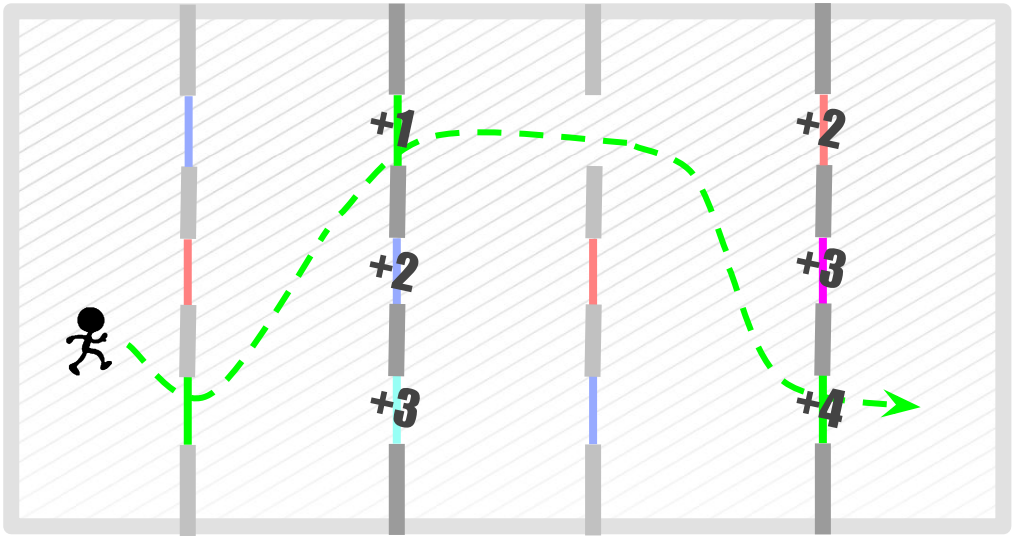
- Case 2: we use some colored gate
- The two consecutive reward rounds are independent of each other

Best: +2 Best: +2 Total best: +4



Subtask 3 (20%): $A_{i,j} \neq 0$ for all $i \equiv 1 \bmod 4$

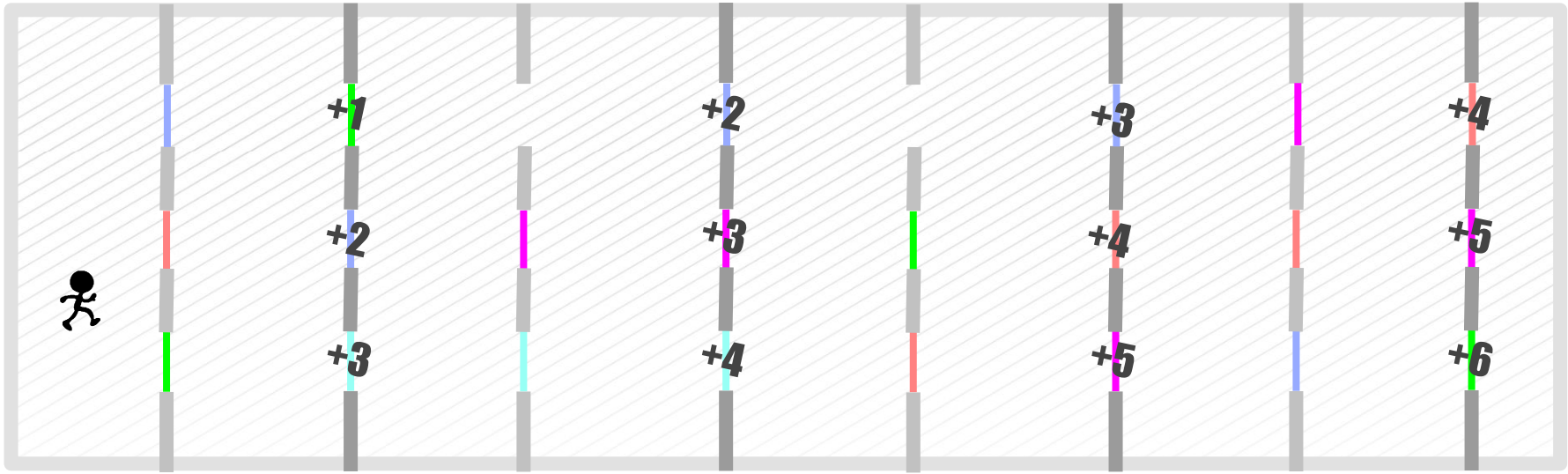
- Choose the best out of two cases for every four consecutive rounds



Total: +5

Subtask 4 (26%): K ≤ 10

- There may be a lot of consecutive uncolor gates
- However, there are at most 10 possible colors
- Keep track of every possible color at each round

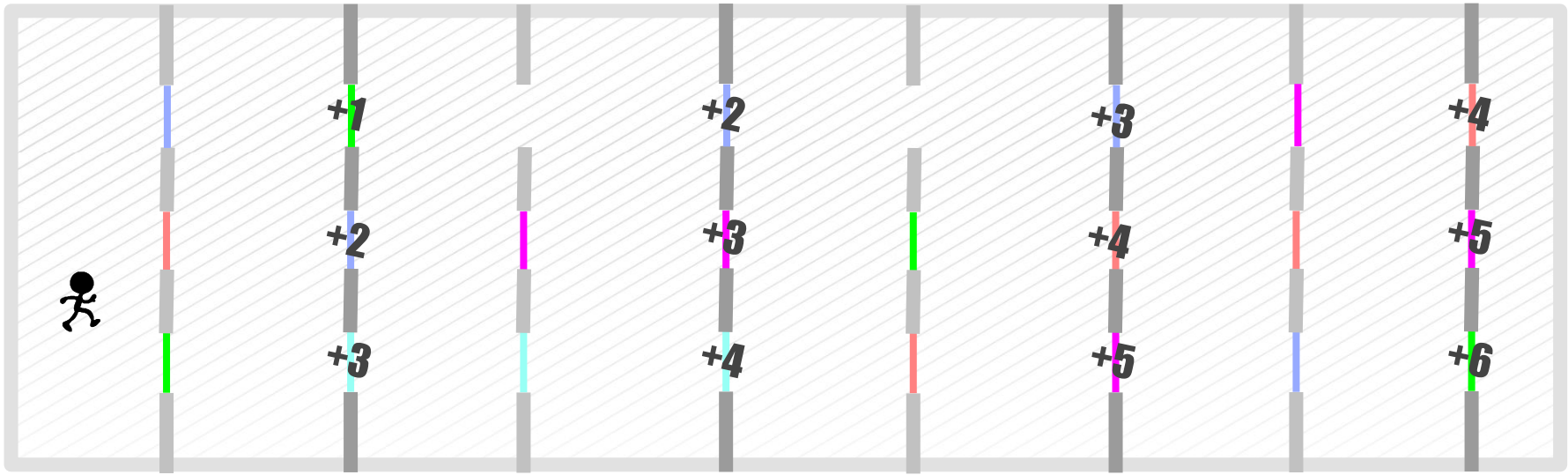


Subtask 4 (26%): K ≤ 10

Reset when no $X_{i,j} = 0$

Do not repeat!

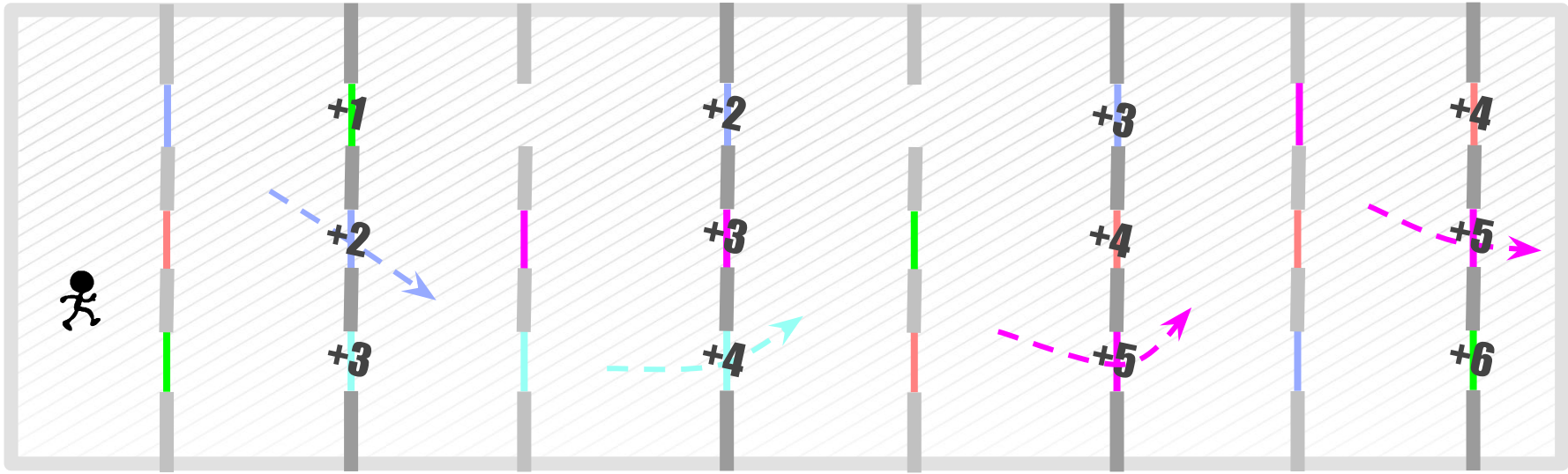
Possible colors:



Subtask 4 (26%): K ≤ 10

- Can we just choose the best possible color in each reward round?

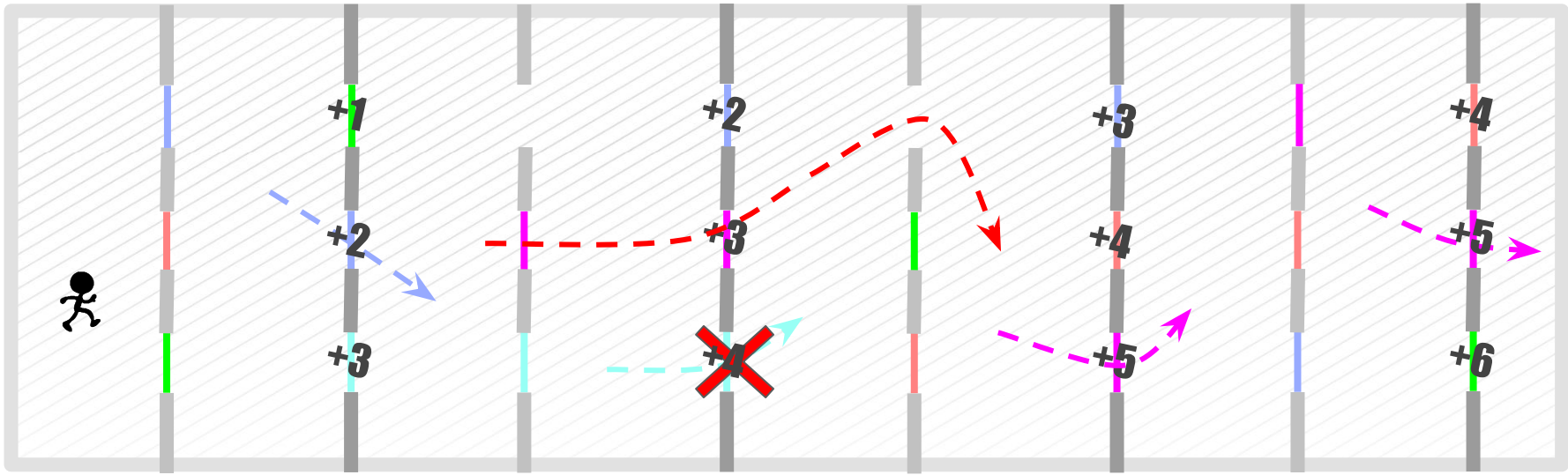
Possible colors:



Subtask 4 (26%): K ≤ 10

- Can we just choose the best possible color in each reward round? NO!
- We must choose purple the color round before

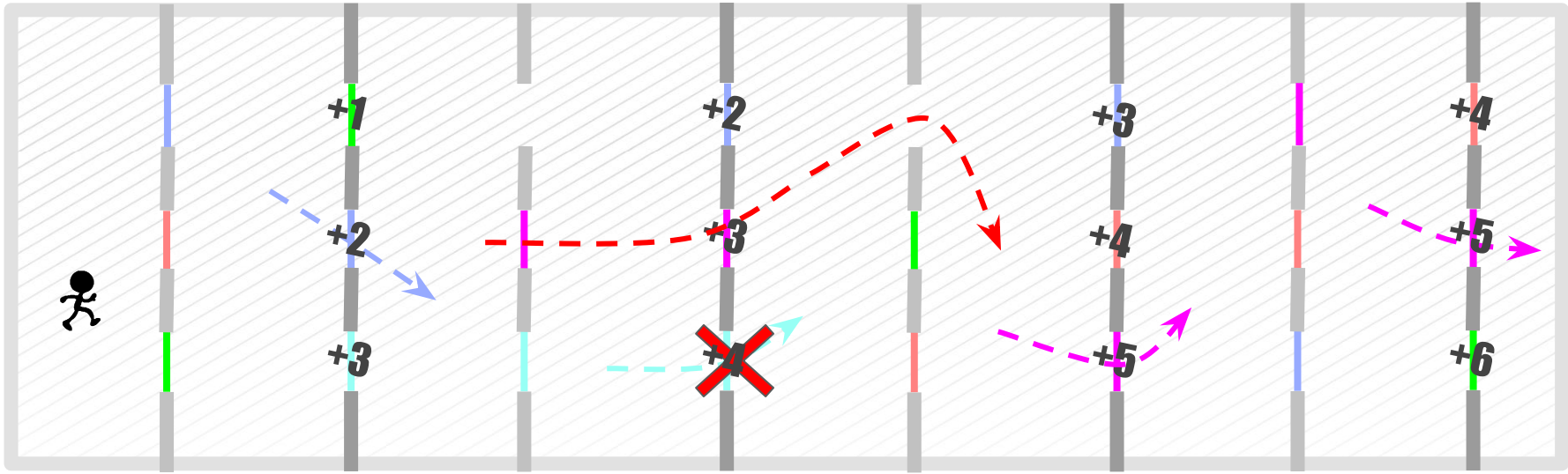
Possible colors:



Subtask 4 (26%): K ≤ 10

- When choosing local optimal does not lead to global optimal → use DP!
- Keep track of the maximum possible number of runners for each color

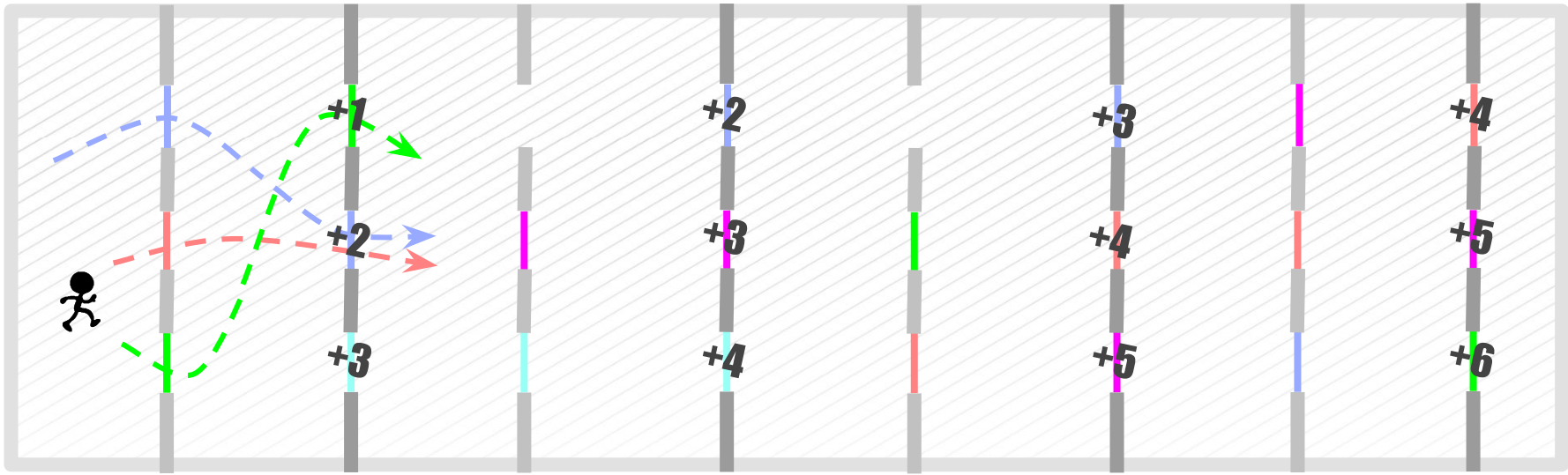
Possible colors:



Subtask 4 (26%): K ≤ 10

- You start with 1 runner

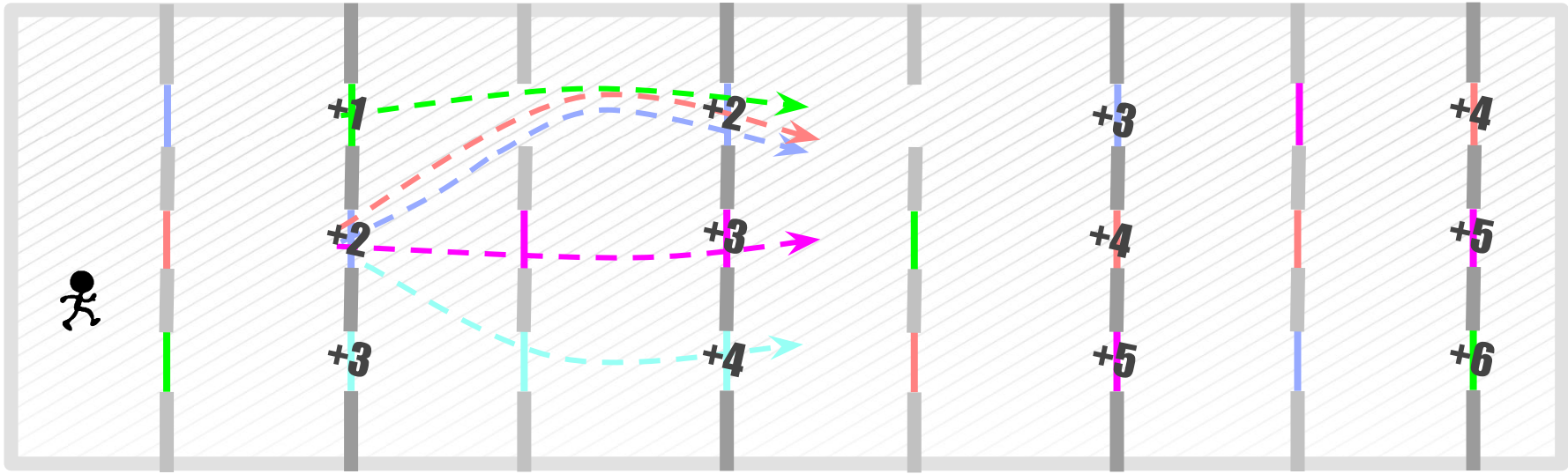
Possible colors: 3 1 2



Subtask 4 (26%): K ≤ 10

- For colors with coloring gate this round, we can transition from the best color

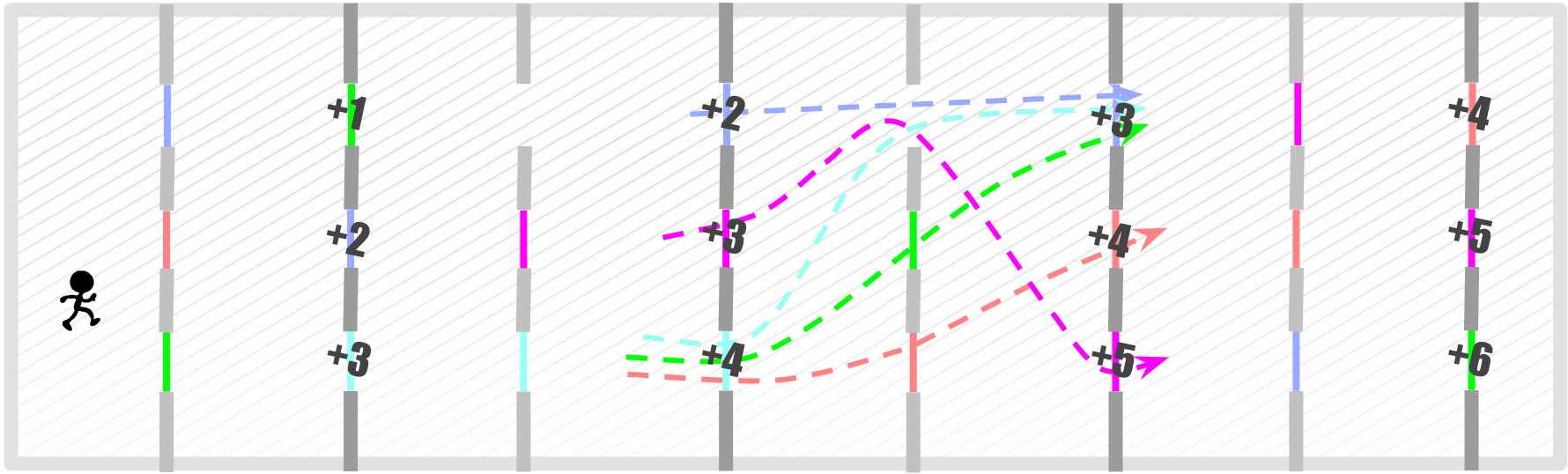
Possible colors: 3 1 2 5 1 2 6 7



Subtask 4 (26%): K ≤ 10

- For colors without coloring gate this round, we must transition from the same color

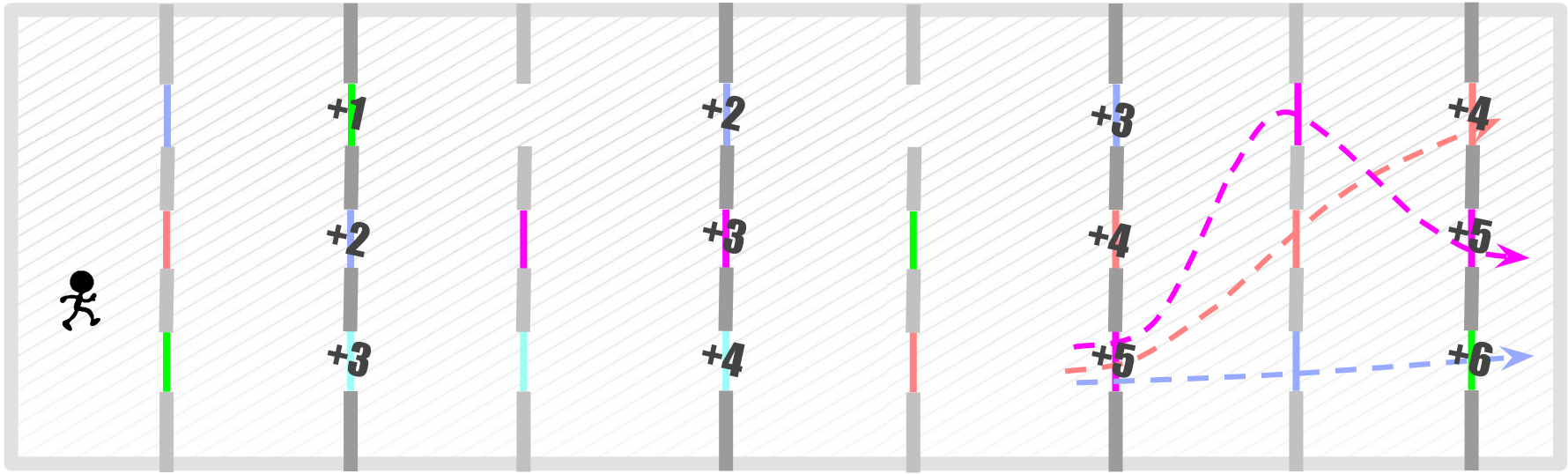
Possible colors: 3 1 2 | 5 1 2 6 7 | 8 11 7 11 7 |



Subtask 4 (26%): K ≤ 10

- When there are no uncolored gate, reset all invalid colors

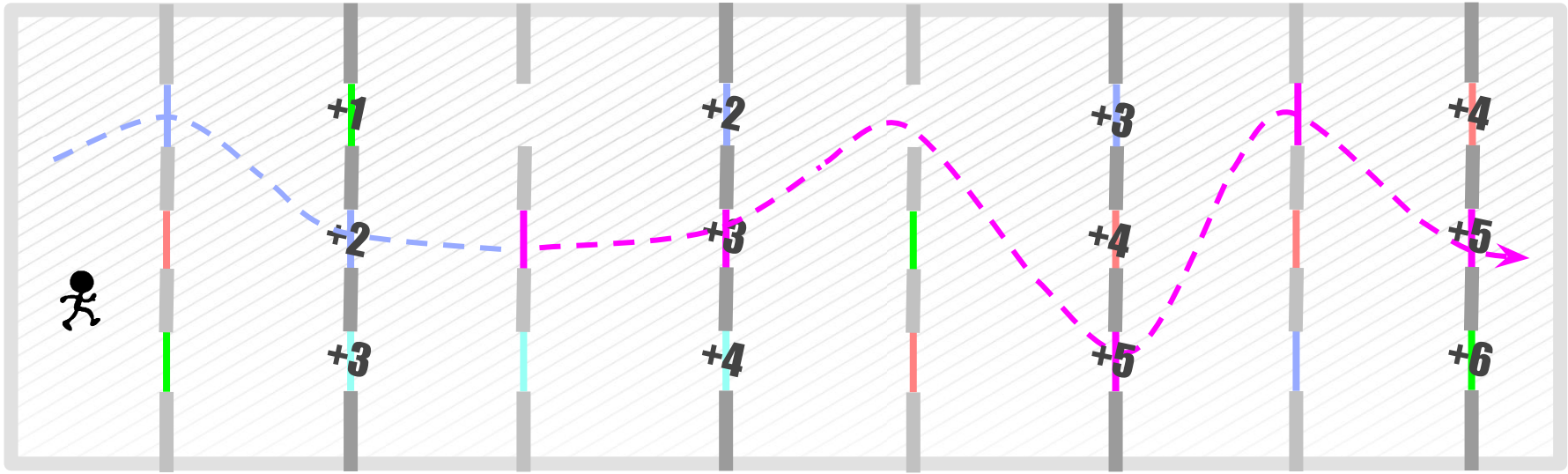
Possible colors: 3 1 2 | 5 1 2 6 7 | 8 11 7 11 7 | 16 15 11



Subtask 4 (26%): K ≤ 10

- Answer = max of all values in the end

Possible colors: 3 1 2 | 5 1 2 6 7 | 8 11 7 11 7 | 16 15 11



Subtask 4 (26%): K ≤ 10

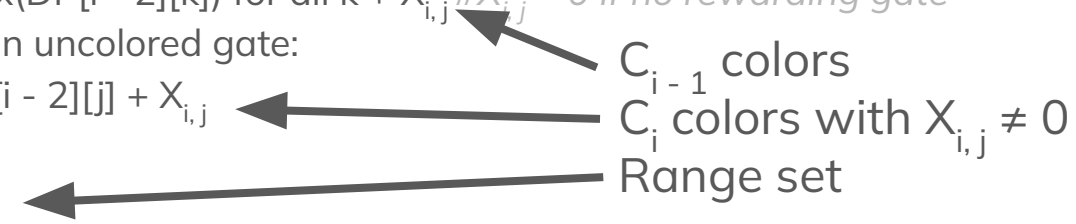
- Pseudocode:

- For i ← 2 to N: *#all reward rounds*
  - For j ← 1 to K: *#all colors*
    - If color exist in previous coloring round (round i - 1):
      - DP[i][j] = max(DP[i - 2][k]) for all k + $X_{i, j}$ *#$X_{i, j}$ = 0 if no rewarding gate*
    - Else if there exist an uncolored gate:
      - DP[i][j] = DP[i - 2][j] + $X_{i, j}$
    - Else:
      - DP[i][j] = -inf

Subtask 5 (23%): No additional constraints

- How to optimize this code?

- For i ← 2 to N: *#all reward rounds*
  - For j ← 1 to K: *#all colors*
    - If color exist in previous coloring round (round i - 1):
      - DP[i][j] = max(DP[i - 2][k]) for all k + $X_{i,j}$ *#$X_{i,j}$ = 0 if no rewarding gate*
    - Else if there exist an uncolored gate:
      - DP[i][j] = DP[i - 2][j] + $X_{i,j}$
    - Else:
      - DP[i][j] = -inf



Subtask 5 (23%): No additional constraints

- Notice that Max(DP[i - 2][k]) is always increasing
- We can just use an extra variable to keep track of it
- Initialize it to 1 instead of -inf! (You start with 1 runner)
- For i ← 2 to N: *#all reward rounds*
  - For j ← 1 to K: *#all colors*
    - If color exist in previous coloring round (round i - 1):
      - DP[i][j] = prevmax + $X_{i,j}$ *#$X_{i,j}$ = 0 if no rewarding gate*
    - Else if there exist an uncolored gate:
      - DP[i][j] = DP[i - 2][j] + $X_{i,j}$
    - Else:
      - DP[i][j] = -inf
  - For j ← 1 to K:
    - prevmax = max(prevmax, DP[i][j])

Subtask 5 (23%): No additional constraints

- Additionally, we can separate processing coloring rounds and reward rounds
- In coloring rounds, we update the set of possible colors, and also transition from the best color. We also remove the first dimension in DP array to reuse it
- For i ← 1 to N: *#all rounds*
  - If i is a coloring round:
    - If there doesn't exist an uncolored gate:
      - Reset whole DP to -inf
    - For j in A_i:
      - DP[j] = prevmax
  - Else:
    - ...

- For colors with coloring gate this round, we can transition from the best color
- For colors without coloring gate this round, we must transition from the same color → Do nothing!

Subtask 5 (23%): No additional constraints

- In reward rounds, we just update the C_i colors with their respective $X_{i, j}$

- For i ← 1 to N: *#all rounds*
  - If i is a coloring round:
    - If there doesn't exist an uncolored gate:
      - Reset whole DP to -inf
    - For j in A_i:
      - DP[j] = prevmax
  - Else:
    - For j ← 1 to C_i:
      - DP[$B_{i, j}$] += $X_{i, j}$
      - prevmax = max(prevmax, DP[$B_{i, j}$])

Subtask 5 (23%): No additional constraints

- Last question: how to reset the whole DP array?
- Use a map! map.clear() → reset whole array
- When the index is not found as a key in the map, the DP value is -inf
- For i ← 1 to N: *#all rounds*
  - If i is a coloring round:
    - If there doesn't exist an uncolored gate:
      - map.clear()
    - For j in A_i:
      - map[j] = prevmax if j is found in map
  - Else:
    - For j ← 1 to C_i:
      - map[$B_{i,j}$] += $X_{i,j}$ if $B_{i,j}$ is found in map
      - prevmax = max(prevmax, DP[$B_{i,j}$]) if $B_{i,j}$ is found in map

Takeaways

- Standard techniques are still very useful
- $K \leq 10$ requires understanding of base cases and transitions in DP

- Sometimes, we need to extend more from standard techniques
- Reuse DP array to avoid unnecessary transitions
- Using set to optimize “reset” action

- You must have a solid understanding in the “standard” version before extending it

Appendix: On std::unordered_map

- Note that in the solution, we just need a key-value lookup dictionary, the order of keys does not matter. **Why don't we use `std::unordered_map` (O(1) lookup) instead of `std::map` (O(log N) lookup)?**

T26G3 - Annoying Advertisements II	C++17	Time Limit Ex	
T26G3 - Annoying Advertisements II	C++17	Accepted	0.516

`std::unordered_map`

`std::map`

Why?

Appendix: On std::unordered_map

- `unordered_map` store elements in different buckets.
- Same hash => Same bucket

- It will always ensure there are enough bucket such that the current `load_factor()` <= `max_load_factor()` (load factor = #elements / #buckets)
- Adding a lot of elements will trigger `std::unordered_map::rehash()`, which would increase bucket count (usually double the number, compiler dependent)

Appendix: On std::unordered_map

- `unordered_map::clear()` operates in linear time of **number of elements**
  - Complexity requirements is on container elements only, did not specify how fast the bucket managements need to be.
- `clear()` do not clear the buckets, only the elements
  - Some implementation will scan through all the buckets in each invoke of clear()
  - This happen even if there is 0 elements in the unordered_map.
- Hence, let say you have a lot of elements in unordered_map in the beginning
- Keep calling clear() is very costly

- Reference: https://stackoverflow.com/questions/63710308/what-does-stdunordered-mapclear-do

Appendix: On std::unordered_map

- Given all of these knowledge, can you still save your solution with `std::unordered_map`?
- Yes, by forcing rehash after every clear.

```
mp.clear();
mp.rehash(0);
```

T26G3 - Annoying Advertisements II	C++17	Accepted	0.284	`std::unordered_map + rehash`
T26G3 - Annoying Advertisements II	C++17	Time Limit Ex		`std::unordered_map`
T26G3 - Annoying Advertisements II	C++17	Accepted	0.516	`std::map`

- Note that there are still another solution by simply using an array as the lookup dictionary, and maintain a list of in-used indices for doing clear.
  - You should be aware of the tradeoffs between different approaches, and also have some knowledge about the inner workings of them to use them effectively