<https://leetcode.com/problems/race-car/description/>

Your car starts at position 0 and speed +1 on an infinite number line. Your car can go into negative positions. Your car drives automatically according to a sequence of instructions 'A' (accelerate) and 'R' (reverse):

When you get an instruction 'A', your car does the following:

position += speed

speed \*= 2

When you get an instruction 'R', your car does the following:

If your speed is positive then speed = -1

otherwise speed = 1

Your position stays the same.

For example, after commands "AAR", your car goes to positions 0 --> 1 --> 3 --> 3, and your speed goes to 1 --> 2 --> 4 --> -1.

Given a target position target, return the length of the shortest sequence of instructions to get there.

**Example 1:**

**Input:** target = 3

**Output:** 2

**Explanation:** The shortest instruction sequence is "AA".Your position goes from 0 --> 1 --> 3.

**Example 2:**

**Input:** target = 6

**Output:** 5

**Explanation:** The shortest instruction sequence is "AAARA".Your position goes from 0 --> 1 --> 3 --> 7 --> 7 --> 6.

**Constraints:**

1 <= target <= 10^4

**Attempt 1: 2024-05-23**

**Solution 1: BFS + Level Order Traversal (30 min)**

class Solution {

    public int racecar(int target) {

        Set<String> visited = new HashSet<>();

        Queue<Node> q = new LinkedList<>();

        q.offer(new Node(1, 0));

        int distance = 0;

        while(!q.isEmpty()) {

            int size = q.size();

            for(int i = 0; i < size; i++) {

                Node cur = q.poll();

                if(cur.position == target) {

                    return distance;

                }

                // A major prune of BFS is "nextPosition > 0 && nextPosition < 2 \* target"

// otherwise it will be TLE, 49/55, target = 5478

                // If 'A'

                int nextPosition = cur.position + cur.speed;

                int nextSpeed = cur.speed \* 2;

                if(nextPosition > 0 && nextPosition < 2 \* target

                && !visited.contains(nextSpeed + "," + nextPosition)) {

                    visited.add(nextSpeed + "," + nextPosition);

                    q.offer(new Node(nextSpeed, nextPosition));

                }

                // If 'R'

                nextPosition = cur.position;

                nextSpeed = cur.speed > 0 ? -1 : 1;

                if(nextPosition > 0 && nextPosition < 2 \* target

                && !visited.contains(nextSpeed + "," + nextPosition)) {

                    visited.add(nextSpeed + "," + nextPosition);

                    q.offer(new Node(nextSpeed, nextPosition));

                }

            }

            distance++;

        }

        return -1;

    }

    class Node {

        int speed;

        int position;

        public Node(int speed, int position) {

            this.speed = speed;

            this.position = position;

        }

    }

}

Time Complexity: O()

Space Complexity: O()

**Refer to**

<https://leetcode.com/problems/race-car/solutions/142360/logical-thinking-with-clean-java-code/>

**Logical Thinking**

Let's define the **State Machine** first.

**State** is represented by state[x] as the shortest instruction sequence to x

**End(Aim) State**: state[target]

**Transition Function**: min(state[canReachX]) + 1 = state[x]

However, it is not easy to get all candidate positions for canReachX.

So we utilize **BFS** template, which is used to get the **Shortest Distance**.

Please note that we create a class StateNode to encapsulate speed and position attributes .

And we serialize StateNode as speed,position, which is used in Set<String> visited to help check if StateNode has been visited during the **BFS**.

**Clean Java Code**

public int racecar(int target) {

Set<String> visited = new HashSet<>();

Queue<StateNode> queue = new LinkedList<>();

queue.add(new StateNode(1, 0));

int distance = 0;

while (!queue.isEmpty()) {

int levelSize = queue.size();

for (int i = 0; i < levelSize; i++) {

StateNode cur = queue.poll();

if (cur.position == target) {

return distance;

}

// if A

int nextPosition = cur.position + cur.speed;

int nextSpeed = cur.speed \* 2;

if (!visited.contains(nextSpeed + "," + nextPosition) && Math.abs(target - nextPosition) < target) {

visited.add(nextSpeed + "," + nextPosition);

queue.offer(new StateNode(nextSpeed, nextPosition));

}

// if R

nextPosition = cur.position;

nextSpeed = cur.speed > 0 ? -1 : 1;

if (!visited.contains(nextSpeed + "," + nextPosition) && Math.abs(target - nextPosition) < target) {

visited.add(nextSpeed + "," + nextPosition);

queue.offer(new StateNode(nextSpeed, nextPosition));

}

}

distance++;

}

return -1;

}

class StateNode {

int speed;

int position;

public StateNode(int speed, int position) {

this.speed = speed;

this.position = position;

}

}

Time Complexity: O(targetlog(target))

Refer to

https://leetcode.com/problems/race-car/solutions/124326/summary-of-the-bfs-and-dp-solutions-with-intuitive-explanation/comments/215841

The BFS is memorizing pairs of speed and positions. So the total time complexity will be the number

of such pairs formed before we hit the target.

Now, How many different positions are possible to reach to target. In worst case we, could have

visited every single position in both the directions i.e. from -target to +target so the total

no of positions possible are O(target).

Now, to check total no of speeds possible we could go either 1, 2, 4, 8.. or -1, -2, -4 , -8 .. ..

upto (target). because we will always reach the target with speed bounded by target position.

So total no of distinct speeds are 2 \* O(log target)

Hence time complexity = Total distinct positions(=target) \* total distinct speeds (=log(target)) = O(targetlog(target)).

**How prune "nextPosition > 0 && nextPosition < 2 \* target" works ?**

<https://leetcode.com/problems/race-car/solutions/124326/summary-of-the-bfs-and-dp-solutions-with-intuitive-explanation/comments/139536>

There are cases when it is desirable to go past the target and then come back.

For example, target = 6, we could go 0 --> 1 --> 3 --> 7 --> 7 --> 6, which takes 5 instructions (AAARA). If you reverse before passing the target, it takes more than 5 instructions to get to the target. That said, we don't want to go too far past the target. The rule of thumb (I have not proved it rigorously though) is that we stay within some limit from the target, and this limit is set by the initial distance from the target, which is target. So from the point of view of the target, we want the car to stay in the range [0, 2 \* target]. This is the primary optimization for the BFS solution.

<https://leetcode.com/problems/race-car/solutions/142360/logical-thinking-with-clean-java-code/comments/1364579>

The point is, once you overshoot target, you should immediately turn around regardless of how far you have overshot. if you are between init\_pos and target, your acceleration cannot take you further than 2 \* target. So that is the upper bound of how far you can get, if you are somewhere between 0 and target. However, the missing part that hasn't been proven is that once you shoot past target, you should not continue going any further away from target. In your Case 2 if nextPos is where you end up after shooting past target, you should not accelerate and start reversing immediately. However, there are cases where you are past target and do want to accelerate further away, but only if it is part of your optimal path to get from your initial overjump position back to target. You should never end up further past target from your initial overjump position. Every acceleration is a power of two, so if you are before the target and jump past it, you are now some (2^n) - k away from target, where k < 2^n. If you keep accelerating past target, you will be some (2^n) - k + (2^(n+1)) away from target. So what should be proven is racecar((2^n) - k) <= racecar((2^n)-k+(2^(n+1))) + 1(+1 step for the extra acceleration). From trial and error this seems true but I haven't formally proven it yet. This equation combined with allenh explanation would prove the upper bound of target\*2.

**Summary of the BFS and DP solutions with intuitive explanation**

**Refer to**

<https://leetcode.com/problems/race-car/solutions/124326/summary-of-the-bfs-and-dp-solutions-with-intuitive-explanation/>

**I -- BFS solution**

Well, the BFS solution is straightforward: we can keep track of all the possible positions of the racecar after n instructions (n = 0, 1, 2, 3, 4, ...) and return the smallest n such that the target position is reached. Naive BFS will run at O(2^n) since for each position we have two choices: either accelerate or reverse. Further observations reveal that there may be overlapping among intermediate states so we need to memorize visited states (each state is characterized by two integers: car position and car speed). However, the total number of unique states still blows up for large target positions (because the position and speed can grow unbounded), so we need further pruning of the search space.

**II -- DP solution**

DP solution works by noting that after each reverse, the car's speed returns to 1 (the sign can be interpreted as the direction of the speed). So we can redefine the problem in terms of the position of the car while leave out the speed: let T(i) be the length of the shortest instructions to move the car from position 0 to position i, with initail speed of 1 and its direction pointing towards position i. Then our original problem will be T(target), and the base case is T(0) = 0. Next we need to figure out the recurrence relations for T(i).

Note that to apply the definition of T(i) to subproblems, the car has to start with speed of 1, which implies we can only apply T(i) right after the reverse instruction. Also we need to make sure the direction of the initial speed when applying T(i) is pointing towards the final target position.

However, we don't really know how many accelerate instructions there should be before the reverse instruction, so theoretically we need to try all possible cases: zero A, one A, two A, three A, ... and so on. For each case, we can obtain the position of the car right before the reverse instruction, which will be denoted as j = 2^m - 1, with m the number of A's. Then depending on the relation between i and j, there will be three cases:

j < i: the reverse instruction is issued before the car reaches i. In this case, we cannot apply the definition of T(i) to the subproblems directly, because even though the speed of the car returns to 1, its direction is pointing away from the target position (in this case position i). So we have to wait until the second reverse instruction is issued. Again, we don't really know how many accelerate instructions there should be in between these two reverse instructions, so we will try each of the cases: zero A, one A, two A, three A, ..., etc. Assume the number of A is q, then the car will end up at position j - p right before the second reverse instruction, where p = 2^q - 1. Then after the second reverse instruction, our car will start from position j - p with speed of 1 and its direction pointing towards our target position i. Since we want the length of the total instruction sequence to be minimized, we certainly wish to use minimum number of instructions to move the car from j - p to i, which by definition will be given by T(i-(j-p)) (note that in the definition of T(i), we move the car from position 0 to position i. If the start position is not 0, we need to shift both the start and target positions so that the start position is aligned with 0). So in summary, for this case, the total length of the instruction will be given by: m + 1 + q + 1 + T(i-(j-p)), where m is the number of A before the first R, q is the number of A before the second R, the two 1's correspond to the two R's, and lastly T(i-(j-p)) is the length of instructions moving the car from position j - p to the target position i.

j == i: the target position is reached without any reverse instructions. For this case, the total length of the instruction will be given by m.

j > i: the reverse instruction is issued after the car goes beyond i. In this case, we don't need to wait for a second reverse instruction, because after the first reverse instruction, the car's speed returns to 1 and its direction will be pointing towards our target position i. So we can apply the definition of T(i) directly to the subproblem, which will be T(j-i). Note that not only do we need to shift the start and target positions, but also need to swap them as well as the directions. So for this case, the total length of the instructions will be given by m + 1 + T(j-i).

Our final answer for T(i) will be the minimum of the above three cases.

**III -- Intuitive explanation of the optimizations**

As I mentioned in section I, we need further optimizations for the BFS solution to work efficiently. This turns out also to be the case for the DP solution. To see why, recall that in the first case of the DP solution, we don't really impose any upper limit on the value of q (we do have limit for the value of m though: j = 2^m-1 < i), while in the third case, we don't really have any upper limit for the value of m. Apparently we cannot explore every possible values of m and q (there are infinitely many).

to be updated...

**IV -- Solutions**

Here is a list of solutions: one BFS, one top-down DP and one bottom-up DP.

The BFS runs at O(target \* log(target)) in the worst case, with O(target \* log(target)) space. The reasoning is as follows: in the worst case, all positions in the range [-target, target] will be visited and for each position there can be as many as 2 \* log(target) different speeds.

Both the top-down DP and bottom-up DP run at O(target \* (log(target))^2) with O(target) space. However, the top-down DP may be slightly more efficient as it may skip some of the intermediate cases that must be computed explicitly for the bottom-up DP. Though the nominal time complexity are the same, both DP solutions will be much more efficient in practice compared to the BFS solution, which has to deal with (position, speed) pairs and their keys for hashing, etc.

**BFS solution:**

public int racecar(int target) {

Deque<int[]> queue = new LinkedList<>();

queue.offerLast(new int[] {0, 1}); // starts from position 0 with speed 1

Set<String> visited = new HashSet<>();

visited.add(0 + " " + 1);

for (int level = 0; !queue.isEmpty(); level++) {

for(int k = queue.size(); k > 0; k--) {

int[] cur = queue.pollFirst(); // cur[0] is position; cur[1] is speed

if (cur[0] == target) {

return level;

}

int[] nxt = new int[] {cur[0] + cur[1], cur[1] << 1}; // accelerate instruction

String key = (nxt[0] + " " + nxt[1]);

queue.offerLast(nxt);

visited.add(key);

}

nxt = new int[] {cur[0], cur[1] > 0 ? -1 : 1}; // reverse instruction

key = (nxt[0] + " " + nxt[1]);

queue.offerLast(nxt);

visited.add(key);

}

}

}

return -1;

}

**Top-down DP:**

public int racecar(int target) {

int[] dp = new int[target + 1];

Arrays.fill(dp, 1, dp.length, -1);

return racecar(target, dp);

}

private int racecar(int i, int[] dp) {

if (dp[i] >= 0) {

return dp[i];

}

dp[i] = Integer.MAX\_VALUE;

int m = 1, j = 1;

for (; j < i; j = (1 << ++m) - 1) {

for (int q = 0, p = 0; p < j; p = (1 << ++q) - 1) {

dp[i] = Math.min(dp[i], m + 1 + q + 1 + racecar(i - (j - p), dp));

}

}

dp[i] = Math.min(dp[i], m + (i == j ? 0 : 1 + racecar(j - i, dp)));

return dp[i];

}

**Bottom-up DP:**

public int racecar(int target) {

int[] dp = new int[target + 1];

for (int i = 1; i <= target; i++) {

dp[i] = Integer.MAX\_VALUE;

int m = 1, j = 1;

for (; j < i; j = (1 << ++m) - 1) {

for (int q = 0, p = 0; p < j; p = (1 << ++q) - 1) {

dp[i] = Math.min(dp[i], m + 1 + q + 1 + dp[i - (j - p)]);

}

}

dp[i] = Math.min(dp[i], m + (i == j ? 0 : 1 + dp[j - i]));

}

return dp[target];

}