**Overview**

Let's simplify the problem statement.

Given a string of characters, where each character represents a senator. Character R represents a Radiant senator and Character D represents a Dire senator.  
↓\downarrow↓  
A particular party wins if only senators of that party are **eligible** in the senate after all **eligible** senators have exercised their rights. A senator is eligible to exercise their right if they are not banned by a previous senator's action.  
↓\downarrow↓  
Let's label "exercised their rights" as "**Voting**".  
↓\downarrow↓  
**Voting** is done from left to right. After the rightmost senator has voted, the voting process starts again from the leftmost senator, marking the beginning of the next Round. A senator cannot participate in any Round after they have been banned.  
↓\downarrow↓  
Now, this "exercise their right", what we have labeled as "**Voting**", is a power that a senator has. By Voting, a senator can ban another senator restricting them from participating in the next Round.  
↓\downarrow↓  
Thus each senator will optimally use their power to help their party win. We have to assume that each senator is rational, and thus have to predict the Winner, "Radiant" or "Dire".

Now, since the problem description mentions that

Every senator is smart enough and will play the **best strategy** for his own party.

Let's try to find this "*best strategy*".

A senator has to ban another senator to increase the chance of their party winning.

**Will they ever ban a senator from their own party? Click to Reveal.**

↓\downarrow↓

No, they won't. Because at last, they want all eligible senators to be from their own party.

Banning senators from their own party not only reduces the number of eligible senators from their party but also restricts one senator from their own party from banning a senator of the other party.

Thus, it's always better to ban a senator from the other party.

↓\downarrow↓

Hence, we can conclude that a senator will always ban an **opponent** senator. That being said, all **opponents** from the other party are potential targets for a senator to ban. And choosing one over the other can give different results.

For Example, let senate be "DRDRDR".

This set of steps will give "Radiant" as the winner.

1 / 14

This set of steps will give "Dire" as the winner.

1 / 10

Thus different strategies can give different results.

**What then is the "best strategy" for a senator to ban an opponent senator? Click to Reveal.**

↓\downarrow↓

Using this "best strategy" we will be able to predict the only possible unique winner for a given senate string.

Let's try to analyze. What if we ban the next opponent senator? Are we gaining anything?

The answer intuitively is Yes. When we ban the next closest opponent senator, we are restricting one senator from the other party which they could have used to ban a senator from our party. Thus, proceeding with this strategy, we are not only eliminating one opponent senator but also trying to preserve our own senators.

Hence, the "best strategy" for a senator is to ban the next closest opponent senator.

↓\downarrow↓

This falls under the category of **Greedy**, whereby the "best strategy" is greedily chosen at each step in hope of local optimum.

Throughout the article, we will be using NN*N* to denote the number of senators, which is the length of the senate string. Before proceeding further, let's try to figure out the maximum number of rounds that we can have.

**How many rounds can we have? Click to Reveal.**

↓\downarrow↓

There are NN*N* senators, and each senator can ban one opponent senator. Thus, after every round, the number of senators will reduce by N2\frac{N}{2}2*N*​.

Hence, in the first round, we will have NN*N* senators. In the second round, we will have N2\frac{N}{2}2*N*​ senators. In the third round, we will have N22\frac{\frac{N}{2}}{2}22*N*​​ senators. And so on.

This will be repeated until we have only one senator from either party.

Hence after every round, the length of the senate will be halved. Thus, the maximum number of rounds that we can have is log⁡2N\log\_{2}Nlog2​*N*. The analysis is similar to [Binary Search](https://leetcode.com/explore/learn/card/binary-search/).

↓\downarrow↓

What perhaps is more interesting is the maximum number of votes that we can have in total.

**How many votes can we have? Click to Reveal.**

Let there be NN*N* senators. In the first round, there will be at most N2\frac{N}{2}2*N*​ votes. In the second round, there will be at most N22\frac{\frac{N}{2}}{2}22*N*​​ votes. And so on.

Thus, the total number of votes will  
N2+N22+N222+...\frac{N}{2} + \frac{\frac{N}{2}}{2} + \frac{\frac{\frac{N}{2}}{2}}{2} + ...2*N*​+22*N*​​+222*N*​​​+...

which simplifies to  
N2+N4+N8+...+N2k\frac{N}{2} + \frac{N}{4} + \frac{N}{8} + ... + \frac{N}{2^k}2*N*​+4*N*​+8*N*​+...+2*kN*​

where kk*k* is the maximum number of rounds that we can have.

This can be approximated to

N2+N4+N8+...\frac{N}{2} + \frac{N}{4} + \frac{N}{8} + ...2*N*​+4*N*​+8*N*​+...

where the number of terms is infinite.

Now, this is a geometric progression with common ratio 12\frac{1}{2}21​ and first term N2\frac{N}{2}2*N*​. Thus, the [sum](https://en.wikipedia.org/wiki/Geometric_series#Sum) of this progression is

N2⋅11−12=N2⋅2=N\frac{N}{2} \cdot \frac{1}{1-\frac{1}{2}} = \frac{N}{2} \cdot 2 = N2*N*​⋅1−21​1​=2*N*​⋅2=*N*

Thus, the total number of votes that we can have is NN*N*.

↓\downarrow↓

↓\downarrow↓

Now, the question boils down to how to find the optimal (next closest) opponent senator **efficiently**. The article presents multiple approaches to solving this problem.

**Approach 1: Greedy**

**Intuition**

As discussed above, the "*best strategy*" for a senator is to ban the next closest opponent senator. Thus, we can use a greedy approach to solve this problem.

For every "eligible senator", we can try to find the next closest opponent senator and ban it. To find this senator, we can linearly scan the senate string ahead of the "current senator", to the right. If no senator is found, we can start scanning from the left of the senate string until one position before the "current senator". If no senator is found, we can return the party of the "current senator" as the winner.

Also, to ban the next closest opponent senator, we can just remove the banned senator from the senate string.

Moreover, one minute optimization that we can do is to keep track of the number of senators from each party. If the number of senators from one party becomes zero, we can return the other party as the winner.

**Algorithm**

1. We need to remove banned senators from the senate string. If strings are immutable in your programming language, then convert the input to an equivalent mutable data structure (like a dynamic array).
2. Count the number of senators from each party, and store it in rCount and dCount respectively.
3. Define a function ban(toBan, startAt) which will ban the next closest opponent senator of type toBan, starting from the position startAt in the senate string. It will return a boolean value, which will be True if we have to loop around the senate string for banning the next closest opponent senator. This will help us in dealing with the indices of the senate string.
4. Define a variable turn which will keep track of the current senator. It will be an integer value, which will be the index of the current senator in the senate string. Initially, it will be 0.
5. While we have senators from both parties, we will keep on banning the next closest opponent senator of the current senator at index turn in the senate string. We will keep on doing this as long as we still have senators from both parties.

After banning, we will decrement the count of the banned senator's party by 1.

*Also, if the senator was banned before this index, it means the senator having the next turn will be the senator at the same index. Only in this case, we will decrement the turn by 1.*

At last, we will increment the turn by 1. Take MOD of turn with the length of the senate string, to handle the wrap-around by keeping it in the range of 0 to senate.length - 1.

1. If the number of senators of one party is 0, we can return the other party as the winner.

**Implementation**

class Solution {

public:

  string predictPartyVictory(string senate) {

    // Count of Each Type of Senator to check for Winner

    int rCount = count(senate.begin(), senate.end(), 'R');

    int dCount = senate.size() - rCount;

    // Ban the candidate "toBan", immediate next to "startAt"

    // If have to loop around, then it means next turn will be of

    // senator at same index. Returns loop around boolean

    auto ban = [&](char toBan, int startAt) {

      bool loopAround = false;

      int pointer = startAt;

      while (true) {

        if (pointer == 0) {

          loopAround = true;

        }

        if (senate[pointer] == toBan) {

          senate.erase(senate.begin() + pointer);

          break;

        }

        pointer = (pointer + 1) % senate.size();

      }

      return loopAround;

    };

    // Turn of Senator at this index

    int turn = 0;

    // While No Winner

    while (rCount > 0 && dCount > 0) {

      // Ban the next opponent, starting at one index ahead

      // Taking MOD to loop around.

      // If index of banned senator is before current index,

      // then we need to decrement turn by 1, as we have removed

      // a senator from list

      if (senate[turn] == 'R') {

        bool bannedSenatorBefore = ban('D', (turn + 1) % senate.size());

        dCount--;

        if (bannedSenatorBefore) {

          turn--;

        }

      } else {

        bool bannedSenatorBefore = ban('R', (turn + 1) % senate.size());

        rCount--;

        if (bannedSenatorBefore) {

          turn--;

        }

      }

      // Increment turn by 1

      turn = (turn + 1) % senate.size();

    }

    // Return Winner depending on count

    return dCount == 0 ? "Radiant" : "Dire";

  }

};

**Complexity Analysis**

Let NN*N* be the number of senators in the senate.

* Time complexity: O(N2)O(N^2)*O*(*N*2).
  + Counting the number of senators of each type is O(N)O(N)*O*(*N*) time.
  + As discussed in [Overview](https://leetcode.com/problems/dota2-senate/editorial/?envType=study-plan-v2&envId=leetcode-75#overview), there will be O(N)O(N)*O*(*N*) turns/votes.  
    *Each turn will take*O(N)O(N)*O*(*N*)*time to find the next senator to ban. Also, removing an element from an array is*O(N)O(N)*O*(*N*)*time. Thus,****each turn****requires*O(2N)O(2N)*O*(2*N*)*operations, which is*O(N)O(N)*O*(*N*)*time.*  
    Thus, O(N)O(N)*O*(*N*) turns/votes requires O(N2)O(N^2)*O*(*N*2) time.

Hence, the overall time complexity will be O(N+N2)=O(N2)O(N + N^2) = O(N^2)*O*(*N*+*N*2)=*O*(*N*2).

* Space complexity: O(N)O(N)*O*(*N*).

If the string is mutable, then we can do it in place.

However, strings are often immutable. Thus, we need to use a new data structure of size NN*N* to store the senate. Hence, the space complexity will be O(N)O(N)*O*(*N*).

**Approach 2: Boolean Array**

**Intuition**

The previous [approach](https://leetcode.com/problems/dota2-senate/editorial/?envType=study-plan-v2&envId=leetcode-75#approach-1-greedy) was not efficient enough and also suffers from nuances of string manipulation, particularly the deletion of characters and maintaining the turn invariant (decrementing if loopAround)

The main purpose of deletion was to maintain only the senators who are still eligible. If we are planning to NOT delete the banned senators, then we can use a boolean array to keep track of the senators who are banned.

This will also help us to maintain the turn invariant. We can simply increment the turn by 1 and take MOD to loop around. The size of the senate string will not change in this approach. Also, there is no need to maintain the loopAround boolean. Further, we don't need to convert the string to a character array.

As done in the previous approach too, we will keep track of the count of each type of senator. If any of the counts reaches 0, then we will return the winner.

**Algorithm**

1. Create a boolean array banned of size NN*N* and initialize it to false. This will keep track of the senators who are banned.
2. Create a count of each type of senator, rCount and dCount.
3. Define a function ban which takes in the type of senator to ban, toBan and the index to start searching for the next senator to ban, startAt.

Inside the function define a pointer pointer and initialize it to startAt. While an eligible senator of type toBan is not found, keep incrementing the pointer by 1 and taking MOD length NN*N* to loop around.

If found, ban the senator by setting banned[pointer] to true.

The loop will terminate because we will call the function only when there is at least one eligible senator of type toBan in the senate. Thus, we will always find an eligible senator of the type toBan in the senate.

1. Define a variable turn and initialize it to 0. This will keep track of the index of the senator whose turn it is.
2. While there is no winner, keep iterating over the senate.

If the senator at turn is not banned, then check the type of the senator, and call the ban function with the type of the opponent senator and the index of the next senator, i.e. (turn + 1) % senate.length(). After executing the ban function, decrement the count of the opponent senator by 1.

Increment the turn by 1 and take MOD NN*N* to loop around. If the senator at turn is banned, do this immediately without checking the type of the senator, otherwise do this after executing the ban function.

1. Return the winner depending on which senator's count has dropped to 0.

**Implementation**

class Solution {

public:

  string predictPartyVictory(string senate) {

    // Count of Each Type of Senator to check for Winner

    int rCount = count(senate.begin(), senate.end(), 'R');

    int dCount = senate.size() - rCount;

    // To mark Banned Senators

    vector<bool> banned(senate.size(), false);

    // Ban the candidate "toBan", immediate next to "startAt"

    auto ban = [&](char toBan, int startAt) {

      // Find the next eligible senator of "toBan" type

      // On found, mark him as banned

      while (true) {

        if (senate[startAt] == toBan && !banned[startAt]) {

          banned[startAt] = true;

          break;

        }

        startAt = (startAt + 1) % senate.size();

      }

    };

    // Turn of Senator at this Index

    int turn = 0;

    // While both parties have at least one senator

    while (rCount > 0 && dCount > 0) {

      if (!banned[turn]) {

        if (senate[turn] == 'R') {

          ban('D', (turn + 1) % senate.size());

          dCount--;

        } else {

          ban('R', (turn + 1) % senate.size());

          rCount--;

        }

      }

      turn = (turn + 1) % senate.size();

    }

    return dCount == 0 ? "Radiant" : "Dire";

  }

};

**Implementation Note :** Instead of using a boolean array to mark banned senators, we can use a set if it supports O(1)O(1)*O*(1) lookup and addition.

**Complexity Analysis**

Let NN*N* be the number of senators in the senate.

* Time complexity: O(N2)O(N^2)*O*(*N*2).
  + Counting the number of senators of each type is O(N)O(N)*O*(*N*) time.
  + As discussed in [Overview](https://leetcode.com/problems/dota2-senate/editorial/?envType=study-plan-v2&envId=leetcode-75#overview), there will be at most NN*N* turns. Thus, if !banned[turn] in while (rCount > 0 && dCount > 0) will be executed at most NN*N* times.  
    *In each turn, we will iterate over the entire senate string to find the next eligible senator to ban. This is bounded by*NN*N as well.*

Thus, the overall time complexity is O(N2)O(N^2)*O*(*N*2).

* Space complexity: O(N)O(N)*O*(*N*).

We use a boolean array of size NN*N* to mark banned senators. However, compared to [previous approach](https://leetcode.com/problems/dota2-senate/editorial/?envType=study-plan-v2&envId=leetcode-75#approach-1-greedy), we have overcome the nuances of maintaining the turn invariant.

**Approach 3: Binary Search**

**Intuition**

In the previous approach, the biggest bottleneck was the **search** for the next eligible senator to ban. We know that we can optimize any **search** using [**binary search**](https://leetcode.com/explore/learn/card/binary-search/) provided the "search space" is sorted.

**What is sorted in this situation? Click to Reveal**

Thus, using binary search, we can optimize the search for the next eligible senator to ban. Let's see if this will help us to improve the overall time complexity or not.

**Algorithm**

1. Declare a Boolean array banned of size NN*N* to flag banned senators.
2. Declare two sorted lists rIndices and dIndices to maintain indices of **eligible senators** of each party.
3. Define a function ban which takes two arguments, indicesArray which is a list of indices of the opponent party, and startAt which indicates we have to find a senator ahead of (including) this index. This function will ban the next eligible senator.

Using binary search, find the index of the next eligible senator to ban ahead of startAt. If not found ahead of startAt, we have to loop around, in this case, we can ban the first eligible senator.

Since the indicesArray will store indices of **eligible senators** only. Therefore, after finding the index of the next eligible senator to ban, mark him as banned in the banned array and remove him from indicesArray to maintain the invariant.

We will call this function only when indicesArray is not empty. Thus answer will always be found.

1. Initialize turn with 0 which indicates the index of the senator whose turn is next.
2. While both parties have at least one senator, do the following:

If senate[turn] is not banned, then find the next eligible senator to ban using the ban function. If senate[turn] is R, then find the next eligible senator to ban from dIndices and vice versa. Start from the turn index to find the next eligible senator to ban.  
*(We can start from (turn + 1) % senate.length() as well. But turn will work too because we know that senate[turn] is from the same party but we want to ban the opponent party senator.)*

Increment turn by 111. Loop around by taking modulo with NN*N* if needed.

1. Return the party which has at least one senator.

**Implementation**

class Solution {

public:

  string predictPartyVictory(string senate) {

    // Number of Senators

    int n = senate.size();

    // To mark Banned Senators

    vector<bool> banned(n, false);

    // List of indices of Eligible Radiant and Dire Senators

    vector<int> rIndices, dIndices;

    for (int i = 0; i < n; i++) {

      if (senate[i] == 'R')

        rIndices.push\_back(i);

      else

        dIndices.push\_back(i);

    }

    // Ban the senator of "indices" array next to "startAt"

    auto ban = [&](vector<int> &indices, int start\_at) {

      // Find the index of "index of senator to ban" using Binary Search

      auto temp = lower\_bound(indices.begin(), indices.end(), start\_at);

      // If start\_at is more than the last index,

      // then start from the beginning. Ban the first senator

      if (temp == indices.end()) {

        banned[indices[0]] = true;

        indices.erase(indices.begin());

      }

      // Else, Ban the senator at the index

      else {

        banned[\*temp] = true;

        indices.erase(temp);

      }

    };

    // Turn of Senator at this Index

    int turn = 0;

    // While both parties have at least one senator

    while (!rIndices.empty() && !dIndices.empty()) {

      if (!banned[turn]) {

        if (senate[turn] == 'R')

          ban(dIndices, turn);

        else

          ban(rIndices, turn);

      }

      turn = (turn + 1) % n;

    }

    // Return the party with at least one senator

    return dIndices.empty() ? "Radiant" : "Dire";

  }

};

**Implementation Note:** For Binary Search, we have used different inbuilt functions in different languages. More about them can be read from the official documentation.

* In Python, we have used the [bisect.bisect\_left](https://docs.python.org/3/library/bisect.html#bisect.bisect_left) function.
* In Java, we have used [Collections.binarySearch](https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html#binarySearch(int%5B%5D,%20int)) function.
* In C++, we have used [lower\_bound](https://en.cppreference.com/w/cpp/algorithm/lower_bound) function.
* In C#, we have used [List.BinarySearch](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1.binarysearch?view=net-5.0) function.

While there is variation in the different functions, the core idea remains the same. The function takes two parameters, an array a and a value x. It returns the index of the first element in a which is not less than x. If all elements in a are less than x, it returns the size of a. The algorithm is based on the fact that the array a is sorted.

**Complexity Analysis**

Let NN*N* be the number of senators in the senate.

* Time complexity: O(N2)O(N^2)*O*(*N*2).
  + Creating the list of indices of eligible senators takes O(N)O(N)*O*(*N*) time.
  + The if !banned[turn] condition in the while (!rIndices.empty() && !dIndices.empty()) loop is executed NN*N* times. Because there will be at most O(N)O(N)*O*(*N*) vote as discussed in [Overview](https://leetcode.com/problems/dota2-senate/editorial/?envType=study-plan-v2&envId=leetcode-75#overview).

Now, each vote will call the ban function. The ban function uses Binary Search to find the index of the senator to ban. The Binary Search takes O(log⁡N)O(\log N)*O*(log*N*) time. But, it is also removing the index from the list using the erase (or equivalent) function. This takes O(N)O(N)*O*(*N*) time. So, the total time taken by the ban function is O(N)O(N)*O*(*N*).

Hence, the total time taken by the while loop is O(N2)O(N^2)*O*(*N*2).

* Thus, the total time complexity is O(N2)O(N^2)*O*(*N*2).
* **Side Note :** **If** popping to maintain invariant of eligible senators was O(1)O(1)*O*(1), then the time complexity would have been O(N+Nlog⁡N)=O(Nlog⁡N)O(N + N \log N) = O(N \log N)*O*(*N*+*N*log*N*)=*O*(*N*log*N*).
* Space complexity: O(N)O(N)*O*(*N*).
  + The space taken by the banned array is O(N)O(N)*O*(*N*).
  + The space taken by the rIndices and dIndices array is O(N)O(N)*O*(*N*).
  + Thus, the total space complexity is O(N)O(N)*O*(*N*).

**Approach 4: Two Queues**

**Intuition**

The biggest drawback of [Approach 1](https://leetcode.com/problems/dota2-senate/editorial/?envType=study-plan-v2&envId=leetcode-75#approach-1-greedy) and [Approach 3](https://leetcode.com/problems/dota2-senate/editorial/?envType=study-plan-v2&envId=leetcode-75#approach-3-binary-search) is the deletion from the array which takes O(N)O(N)*O*(*N*) time.

Let's revisit [Approach 3](https://leetcode.com/problems/dota2-senate/editorial/?envType=study-plan-v2&envId=leetcode-75#approach-3-binary-search).

* We need a boolean array banned to keep track of the banned senators. This helped the variable turn to move forward.
* We need rIndices and dIndices to keep track of the eligible senators separately in sorted order. This helped us to find the next target to ban.

**Can we somehow combine these two? Is banned really needed? What do the rIndices and dIndices actually store?**

Let's say the first radiant senator from left is r0 and the first dire senator from left is d0. Then, rIndices and dIndices will store the indices of r0, r1, r2, ..., rp and d0, d1, d2, ..., dm respectively.

* Since indices (rIndices and dIndices) are sorted, and they also cover the entire senate, we can say that the turn will be the minimum of the first index/element of rIndices and dIndices array.
* and who will be the next target to ban? The person with the first turn will choose the immediate next opponent. And where is it? It will be the maximum of the first index of the rIndices and dIndices array. Because the minimum one got the turn and the maximum (or other) one will get banned.

Thus, here comes the **key driving idea**. Take two arrays rIndices and dIndices to keep track of the indices of the eligible senators separately in sorted order. Take the first element *(which represent indices of senators)* of rIndices and dIndices and compare them.

* The minimum of these two will be the turn. It will not get banned, at least as of now. Thus, it will again be added to the array. Since it should get turn in the next round, we will add it to the end of the array, and the index will be turn + n because, in the next round, this would be the first index.
* The maximum of these two will be the next target to ban. It will get banned. Thus, it will not be added back to the array.

Now, in this approach, we are removing the element from the front of the array and adding the element to the back of the array. This is nothing but the working principle of a [Queue](https://leetcode.com/explore/learn/card/queue-stack/228/first-in-first-out-data-structure/). Removing from the front is DE-queuing and adding to back is EN-queuing.

[Queue](https://leetcode.com/explore/learn/card/queue-stack/228/first-in-first-out-data-structure/) is an efficient data structure that can help us find the next closest opponent senator as well as the next eligible voter. It also helps us in simulating the voting process from left to right. Also, it is easier to keep track of rounds of voting by assuming the index increase by NN*N* after each round.

**Algorithm**

1. Create two queues rQueue and dQueue to keep track of the eligible senators separately in sorted order.
2. Populate the queues with the indices of the respective senators from left to right.
3. While both parties have at least one Senator, do the following:
   * Pop the Next-Turn Senator index from both queues.
   * ONE having larger index will be banned by lower index. Thus, the lower index will again get Turn, so EN-Queue in the same queue with the index/turn increased by NN*N*.
4. Return the party name of the queue which is not empty.

**Implementation**

class Solution {

public:

    string predictPartyVictory(string senate) {

        // Number of Senator

        int n = senate.size();

        // Queues with Senator's Index.

        // Index will be used to find the next turn of Senator

        queue<int> rQueue;

        queue<int> dQueue;

        // Populate the Queues

        for (int i = 0; i < n; i++) {

            if (senate[i] == 'R') {

                rQueue.push(i);

            } else {

                dQueue.push(i);

            }

        }

        // While both parties have at least one Senator

        while (!rQueue.empty() && !dQueue.empty()) {

            // Pop the Next-Turn Senate from both Q.

            int rTurn = rQueue.front();

            rQueue.pop();

            int dTurn = dQueue.front();

            dQueue.pop();

            // ONE having a larger index will be banned by a lower index

            // Lower index will again get Turn, so EN-Queue again

            // But ensure its turn comes in the next round only

            if (dTurn < rTurn) {

                dQueue.push(dTurn + n);

            } else {

                rQueue.push(rTurn + n);

            }

        }

        // One's which Empty is not winner

        return rQueue.empty() ? "Dire" : "Radiant";

    };

};

**Implementation Note**

Python does not have a built-in Queue data structure. We can use [deque](https://docs.python.org/3/library/collections.html#collections.deque) from the collections module. deque is a double-ended queue that supports adding and removing elements from both ends in O(1)O(1)*O*(1) time.

Also, we have EN-queued the eligible senator out of two by adding NN*N* to the index. Although this new index is not necessarily the turn number because the number of senators in the senate can be less than NN*N*. But this is not a problem because we are only interested in ordering the senators, and not the actual turn number.

**Complexity Analysis**

Let NN*N* be the number of senators in the senate.

* Time complexity: O(N)O(N)*O*(*N*).
  + Populating the queues takes O(N)O(N)*O*(*N*) time.
  + While loop will give chance to each eligible senator to vote until the last round. The voting process for one senator takes O(1)O(1)*O*(1) time because of constant queue operations. There will be O(N)O(N)*O*(*N*) such votes as discussed in [Overview](https://leetcode.com/problems/dota2-senate/editorial/?envType=study-plan-v2&envId=leetcode-75#overview) section.
  + Hence, total time complexity is O(N+N)=O(N)O(N + N) = O(N)*O*(*N*+*N*)=*O*(*N*).
* Space complexity: O(N)O(N)*O*(*N*).

Storing the index of senators in the queues takes O(N)O(N)*O*(*N*) space. The queues will either decrease or remain the same in size in each round. They will never increase in size. Hence, space complexity is O(N)O(N)*O*(*N*).

**Approach 5: Single Queue**

**Intuition**

As clear from the heading, let's try to use a single queue of senators. The front of the queue will be the senator having the next turn.

We can pop the front of the queue.

* if the senator is eligible to vote, we can "float" the ban on the opponent senator. By "floating", we mean that we are not banning the opponent senator right now, but we are just making a note that we will ban the opponent senator if encountered in the future. This will also ensure that the banned senator is the immediate opponent of the current senator.

After doing this, we can simply push the current senator to the back of the queue because it will again get a chance to vote in the next round.

* else if the senator is not eligible to vote because previously its opponent has "floated" a ban on it, we can simply ignore it. Banning is marked by NOT adding it again to the queue. Moreover, we can also decrement the floating ban count on this party.

Thus, this thought process ensures the implementation of the greedy approach by floating the ban.

A minute optimization, we can also maintain the count of "eligible senators" of each party, and decrement it when a senator is banned. This will help us to stop the voting process when one party has no eligible senators left.

**Algorithm**

1. Count the number of senators of each party. Let's call them rCount and dCount.
2. Initialize the floating ban count of each party to 0. Let's call them rFloatingBan and dFloatingBan.
3. Initialize a queue of senators with the order same as senate.
4. While both parties have at least one eligible senator, do the following:
   1. Pop the next-turn senator from the queue.
   2. If the senator is eligible to vote, then:
      * Float the ban on the opponent party.
      * Push the current senator to the back of the queue.
   3. Else if the senator is not eligible to vote, then:
      * Decrement the floating ban count of the party.
      * Decrement the count of the party.
5. Return the party which has at least one eligible senator.

**Implementation**

class Solution {

public:

    string predictPartyVictory(string senate) {

        // Number of Senators of each party

        int rCount = 0, dCount = 0;

        // Floating Ban Count

        int dFloatingBan = 0, rFloatingBan = 0;

        // Queue of Senators

        queue<char> q;

        for (char c : senate) {

            q.push(c);

            if (c == 'R') rCount++;

            else dCount++;

        }

        // While any party has eligible Senators

        while (rCount && dCount) {

            // Pop the senator with turn

            char curr = q.front();

            q.pop();

            // If eligible, float the ban on the other party, enqueue again.

            // If not, decrement the floating ban and count of the party.

            if (curr == 'D') {

                if (dFloatingBan) {

                    dFloatingBan--;

                    dCount--;

                } else {

                    rFloatingBan++;

                    q.push('D');

                }

            } else {

                if (rFloatingBan) {

                    rFloatingBan--;

                    rCount--;

                } else {

                    dFloatingBan++;

                    q.push('R');

                }

            }

        }

        // Return the party with eligible Senators

        return rCount ? "Radiant" : "Dire";

    }

};

**Implementation Note :** Python does not have a built-in Queue data structure. We can use [deque](https://docs.python.org/3/library/collections.html#collections.deque) from the collections module. deque is a double-ended queue that supports adding and removing elements from both ends in O(1)O(1)*O*(1) time.

**Complexity Analysis**

Let NN*N* be the number of senators in the senate.

* Time complexity: O(N)O(N)*O*(*N*).
  + Counting the number of senators of each party takes O(N)O(N)*O*(*N*) time. So does populating the queue.
  + The condition while (rCount && dCount) will be executed O(N)O(N)*O*(*N*) times because they are the simulation of the voting process, which is bounded by O(N)O(N)*O*(*N*) as discussed in [Overview](https://leetcode.com/problems/dota2-senate/editorial/?envType=study-plan-v2&envId=leetcode-75#overview) section.

Inside the loop, there are O(1)O(1)*O*(1) operations.

* + So the total time complexity is O(N+N)=O(N)O(N + N) = O(N)*O*(*N*+*N*)=*O*(*N*).
* Space complexity: O(N)O(N)*O*(*N*).

The Queue will have NN*N* senators initially. The number can only decrease but can never increase. So the space complexity is O(N)O(N)*O*(*N*).