# Introduction / Problem Statement

## What's the problem?

If you've spent time programming, you've likely encountered the concept of **palindromes**—sequences that read the same forward and backward. While palindromes might seem like a trivial concept at first, they actually have meaningful applications in fields like **bioinformatics**, **natural language processing (NLP)**, and **data compression**. Here are a few examples:

- **DNA/RNA Structures**: In molecular biology, palindromic regions in DNA and RNA can fold into hairpin or stem-loop structures, which play key roles in gene regulation and function.
- **Pattern Recognition**: In data compression, algorithms like **LZ77** and **LZ78** can exploit palindromic patterns to identify and compress repeated structures more effectively.
- **Error Correction**: Some advanced encoding schemes use palindromic properties to enhance data integrity and support robust error detection.

Today's problem is one of the most classic and elegant about palindromes:

> Find the **longest palindromic substring** in a given string (A *substring* refers to a contiguous sequence of characters within a string).

## Some Naive Approaches

To find the longest palindromic substring in a given string, the most naive approach uses two nested loops to generate all possible substrings and check each one:

```
for start in range(len(s)):
  for end in range(start + 1, len(s)):
    check_is_palindrome(s[start:end+1])
```

This method has a time complexity of **$O(n^3)$**: one layer for the start index, one for the end index, and another for checking if the substring itself is a palindrome (which takes O(n) time). Clearly, this is inefficient and impractical for long strings (Like DNA/RNA sequences or massive data compression).

If we look more closely at the structure of palindromes, we realize that using `start` and `end` to define substrings is not optimal. Palindromes are **symmetric**, so instead of scanning all possible substrings, we can iterate through each character and expand outward from it as a **center**:

```
for i in range(len(s)):
```

```
    radius = 0
    while i - radius >= 0 and i + radius < len(s) and s[i - radius] == s[i
+ radius]:
      radius += 1
    max_length = 2 * radius + 1
```

- This approach is more intuitive and avoids checking every substring, but it still has two major drawbacks:
  - **Time complexity is still O(n²)**: In the worst case, each center may expand up to the entire string.
  - **It only detects odd-length palindromes**: since the center is assumed to be a single character. Even-length palindromes (like `"abba"`) are ignored.

### Our Goal

To overcome these limitations, the algorithm we seek should:

> - **Achieve a target runtime of O(n).**
> - **Be able to detect both even- and odd-length palindromes efficiently in any string.**

## Manacher's Algorithm – Idea

### Detect both even- and odd-length palindromes

Let's start solving the problem!

Before we dive into optimizing performance, we'll first address a foundational challenge: **how to uniformly detect both odd- and even-length palindromes**.

Odd-length palindromes are naturally handled when we expand around a single character. The real difficulty lies in **even-length palindromes**. Intuitively, we can interpret these palindromes to have a center between characters. For example:

```
ab ba
  ^
```

How can we formalize this idea?

We can treat the *space* between every pair of characters as a valid center by **inserting a special character** between each letter. This character should not interfere with the original string's content—so instead of inserting an actual letter (which could create false matches), we typically use a **non-alphabetic symbol like `#`**. For example:

```
abba => #a#b#b#a#
abcdef => #a#b#c#d#e#f#
```

After this transformation, we no longer need to distinguish between even- and odd-length palindromes—**every palindrome is now centered on a single character**, whether it's a letter or a `#`.

## Achieve O(n) runtime

Now comes the real challenge: **how do we achieve O(n) runtime?**

Let's look at our second brute force algorithm. The good news is that we already have a single outer loop, iterating through all possible centers. So, if we can reduce the **inner palindrome expansion** to **amortized O(1)** per center, the entire algorithm will run in **O(n)** time.

This is where we borrow an idea similar to **dynamic programming**: instead of recomputing everything from scratch, we **store useful information** from earlier computations and reuse it when possible.

What kind of information can we store? Let's examine an example:

```
#a#b#c#b#c#b#a#
     ^ ~ ^
```

What can we say about the length of the longest palindromic substring centered at two characters marked by `^`? They are the same! How do we know? Except for just counting them, we can also tell this by the fact that they are at two **symmetrical positions** with respect to the center `~` (which is `b` in the middle). If we already know that the entire region around `~` forms a palindrome, then:

- The substring centered at the left `^` has a known radius.
- The substring centered at the right `^` must have **at least the same radius** unless it hits the boundary of the longer palindrome.

This observation leads to a key optimization:

> **If a center lies within a previously known palindrome, its mirrored counterpart (with respect to the current center of the known palindrome) has already computed useful information.**

Therefore, instead of expanding every center from scratch, we can:

- Maintain the **rightmost boundary** of any palindrome found so far and its **corresponding center**.
- Keep an array `P` to store the radius of the longest palindromic substring centered on each character.
- Use the **mirror** of the current character about the stored center to guess the radius (stored in `P`).
- Only expand further if the guessed radius reaches the boundary.

This is the core insight behind **Manacher's Algorithm**, which leverages symmetry and previously computed results to eliminate redundant computations and bring the runtime down to **O(n)**.

# Manacher's Algorithm – Implementation

## Implementation using python

Now let's dig into detailed implementation. First, insert `#` into the string:

```
new_s = '#' + "#".join(s) + '#'
```

We need to keep track of the following three values:

```
p = [0] * n  # p[i] will hold the radius of the longest palindrome centered at i
center = 0   # The center of the palindrome that extends farthest to the right (i.e. rightmost palindrome)
right = 0    # The right boundary of the palindrome centered at 'center'
```

Next, we begin scanning the string to find the longest palindromic substring centered at each character. Before expanding from a center to check for palindromes, we first try to **leverage previously computed information**. Since we've already recorded palindromic spans up to "`right`", we can use this information to make an **initial guess** about the radius at the current center.

```
for i in range(k):
  mirror = 2 * center - i  # Mirror position of i around the current center

  if i < right:
```

```
        # Use the mirror's result if it doesn't go beyond the right boundary
        # Length of the new palindrome should be at least the same as its
    mirror
        p[i] = min(right - i, p[mirror])
```

Next, we attempt to **expand beyond the guessed radius** to find the actual longest palindrome centered at the current position—just like in the traditional center-expansion approach. If this newly found palindrome extends past the current `right` boundary, we update both `center` and `right` to reflect the new rightmost-reaching palindrome.

```
    for i in range(k):
            mirror = 2 * center - i
            if i < right:
                p[i] = min(right - i, p[mirror])

            # Attempt to expand the palindrome centered at i beyond guessed
    raduis
            # Compare characters symmetrically around i as long as they
    match
            while i + p[i] + 1 < n and i - p[i] - 1 >= 0 and t[i + p[i] + 1]
    == t[i - p[i] - 1]:
                p[i] += 1

            # If the expanded palindrome goes beyond the current right
    boundary,
            # update the center and right to reflect the new rightmost
    palindrome
            if i + p[i] > right:
                center = i
                right = i + p[i]
```

After getting the entire array `p`, you can extract either the length of the longest palindrome or its position.

The detailed implementation using different programming languages, including Python, Java, and C++, can be found in our bundled files.

## Time/Space complexity analysis

As discussed earlier, the algorithm contains only a single outer loop, which iterates `k` times — where `k` is the length of the preprocessed string with inserted `#` characters. Although this transforms the original string of length `n` into one of length `2n + 1`, the overall complexity remains **O(n)**.

The only potentially non-constant operation inside the loop is the **while loop** used for

expanding palindromes. However, this expansion only occurs when it **extends the `right` boundary** of the currently known longest palindrome. Since `right` can only move forward—from `0` to at most `k`—the total number of expansions across the entire algorithm is bounded by `k`.

This means the expansion step has **amortized O(1)** time complexity per iteration, leading to a total runtime of **O(n)**.

For space complexity, the only thing that needs a lot of space is `P` (which stores longest radius at each position) and `new_s` (the expanded string), and this costs **O(n)**.

# Programming Challenge

## Problem idea – Maximum Valid DNA Helix

To encourage deeper learning and hands-on understanding, we challenge students to **modify Manacher's Algorithm** to solve a broader class of problems involving symmetry.

Instead of finding the longest palindromic substring, we ask students to find the longest **DNA Helix**—a substring that forms a valid double-stranded DNA structure. In this version, characters no longer match by identity. Instead, valid pairings follow the biological base-pair rules:

- `'A'` pairs with `'T'`
- `'C'` pairs with `'G'`

## Idea for solution

This problem is almost identical to the classic longest palindromic substring problem!

The key difference is that instead of matching identical characters, we now want to match each character with its **corresponding pair**. To handle this, we can define a **mapping dictionary** that specifies valid character pairings.

```
mapping_dict = {
    'A': 'T', 'T': 'A', 'C': 'G', 'G': 'C', '#': '#',
}

def matches(c1, c2):
  return mapping_dict[c1] == c2
```

But wait—is that really all? Is this problem **that** simple?

At first glance, it may seem like just a minor tweak to the original palindrome problem. However, after implementing the naive solution and running it on several test cases, you may start to notice some unexpected behavior.

Consider the following example:

```
  G     G     C     C     C
        ^     ^     ^
     mirror center  i
```

When using Manacher's Algorithm, calculating the longest radius at a position `i` involves looking at its `mirror` with respect to the current `center`.

Now, in the above example, the mirror position has a radius of 1. Intuitively, this suggests that the position `i` should also have at least a radius of 1. However, in our modified problem, we might find that the radius at `i` is actually 0—'C' and 'C' **do not form a valid pair** under our custom mapping.

The problem arises because:

> **`center` does not record enough information for itself.**

What does this mean? First, look at our original palindrome case:

```
  1     2     1     2     1
  ^     ^     ^     ^     ^
  r  mirror center  i     l
```

We see that `center`, `r` (The mapping of `center` by `mirror`), and `l` (The mapping of `center` by `i`) must be the same.

But in our altered case, things are different:

```
  G     G     C     C     C
  ^     ^     ^     ^     ^
  r  mirror center  i     l
```

We see that `center` matches `r` but does not match `l`. This means that our algorithm only ensures that characters AROUND `center` matches, but NOT `center` itself.

So how do we fix this? Once we understand the source of the issue, the solution becomes simple and intuitive! We just add one more check: check whether `s[center] == s[i + (i - center)]`. If so, do things as usual. If not, this means that the longest radius at `i` is `i - center - 1`.

### Inspiration

At first glance, the problem we designed seemed deceptively simple—just a small twist on the classic longest palindromic substring task. However, this illusion fades when the **"magic" of Manacher's Algorithm breaks down** at the `center` under our modified matching rules.

This challenge highlights an important lesson: to adapt powerful algorithms like Manacher's to new settings, one must have a **deep understanding of their inner workings**—not just how to implement them, but why they work.

By confronting this breakdown, students gain valuable insights into:

- The structural assumptions underlying Manacher's Algorithm,
- The importance of considering edge cases when applying what they learned before to something similar,
- How to generalize or repair an algorithm when its assumptions no longer hold.

This exercise also opens the door to applying Manacher's approach to **other symmetry-based problems**, teaching students not only where this algorithm shines, but also **where and why it needs modification**.

## Conclusion

Today, we explored **Manacher's Algorithm**—a remarkably powerful technique for finding the longest palindromic substring in linear time. This algorithm opens the door to a new way of thinking: a world that is **symmetric, but not linear**.

Manacher's approach beautifully illustrates a fundamental principle in algorithm design:

> **Store useful information along the way, and use it wisely.**

By analyzing a slightly altered version of the palindrome problem, we gained a **deeper understanding** of how Manacher's Algorithm works, where its assumptions lie, and how it can be modified to handle new challenges involving symmetry.

This experience shows not only the power of the algorithm itself, but also the importance of **understanding algorithms at a conceptual level**—so that we can extend, adapt, and apply them to a wider range of problems.