Abstract

Recently I came across a simple palindrome problem. Which inputs an integer number and check if it's a palindrome. In case you don't know, palindrome sequence of characters or number which reads the same backward as forward.

The Palindrome Algorithms.

These are very basic and simple problems to solve. And most of the time's people tend to use an array, string, list, or any data structure of a similar manner to store data. I had solved loads of palindrome problems in past. The main algorithm is to split the data in half, then reverse check. bingo !!! you're done. But I was skeptical if the same can be done without using any data structure. palindrome general algorithm

The benefit of ditching data structures.

I was optimistic if I can slice and reverse a piece of number, which is technically a data structure niche. Maybe i can optimize this palindrome algorithm to a whole new level. Since using and iterating through unsorted data structure is a resource-consuming task that rises with the amount of data linearly or sometimes exponentially depending on related factors.

So my journey toward excellence begins. Soon and pretty simple enough I found the first piece of grail. Before I can quantitively slice anything. I need to find the size of that entity. and for integer, it's simple math equitation.

Find the size of an integer.

$$f(n) = \lfloor \log 10 \ (n) + 1 \rfloor$$

See! pretty simple you just need to log a number with a 10 base logarithmic function. then add 1 and voila!!

```c
#include <math.h>
int sizeof_int(int* n){
    return (int)floor(log10(*n) + 1);
}
```

You just calculate the size of an integer without a loop or any recursive operation. and its time complexity is `O(1)`.

Slicing integer.

Since we calculate the **size of** an integer, now we can slice them in two with a **division** and **modulus** operation. Just like we did in MSD and LSD Calculation.

Splitting algorithm

```c
int integerSlicer(int *number, int *first_part, int *second_part)
{
    //size of this number.
    int sizeof_number = sizeof_int(number);
    //trimming for odd number :)
    int range = sizeof_number - sizeof_number % 2;
    *first_part = floor(*number / pow(10, sizeof_number - (range / 2)));
    *second_part = *number % (int)pow(10, range / 2);
    return 0;
}
```

now its time complexity is `O(1)`. Though the Modulus operator is a bit slower. But it's insignificant.

Integer reversing

Up until now, We perform some critical operations with peerless efficiency. And avoid iteration, repetition as much as possible. But Decimal Reversing is a process where each element has to be picked individually and place at its significant figure subsequently.

First, we have to take an empty integer variable assigned to zero for storing. Then we will slice the Most significant digit and add them to the new variable according to its significant figure.

```
int digit_reverser(int digit, int size_of_digit)
{
    if (size_of_digit <= 1)
        return digit;

    int local_significant_position = floor(pow(10, size_of_digit - 1));
    int data = floor(digit % 10) * local_significant_position;
    return data + digit_reverser(floor(digit / 10), (size_of_digit - 1));
}
```

Visually speaking:

Reverse algorithm math

This particular function's time complexity is O(log n). it is the bottleneck of the entire program. Now, all we have to do is check the other part if those are equal.

End

This concludes the article.

Since we have all the necessary elements to complete this program. First, try to complete this code on your own.
And Check hereafter
You Can try some problems with your newly gained knowledge.